

# Reducing the Availability Management Overheads of Federated Content Sharing Systems

Christopher Peery, Thu D. Nguyen  
Department of Computer Science  
Rutgers University, Piscataway, NJ 08854, USA  
{peery, tdnguyen}@cs.rutgers.edu

Francisco Matias Cuenca-Acuna  
FaMAF, Universidad Nacional de Córdoba  
Córdoba 5000, Argentina  
matias@hal.famaf.unc.edu.ar

## Abstract

*We consider the problem of ensuring high data availability in federated content sharing systems. Ideally, such a system would provide high data availability in a device transparent manner so that users are not faced with the time-consuming and error-prone task of managing data replicas across the constituent devices of the system. We propose a novel unified availability model and a decentralized replication algorithm to approximate this ideal. Our availability model addresses three different concerns: availability during connected operation (online), availability during disconnected operation (offline), and availability after permanent disconnection from the federated system (ownership). Our replication algorithm centers around the intuition that devices should selfishly use their local storage to ensure offline and ownership availability for their individual owners. Excess storage, however, is used communally to ensure high online availability for all shared content. Evaluation of an implementation shows that our algorithm rapidly reaches stable and communally desirable configurations when there is sufficient space. Consistent with the fact that devices in a federated system are owned by different users, however, as space becomes highly constrained, the system approaches a non-cooperative configuration where devices only hoard content to serve their individual owners' needs.*

## 1. Introduction

Increasingly, users are accessing their computing environments from multiple personal devices such as PDAs, laptops, and PCs. Users often maintain significant state on each of these devices to increase responsiveness during connected operation as well as for availability during disconnected (or low-bandwidth) operation. Simultaneously, complex sharing patterns are driving users to share data through multiple logical systems such as Web-based, CVS, and file

systems. These trends are combining to fundamentally raise the overheads of data management, requiring users to track data replicas across multiple logical systems, each of which may span numerous heterogeneous devices.

In this paper, we consider the problem of automatic data replication and placement in an infrastructural layer such as a federated file system that can be used as a unifying sharing mechanism. Specifically, suppose we have a content sharing community comprised of groups of personal devices belonging to specific users and servers providing communal resources—an environment that is quite common to today's enterprise systems as well as many peer-to-peer networks. Ideally, we would replicate content across these devices so that any shared content would be accessible from any device at anytime with high probability, regardless of the device's connection status at the time of an access. This device and connection transparency allows users to avoid the need to reason about the placement of data replicas on specific devices. In addition, if a user becomes permanently disconnected from the community,<sup>1</sup> he should still be able to access any content in a snapshot of the system from his personal devices with high probability. This latter requirement removes the need for users to hoard content external to the federated system to ensure availability in case of such a permanent disconnection.

The above ideal situation is, of course, impossible to achieve. We instead propose the following intuitive 3-part availability model to approximate the ideal: each user should be able to (1) access any file from any device connected to the federated system with high probability—this part corresponds to the traditional definition of availability, which we will call *online availability*, (2) access files within a *working set* from any device within a set of *personal* devices when operating in disconnected mode with high probability—we call this *offline availability*, and (3) access files that he *owns* from his personal devices with high

---

<sup>1</sup>Permanent disconnection can arise for a number of reasons, including catastrophic failure of the federated system, dissolution of the federation, and the user simply leaving the federation permanently.

probability, even if he becomes permanently disconnected from the community—we call this *ownership availability*.

Our model departs from the ideal model in two ways. First, during disconnected operation, the user is limited to working from one of his personal devices and can only access files local to that device. With high probability, these local files will include his working set. This deviation is both reasonable and intuitive given that users do typically constrain disconnected operation to their personal devices rather than arbitrary devices in the federated system. Further, when operating in disconnected mode, the user is limited to the resources available on his local device and thus cannot expect to have access to all shared content. Having a well-defined working set automatically placed on his personal devices should go a long way toward meeting a user's need for disconnected operation with minimal data management overheads [11]. The second deviation occurs when a user becomes permanently disconnected from the community. In this case, his devices may only contain the subset of files that he owns, rather than a snapshot of the entire system. Again, this difference is reasonable and intuitive given the finite resources of a user's device set and the fact that the user is unlikely to be interested in the content of the entire federated system.

The core contributions of this paper, in addition to the availability model itself, is then the design (Section 2), implementation (Section 3), and evaluation (Section 4) of a replication algorithm to support this availability model. The key intuition behind our replication algorithm is that devices belonging to a user should prioritize offline and ownership availability for that user over online availability for the community. This is because the primary use of a personal device is to store content that the user cares about; i.e., content in the user's working set, which he will likely access in the near future, and content that the user owns, which he would want in the case of permanent disconnection. However, it is beneficial for devices to collaborate to maintain high online availability for all shared content because this allows all users to find new content of interest to them as well as easy access to content that they have not used in a long time. Thus, in our algorithm, devices selfishly use their local storage to store files in their owners' working and ownership sets (a user's ownership set is the set of all files owned by that user). However, excess storage across the federated system is used to collaboratively ensure high online availability for all shared content. Server-like devices that do not belong to any single user can also be added to the system to ensure that sufficient communal storage is available to maintain high online availability.

Critically, our replication algorithm explicitly considers the impact of the selfish hoarding actions of individual devices on the online availability of shared content. This means that *hot* content, i.e., content that has been re-

cently accessed by many users, typically does not need to be replicated for online availability. On the other hand, as content becomes cold, the algorithm will ensure that sufficient replicas remain in the system to maintain a target online availability level. Also, all key replication decisions are made autonomously by individual devices using only a small amount of loosely synchronized global state—this autonomy is important because devices in a federated system may join and leave the online system unpredictably and may have low online availability (e.g., laptops that are often turned off). Devices can also arbitrarily leave the system permanently.

Our replication algorithm is based on previous work as described in [5]. We have extended this previous work significantly, however, to support the above unified availability model, to support changes to the subset of shared content that is hoarded local to each device, to loosely coordinate the actions of multiple devices belonging to a single user, and to accommodate mutable shared content.

We have also implemented the replication algorithm in a prototype federated file system called Wayfinder [15]. Wayfinder is a file system designed to provide a unified data view over a federation of devices. One of the critical abstractions that Wayfinder exports is a dynamically constructed global namespace that is uniformly accessible across connected and disconnected operation. As such, it provides a perfect hosting infrastructural layer for our replication algorithm. Coupling our replication algorithm with a global namespace removes the need for users to reason about the placement of replicas for both locating data and ensuring data availability. The burden of data management for each user is thus reduced to reasoning about what availability properties he desires for specific portions of the global namespace. Note that the latter is an already existing and necessary part of participating in a federated system unless the user's devices have sufficient resources to hoard all shared content.

