Autonomous storage management for personal devices with PodBase

Ansley Post[†] Juan Navarro[¶] Petr Kuznetsov[§] Peter Druschel[†]
[†]MPI-SWS [§] TU Berlin/Deutsche Telekom Laboratories [¶]TU Munich

Technical Report MPI-SWS-2011-001*

Abstract

People use an increasing number of personal electronic devices like notebook computers, MP3 players and smart phones in their daily lives. Making sure that data on these devices is available where needed and backed up regularly is a time-consuming and error-prone burden on users. In this paper, we describe and evaluate PodBase, a system that automates storage management on personal devices. The system takes advantage of unused storage and incidental connectivity to propagate the system state and replicate files. PodBase ensures the durability of data despite device loss or failure; at the same time, it aims to make data available on devices where it is useful.

PodBase seeks to exploit available storage and pairwise device connections with little or no user attention. Towards this goal, it relies on a declarative specification of its replication goals and uses linear optimization to compute a replication plan that considers the current distribution of files, availability of storage, and history of device connections. Results from a user study in ten real households show that, under a wide range of conditions, PodBase transparently manages the durability and availability of data on personal devices.

1 Introduction

Modern households have multiple personal electronic devices, such as digital cameras, MP3 players, gaming devices and smart phones, in addition to desktop and notebook computers. As users increasingly depend on such devices, it is important to ensure the *durability* of data in the event of loss or failure of a device, and the *availability* of the latest data on all appropriate devices.

Ensuring that data is durable is an onerous task even for a single home computer, and the situation is getting worse as the number and diversity of devices increase. Users must keep track of all devices that need to be backed up and perform the appropriate actions on a regular basis. Anecdotal evidence suggests that many users fail to ensure the durability of their data [14, 17]. Thus, users face the risk of data loss, just as they are becoming increasingly dependent on digital information.

Making sure that a given data object is available on all the devices that need it is equally burdensome. A user must regularly connect and synchronize devices to ensure, for instance, that changes to her address book are propagated to all communication devices, and that additions to her music library are present on all devices capable of playing music.

In this paper we present *PodBase*, a system that manages data on personal devices in an autonomous, decentralized, device- and operating system-independent manner. The system is transparent to the user, takes advantage of unused storage space and exploits incidental pairwise connectivity that naturally occurs among the devices, (e.g., via Wi-fi, Bluetooth or USB).

With PodBase, each device stores metadata that describes a household's devices and data. During pairwise connections, devices reconcile their metadata and exchange data. Over time, metadata and data propagate among a household's devices. PodBase progresses toward a state where, subject to available storage and in order of decreasing priority, (i) the contents of any failed device are restored to a replacement device, (ii) each object has a certain minimal number of replicas, and (iii) each object is available on devices that can potentially use it.

Results from our user study show that many households have sufficient storage and connectivity

*A short version of this paper appears in the Proceedings of the Usenix Annual Technical Conference (Usenix ATC 2011), Portland, OR, June 2011.

to permit full replication. However, there is typically not one hub device with plenty of storage to which all other devices are regularly connected with sufficient bandwidth. To ensure full and timely replication, PodBase must therefore be able to use free space on all and replicate data between any pair of devices, and possibly move data via multiple pairwise connections.

Given the vast space of possible configurations, device connection sequences and replication plans, designing an appropriate replication algorithm for PodBase is not straightforward. Simple, greedy algorithms are stable and robust but tend to get stuck in local minima. PodBase instead uses linear optimization to compute an adaptive replication plan from a declarative specification of the goal state, and a local view of the current replication, available storage and history of device connections. As a result, PodBase is highly adaptive and provably stable. Moreover, it finds sophisticated solutions in unexpected scenarios. For instance, without being programmed for this case, the system takes advantage of "sneakernets", i.e. mobile devices to transport data between home and office, thus avoiding slow broadband connections.

In the rest of the paper, Section 2 states the requirements. We discuss related work in Section 3. Section 4 presents the design of PodBase and Section 5 describes its replication algorithm. Section 6 presents our evaluation and Section 7 concludes.

2 Requirements

PodBase is intended for a household with one or more users and a set of shared personal devices. Based on the results of a feasibility study [20], we can characterize this environment as follows:

- Devices are periodically connected, such that any pair of devices can eventually communicate via a series of sequential pairwise connections.
- A device may fail or be lost at any time. However, the failure or loss of many devices during a short period of time is unlikely.
- Devices may be turned off when not in use; it cannot be assumed that any one device is always online.
- The system must be able to handle a wide range of usage patterns and device configurations, without attention from an expert system administrator.

An important aspect of the target environment is that most users don't have the expertise, interest or time to manage data and storage on their devices. They expect the system to do something reasonable automatically. Unlike a system designed for expert users (like the authors and readers of this paper), PodBase must be able to achieve its goals with little user expertise and attention.

2.1 Desired system behavior

In this section, we describe the desired system behavior intuitively and by example. A more detailed description of PodBase's properties, design and implementation follows in subsequent sections.

PodBase aims to relieve users from having to worry about the *durability* and *availability* of their data. Durability requires that the failure or loss of a device not result in the loss of user data. Availability requires that each device store the latest collection of data *relevant* to that device. For example, each communication device should store the latest version of the address book and, subject to available storage space, a shared music collection should be available on all devices capable of playing music.

As an example, Alice and Bob share a household. Alice has a notebook, an MP3 player and an external USB hard drive. Bob has a notebook and a desktop computer at his office. Their home has a wireless network connected to the Internet via a broadband connection. On workdays Alice and Bob bring their notebooks to their offices and perform their daily work, such as writing documents and using email.

At night both return home with their notebooks and use them to surf the web, play games, or listen to music. Although they have important data stored on their notebooks, they rarely back up their data.

PodBase should automatically perform the following tasks without any explicit action by Alice or Bob:

- Every night, new or modified files are replicated, in cryptographically sealed form, between Alice and Bob's notebooks via the wireless network. (This works even when they are on vacation, e.g., when the pictures Alice uploads from her camera are replicated on Bob's notebook.)
- When Bob purchases a new CD and rips it to his hard drive, a replica of the mp3 file is later moved to Alice's notebook. When Alice connects her MP3 player to charge, it also receives the new music.
- Whenever Alice or Bob edit their personal address books, the changes are automatically

propagated to their other communication devices.

- Whenever Alice's USB hard drive is connected to her laptop, additional replicas of the files and replicas on her laptop are made.
- Bob's office desktop is connected to his home via a broadband connection. Rather than transfer data using the slow connection, the system uses Bob's notebook disk to rapidly replicate data between home and work.
- When Bob's notebook is running low on disk space (after removing any replicas), the system asks Bob if it should move not recently accessed movie files to Alice's USB drive, which has plenty of space.

PodBase can recover from otherwise costly incidents. For example, imagine Alice's laptop is stolen. With PodBase, she is able to restore the data on the lost device's hard drive to her replacement notebook. When she connects over the wireless network to Bob's notebook, some files from her stolen notebook are restored on the new device. When she later connects her new notebook to the USB drive, the remaining files are restored. Thanks to the replication between home and Bob's office, they could recover all data even after a total loss of the home or office devices.

An important goal we set ourselves for PodBase is transparency: the system's background activity should not affect users' experience during normal operation. By default, the system does not remove user files, automatically propagate changes to user files or attempt to reconcile conflicting versions of concurrently modified files. Instead, PodBase maintains all versions of a file along with their modification history. Optional plug-ins can define file type-, device-, or application-specific consistency semantics.

PodBase's transparency is consistent with the principle of least surprise: by default, the installation of the system should not change a device's user-visible behavior during normal operation. Advanced behavior (e.g., automatic propagation of changes to the address book) can be enabled explicitly by enabling appropriate plug-ins.

3 Related work

PodBase is in the spirit of Weiser's Ubiquitous Computing vision [39], as it transparently manages storage on personal devices. To the best of our knowledge, no prior system provides automatic durability and availability of data on personal devices, without relying on central storage, a fast Internet connection or explicit user attention.

With Personal Server [37], users carry a personal storage device and use input/output devices found in the environment. In Omnistore [10], data is maintained on a central store, while other devices interact to cache data or relay data to the store. The Roma system [32] provides a shared, centralized metadata service that can be used to build higher-level services for synchronization, consistency and availability. Apple TimeMachine [34] and Windows Home Server [40] provide automatic backup to a dedicated storage node. Unlike PodBase, the above systems rely on a dedicated storage device, are vulnerable to the failure or loss of that device, and cannot exploit unused storage on other devices.

