Pastwatch: a Distributed Version Control System

Alexander Yip, Benjie Chen and Robert Morris
MIT Computer Science and AI Laboratory
yipal@mit.edu, benjie@csail.mit.edu, rtm@csail.mit.edu

Abstract

Pastwatch is a version control system that acts like a traditional client-server system when users are connected to the network; users can see each other's changes immediately after the changes are committed. When a user is not connected, Pastwatch also allows users to read revisions from the repository, commit new revisions and share modifications directly between users, all without access to the central repository. In contrast, most existing version control systems require connectivity to a centralized server in order to read or update the repository.

Each Pastwatch user's host keeps its own writable replica of the repository, including historical revisions. Users can synchronize their local replicas with each other or with one or more servers. Synchronization must handle inconsistency between replicas because users may commit concurrent and conflicting changes to their local replicas. Pastwatch represents its repository as a "revtree" data structure which tracks the relationships among these conflicting changes, including any reconciliation. The revtree also ensures that the replicas eventually converge to identical images after sufficient synchronization.

We have implemented Pastwatch and evaluate it in a setting distributed over North America. We have been using it actively for more than a year. We show that the system is scalable beyond 190 users per project and that commit and update operations only take 2-4 seconds. Currently, five users and six different projects regularly use the system; they find that the system is easy to use and that the system's replication has masked several network and storage failures.

1 Introduction

Many software development teams rely on a version control system (VCS) to manage concurrent editing of their project's source code. Existing tools like CVS[7] and Subversion[22] use a client-server model, where a repository server stores a single master copy of the version history and the clients contact the server to read existing revisions and commit new modifications. This model works well when the users can contact the server, but as portable computers gain popularity, the client-server model becomes less attractive. Not only can network partitions and server failures block access to the repository, but two clients that cannot contact the server cannot share changes with each other even if they can communicate directly.

One approach to solving this problem is to optimistically replicate the repository on each team member's computer. This would allow users to both modify the replica when they are disconnected and to share changes with each other without any central server. The challenge in this approach is how to reconcile the *write-write conflicts* that occur when two users independently modify their replicas while disconnected. Conflicts can occur at two levels. First, the repository itself is a complex data structure that describes the revision history of a set of files; after synchronizing, the repository must contain all the concurrent modifications and the system's internal invariants must be maintained so that the VCS can still function. The second level is the source code itself which also contains interdependencies. The VCS should present the modification history as a linear sequence of changes when possible but if two writes conflict, the system should keep them separate until a user verifies that they do not break interdependencies in the source code.

Pastwatch is a VCS that optimistically replicates its repository on each team member's computer. To manage concurrent modifications, Pastwatch formats the repository history as a *revtree*. A revtree is a data structure that represents the repository as a set of immutable key-value pairs. Each revision has a unique key and the value of each pair represents one specific revision of all the source code files. Each revision also contains the key of the parent revision it was derived from. Each time a user modifies the revtree, he adds a new revision to the revtree without altering the existing entries. Revtrees are suitable for optimistic replication because two independently modified replicas can always be synchronized by taking the union of all their key-value pairs. The resulting set of pairs is guaranteed to be a valid revtree that contains all the modifications from both replicas. If two users commit changes while one or both is disconnected, and then synchronize their replicas, the resulting revtree will represent the conflicting changes as a *fork*; two revisions will share the same parent. Pastwatch presents the fork to the users who examine the concurrent changes and explicitly reconcile them.

Although Pastwatch users can synchronize their replicas with each other directly, a more efficient way to distribute updates is for users to synchronize against a single rendezvous service. In a client-server VCS, the repository server functions as the rendezvous but it must enforce single copy consistency for the repository. The consistency requirement makes it challenging to maintain a hot spare of repository for fail-over because a server and a spare may not see the same updates. Revtrees, however, support optimistic replication of the repository, so Pastwatch can easily support backup rendezvous servers with imperfect synchronization between servers. Pastwatch exploits the revtree's tolerance for inconsistency and uses a public distributed hash table that makes no guarantees about data consistency as a rendezvous service.

This paper makes three contributions. First, it describes the revtree data structure which makes divergent replicas easy to synchronize. Second, it shows how revtrees can handle many classes of failure and present them all to the users as forks. Finally, it describes Pastwatch, a distributed version control system that uses a replicated revtree to provide availability despite system failures, network failures and disconnected users.

We have implemented Pastwatch and have been using it actively for more than a year. We show that the system scales beyond 190 members per project and that commit and update operations only take 2-4 seconds. Currently, five users and six projects use the system, including this research paper and the Pastwatch software itself. The system has performed without interruption during this time despite repeated down-time of rendezvous nodes. During the same time, our CVS server experienced three days with extended down-time.

The remainder of this paper is organized as follows: Section 2 motivates Pastwatch and gives concrete requirements for its design. Section 3 discusses revtrees and section 4 describes how Pastwatch presents optimistic replication to its users. Sections 5 and 6 describe implementation details and system performance. Section 7 describes related work and Section 8 concludes.