We evaluate the above implementation using a micro-benchmark as well as a macro-benchmark derived from traces of a wiki web. The macro-benchmark is particularly relevant because wikis are designed to allow communities of users to collaboratively maintain shared sets of web pages. While wikis are extremely useful tools for collaboration, they force users to explicitly partition their data between their local file systems and the wiki systems. Further, users are forced to learn and use the wikis' specific tools for editing, search, and file management. In contrast, when using Wayfinder, the web pages can be maintained as a shared file system that is a natural part of each user's normal file system namespace. Users can use their standard everyday tools to manipulate these files. Further, a user can access these files from any of their personal devices without having to worry about managing replicas. Finally, the pages

can still be served in the Web browsing format by having a vanilla Web server serving the appropriate content.

Our evaluation shows that the algorithm achieves our design goals. In particular, the algorithm rapidly reaches stable and communally desirable configurations when there is sufficient space. Consistent with the fact that devices in a federated system are owned by different users, however, as space becomes constrained, the system approaches non-cooperative configurations where devices only hoard content to serve their individual owners' needs. We also show that the algorithm imposes very modest processing and communication overheads.

## 2. Replication Algorithm

As already mentioned, the basic idea behind our algorithm is for personal devices belonging to a user  $u$  to collaborate among themselves to maintain offline and ownership availability for  $u$ 's working set and ownership set, respectively, and for all devices in the federated system to collaborate to maintain the online availability of all shared content. Toward this goal, devices belonging to a user  $u$  would: (1) hoard replicas of files in  $u$ 's working and ownership sets, and (2) push replicas of files owned by  $u$  throughout the system to achieve a target online availability level.<sup>2</sup> Assuming that each file is owned by at least one user, then the push component of our algorithm serves to maintain online availability for all files. However, because devices prioritize offline and ownership availability over online availability, the target online availability is only achievable when there is sufficient excess storage space. In the following subsections, we first introduce some notations and several important assumptions. We then describe the algorithm in detail.

### 2.1. Terminology and Assumptions

With respect to notation, let  $WS_u$  denote the working set of a user  $u$ ,  $Own_u$  denote  $u$ 's ownership set, and  $DS_u$  be the set of all personal devices belonging to  $u$ . Also, let us divide the local storage of each device into two logical regions called *personal space* (Pspace), which is used to achieve offline and ownership availability for the device's owner, and *communal space* (Cspace), which is used to achieve the communal online availability.

To support the availability model introduced in Section 1, we assume that up to two tags can be associated with each file  $f$  for each user  $u$ : (1)  $\langle u, f, OffA, t \rangle$ , specifying that  $u$  wants high offline availability for  $f$  until the expiration time  $t$ , and (2)  $\langle u, f, OwnA \rangle$ , indicating that  $u$  wants high ownership availability for  $f$ . The set

<sup>2</sup>Although we talk about replicating *files* because files are a well-understood content encapsulating abstraction, our replication algorithm should be applicable to arbitrary data objects.

$\{f | \exists \langle u, f, OffA, t \rangle, \text{ where } t > \text{the current time}\}$  then defines  $u$ 's working set at any point in time while the set  $\{f | \exists \langle u, f, OwnA \rangle\}$  defines  $u$ 's ownership set.<sup>3</sup> As shall be seen in Section 3.2, in an implementation, *tag inheritance*, e.g., tagging a directory in a federated file system and specifying that files and subdirectories inside that directory should inherit the tags, and *automatic system tagging*, e.g., automatic tagging of files recently accessed by a user for high offline availability, makes this tagging scheme practical.

We also assume that: (1) for each user  $u$ ,  $DS_u$  contains at least 1 device that is online most of the time—that is, this device has high online availability with respect to the federated system; (2) given a file  $f$ , each device can inexpensively determine the latest version of  $f$  and the approximate location of all replicas of  $f$  as well as all replicas of a specific version of  $f$ ; (3) each device can inexpensively track the online availability of all other devices in the federated system; and (4) devices in  $DS_u$  can inexpensively track  $WS_u$  and  $Own_u$ . In Section 3, we will show how these assumptions can be supported in an actual system.

### 2.2. Replication

The replication strategy for each user  $u$  is then to: (1) replicate each file in  $Own_u$  in the Pspace of at least one device in  $DS_u$ ; (2) replicate each file in  $WS_u$  in the Pspace of all devices in  $DS_u$ ; and (3) replicate each file in  $Own_u$  in the Cspace of devices throughout the system as needed to achieve a communal online availability target  $TOA_C$ . The first component ensures that the user will have at least one copy of each file that he owns in the case of permanent disconnection. The second component ensures that the user will have access to his working set during disconnected operation, regardless of which personal device he is using. Finally, the last component ensures online availability for all shared content when there is sufficient space.

To simplify the implementation of the above strategy, we introduce the notion of *champion* devices. Each user  $u$  must designate at least one *champion* device  $C_u$  from  $DS_u$ —typically the per-user highly available device assumed above. Ideally,  $C_u$  also has plentiful storage and processing capacity. The role of  $C_u$  is then to maintain ownership availability for  $u$  and to shoulder's  $u$  portion of ensuring online availability for all shared content. (Note that while we describe the algorithm as if there is a single champion per user, each user can in fact have multiple champions without introducing added complexity.) The remainder of the devices in  $DS_u$  are only concerned with maintaining offline availability for  $u$ .

<sup>3</sup>Note that it is possible for multiple users to own the same file. In our context, ownership only means that the owner wants a copy of the file to guard against permanent disconnection and not that he has any special rights to the file compared to other users.

Further assume for the moment that  $C_u$  has sufficient capacity to store all files in  $Own_u$ . Then,  $C_u$  would monitor  $Own_u$  and download any new member of this set to its Pspace. If its local storage is full, it will evict enough files from its Cspace to accommodate the new file. Eviction is described in Section 2.3 below.  $C_u$  would also periodically, say every  $T_r$  time units, randomly select a file  $f$  from  $Own_u$  with lower online availability than  $TOA_C$  and push a replica of  $f$  to a randomly selected peer that does not yet store  $f$ .

Simultaneously, each non-champion device in  $DS_u$  monitors  $WS_u$  and downloads any new member of this set to its Pspace. If its local storage is full, then the device will evict enough files from its Cspace to accommodate the new file. If  $WS_u$  becomes larger than the device's local storage capacity, then files in  $WS_u$  are evicted in order of expiration time, nearest to furthest in the future.

If  $C_u$  cannot hold all files in  $Own_u$ , then it must ask one or more peer devices in  $DS_u$  to hold some of the files (in their Pspaces) on its behalf. This is a "golden" copy with respect to ownership availability and thus cannot be dropped by the peer device without the consent of  $C_u$ . Further, the peer device must become the champion for maintaining the online availability target for the subset of  $Own_u$  that it is storing. This is the only instance in our algorithm when two devices must explicitly coordinate. Note that if a user has multiple champion devices, it is quite easy for these devices to coordinate the partitioning of files in  $Own_u$  among themselves; they are highly available and so can easily run a standard commit protocol. This ensures that the size of a user's ownership set is not limited to the storage capacity of a single machine.