Availability of data on a set of devices can be provided by a distributed file system that supports disconnected operation, like Bayou [33], Ficus [19], and Coda [12]. Some systems additionally support partial replication to meet the needs of mobile devices, e.g. PRACTI [1], WinFS [42], Roam [24], Ensemblue [18], the Few File system [22] and Segank [29]. Oasis [25] is an SQL-based data management system for pervasive computing applications. PodBase differs from these systems in that it replicates data for availability *and* durability, is fully automatic, takes advantage of pairwise connections and unused storage efficiently, requires no centralized server, and is device, vendor and OS-independent.

Cimbiosys [23] is a platform for content-based partial replication. Like PodBase, Cimbiosys carefully manages the amount of information that has to be exchanged during pair-wise connections. The goal of Cimbiosys is to facilitate replication by propagating updates between peer devices. Applications or users are expected to specify filters for what each device should store. Unlike PodBase, Cimbiosys does not specify a replication policy for either availability or durability, and instead provides a replication platform for higher level applications. For replication to eventually reach the desired state, Cimbiosys assumes that all devices that replicate a given collection of objects form a tree, such that a parent stores a superset of the objects stored by its children and children regularly connect to their ancestors. PodBase, on the other hand, achieves eventual consistency as long as any two devices are repeatedly connected via a sequence of pairwise connections.

Like PodBase, Perspective [27] supports automatic partial replication among mobile devices, without relying on a centralized server. However, Perspective assumes that a view is defined for each

device, which specifies the set of files that should be present on the device. Files are then replicated along sequences of pairwise connections, where a file must be contained in the view of each device that appears on the path. PodBase, on the other hand, uses multi-step replication plans, where files can be placed on intermediate devices solely for the purpose of transporting them to another device. PodBase computes a replication plan automatically and dynamically to maximize durability and availability given the available free space on devices, without requiring the specification of per-device views.

One could try to simulate the effect of PodBase's replication policy in Perspective by specifying that each device's view include all files. Perspective would then replicate all files greedily as device connections occur, until each device either replicates all files or its space is exhausted. Unless most devices have enough space to store most files, however, this would likely lead to uneven replication levels and poor availability. Finally, PodBase was evaluated using an actual user deployment.

Device Transparency [30] is a storage model for mobile devices, where each device maintains global metadata. PodBase uses a similar capability as a building block to support transparent data replication for availability and durability. Moreover, PodBase can also support devices too small to store metadata for all objects in the system.

Synchronization tools like Unison [36] synchronize data among devices, and attempt to reconcile replicas that have diverged due to concurrent edits. Windows Live Sync [41] and Live Mesh [13] allow users to sync folders on their machines. File synchronization tool like these can be used as a plug-in for PodBase. Groove [9] provides a collaborative workspace that propagates file edits automatically among a group of users. PodBase is also concerned with durability and focuses on intermittently connected devices in the home.

Pastiche [4] and FriendStore [35] implement cooperative backup on users' machines in a peer-to-peer network. PodBase replicates data for availability and durability, within a household, on intermittently connected devices, and without relying on third-party storage.

Cloud storage services (e.g. [5, 15, 28, 31]) provide automatic backup or synchronization for mobile devices at a charge. PodBase is free, can replicate much faster because it is not limited by the upstream bandwidth of a broadband connection, exploits unused storage on existing devices, replicates among devices that are away from home (e.g. on vacation), and avoids the dependence on a single provider for data protection. Nevertheless, PodBase can take advantage of a Cloud storage service to maintain additional off-site replicas for added safelty.

Keeton et al. [11] advocate the use of operations research techniques in the design and implementation of systems. PodBase is an example of a system that uses linear optimization to adapt to its environment. Other examples include Rhizoma [43] and Sophia [38], which use logic programming to optimize cloud computing and network testbed environments, respectively. Pandora [2] uses linear optimization to optimize bulk data transfers for cost and timeliness, using a combination of Internet data transfers and the shipping of storage devices.

Since PodBase shares data among a set of intermittently connected devices, it implements a form of delay tolerant network (DTN) [6]. PodBase can be viewed as a data management application on top of a specialized DTN. The Unmanaged Internet Architecture [8] (UIA) provides zero-configuration naming and routing for personal devices. PodBase addresses the complementary problem of data management for personal devices.

A prior workshop paper [20] sketches a preliminary design of PodBase and presents results from a trace-based feasibility study. This paper contributes a revised design, a full implementation, a new replication algorithm, support for space-constrained devices, a plug-in architecture to add file type and device specific behavior, an extensive evaluation and a user study. A short version of this paper appears in [21].

4 PodBase design

We start with an overview of PodBase, its user interface, operation, plug-in architecture and security aspects.

4.1 Overview

PodBase is implemented as a user level program. It keeps track of user data at the granularity of files. PodBase is oblivious to file and device types. However, PodBase supports a plug-in architecture, by which file type and device specific data management policies can be added.

PodBase distinguishes between *active devices* and *storage devices*. Storage devices include hard drives, media players and simple mobile phones. Active devices run the PodBase software and provide a user interface. An active device contains at least one storage device; additional storage devices can be connected internally or via Bluetooth or USB. The set

of devices in a household form a PodBase *pool*. In each pool, there must be at least one active device, which runs the PodBase software.

Active devices communicate via the network and handle the exchange of data. Whenever two active devices communicate, a storage device is attached to an active device, or two storage devices are attached to the same active device, we say that these devices are connected. Data propagates during these pairwise connections.

There are three different types of data on each storage device: (1) regular *user data*, (2) PodBase *file replicas*, and (3) PodBase *metadata*. Although logically separate, all of these data are stored in the device's existing file system. The PodBase replicas and metadata are cryptographically sealed and stored under a single directory.

Metadata describes a device's most recent view of the pool's state. Included in the metadata is the set of known devices and their capacities, a logical clock for each storage device and a list of all user files that PodBase manages, along with their replication state. Capacity constrained devices may store only a subset of the system's metadata, as described in Section 4.3.2.

Some of the space on a device not occupied by user data or metadata is used to replicate files for durability and availability. User data has priority over replicas. PodBase continuously monitors its storage use and seeks to keep a proportion f_{min} of the device's capacity free at all times.

When a file is modified by an application or the user, PodBase creates a new version of the file and replicates both the old and new version independently. Plug-ins (see Section 4.4) can be used to automatically apply consistent file updates, reconcile conflicting versions or purge obsolete versions in a file type-specific manner. Users can manually retrieve copies of old versions or even deleted files.

4.2 User interaction

Next, we describe how users typically interact with PodBase. Though PodBase is designed to minimize user involvement, some interaction is required. Moreover, interested, tech-savvy users have the option to change its policies.

Device Registration. When a new device is connected for the first time, PodBase asks the user if the device should be added to the storage pool.

Device Deregistration. A storage device may permanently disappear due to loss, permanent failure or replacement. If a device has not been connected for an extended period (e.g., a month), PodBase prompts the user to connect the device or else deregister it.

Data Recovery. When a storage device fails, PodBase can recover the files it stored. The user informs PodBase that she wishes to recover the data from a particular lost device onto a replacement device or onto an existing device. The PodBase software on the recovery device then obtains copies of the appropriate files during each connection.

Externalization. By default, users and applications cannot directly access replicas stored on a device. However, users with the appropriate credentials can *externalize* replicas, that is decrypt and move the cleartext of a replica into the user file portion of the device. Alternatively, externalization can be automated using a plug-in.

Warnings. PodBase warns the user when it is unable to replicate files because there is insufficient storage space or connectivity, with specific instructions to buy an additional disk or connect certain devices.

4.3 Device interaction

When two devices are connected, they reconcile their view of the system and exchange data. First, the devices reconcile their metadata. Then, PodBase determines if any of the replicas on either device should be moved, copied or deleted. Next, we detail these steps.

4.3.1 Metadata contents

The metadata consists of the following items (their purpose will become clear in the subsequent discussion):

1. Vector Clock: A vector clock, consisting of the most recent known logical clock values for each device in the pool. A device's logical clock is incremented upon each metadata change. When a device is removed from the system, its logical clock is set to a special tombstone value. Also, the metadata includes the most recently observed vector clock of each device in the storage pool.

2. Connection History: A list of the past 100 connections that have been observed between each pair of devices, their time, duration, their average and maximum throughput, as well as the network addresses used by the devices.

3. Policies: The current policy settings. Policies can be modified by sophisticated users. Installed plug-ins (Section 4.4) can also modify the policies.

Items 1–3 are included in the metadata of all devices.

4. Set of user files: Keeps track of the user files stored on each device in the pool. The content hash[1] value, size and last modification time are recorded for each unique file. In addition, the content hashes of the last v ($v = 10$ by default) versions of each file are included (modification history).

5. Set of replicas: Keeps track of the replicas stored on each device in the pool. For each replica, its size, content hash value, and replica id are recorded.

6. Reverse map of unique files in the pool: Maps a content hash value to the set of files whose content matches the value. This mapping is used to determine the current replication level for each unique data file, considering that different files may have identical content. (PodBase de-duplicates files prior to replication.)

Each record in items 4–6 contains a version number, which corresponds to the device's logical clock at the time when the record was last modified. A small device may include only a subset of the records in items 4–6.

4.3.2 Metadata reconciliation

Metadata reconciliation is straightforward in the common case when two devices that carry the full metadata are connected. They compare their vector clocks to determine which has the more recent metadata for each device in the pool. For each such device, the more recent metadata is then merged into the reconciled metadata.

PodBase also supports devices too small to hold the full metadata. (In practice, devices smaller than about 100 MB are excluded. This is a mild limitation, since smaller storage devices are already rare at the time of this writing.) Such devices hold the full metadata for the files and replicas they store, plus some amount of partial metadata about other devices.

PodBase ensures progress and eventual consistency of metadata, even if some devices are only ever sequentially connected via small devices. To this end, PodBase places metadata on a small device that are needed to update other devices. For this purpose, it checks the last known vector clocks of all devices. PodBase selects partial metadata subject to the available space on the small device, while ensuring that (i) metadata needed by more devices are more likely to be chosen, and (ii) a roughly equal number of metadata items are included for each device that the small device may encounter. This policy seeks to maximize the spread of useful information and ensure convergence of device metadata even in extreme situations where different sets of devices are connected only via a small device.

When reconciling any device L with a small device S, PodBase checks if the metadata on S can be used to update L. For a given device d whose partial metadata appears in a small device, all metadata are included that have changed within some range of versions $i < j$ of d's metadata. This metadata can be used to update L if L's current metadata version for d is at least i and less than j. If so, PodBase merges the metadata about d from S into L's metadata.

4.3.3 Replication

Once the metadata is reconciled, PodBase determines the actions, if any, that should be performed on the data. PodBase may *copy* a replica of a file, in which case the file is stored on the target device with a new random replica id (used to distinguish between replicas), while the original replica remains on the source device. A device may also *move* a replica, in which case the replica is stored on the target device with the same replica id and then deleted from the source device. Finally, a device may *delete* a replica, to make room for another replica that it believes is more important. During replication, data is transmitted in a cryptographically sealed form, and a hash of each replica's content is attached to ensure data integrity. How PodBase determines the actions that should be performed is described in Section 5.

4.3.4 Data recovery

After a device loss or failure, data can be recovered onto a replacement device at users' request. During each connection to another device, the replacement device restores as many files as possible, guided by the reconciled metadata. The most recent available version of each file is restored. Users can speed up the recovery process by connecting appropriate devices under the guidance of PodBase. The restoration is complete when the replacement device has received, directly or indirectly, from each device in the pool a metadata update no older than the time at which the lost device went out of service, and the reconciled metadata indicates that all files were restored.

4.3.5 Replica deletion

PodBase removes replicas when the free space on a device falls below f_{min}, the minimal proportion of

[1] A second preimage resistant hash function is used

a device's storage that PodBase keeps available at all times (by default, $f_{min} = .15$). When PodBase frees space, it considers the most replicated files first. Among files with the same replication level, PodBase first deletes replicas that have the lowest (randomly assigned) replica id among the replicas of a file, then the second lowest id, and so on. This policy ensures that different devices delete replicas of the same file only when a shortage of space dictates it, but never as a result of inconsistent metadata in partioned sets of devices. (PodBase never deletes the original or any externalized replica.)

4.4 Plug-ins

Plug-ins can be used to implement policies and mechanisms that are specific to particular file types, collections of files, device types or specific devices. Following are some example plug-ins.

Consistency: PodBase replicates each version of a file independently. A plug-in can be used to automatically propagate changes or reconcile concurrent modifications under a given consistency policy. There is a large body of work on consistency, and powerful tools exist for reconciling specific file types, e.g. [7, 16, 26]. Such tools can be integrated as plug-ins in PodBase.

Unified Namespace: By default, PodBase does not automatically externalize replicas. A plug-in could export files as part of a global uniform namespace on all devices. This would allow users to browse the contents of all devices, and access files available locally (subject to user access control restrictions). In combination with a plug-in that provides consistency, this would provide a simple distributed file system.

Digital Rights Management (DRM): Media files stored on a user's devices may be protected by copyright. Usually, copyright regulations allows users to maintain copies on several of their personal devices. However, if restrictions apply, then the policies appropriate for a given media type can be implemented as a plug-in.

Archiving: A plug-in can automatically watch for large, rarely accessed user files (e.g. movies). If such files occupy space on a device that is nearing capacity, the plug-in suggests moving the collection to a different device with sufficient space. If the user approves, PodBase automatically moves the files.

Content-specific policy: A content-specific plug-in can, for example, replicate and automatically externalize mp3 files on devices capable of playing music. Moreover, the plug-in can select a subset of the music collection for placement on small devices. For instance, when replicating music on a device with limited space, a plug-in may select the most recently added music, the most frequently played music, and a random sample of other music.

As a proof of concept, we developed a plug-in that automatically externalizes replicas of mp3 files and imports them into iTunes. The plugin required around 100 lines of Java code, and two simple OS specific AppleScript scripts to interact with iTunes.

4.5 Security

PodBase uses authenticated and secure channels for all communication among devices within a pool. When a device is introduced to a PodBase pool, it receives appropriate key material to enable it to participate. Users have to present a password when they wish to interact with PodBase. Metadata and replicas are stored in cryptographically sealed form when stored on devices, in order to minimize the risk of exposing confidential data when a device is stolen. PodBase respects the file access permissions of user files – encrypted replicas can be externalized only by a user with the appropriate permissions on the file. By default, PodBase manages all of a device's contents; it can be configured to manage only specific subtrees in the namespace of a device.

The strength of PodBase's access control within a household is designed to be at least as strong as the access control between different users on the same computer. If stronger security isolation is required between devices or users, then they should not join the same pool. For instance, if a user's office computer contains confidential material that must not leave company premises, then it must not join the user's home PodBase pool.

5 Replication

We considered a number of replication algorithms. Greedy algorithms place under-replicated files on the first connected device that has space. These algorithms are simple, stable, and replicate files at the first opportunity, which is good. Unfortunately, the initial placement of a file is often sub-optimal and cannot be changed. (By definition, greedy algorithms never reconsider an earlier choice and cannot move replicas if a better placement turns out to be possible in the future.) A more sophisticated class of algorithm seeks to equalize the storage utilization of connected devices, thereby moving replicas toward devices that have space. Unfortunately, these algo-

rithms cannot take advantage of a "shuttle device" to transport data between clusters of devices, e.g., home and office.

Extending the algorithms to cover these and other important cases while avoiding degenerate performance in unexpected cases seemed daunting. Instead, we decided to pose optimal replication declaratively as a linear optimization problem. This approach minimizes design time assumptions about system configurations and usage patterns, computes optimal solutions to unexpected cases at runtime, and has provable stability properties.

Whenever two devices connect, PodBase uses an LP solver to compute a multi-step replication plan that moves the system toward the goal state. The plan considers the current system state and likely future device connections, and specifies which replicas should be deleted, copied or moved during each connection accordingly.

In general, only the first step of the replication plan is relevant, as it concerns the currently connected devices. The subsequent steps are speculative, since they depend on which future device connections actually occur. If the actual device connected next differs from the current plan, a new plan is computed. The following subsections describe the approach in more detail. Additional detail can be found in Appendix A.

5.1 Replication objective

First, we wish to guarantee that files are evenly replicated on as many devices as the available space allows. As a secondary goal, we want to maximize availability by placing copies of each file on devices where it is potentially useful. In the rest of this section we define these two properties more formally.

Let D be the set of participating devices and let F be the set of files that are managed by the system. For each device $d \in D$, let $space_d$ denote its capacity, i.e., the amount of space available at d for storage of both user and replica files. For a set of files $S \subseteq F$, $size(S)$ denotes the amount of storage required to keep a copy of S. For each device d, the set of *user files* stored in that device is denoted by $\mathsf{user\text{-}files}(d)$. In particular, for each device d, $size(\mathsf{user\text{-}files}(d)) \leq space_d$.

The goal of a storage management system is to determine and maintain, for each device d, a suitable selection of files, $\mathsf{store\text{-}files}(d) \subseteq F$, to be stored on it. Files are replicated when they are selected for storage at several different devices. Moreover, at any time, such a selection must satisfy

- $\mathsf{user\text{-}files}(d) \subseteq \mathsf{store\text{-}files}(d)$, user files are never moved or deleted from devices;
- $size(\mathsf{store\text{-}files}(d)) \leq space_d$, the files stored on a device may not exceed its capacity.