If  $C_u$  has plentiful storage, it can also monitor and hoard files in  $WS_u$  in the case that  $u$  ever needs to use it in the disconnected mode.

All devices may receive push requests from peers in the system to increase the online availability of under-replicated files. When a device receives such a request, it accepts and stores the replica in its Cspace if it has sufficient free space. Otherwise, it can either reject the request or evict from its Cspace to free up space.

### 2.3. Eviction

Eviction is a two-part process: (1) migrating files from Pspace to Cspace, and (2) evicting files from Cspace. Specifically, each device migrates each file  $f$  in its Pspace to its Cspace whenever  $f$  is no longer a member of  $Own_u \cup WS_u$ . A non-champion device holding a golden copy of a file in  $Own_u$  but not in  $WS_u$  can also migrate that file to its Cspace after  $C_u$  negotiates to take back the responsibility for the golden copy.

When evicting files from Cspace, each device should evict the files with the highest online availability. If each de-

vice deterministically evicts the most over-replicated files, however, then multiple devices running the same algorithm autonomously may simultaneously victimize the same set of files, leading to drastic changes in the files' availability. Thus, we instead use a weighted random selection process, where files with higher availability have higher chances of being selected for eviction.

This availability-conscious eviction policy can be implemented as follows. Periodically, each device computes the average number of nines in the availability of all files in its Cspace. If a push request requires eviction, then the request would be rejected if the availability of the file to be replicated is more than a threshold percentage above the computed average availability. This prevents the acceptance of a replica that will likely be evicted the next time a replication request is received by the target device. Otherwise, lottery scheduling is used to affect a weighted random selection of victims where over-replicated files are heavily penalized for their excess availability, making it highly probable that a replica of an over-replicated file will be evicted.

### 2.4. Updates

Thus far, we have described our replication algorithm as if files are immutable. When a file is updated, however, we must ensure that the update is propagated to maintain the availability of the *latest* version. (We are, of course, assuming that the file system itself does not ensure that an update is applied to all existing replicas.) If a file is updated on a non-champion device, then the device would push the new version of the file to the champion as soon as possible. Further, the device cannot evict the file until the new version reaches the champion. As shall be seen, when a system automatically tags recently accessed files to be in the accessing users' working sets, the accessing devices would naturally avoid evicting these files for some time to ensure offline availability.

When an updated file reaches the champion  $C_u$ , or if the update was performed on  $C_u$ , there are two possible cases: (1) the file is owned by  $u$ , and (2) the file is not owned by  $u$ . In the first case,  $C_u$  uses the standard periodic pushing process described above to push the new version of the modified file. One complication is that  $C_u$  is faced with the problem of computing the availability of the latest version as opposed to that of old versions of the file. We address this problem by distinguishing between *file online availability* and *version online availability*, where file availability is computed by considering all replicas of the file, regardless of the version, and version availability is computed by considering only replicas of a particular version. The champion then uses the version availability of the latest version of a file when it is considering whether a file still needs to be pushed for increased online availability. Eviction re-

mains based on file online availability. This ensures that replicas of out-of-date versions will eventually be flushed from the system since they inflate file online availability but are not maintained by any champion.

In the second case,  $C_u$  becomes a temporary champion for the updated file and pushes the new version to achieve  $TOA_C$  for it.  $C_u$  stores a replica of this version in its Pspace until it stops championing that file, at which time the replica would be migrated to its Cspace. All champion devices periodically look for updates to files in their users' *Own* sets and download the new versions. Once the true champion of an updated file has downloaded the new version, it takes over the responsibility of maintaining the online availability of that version. (Ownership availability is already ensured by the fact that the champion downloaded the new version.) Note that this hand-off is implicit in that the temporary champion will push the new version of the updated file for a period of time, after which it quits under the assumption that the true champion has found the update.

When pushing an updated file, the champion will preferentially select a device that already has a replica of a previous version of the file by giving these devices more weight in its random selection of replication targets. This limits the storage devoted to replicas of out-of-date versions (although, even without this bias, out-of-date versions would eventually be flushed from the system as explained above).

All devices, including the champions, also periodically look for updates to files in their users' working sets and download all updates to ensure high offline availability.

## 2.5. Estimating Online Availability

Assuming that devices going online and offline are independent events, the availability of a file  $f$  can be computed as  $A(f) = 1 - \prod_{d \in D(f)} (1 - A_d)$ , where  $D(f)$  is the set of devices that contain replicas of  $f$  and  $A_d$  is the availability of device  $d$ . Recall that  $D(f)$  can be computed inexpensively according to assumption (2) in Section 2.1 and the availability of all devices are known according to assumption (3). The availability of a particular version  $f_v$  of file  $f$  can be computed similarly using the set of devices that contain replicas of  $f_v$ .

## 2.6. Discussion

In this section, we discuss several important implications of the above replication algorithm. First, it is quite easy to extend our algorithm to support per-file online availability targets, as opposed to a single common target. As space becomes constrained, however, our eviction algorithm will essentially place a cap on the maximum online availability that is achievable.

Second, while users do not have to reason about replicas and their placement, they do need to be aware of the amount of storage available on their personal devices vs. the files that they want to own. As a user owns more files, more storage on his personal devices will be devoted to maintaining ownership availability, which may directly reduce offline availability for the user and online availability for the community as a whole. As already mentioned, a user can easily aggregate the resources of multiple champions to store and maintain the online availability of a large ownership set.

Third, it is easy to add devices not belonging to any user, e.g., server-like machines, to the system to provide communal space for maintaining online availability. These devices could just passively provide added storage (Cspace) to the community or actively take ownership of a portion of the shared content to help monitor and maintain online availability. Thus, our replication algorithm extends quite naturally to hybrid environments containing both personal devices and shared servers, which are quite prevalent today.

Finally, note that our replication algorithm is *not* concerned with durability. If a champion device has sufficient availability, then some of the files stored on that device may only have one replica in the system. Thus, durability must be maintained through traditional means such as external data backups.

## 3. Implementation

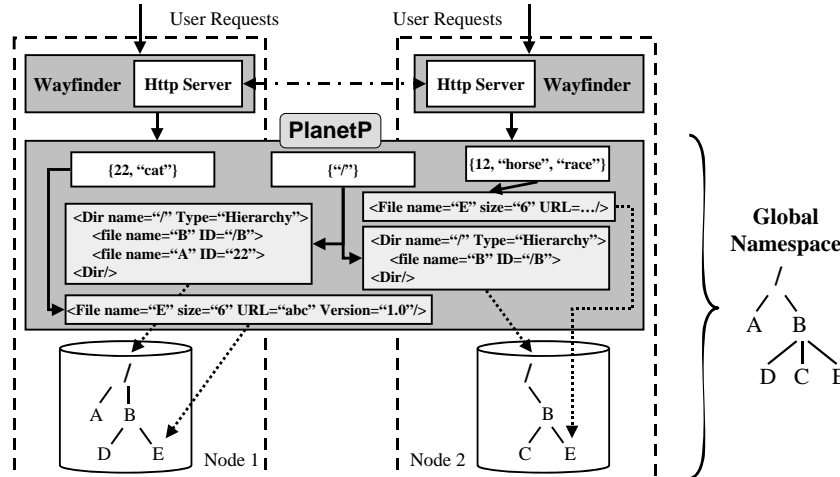
We have implemented the above replication algorithm in a prototype federated system called Wayfinder. In this section, we describe our implementation after first describing Wayfinder briefly to provide the necessary context.

### 3.1. Wayfinder

Wayfinder is a file system designed to provide a unified view of data storage across a federation of devices. As shown in Figure 1, Wayfinder stores its data in the local file systems of the individual devices comprising the federated system and its meta-data in a peer-to-peer (P2P) layer called PlanetP [6]. Thus, we begin by briefly describing PlanetP before describing how Wayfinder uses PlanetP to construct its unified data view.

**PlanetP.** PlanetP is a toolkit for building P2P applications that exports three key abstractions, a membership directory, a multidimensional indexed storage system, and a gossiping service. PlanetP's membership directory is a globally replicated database containing the name and a small set of properties for each member of a PlanetP system.

PlanetP's storage system stores bindings of the form  $\{k_1, k_2, \dots, k_n\} \rightarrow o$ , where  $k_i$  is a key and  $o$  is an arbitrary object, and we say that  $keys(o) = \{k_1, k_2, \dots, k_n\}$ .



**Figure 1.** An overview of Wayfinder’s architecture. Solid arrows inside PlanetP indicate bindings of keys to waynodes while dashed lines indicate implicit bindings of waynodes to file/directory replicas. Note that only a few waynodes are shown for clarity.

Stored objects are retrieved by specifying key-based queries using a simple query language comprised of three operators, *and* ( $\wedge$ ), *or* ( $\vee$ ), and *without* ( $-$ ). For example, a query (“cat”  $\wedge$  “dog”  $-$  “bird”) would retrieve the set  $\{o \mid (\{cat, dog\} \subseteq keys(o)) \wedge (\{bird\} \not\subseteq keys(o))\}$ .

When a binding  $\{k_1, k_2, \dots, k_n\} \rightarrow o$  is inserted into PlanetP at a node  $n$ , PlanetP stores  $o$  in a persistent store local to  $n$ , e.g., a BerkeleyDB database [17], and stores the binding itself in a persistent two-level index. The top level of this index is a globally replicated key-to-node index, where a mapping  $k \rightarrow n$  is in this *global* index if and only if at least one binding  $\{\dots, k, \dots\} \rightarrow o$  has been inserted at node  $n$ . The second level is comprised of a set of *local* indexes, one per node, which maintains the key-to-object mappings for all bindings inserted at each node. The global index is currently implemented as a set of Bloom filters [3], one per node that summarizes the set of unique keys in that node’s local index.

PlanetP evaluates a query posed at a node  $n$  by using  $n$ ’s replica of the global index to identify target peers that may contain relevant bindings. PlanetP can then either return this list of peers or directly forward the query. Contacted peers evaluate the query against their local indexes and return URLs for matching objects to  $n$ .  $n$  then merges the results and returns the list of matching objects in the form of an iterator to the calling application. The application uses the iterator to control how many and when matching objects are retrieved.

PlanetP uses gossiping to loosely synchronize both the global index and the membership directory. PlanetP also exports its gossiping layer as a tool for synchronizing application-specific data structures.

**Data Storage.** Each member device in a Wayfinder federated system stores some amount of data in a local file system, called its *hoard* (Figure 1). Each hoard contains a subset of the files in the global namespace and the corresponding directory structure. Whenever a user accesses a file  $/pathname/f$  at a device  $d$ , a replica of  $f$  and partial replicas of all directories in  $/pathname$  are downloaded to  $d$ ’s hoard. Thus, each file and directory can have many replicas across the system. Each replica is described by a small data structure called a *waynode* (the equivalent of an inode in the Unix file system). A waynode describing a file replica contains attributes such as the file ID, version, and hosting device. A waynode describing a partial directory replica contains attributes such the hosting device and the name-to-ID bindings for files and subdirectories inside that directory that are hoarded on the hosting device. The set of all waynodes comprises Wayfinder’s meta-data and is stored in PlanetP, where each waynode representing a file replica is bound to a unique file ID and version and each waynode representing a directory is bound to a directory pathname.

**Global Namespace.** Wayfinder then constructs the global namespace and locates individual files by posing appropriate queries to PlanetP. For example, to compute a *view* of the root directory for the system shown in Figure 1, Wayfinder would query PlanetP for all waynodes bound to the key “/”. This query would return two waynodes, one each from Node 1 and Node 2. The view constructed from these two waynodes would then show that root contains the file  $A$  and sub-directory  $B$ .<sup>4</sup> Note that in essence, Wayfinder con-

<sup>4</sup>To increase the efficiency of this query-based design, Wayfinder caches directory and file views using a lightweight DHT exported by Plan-

structs a directory in the global namespace by *merging* the content of the individual hoarded directory replicas with the same pathname. A key property of this merging approach for our availability work is that any node can contribute data to any portion of the global namespace. Also, any file can be placed on any node in the same manner, whether it is because of a user access or it has been placed there automatically by the system for availability.

**Access Model.** Wayfinder supports a single copy access model with eventual consistency semantics. That is, any device can modify any file as long as it can download a replica of any version of the file to its local hoard. When a file is modified, Wayfinder computes a diff that can be used to efficiently update all replicas. The diffs can also be used by users to explicitly roll back changes if they need to resolve write conflicts.

### 3.2. Replication

Relevant aspects of the implementation of our replication algorithm inside Wayfinder are as follows.

**Locating Replicas.** A device can locate the replicas of a file  $f$  with the unique ID  $f_{ID}$  by querying PlanetP for the approximate set of devices containing the key  $f_{ID}$ ; recall that the waynode for each replica of  $f$  is bound to  $f_{ID}$  on each hosting device. File IDs are obtained from directory views as described above. Since each waynode is also bound to the version of the file, we can compute the approximate set of devices holding a particular version of the file in the same manner. Note that both these computations only involve consulting the local copy of PlanetP’s global index and so does not require any network communication.

**Tracking Device Online Availability.** Each device in a Wayfinder system tracks when it is connected to the system (online) and when it is not (offline) and computes its availability as  $P(\text{online}) = \frac{\text{online time}}{(\text{online time} + \text{offline time})}$ . Each device publishes its own availability as a property in the PlanetP membership directory.