Given a particular store-files selection, we say that its *replication factor* is the number of copies k of the least replicated file in the system. More formally,

$$k = \min_{f \in F} |\{d \in D : f \in \mathsf{store\text{-}files}(d)\}| \ .$$

Moreover, we say that the replication factor is *optimal* if there is no other file selection $\mathsf{store\text{-}files}'$ with a higher replication factor.

In order to model availability, plug-ins have the option to provide an *availability selection* that assigns to each device $d \in D$ a set of files $\mathsf{like\text{-}files}(d)$ that it should *preferably* store. The *availability score*, or *av-score*, of a file selection store-files is then defined as the number of file copies that match the preference expressed by like-files, i.e.,

$$av\text{-}score = \sum_{d \in D} |\mathsf{like\text{-}files}(d) \cap \mathsf{store\text{-}files}(d)| \ .$$

In a desired *goal state*, PodBase places at each device $d \in D$, a set $\mathsf{store\text{-}files}(d)$ of replicas such that the following properties are satisfied:

Durability. The replication factor is optimal, i.e., files are maximally replicated on the existing devices.

Availability. Among the file selections with optimal replication factor, store-files has a maximal *av-score*; i.e., files are replicated in devices where they are useful.

5.2 Problem formulation

The system state, the effects of the actions, as well as the objectives are modeled as a set of linear arithmetic constraints. Care must be taken to ensure the problem formulation scales. To make the optimization problem tractable, we group files into equivalence classes called categories. All files that are stored on the same set of devices are in the same category. The system state is then encoded by specifying, for each category, the total amount of space occupied by all files in that category. This significantly reduces the number of variables in the problem formulation, which no longer depends on the number of files but on the number of devices in a pool, without any loss of accuracy.

To model the connectivity among devices, a graph is constructed with a link between each pair of devices that can potentially be connected. The link

weight specifies the estimated *cost* of data transfer among the devices. This cost is calculated based on the maximum connection speed and the probability that the devices will be connected on a given day, based on the history of past connections. In this calculation, more recent connections are weighted more heavily; individual measurements are filtered appropriately to reduce noise (see Appendix A).

Finally, we model the actions (copy, move, delete) PodBase can perform, and their effects on the system state. In general, a sequence of connections may be required in order to affect a certain state change (e.g., copy some files from A to B, and then from B to C). The formulation then encodes how the possible sequences of actions modify the number of bytes in each category.

Encoding the problem this way enables us to symbolically describe all the possible plans that PodBase could execute in order to manipulate the distribution of files. Given this formalization, the goal is to find a plan that optimizes the desired goals.

The optimization involves multiple stages, narrowing the set of candidate replication plans in each. First, the maximal replication factor k is computed based on the available space in the system. Then, we optimize for durability by computing replication plans that can achieve a k-replication for all files. Next, we optimize for cost by narrowing the set of plans to those that minimize the sum of the link weights. In the next stage, we select among the remaining plans those that maximize availability. In the final stage, we select a plan that minimizes the number of necessary replication steps. PodBase then executes the first step of the resulting replication plan, by copying, moving or deleting replicas on the currently connected devices. In practice, we do not consider plans with more than three replication steps for efficiency (few interesting plans with more steps occur in practice).

The optimization favors cost over availability, because high cost plans are highly undesirable: they may rely on links with low bandwidth or rare connectivity. Notably, this choice still permits good availability, because the cost optimization generally leaves many candidate plans from which the availability optimization can select. The reason is that all plans involving the same set of connections have the same cost, and there is a combinatorially large number of such plans, corresponding to the different placements of replicas that can occur as a result of these connections.

The cost optimization does, however, eliminate plans that create more than k replicas, even if availability calls for more. To enable additional replication for availability, PodBase changes the order of optimizations once the durability goal has been achieved. In this case, availability is optimized before cost.

In a final step, the categories are mapped back onto individual files. In cases where the solution would require a file to be split by an action, file integrity can be fed back into the optimizer as an additional constraint. The replication process is guaranteed to converge in a bounded number of steps after the set of primary data files stabilizes.

Additional parameters could be added to the optimization. For example, if device reliability data is available, this information can be considered by modeling a replica stored on a less reliable device as contributing less to the durability of the associated file than a replica stored on a more reliable device.

6 Experimental evaluation

Next, we present experimental results obtained with a prototype implementation of PodBase. We sketch the implementation, report on its overheads and verify that the system behaves as expected. Then we present measured results from a user study. Additional results are shown in Appendix B.

6.1 Implementation

PodBase is implemented as a user-level program written in Java. Most of the code (48,512 lines) is platform-independent, with the exception of a small amount (about 1000 lines) of custom code for each supported platform (Windows 2000 and higher, Mac OS X). The platform-specific code deals with mounting disks and naming files. The implementation currently requires that storage devices export a file system interface, and that active devices are able to run Java 1.5 bytecode.

Running PodBase on platforms like cell phones or game consoles is feasible, but requires additional engineering effort. We feel that our prototype strikes a reasonable trade-off between engineering effort and research goals, because it can use the majority of devices in our study.

In our deployment, active devices contact a server (2.6Ghz AMD Opteron) running CPLEX 11.2.1 (a commercial LP solver) to compute replication plans. Using the server simplifies the installation of PodBase and is not fundamental to the system. With an additional installation step, PodBase can be configured with a local solver, like the free LP solver package clp [3].

PodBase rate-limits network and disk I/O, marks I/O as non-cacheable and runs single-threaded to avoid competing with other applications for resources. To the extent possible, we tried to ensure that users did not notice that PodBase was running in the background.

6.2 Computation and storage overhead

PodBase periodically crawls file systems to monitor the state of files. Each time a new file is discovered or an existing file is modified, the file is hashed and added to the pool's metadata. We measured the amount of time the first crawl took when a new drive was added to the system. The measurements were taken on a 2.4 GHz Apple MacBook Pro, running OS X, one author's primary computing device. The internal notebook disk contained 165,105 files with a total size of 87.4GB. The initial crawl took approximately 5 hours to complete. Subsequent crawls, which only re-compute hashes for new or modified files, took on the order of 10 minutes. (Both OS X and Windows support APIs that notify applications of any folder or file modifications. Using these APIs can dramatically reduce the need for crawling, but our implementation did not use them.)

The size of the system's metadata grows proportionally with the number of files and replicas managed by a PodBase pool. In our user study, the uncompressed metadata size ranged from 270MB to 2.5GB. This amounts to only a small fraction of the capacity of most modern storage devices. For the devices in our user study, storing the full metadata was possible in all cases. However, smaller storage devices (e.g. older USB sticks or cameras) are supported via the partial metadata mechanism.

Using the LP solver to compute a replication plan takes between one and thirty seconds for most households, and 180 seconds for the largest household in the user study. When two devices connect, replication starts immediately on a speculative basis, while the optimization runs in the background. For instance, PodBase starts to replicate greedily those files that appear the most underreplicated or should reside on one of the devices for availability, according to the reconciled metadata. This replication can later be undone, in the (uncommon) case that it is inconsistent with the computed plan.

6.3 Data restoration

Next, we test PodBase's ability to successfully restore the contents of a lost device. We simulated the loss of a notebook after the replication phase was completed. PodBase successfully restored to a USB hard drive all 211206 files (75GB) that were present at the time of the last crawl of the "lost" notebook. The restoration took 5 hours 27 minutes to complete, which includes decrypting the replicas.

6.4 Partial metadata reconciliation

Next, we experiment with small devices that carry partial metadata. In our example, there are three devices: two full metadata devices, which never directly connect to each other; and a small device, which is connected to each of the other devices once per day. The small device is able to carry 100MB of metadata about other devices, and unable to carry actual data. The total metadata size is 2GB.

Initially, the large devices are completely unaware of each other. No new data is added after the experiment begins. It took ten days or 21 connections for the metadata on the two large devices to converge, which is expected based on the relative size of the metadata and the small device. This example shows that metadata converges even in extremely constrained cases. In our experience, most devices are larger and connectivity tends to be much richer in practice, leading to much faster convergence.

6.5 User study

To study how PodBase performs in a real deployment, we asked ten members of our institute to deploy the system in their households and collected trace data over a period of approximately one month. We asked the users to, as much as possible, ignore the presence of PodBase and use their devices the way they would normally use them. Three users were given an external one terabyte USB disk, because they had insufficient free space to allow their files to be replicated.