While the above definition of online availability is quite standard, in a federated system such as Wayfinder where any subset of devices can form a working subsystem, the question arises of what constitutes being connected to the system. Under the assumption that any Wayfinder system is likely to have at least a small number of highly available devices if it is to provide reasonable data availability, we define a *core set* that contains one or more of these highly available devices for each file system. Then, a device is defined to be connected to the file system if it can connect to at least one device in the core set.

---

etP. We do not describe PlanetP’s DHT and this caching process because they are orthogonal to our work here.

**File Tagging.** To minimize the overhead of reasoning about availability, Wayfinder allows users to specify tags for directories that should be inherited by descendant files and sub-directories. In essence, this allows the user to specify an availability directive for an entire portion of the namespace with just one explicit tag. In addition, Wayfinder can optionally tag files automatically for each user  $u$  as follows: (1) whenever  $u$  accesses a file  $f$ ,  $f$  is tagged with  $\langle u, \text{OffA}, t \rangle$ , where  $t$  is the current time plus a user-specified drop period; and (2) whenever  $u$  creates a new file  $f$ ,  $f$  is tagged with  $\langle u, \text{OwnA} \rangle$ . Tags are stored in waynodes.

**Maintaining Offline and Ownership Availability.** Each device in  $DS_u$  maintains copies of  $u$ ’s working and ownership sets and loosely synchronize them with its peers in  $DS_u$  using PlanetP’s gossiping service. (Non-champion devices only have to maintain their additions to the ownership set.) Whenever a device learns of an addition to the working set, it downloads the new file to its hoard as part of Pspace. The champion device does the same thing for the ownership set.

An interesting consequence of tag inheritance is that a user may become the owner of a file that he did not create. An example of this situation arises when a user is interested in maintaining a copy of all content in a directory, regardless of who first introduces that content. Thus, to properly maintain the ownership set, each champion device periodically traverses all portions of the namespace tagged for ownership by its owner. In fact, the champion does not maintain the full ownership set; rather, it only maintains the set of roots defining the portions of the namespace that it must traverse to compute the ownership set.

**Maintaining Online Availability.** Recall that each champion device  $C_u$  must periodically choose a file from  $\text{Own}_u$  that has not achieved an online availability of  $\text{TOA}_C$  and push a replica of that file out to the system. As  $C_u$  traverses the namespace to compute the ownership set, it computes the current version online availability of the latest version of each member of  $\text{Own}_u$  and inserts it into a pool of candidates if the file is under-replicated. The champion then periodically selects a file from this pool, recomputes its version availability if a threshold time period has passed since the availability was last computed, pushes a replica if still necessary, and removes the file from the pool if it has now achieved its availability target. The pool has finite size. If the pool fills up, candidates are removed from the pool using a random selection similar to that described in Section 2.3 but weighted by the version availability instead of the file availability.

**Eviction.** A pool of candidates for eviction is maintained on each device similar to the pool on each champion for replication. Each device traverses its Cspace and fills the pool whenever free space falls below some threshold. Eviction victims are then chosen from the pool as described in

Section 2.3.

**Updates.** When a file is updated so that a diff is computed, the updating device pushes the diff to its champion rather than the entire file. Whenever a device learns of an update to a file in its Pspace, it downloads the appropriate diffs and apply them to the local replica to bring it up to date.

When a champion receives a diff from a device in its device set (or generates one because of a write), if the updated file is not in its owner's ownership set, then it becomes the temporary champion for the diff as if the diff is a newly created file. Otherwise, it uses the diff to update its replica of the modified file. It then uses the standard periodic random pushing process described above to push both the diff and the new version of the modified file. Over time, however, it will become less important to maintain high online availability for the diff since all replicas of the old version should have been rolled forward and any write conflicts should have been caught. Thus, Wayfinder decreases the probability with which a diff is chosen to be pushed for online availability proportional to its age.

To reduce the overhead of detecting updates, in our implementation, only the champions look for updates to files in their owners' working sets. When a champion learns of an update, it downloads the appropriate diffs and gossips the presence of the update to its peer devices in the device group.

## 4. Evaluation

We now explore the behavior of our replication algorithm as implemented in Wayfinder. Our Wayfinder prototype is written in Java and supports an NFS facade derived from JNFSD, an open source NFS server [10]. The NFS facade allows users to access Wayfinder through any standard NFS client. Our prototype is sufficiently complete to support the transparent execution of many common applications on Linux such as emacs and cvs.

Our study centers around two benchmarks, a micro-benchmark that injects bursts of file creations and file updates, and a macro-benchmark derived from a read/write trace of a wiki. The former studies the efficiency of our randomized replication algorithm while the latter studies our algorithm's behavior under a real, albeit condensed, workload.

Our study only focuses on Wayfinder's maintenance of online and ownership availability. While the full replication algorithm is always executed, we do not measure offline availability as we do not have sufficient data on disconnected access patterns. Further, our contribution is a combined availability model that includes hoarding as a component, rather than any novel hoarding algorithm for disconnected operation. Other efforts such as [12] have explored this latter problem and our replication algorithm can

directly leverage their algorithms if desired. In this paper, as already described, our algorithm maintains an LRU working set for simplicity.

Our benchmarks are run in two different environments designed to represent two different styles of federated communities.

**Corporate (CO).** This community represents what we might see in a standard corporate or university environment, where each employee is assigned a desktop. The goal of this community is to provide high online availability for all content. This community is parameterized using data from Bolosky et al.'s [4] study of a large corporate environment. Specifically, we set each node in a cluster of 12 nodes to have 80% availability with an average uptime of 272 minutes. (The average uptime was shortened by a factor of 11 from that reported by Bolosky et al. because our benchmark is a compressed trace designed to reduce experimental time.) Each node belongs to a distinct device set. All files in the system are tagged as owned by a single node, which represents the infrastructural resource devoted to maintaining availability for all shared content. In essence, this node is the champion for the entire community. We could have also chosen to have all nodes own all files. However, this would have put the algorithm in the best possible case of fully concurrent downloads by all nodes.

**Heterogeneous Workgroup (HW).** This community represents an extended, more mobile environment, where each user may have a desktop at work, a laptop for mobile computing, and a home machine. This community is quite representative of our research lab and is parameterize using measurements obtained from our lab. Specifically, the community contains 3 device sets belonging to 3 distinct users. Each device set has a work desktop with 80% availability, a home desktop with 50% availability, and a laptop with 32% availability. The community also has 3 additional server-like nodes that do not belong to any user but rather provide resources for the entire community. Each of these server-like nodes has 95% availability. The average node uptimes were set to 387 min., 225 min., 60 min., and 55 min. for nodes with 95%, 80%, 50%, and 32% availability, respectively. Files were partitioned into 3 non-overlapping ownership sets, one per distinct user.