For practical reasons, the number of households and users in our study is limited and covers a relatively short period of time. Moreover, at least one member of each household was a computer science researcher. Therefore, there is a likely bias towards users who have an interest in technology. As a result, our results may not be representative of a larger and more diverse user community, or a long-term deployment. Nevertheless, we feel that the study was tremendously valuable in identifying the difficult issues, in building our confidence that the system is feasible and addresses a real need, and in understanding the system's performance in practice.

The system was deployed and actively used over

the course of two years. The data collected for the results presented in this paper were collected between July and September 2009. During this period, we collected anonymized data about file creation, modification and deletion on each device, when and where replicas were created, and which devices were connected at what times. We use these logs to generate the graphs used in the rest of this section.

First, we provide a brief overview of the households used in our deployment and the characteristics of the devices used in each.

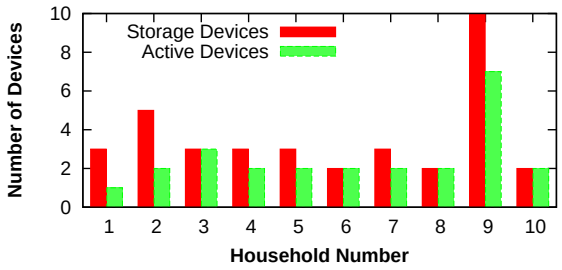


Figure 1: Number and type of devices, by household.

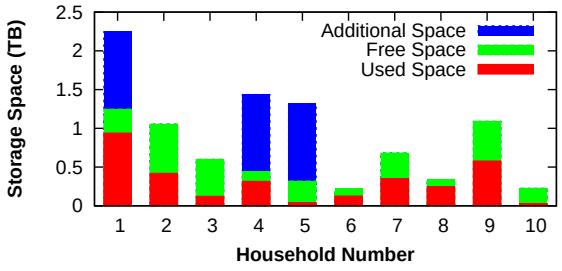


Figure 2: Storage capacity and free space on devices before PodBase begins replication. Additional space corresponds to the USB disks given to households 1, 4, 5.

Figure 1 shows the number of storage and active devices in each household. The number of active devices ranged from one to seven. Some households had no additional storage devices, while others had up to three. Households 1, 4 and 5 received an additional one terabyte USB disk, which is reflected in the data. Household 4 has a virtual device that is backed by 10GB .Mac cloud storage. PodBase uses this device like any other, considering its capacity and connection bandwidth.

Figure 2 depicts, for each household, the total size of the household's storage pool, divided into used storage and available storage at the beginning of the deployment and before PodBase was activated. The additional storage given to households 1, 4 and 5 is shown as "additional space". After this addition, seven of the households had at least half of their total storage capacity available. This does not imply that the remaining three households cannot replicate their data; whether they can depends on how much duplication there is among their existing user files.

6.5.1 Replication results

In this section, we evaluate the performance of PodBase by looking at the replication state at the beginning and the end of the (one month long) trace collection.

Let us look at the replication state of the system before the households ran PodBase. As shown in Figure 3 (left bars), many households had files that existed on only one device, leaving these files vulnerable to data loss if the device were to fail. Also, many households had a significant number of files already replicated, either as copies of the same file or different files with identical content.

The right bar in Figure 3 shows the replication state at the end of the trace collection[2]. Five households (1–3, 8–9) had most (more than 97%) of their files replicated. With the exception of household 9, which had not quite finished replicating its original files, the remaining households' unreplicated files were recently created or modified and had not yet been replicated at the end of the trace. Households 4, 5, and 7 were not able to replicate as much, as these households had only intermittent connectivity between a pair of their devices. These households each had two well-connected devices and one device that was either mostly offline or connected via a slow DSL connection. In these cases, all of the data was replicated between the well connected devices, but the data on the poorly connected device was not replicated fully.

Households 6 and 10 did not have enough space to replicate the remaining 19% and 10% of their files, respectively. In order to improve upon these results, the users would have had to purchase inexpensive additional storage. As a sanity check we had users from households 4 and 10 bring in their notebooks in order to confirm the diagnosis described above. Simply having household 4 bring its notebook into the office, where there was good connectivity between devices, allowed its data to be fully replicated. For household 10, we attached a one terabyte external drive to an active device that had data to be replicated. After doing this, less than 0.5% percent of

[2]The result for household 7 was obtained by re-playing the trace, because a bug was discovered during the user study that had influenced the final state of this household

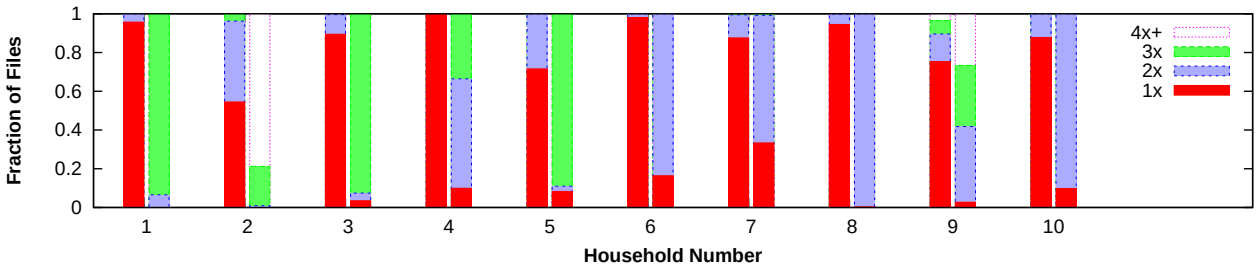


Figure 3: The initial (left bar) and final (right bar) replication status of each household.

files remained to be replicated.

Several households (1–5, 7 and 9) were able to achieve a replication factor greater than two for some of their files, enabling these files to survive multiple device failures. In Household 2, 80% of the user files were replicated 4 times or more.

6.5.2 Availability results

A secondary goal of PodBase is to place replicas on devices where they are likely to be useful. Specifically, our mp3 plug-in causes music files to be preferentially placed on devices that are capable of playing music.

In analyzing the trace, we found that one household had no mp3s and three households had already replicated all of their music files on the relevant devices. Thus, PodBase did not have an opportunity to improve availability. However, it did provide a significant gain in availability for several other households. Household 3 had its entire music library of 415 music files made available on all three of its devices. Households 7 and 8 had 851 and 1318 music files made available by PodBase, respectively. Household 9 had 1500 music files from a music library, which was otherwise loosely synchronized between its devices, made available on two additional devices. An additional two households originally had a significant number of mp3 files on their laptops but not on their desktops. PodBase replicated these files onto the desktops, and the mp3 plug-in described in section 4.4 had externalized the music files. This happened during an earlier run of PodBase, therefore it did not show up in our trace. The users gained access to 426 songs and 2611 songs, respectively, on their desktop computers. (The songs were previously stored only on their notebooks.)

As described in Section 5.2, the replication first optimizes for durability, then cost (time to complete), and finally availability. A concern might be that this choice limits the availability the system can provide. We looked at the impact of this optimization process on household 9, for which the final replication plan had not achieved full replication for availability. In this household the final replication plan yields 95% of the optimal availability. The remaining 5% were not achieved because the replication had not yet finished at the end of the trace, and not because of a limitation in the algorithm.

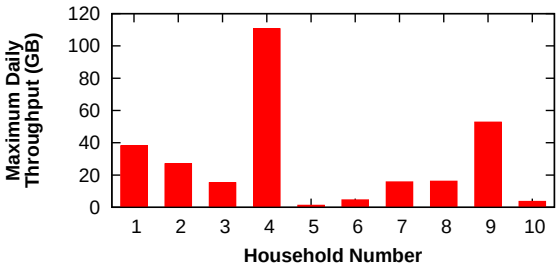


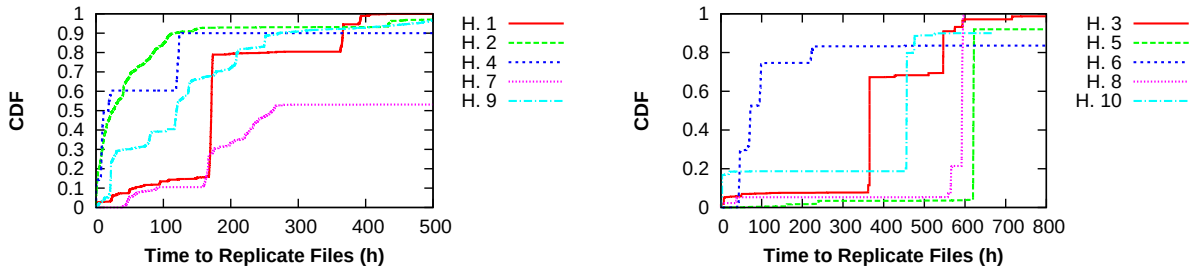
Figure 4: Peak daily throughput for each household

6.5.3 Replication latency and throughput

We next look at the maximal replication throughput in each of the households. Since all households had many files to replicate at the beginning of the trace collection, the rate at which data was replicated early in the trace is a lower bound for the total replication throughput of a pool. This value in turn provides a lower bound for the rate of new or modified data that a household could generate, such that PodBase would still be able to keep up with replicating.

Figure 4 shows that the peak throughput ranges from 1.4 to 110 GB per day. This result shows that PodBase can keep up with a high to very high rate of data generation, using only existing pair-wise connectivity.

We now examine the replication latency, i.e., the elapsed time until a new or modified file becomes replicated. If a file is not yet replicated at the end of the trace, we include it in the CDF as having an infinite latency. We first examine those households



(a) Replication latency for households 1,2,4,7,9

(b) Replication latency for households 3,5,6,8,10.

Figure 5: Replication latency for all households

with relatively short latencies. Figure 5(a) shows a CDF of how long it took to replicate a file. For households 2 and 4, over 50% of files were replicated within approximately one day. Households 1 and 7 took longer because there were extended periods with no connectivity. Household 9 replicated gradually over the course of the trace, as connectivity allowed. Second, we show the latency of the households that took significantly longer to replicate their files in Figure 5(b). In these households, device connectivity is the dominant factor in the replication latency. When there is connectivity, there are sharps jumps as files get replicated, followed by periods of disconnection, where no replication happens.

We note that our measured replication latencies are conservative, because in most households, PodBase was busy replicating the user files found initially on the devices during a large part of the trace collection. In steady state, PodBase would have to replicate only newly created or modified files, reducing the latencies considerably. Nevertheless, PodBase was able to replicate data in a timely fashion, subject to available storage and device connectivity.

6.5.4 Rate of new or modified data

Next, we look at the rate of new or modified data that is being generated. Each of the households in the user study had on average 528,187 files taking up 332GB. After the initial crawl, an average of 21GB per day was generated by the addition of new and modifications of existing files. These numbers are skewed by a household that stored the disk image of an active virtual machine in the file system; without this household, the value was 381MB per day. (Of course, PodBase could be optimized to handle this case more efficiently.)

Our normal households generate new or modified data at a minimal/average/maximal rate of 4.5/36.1/316 Kb/s, while the "heavy" household generates 2.3 Mb/s. Let us consider how well a backup system based on cloud storage alone would perform in our households. At an assumed broadband upload bandwidth of 1 Mb/s, transferring the initial data to the cloud while keeping up with updates would require between 3.7 and 121.6 (median 31.82) days for the normal households. For the heavy household, cloud storage would be infeasible, because the rate of new data exceeds the network bandwidth.

These results show that for timely replication of data, PodBase's use of peer connections and local storage devices is important. For the normal households, a broadband connection would suffice to replicate new data, but the heavy household would require a faster Internet connection. Even for the normal households, relying solely on a broadband connection to the cloud would require a long period of full network utilization to replicate the initial data, and increase the replication latency in steady state (and therefore the window of vulnerability for new and modified files that have not yet been replicated).

6.6 Discussion

PodBase has been developed by the first author over a period of two years, with three user deployments at different stages. Significant engineering effort was required to make sure our users (most of whom where not affiliated with the project) and their families felt comfortable running it on their personal devices. Users demanded not to have to notice the presence of the system in their daily activity or be surprised by it actions, yet expected the system to do "the right thing" without requiring their attention. Moreover, different households used their devices in very different ways, some of which we could not have imagined (see the discussion of results for different households in Section 6.5). This forced us to emphasize non-intrusiveness (not interfering with user's activities), autonomy (making reasonable choices without user's input) and adaptivity to

unexpected scenarios far more than efficiency. Apart from the quantitative results reported in this section, the most important indicator of the project's success may be the fact that ten households (which included members who had little interest and patience for our project) agreed to use the system for the duration of the study and beyond.

7 Conclusion

PodBase transparently manages the data stored on personal devices for durability and availability. The system takes advantage of existing free storage space and incidental connectivity among devices. Thus, it reduces the need for dedicated backup storage or an external storage provider and avoids the bottleneck of a home broadband uplink. PodBase relies on optimization techniques to achieve highly adaptive replication. The system is fully decentralized and does not depend on the health of any one device. Experimental results from a user deployment in ten real households indicate that the system is effective in replicating data without any user attention, and in many cases without requiring additional storage.

References

[1] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. NSDI '06*, 2006.

[2] B. Cho and I. Gupta. New algorithms for planning bulk transfer via internet and shipping networks. In *Proc. ICDCS '10*, ICDCS '10, 2010.

[3] COIN-OR Linear Programming Solver. `http://projects.coin-or.org/Clp`.

[4] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. *SIGOPS Operating Systems Review*, 36(SI):285–298, 2002.

[5] Dropbox. `https://www.dropbox.com/`.

[6] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proc. SIGCOMM '03*, Aug 2003.

[7] A. D. Fekete and K. Ramamritham. Replication. volume 5959 of *Lecture Notes in Computer Science*, in chapter Consistency Models for Replicated Data, pages 1–17. Springer Berlin / Heidelberg, 2010.

[8] B. Ford, J. Strauss, C. Lesniewski-Laas, S. Rhea, F. Kaashoek, and R. Morris. Persistent personal names for globally connected mobile devices. In *Proc. OSDI '06*, Nov 2006.

[9] Groove. `http://office.microsoft.com/groove`.

[10] A. Karypidis and S. Lalis. Omnistore: A system for ubiquitous personal storage management. In *Proc. PerCom '06*, Mar 2006.

[11] K. Keeton, T. Kelly, A. Merchant, C. Santos, J. Wiener, X. Zhu, and D. Beyer. Don't settle for less than the best: Use optimization to make decisions. In *Proc. HotOS '07*, May 2007.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[13] Live mesh. `http://www.mesh.com`.

[14] Many PC Users don't backup valuable data. `http://money.cnn.com/2006/06/07/technology/data_loss/index.htm`.

[15] Mozy. `https://www.mozy.com/`.

[16] J. P. Munson and P. Dewan. A flexible object merging framework. In *Proc. CSCW '94*, pages 231–242, New York, NY, USA, 1994. ACM.

[17] PC Pitstop Research. `http://pcpitstop.com/research/storagesurvey.asp`.

[18] D. Peek and J. Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proc. OSDI '06*, Nov 2006.

[19] G. J. Popek, R. G. Guy, T. W. Page, Jr., and J. S. Heidemann. Replication in Ficus distributed file systems. In *Proc. WMRD*, pages 20–25, Nov 1990.

[20] A. Post, P. Kuznetsov, and P. Druschel. PodBase: Transparent storage management for personal devices. In *Proc. IPTPS '08*, Feb 2008.

[21] A. Post, J. Navarro, P. Kuznetsov, and P. Druschel. Automomous storage management for personal devices with Podbase. In *Proc. Usenix ATC 2011*, June 2011.

[22] N. Preguiça, C. Baquero, J. L. Martins, M. Shapiro, P. S. Almeida, H. Domingos, V. Fonte, and S. Duarte. Few: File management for portable devices. In *Proc. IWSSPS 2005*, March 2005.

[23] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: a platform for content-based partial replication. In *Proc. NSDI'09*, 2009.

[24] D. Ratner, P. Reiher, and G. J. Popeky. Roam: A scalable replication system for mobility. *Mobile Networks and Applications*, 9(5):537–544, 2004.

[25] M. Rodrig and A. LaMarca. Oasis: an architecture for simplified data management and disconnected operation. *Personal and Ubiquitous Computing*, 9(2):108–121, 2005.

[26] Y. Saito and M. Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, Mar. 2005.

[27] B. Salmon, S. W. Schlosser, L. B. Mummert, and G. R. Ganger. Putting home storage management into perspective. Technical Report CMU-PDL-06-110, Parallel Data Laboratory, Carnegie Mellon University, 2006.

[28] Skydrive. `http://skydrive.live.com/`.

[29] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: A distributed mobile storage system. In *Proc. FAST '04*, March 2004.

[30] J. Strauss, C. Lesniewski-Laas, J. M. Paluska, B. Ford, R. Morris, and F. Kaashoek. Device transparency: a new model for mobile storage. *SIGOPS Oper. Syst. Rev.*, 44(1):5–9, 2010.

[31] SugarSync. `http://www.sugarsync.com/`.

[32] E. Swierk, E. Kıcıman, N. C. Williams, T. Fukushima, H. Yoshida, V. Laviano, and M. Baker. The Roma Personal Metadata Service. In *Proc. WMCSA 2000*.