**Experimental Platform.** All results reported below were obtained on a cluster of PCs, where each node was equipped with a 64-bit 2.8 MHz Intel Xeon processor, 1 GB of memory, and a 10K RPM 70 GB SCSI disk. All nodes ran the Linux 2.6.12 kernel and Sun's Java 1.4.1\_2 JVM. The cluster is interconnected by a 100Mb/s Ethernet switch. The Wayfinder prototype was configured with a 1 second PlanetP gossiping interval and a 3 seconds inter-push time for replication for online availability.

No. background files/diffs ( $N_i$ )	Time to achieve $TOA_C$ in CO (sec)			Time to achieve $TOA_C$ in HW (sec)		
	Burst Size ( $N_b$ )			Burst Size ( $N_b$ )		
	10	20	30	10	20	30
100	153	296	433	74	155	227
200	155	277	416	81	142	206
400	177	277	416	84	141	212

**Table 1.** Total time required for Wayfinder to achieve the target online availability  $TOA_C$  for a group of newly created files.

#### 4.1. Online Availability for Creation and Write Bursts

We begin by studying the time required for Wayfinder to achieve a stable configuration, i.e., one in which all files and diffs have achieved  $TOA_C$ , after a single node injects a burst of new data into the system. Specifically, we use a benchmark that first creates  $N_i$  files and  $N_i$  diffs and allow the system to reach a stable configuration. The creation of these initial files and diffs is designed to evaluate whether the time to reach a stable configuration depends on the number of existing files and diffs as well as the size of a creation or write burst. Then, the benchmark creates an additional  $N_b$  files and measure the time required for the system to achieve  $TOA_C$  for these files. Finally, the benchmark creates an additional  $N_b$  diffs and measure the time required for the system to achieve  $TOA_C$  for these diffs. All created files and diffs are small, on the order of several hundred bytes, and so the actual time to create a replica is negligible.

Table 1 shows the results for the file creation bursts when the above benchmark is run on a cluster of 12 nodes with  $TOA_C = 0.999$ . In both the CO and HW environments, the bursts were performed on a champion device. During the experiment, node arrival to and departure from the online system were driven by exponential arrival processes based on the means given above. The times presented were averaged over four runs of the benchmark.

Observe that the time required for Wayfinder to achieve the online availability target for all newly created files is roughly linear to  $N_b$  but independent of  $N_i$ . This independence from the number of existing files and diffs arises from the “pool of candidates” implementation described in Section 3 and is quite important because Wayfinder systems may contain very large numbers of files (e.g., millions). The linear dependence on the burst size shows that Wayfinder’s randomized selection process is (almost) as efficient as the use of a predetermined deterministic schedule.

Observe also that Wayfinder reaches stability faster in

HW than in CO. This is because the ownership of the created files in HW was partitioned among the three device sets—they were created in different directories, each of which was owned by a different user with tag inheritance turned on—and so the new files were pushed by all three champions as opposed to the single champion in CO.

We do not show the results for the diff bursts because they are not significantly different.

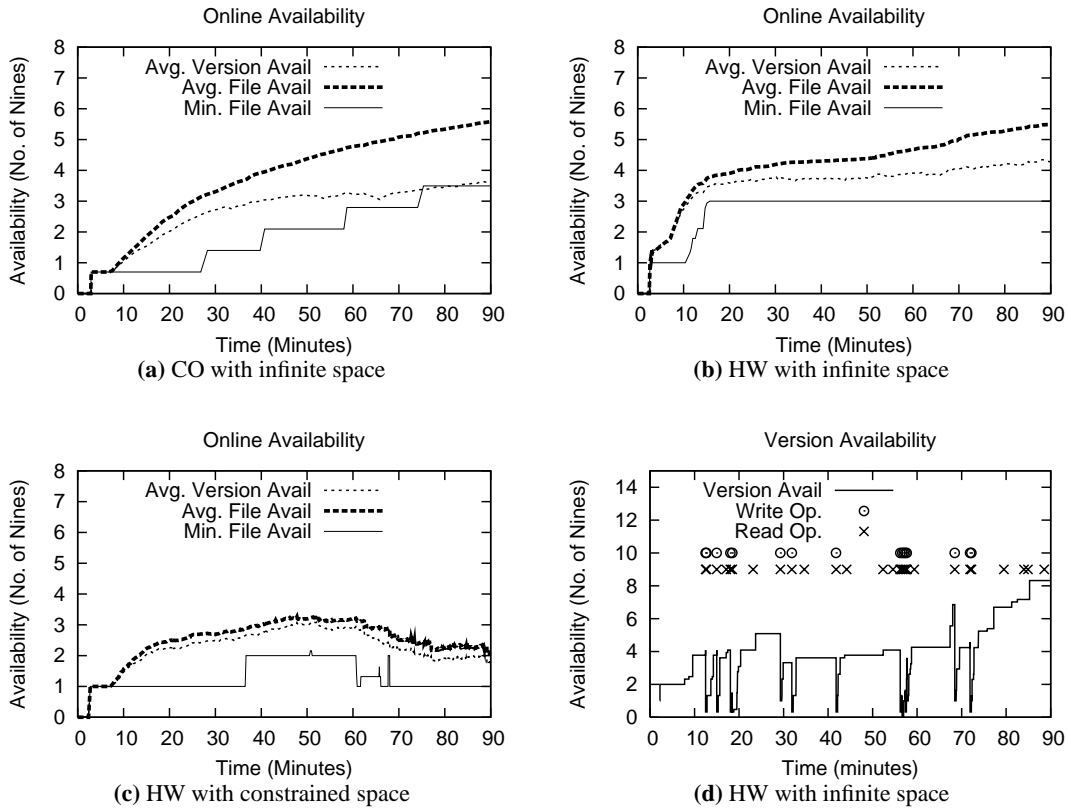
#### 4.2. Online Availability for a Collaborative Workload

We now explore Wayfinder’s behavior using a trace of a collaborative workload. In particular, we derive a benchmark based on data collected from the PlanetLab documentation wiki web, which allows the PlanetLab community to collaboratively self-document the system.<sup>5</sup> We were able to obtain information about file creations (one file per web page) and updates, when they occurred, and who performed them because the wiki software preserves the history of the web’s evolution much like a version control system. When we collected our trace, the wiki web contained 457 distinct files with 2800 distinct versions. The updates spanned a period of several years and were performed by 153 distinct users.

We created a benchmark from the above data as follows. First, we chose a period of 134 days during which the wiki was particularly active, containing 532 updates spanning 128 files, and compressed the update stream so that it could be replay in approximately 90 minutes. Second, to actually run the benchmark, we needed to map the updates in the trace to authoring nodes in the federated system. Since we are not evaluating offline availability, all accesses needed to be mapped to nodes that are online at the time of the access. We ensured this by first mapping accesses of each user to a node in the community—typically, this meant that multiple users were assigned to each node—and then constructing the online and offline behavior of each node around the accesses assigned to that node. Each node’s behavior corresponded to one possible sequence of arrival to and departure from the online system according to an exponential arrival process with the appropriate mean. Third, we added in read traffic by injecting reads to random files in the web during nodes’ uptimes according to an exponential arrival process with a mean inter-arrival time of 5 seconds. Finally, to avoid name conflicts, which are difficult to handle in an automated trace-driven experiment, we set the benchmark to pre-create all files at the beginning of the experiment with an initial size of zero.