[33] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP'95*, December 1995.

[34] Time Machine. `http://www.apple.com/macosx/features/timemachine.html`.

[35] D. N. Tran, F. Chiang, and J. Li. Friendstore: Cooperative online backup using trusted nodes. In *SocialNet'08: First International Workshop on Social Network Systems*, Glasgow, Scotland, 2008.

[36] Unison File Synchronization. `http://www.cis.upenn.edu/~bcpierce/unison/`.

[37] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light. The Personal Server: Changing the way we think about ubiquitous computing. In *Proc. of Ubicomp'02*, 2002.

[38] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: an information plane for networked systems. In *Proc. HotNets-II*, 2003.

[39] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, September 1991.

[40] Windows home server. `http://www.microsoft.com/windows/products/winfamily/windowshomeserver/default.mspx`.

[41] Windows live sync. `http://sync.live.com/`.

[42] Winfs team blog. `http://blogs.msdn.com/winfs/`.

[43] Q. Yin, J. Cappos, A. Baumann, and T. Roscoe. Dependable self-hosting distributed systems using constraints. In *Proc. HotDep '08*, Dec 2008.

A Adaptive Replication Formulation

In this appendix, we discuss the adaptive replication LP formulation and its inputs in more detail.

A.1 Definitions

We begin by defining some terminology. A *device* is a container for data. A device has a finite capacity, which can be consumed by storing files or replicas. A device can have *connections* to other devices; these connections have a speed associated with them. A *configuration* is the current set of devices and connections.

A *state* is the current configuration of data stored on devices. A *step* describes how a set of *actions* moves one state into the next state. The set of possible actions is restricted by *constraints* on valid states. A *plan* is a set of actions to be taken over a series of steps.

A.2 Constraints

A constraint describes the set of valid states that can exist. An example of a constraint is that no device can store more data than it has capacity for. Through the use of constraints the state space of possible plans is limited to those of interest. Next, we briefly describe the constraints used in the PodBase LP formulation.

- The amount of data stored on a device cannot exceed its capacity.
- A non-replica file can not be deleted or moved.
- A copy or a move action can only occur between connected devices.
- The amount of data copied or moved must not exceed the original amount of data.

This simple set of constraints is enough to limit the actions of the system to those that are feasible in practice.

A.3 Actions

We define two simple actions (copy and delete), which can be combined to create a third logical action (move). The basic actions are to make a copy of a file on another device, or to remove a copy of a currently stored file from a device. Moving a file from one device to another can be accomplished by first copying to the destination and then removing the file from the source.

More precisely, copy takes in a state, and then adds a device to the set of devices that are storing the file. Similarly, remove takes in the state and removes a device from the list of devices storing the file. Associated with each action is a cost. This cost describes roughly the amount of time it would take to execute the action. The cost of a remove is defined as zero, as it can happen more or less instantaneously. The cost of copying between two devices is the amount of time that it would take to move the data over the connection between the devices. This cost is based on a model described later in this section.

A.4 Goals

Up to this point we have constructed the building blocks of the LP. We have defined the valid system states, and actions that evolve the system state over time. A goal describes the system state that the system seeks to eventually reach. PodBase defines several different goals, which can be combined to compute replication plans that have a certain set of properties. These goals are the following:

- Maximize durability: maximize the number of copies of all data (see Section 5.1)
- Maximize availability: maximize the number of files that meet the availability goals (see Section 5.1)
- Minimize cost (time): minimize the cost of the sum of all actions in the plan.
- Simplicity (minimize the number of steps): maximize the number of actions that occur in the first n steps of the plan. PodBase optimizes for simplicity once other goals have been optimized for; it is used to ensure the actions in the plan occur as early as possible. PodBase optimizes for simplicity by iterating the optimization for decreasing values of n to find the minimal n for which the solution remains feasible.

An individual LP formulation is a set of of actions, constraints, and a fixed goal. However, in PodBase we often want to optimize multiple goals. To do this we iteratively solve multiple LP problems, and fix some components of the next problem with values derived from the previous problem. While these building blocks can be arranged in any order, in our actual PodBase deployment we fixed them as described in the section 5.2.

A.5 Categories

A simple approach would be to use the LP formulation with all of the above concepts on a per-file level, with one variable for each file. However, given the large number of files in a typical PodBase deployment, this is not feasible given the current state of linear optimizers. To make the problem tractable, we keep track of categories of files, rather than individual files. A category is the set of all files stored on the same set of devices in the same form. An example is all files stored as user files on A and as replicas on B. This set is associated with a number, which is the total size (in bytes) of all files in the set. An action can move data between these categories by copying or deleting. The use of categories limits the number of variables to be included in the LP to the number of categories, which is much smaller than the number of files in a PodBase deployment.

A.6 Defining action costs

An important aspect of the formulation is the cost of a copy action. (Recall that a delete is assumed to have zero cost. A move consists of a copy followed by a delete, and therefore costs the same as a copy.) The cost of a copy between a given pair of devices is based on the expectation that the connection will occur in the near future, as well as the expected bandwidth and duration of the connection should it occur. The cost is assigned based on the history of observed connections between each pair of devices.

More specifically, the cost is based on the throughput, frequency and duration of past connections observed between any pair of devices. We combine these factors, filter them to make older samples worth less than newer samples, and assign a weight to each pair (link). For each $< source, destination >$ IP pair, we first define a value $throughput_{peak}$, which is the maximal measured throughput between that pair of devices. Then, for each day during which a connection was observed, the daily connectivity $connectivity_i$ on day i is calculated as follows:

$$connectivity_i = \frac{secondsconnected_i}{86000} \quad (1)$$

This value is the proportion of time during which a pair of devices was connected. The value is then multiplied by $throughput_{peak}$ to arrive at the expected average throughput between those two devices:

$$throughput_i = throughput_{peak} * connectivity_i \quad (2)$$

Now a discount function $w(age)$ is applied, which weights samples based on how long ago they occurred in the past. For a sample of age age (in days) the discount function $w(age)$ is defined as:

$$w(age) = 1/e^{-log(0.1)/30*age} \quad (3)$$

The function weights the more recent samples exponentially higher than older samples. If a sample is older then 30 days, it has a weight of zero. The parameters in this weighting function where determined empirically as reasonable choices in our deployment.

The final expected throughput for a pair of devices is the sum of all samples divided by the maximum age:

$$\frac{\sum_{i=0}^{30} w(i) * throughput_i}{\sum_{i=0}^{30} w(i)} \quad (4)$$

The result of this calculation is the rate at which the two devices are expected to be able to transfer data between each other.

The calculated rates are then used to label the edges in a graph connecting all devices that have ever been connected to each other. In this graph edges with higher weight are cheaper to use in that data can flow along the link in less time. From this weight, the cost can be defined as the amount of time it would talk data to flow across a link. For example, a link with a weight of 100KB/s, carrying 1MB, would have a copying cost of 10s. Through the use of the link costs, a total cost for a set of actions can be computed.

To illustrate how the link weights adapt to changes in the actual connectivity, we present a brief example. Two devices A and B are connected via a single DSL link, A can upload to B with a maximal speed of 1MB/s. Assume that the devices are always connected via DSL, and have been for more than 30 days, then the predicted throughput would be 1MB/s. Let us assume a USB stick (C) can transfer data at 100MB/s when connectyed to A or B. Starting at day zero, the device is now connected to A and B ten percent of the time, respectively. Figure 6 shows how the edge weight from A to B adapts as the USB stick is connected for a number of days starting from day zero.

Note that both the direct (via the DSL connection) and the indirect path (via the USB stick) can be be used by the planner. Some fraction of the data to be transferred will always be assigned to the direct path, and as such it will remain utilized.

A.7 Optimization example

We now look at an example to illustrate the process. The goal of the planner is to find a series of steps that reach the goal state. In our example, data must be transported by intermediate devices, and the system must correctly prioritize durability replicas.

We have three devices, A, B and C. The initial distribution of files is the following: A has 100 units of data, 50 of which is of a type that should be made available, B has 0 units of data, and C has 5 units of data. A and C are capable of storing 500 units of data, and B is capable of storing 50 units of data. A and C never directly connect to each other, but B is connected to both A and C. In the beginning, no data is replicated on any of the devices.