We then ran the above benchmark in several different environments: CO with infinite storage, HW with infinite

<sup>5</sup>PlanetLab is an open platform for developing, deploying, and accessing planetary-scale services (<http://www.planet-lab.org>).



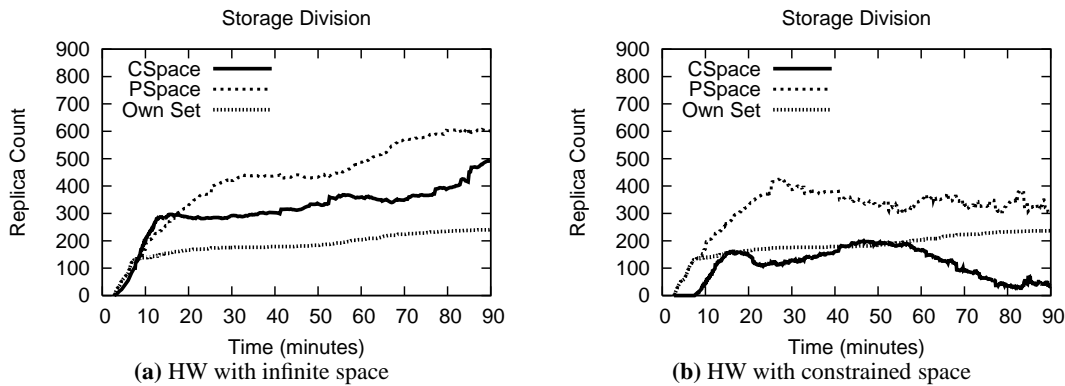
**Figure 2.** Average file online availability, average version availability (for the latest version of each file at each instant in time), and minimum availability plotted as functions of experiment time for (a) CO with infinite space, (b) HW with infinite space, and (c) HW with constrained space. (d) Online version availability for a representative file from HW with infinite space.

storage, and HW with constrained storage. In the last environment, each non-champion node was given sufficient local storage to hold about 15% of the shared data set, while each champion was given sufficient local storage to hold the entire data set. (The shared data set grows to approximately 11 MB by the end of the benchmark.)  $TOA_C$  was again set to 0.999.

Figures 2 and 3 show some of the results. We make several observations based on these results. First, *when there is sufficient storage space, the system consistently achieves and maintains the target online availability for all files and diffs.* Figures 2(a-b) show that Wayfinder takes some time to achieve the target online availability for all files, particularly when there is only one champion as in the CO environment, because of the creation burst at the beginning of the benchmark. (In CO, the champion has to push all files created at the beginning of the benchmark as well as updates that occur before it can achieve  $TOA_C$  for the files.) However, once the minimum file availability reaches the target

online availability, it never drops below this target again. In fact, the average file online availability is greater than the target as the benchmark continues to run. This is because of the hoarding of files in the working set for offline availability; as files are read or written on a device, they become members in the user’s working set and are hoarded on that device. Since we have infinite storage, no file is ever discarded from any device’s local storage.

Second, *when there is insufficient storage space, the system approaches a non-cooperative configuration where devices selfishly use their local storage to achieve offline and ownership availability for their owners.* Observe that there is significant separation between the minimum and average file availability in Figure 2(c) as space becomes constrained. This is because devices are not cooperating to maintain the target online availability for shared content. Rather, online availability is just a consequence of devices hoarding content for offline and ownership availability. This is shown clearly in Figures 3(a) and (b), where Cspace is much



**Figure 3.** Division of files between Pspace and Cspace for (a) HW with infinite space and (b) HW with constrained space.

smaller in the space constrained case (b) than in the case with infinite space (a). In fact, observe that Cspace eventually drops close to zero in (b) as more content is added to the system.

Third, our unified availability model allows the system to account for the selfish behaviors of devices in attempting to achieve the best possible online availability for all shared content. Observe that even though Cspace is much more constrained in Figure 3(b) than in (a), the average file availability is still not far from the target of 3 nines. This is because replicas of files that are relatively over-replicated by selfish hoarding are preferentially evicted from Cspace. Also, because each file is owned by at least one device, typically a champion, the minimum file online availability remains at least at the availability of the champion—in this case, 1 nine.

Finally, the system efficiently maintains the target online availability for all files, despite a stream of updates that requires re-replication of the modified files. Figure 2(d) shows that although every write causes the version availability of a file to drop, the system rapidly regains the target online availability for the new version. This is particularly true in the HW environment because each modified file becomes a member of a user’s working set and so a new version is rapidly downloaded by the 3 devices in that user’s device set. This leads to an average version availability that is only slightly lower than the average file availability as shown in Figures 2(a)–(c).

### 4.3. Ownership Availability for a Collaborative Workload

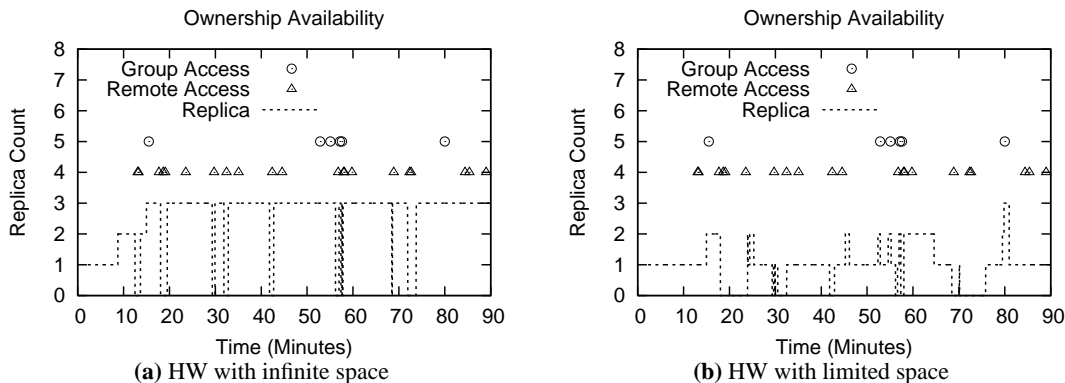
Figure 4(a) plots the ownership availability of a representative file against experiment time as the wiki benchmark

is executed in the HW with infinite storage environment. Ownership availability is expressed as the number of up-to-date file replicas in the owning user’s device set plus the number of out-of-date replicas in the device set which can be brought up-to-date using diff information also hoarded in the device set. This count can drop to zero if the file is updated by a device outside of the owning user’s device set. However, observe that even in this case, at least one device in the device set notices the update in a short time so that there is almost always at least one up-to-date replica in the device set. Figure 4(b) plots the ownership availability when the benchmark is run in the HW environment with constrained space. In this case, we see that only one replica of the representative file is kept unless that file is accessed by the user and so becomes part of the working set, resulting in replicas being created on all online devices. These new replicas are then rapidly discarded from the non-champion devices as other files are accessed.