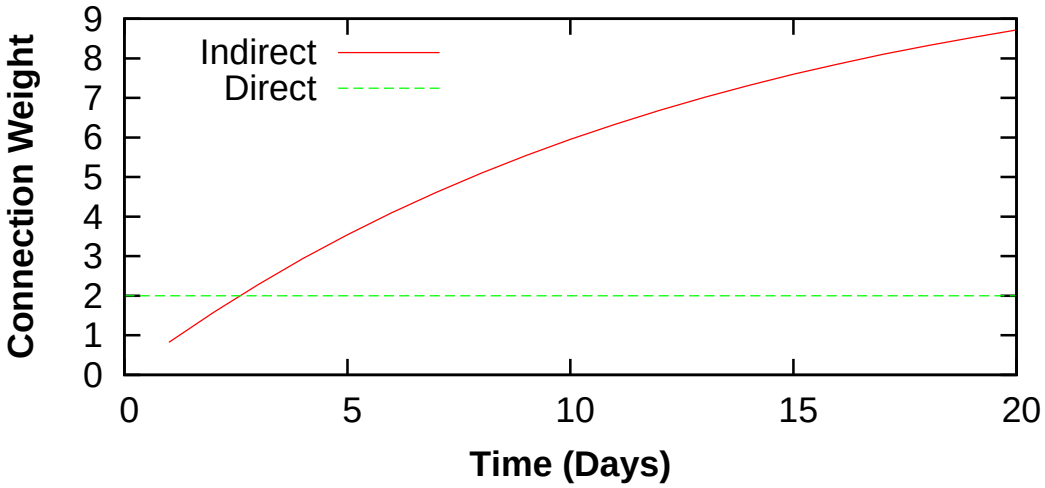


Figure 6: Evolution of link weight between two devices on the direct connection versus an indirect path via a USB stick that starts to be periodically connected to either device, starting on day zero.

```
Initial File Distribution
A=50,50
B=0
C=50

Device Capacities
A = 500
B = 50
C = 500

Connectivity
A - B = 1
B - A = 1
B - C = 1
C - B = 1

Maximum Steps
steps:  10
```

Figure 7: Summary of system state and environment, which is used to generate a linear programming problem.

Problem Instance

In this scenario, the goal state should be that C is storing a replica of all of A's data and vice-versa. B should be storing the 50 units of data for availability. A greedy algorithm will never arrive at this state. B would take 50 units of data from either A or C (whichever it was attached to first), and then the system would not be able to make any further progress.

The input to the planner is a description of the scenario, and a number of steps that are allowed in the final plan. The problem description is shown below in Figure 7. The goal state is based on the definitions in 5.1. That is, have all data maximally durable, the data that is useful for availability present on all devices, and with minimum cost.

From this specification, a linear programming problem is generated and passed to a solver. The result is used to determine the actions at each step, and must be repeated once for each step.

Generated Plan

The result of the above described process is used to generate a series of steps that PodBase must take in order to reach the goal state. We describe below the steps returned by the solver for the example scenario. The actual solution is a set values of certain variables in the solution of a linear program. We translate these results to English for the sake of clarity.

- Step 1: Copy 50 units of data that are not needed for availability from A to B
- Step 2: Copy 50 units of data that are not needed for availability from B to C
- Step 3: Delete all data from B; Copy 50 units of data from C to B
- Step 4: Copy 50 Units from B to A
- Step 5: Delete 50 all data from B; Copy 50 units of data needed for availability from A to B

```
Minimize
totalcost
Subject to
durable1 = 150.000000
available4 = 50.000000
durable3 = 100.000000
durable4 = 50.000000
durable5 = 50.000000
available1 = 50.000000
available3 = 50.000000
available2 = 50.000000
durable2 = 100.000000
......
fa01 + fa11 + .....  + fabc11 - capicitya =
0 ......
cpya0b1 + 1 cpya1b1 + 1 cpyab0c1 + 1
cpyab1c1 + 1 cpyac0b1 + 1 cpyac1b1 +
1 cpyb0a1 + 1 cpyb0c1 + 2 cpyb0ac1 +
1 cpyb1a1 + 1 cpyb1c1 + 2 cpyb1ac1 + 1
cpyba0c1 + 1
cpyba1c1 + 1 cpybc0a1 + 1 cpybc1a1 + 1
cpyab0c1 + 1 cpyab1c1 + 1 cpyc0b1 + 1
cpyc1b1 + 1 cpyca0b1 + 1 cpyca1b1 + 1
cpycb0a1 + 1 cpycb1a1 + 1 cpyac0b1 + 1
cpyac1b1 + 1
cpybc0a1 + 1 cpybc1a1 - cost1 = 0
......
cost1 + cost2 + cost3 + cost4 + cost5 +
cost6 + cost7 + cost8 + cost9 + cost10 -
totalcost = 0
```

Figure 8: Simplified snippet of LP program generated in the example problem. The objective function in this stage of the formulation is to minimize the total cost, given that the durability and availability goals are satisfied (the lines directly following "Subject to"). The next constraint ensures that the files stored on a device in a step never exceed the capacity of that device (the data being stored in each category is captured by the variables starting with f. Each letter following f is a device which the data is present on, and the following number indicate what step, and what type (0 = durability, 1 = availability)). Next, the variables encoding actions are shown, and they are related to the cost. Finally, the costs are aggregated into the objective function. A variable above is always indexed by first the devices that it is / could be stored on, then the type of file (availability=1, durability = 0), and finally the step in the output plan. Action variables (above copy) also include a target device, in the case of the copy variables the step is omitted to shorten the variable names.

- Step 6: Copy 50 units of data needed for availability from B to C.

The final result of following this plan is the optimal goal state. All data is durable, and the data should be available is present on all devices. At the intermediate steps, no replicas are made for availability before the durability goal has been reached. While this example is simple, the adaptive algorithm will work in much more complex scenarios. The units of data and edge weights are set to simple values for presentational reasons, and do not represent the values that one would expect while running PodBase.

B Replication comparison

In this section we compare the results from our adaptive replication algorithm with those from a simple greedy replication algorithm. To make the results comparable, we used the data from the user study described in 6.5. We replayed the trace from this user study against an implementation of PodBase using the greedy algorithm. The rest of this section describes the results of this experiment.

Figure 9 corresponds to Figure 3 in Section 6.5, with an additional (right) bar for each household, which shows the replication state at the end of the trace when the greedy replication algorithm is being used, with a replication factor of two. Household 1 chose not to provide their trace for privacy reasons, and thus we can not include the results.

In households 6, 8 and 10, which have only two devices, the behavior of the two algorithms is almost identical. In households 2–5 and 9, the adaptive replication algorithm performed better, showing the advantages of dynamically calculating the replication factor in the adaptive algorithms. Additionally, household 9 benefited from a multi-step replication plan used by the adaptive algorithm, which was able to make effective use of a device to transport data.

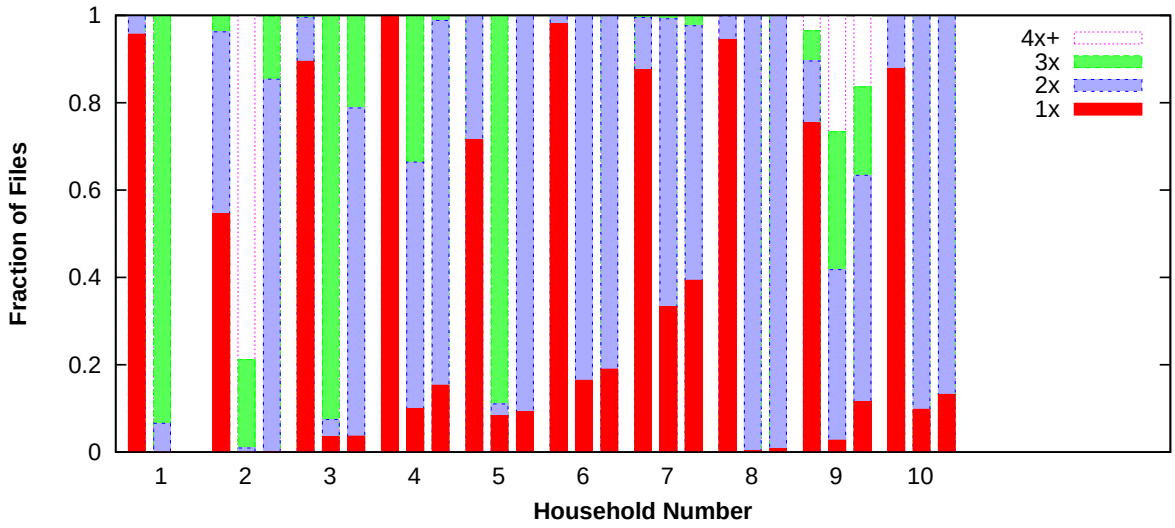


Figure 9: The initial (left bar) and final (center bar) replication status of each household. Final results for the greedy algorithm (right bar) are shown for comparison.