### 4.4. CPU and Network Utilization

Our replication algorithm relies on a number of periodic processes—e.g., pushing under-replicated files, looking for and downloading updates to files in the user’s working and ownership sets, etc.—that must run continuously in an implementation. Thus, we measured the CPU utilization during the execution of the wiki benchmark to evaluate the CPU overhead imposed by these periodic processes. For all environments, including the one with constrained storage space, the average CPU utilization was less than 3% on a champion device and less than 2% on a non-champion device.

We also measured the bandwidth used while running the wiki benchmark. The entire data set, i.e., a single copy of



**Figure 4.** Ownership availability of a representative file vs. experiment time for the HW environment with (a) infinite space and (b) limited space .

the final version of each file and all the created diffs, is approximately 11 MB in size. For the HW community, each champion node communicates approximately 22 KB/s, including both sends and receives, over the 90 minute benchmark execution. Non-champion nodes each communicates approximately 10 KB/s. In the space constrained scenario, each champion’s communication rises to approximately 25 KB/s. For the CO community, the champion node communicates approximately 20 KB/s while each non-champion node communicates approximately 10 KB/s.

In general, transfer of content because of the general access pattern and because of replication accounted for about 20% of the bandwidth usage. (This amount increases to 34% in the space constrained scenario, reflecting the file churn because of eviction.) Constructing Wayfinder’s global namespace for accesses as well as for the champions to walk the namespace (see Section 3.2) accounted for another 64% of the bandwidth usage. PlanetP’s gossiping of the membership directory and the global index together with Wayfinder’s gossiping of the working sets accounted for the remaining 16%.

## 5. Related Work

Total Recall [2] is perhaps the work that is the most related to ours. Total Recall explores the problem of maintaining online availability for P2P content sharing systems. Similar to our work here and in [5], Total Recall monitors the availability of nodes and placement of file replicas, and dynamically adjusts the number of replicas to maintain an online availability target. Total Recall only considers online availability, however, whereas a key contribution of our work is a framework that unifies replication for online availability, offline availability, and ownership availability.

Farsite [1] is a decentralized file system similar to Wayfinder in spirit. Like Total Recall, however, Farsite only considers online availability. Further, Farsite targets environments that are much more homogeneous in availability than those we target. This leads Farsite to adopt a replication algorithm that uses a fixed number of replicas for all files in contrast to the dynamic determination of the number of replicas needed for a target availability used in our algorithm.

Segank [18] is a mobile file system that targets multi-device-per-user environments. However, Segank [18] is more concerned with maintaining a consistent view of the file system across the device set of a single user rather than across the device sets of multiple users in a federated system. This lead to considerable differences in the design of Segank compared to our replication algorithm and its implementation in Wayfinder. Also, Segank never developed an availability model such as the one we discuss here (although the authors did mention it as future work).

Glacier [9] is a distributed storage system that relies on massive redundancy to mask the effect of large-scale correlated failures. In contrast to our work, Glacier is more concerned with durability across these large-scale failures rather than availability. Also, Glacier does not rely on introspection, i.e., tracking the availability of individual nodes, to optimize replication because such introspection typically cannot accurately account for the impact of large-scale correlated failures.

The Coda File System originally introduced the concept of hoarding for disconnected operation [11]. Seer [12] attempted to improve hoarding using semantic distance and clustering for prefetching content. These works are complementary to our work in that our replication algorithm can leverage their approaches to improve its hoarding algorithm

(currently based on LRU) for improved offline availability.

Recently, there has been a flurry of work in building P2P file systems (e.g., [8, 7, 14, 13, 16]). The main difference between these efforts and our work here is that they consider only online availability and only as a peripheral issue.

## 6. Conclusions

We have presented a novel unified availability model targeted to content sharing federated systems. Such federated systems are typically comprised of a number of users collaborating to share data, each of whom may be using a distinct set of personal devices. To accommodate this environment, our model differentiates between three types of availability, online, offline, and ownership availability. This leads to a user-centric availability model that tries to make data available to users across periods of connected and disconnected operation. This model also helps users to preserve data they care about in case they become permanently disconnected from the federated system. Consequently, the model removes the need for users to explicitly manage data replicas and to hoard data external to the federated system.

We have shown how a single replication algorithm can be designed to achieve all three types of availability. This algorithm allows devices to selfishly prioritize ownership and offline availability for their owners over online availability for the community. However, the algorithm explicitly accounts for the impact of devices' selfish hoarding actions on online availability in order to minimize the space required to achieve a target online availability level for all shared content. Further, this algorithm is based on autonomous actions from devices in the community, allowing the system to tolerate the fact that devices in a federated system is not under centralized control and so may have unpredictable prolonged periods of disconnection or even leave the system permanently.

Finally, we have shown that it is practical to implement the proposed replication algorithm and that the implementation efficiently achieves our design goals.

## References

- [1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.
- [2] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total Recall: System Support for Automated Availability Management. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004.
- [3] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 1970.
- [4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2000.
- [5] F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. Autonomous Replication for High Availability in Unstructured P2P Systems. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, Oct. 2003.
- [6] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *International Symposium on High Performance Distributed Computing (HPDC)*, June 2003.
- [7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage With CFS. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [8] K. Fu, M. F. Kaashoek, and D. Mazires. Fast and Secure Distributed Read-Only File System. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2000.
- [9] A. Haebleren, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.
- [10] Java NFS Server. [http://members.aol.com/\\_ht\\_a/markmitche11/jnfsd.htm](http://members.aol.com/_ht_a/markmitche11/jnfsd.htm), Oct. 2002.
- [11] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 1991.
- [12] G. H. Kuenning and G. J. Popek. Automated Hoarding for Mobile Computers. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.
- [13] T. D. Moreton, I. A. Pratt, and T. L. Harris. Storage, Mutability and Naming in Pasta. In *International Workshop on Peer-to-Peer Computing*, May 2002.
- [14] A. Muthitacharoen, R. Morris, T. Gil, and I. B. Chen. Ivy: A Read/Write Peer-To-Peer File System. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.
- [15] C. Peery, F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. Wayfinder: Navigating and Sharing Information in a Decentralized World. In *Proceedings of Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, Aug. 2004.
- [16] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The Oceanstore Prototype. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Mar. 2003.
- [17] Sleepycat Software. Berkeley DB. <http://www.sleepycat.com/>.
- [18] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Wang. Segank: A Distributed Mobile Storage System. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Mar. 2004.