2 Design Requirements

The task of a VCS is to store historic revisions of a project's files and to help programmers share new changes with each other. Ideally, a VCS would be able to accomplish these goals despite network disconnections, network failures and server failures. We outline the requirements of such a VCS below.

Conventional Revision Control: Any VCS should provide conventional features like checking out an initial copy of the source code files, displaying differences between file revisions and committing new revisions to the repository. In most cases, users will expect the system to have a single latest copy of the source code files, so when possible the VCS should enforce a linear history of file modifications.

At times, one or more project members may choose to *fork* to keep their modifications separate from other users. A fork is a divergence in the change history where two different revisions are derived from the same parent revision. A *branch* is a sequence of changes from the root revision to one of the current leaf revisions. After a fork, each of the two branches will maintain a separate sequential history and they will not share changes until they are explicitly reconciled. Forking is a common practice in software projects; for example, many projects use a main development branch and fork at each major release to create a maintenance branch. Some projects even use separate branches for each individual bug fix. This way, a programmer can make intermediate commits for the bug fix in her own branch without interfering with other programmers.

Disconnected Repository Operations: A VCS should support as many functions as possible even if it is disconnected from the network, for example when a user is traveling. The ability to retrieve old revisions from a local replica of the repository while disconnected is useful and easy to support. Being able to commit new revisions to the repository while disconnected is also useful, because programmers often commit changes several times a day.

For example, we will show in Section 6.1.1 that the average developer in the Gaim open-source project commits an average of 3 times per day when he is active and on the busiest day, a single user made 33 commits. Frequent commits are encouraged in software projects like PHP and libtool; their coding standards encourage programmers to make several smaller commits rather than a single large commit because it simplifies debugging.

A VCS that allows disconnected commits must handle conflicting commits. When two disconnected users commit changes, they do so without knowledge of the

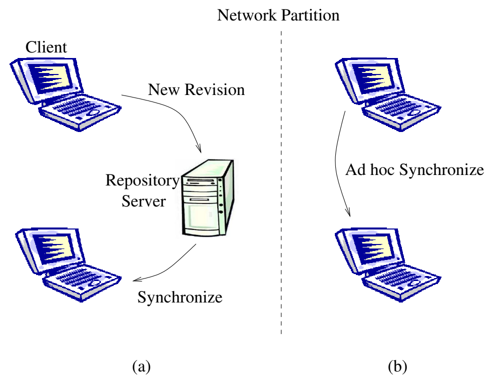

Figure 1: Flexible sharing of updates. Clients in one partition can share new revisions directly, without any servers.

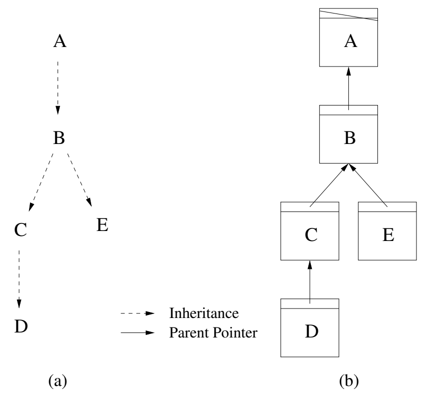

Figure 2: (a) Inheritance graph. Letters depict revisions. (b) Example revtree. Each box is a revision; it specifies one version of every file in the project. Nodes D and E are leaves and node B is a fork point.

other person's changes. This means that they may commit changes that conflict at a semantic level in the source code. In one example, two users may independently implement a `sqrt` function in a math library while disconnected. When they reconnect, the VCS could reconcile their changes automatically by including both `sqrt` implementations, but then the math library would fail to compile because the semantics of the resulting source code are invalid. Although concurrent modifications may not always conflict in this way, it is best to avoid situations where the repository revision does not compile or contains inconsistencies. Since it is difficult to automatically determine if concurrent changes will cause a source code inconsistency, the VCS should record the conflict and allow a human to make the distinction when convenient.

Flexible Update Sharing: A VCS should allow users to share their changes with each other whenever their computers can communicate. This includes scenarios where two users are sitting next to each other on an airplane; they are able to connect to each other but not to the VCS server or the other client hosts (see Figure 1b). They should be able to commit changes and share them with each other via the VCS, even though the repository server is not reachable.

Server Failures: Another scenario that the VCS should handle gracefully is a server failure. If a VCS server fails, the system should be able to switch to a backup server seamlessly. The event that motivated the Pastwatch project was a power failure in our laboratory one day before a conference submission deadline; the failure disabled our CVS server. We were able to create a new CVS repository off-site, but our history was unavailable and there was no simple way to reconcile the change history between the old and new repositories after the power was restored. Ideally, the VCS would be able to manage update consistency between replicas so that switching between repository replicas would be easy.

3 Revtrees

Pastwatch supports disconnected operation by storing a full repository replica on each member's computer. If two users concurrently modify their repository replicas, there is a risk that the modifications will conflict when the users attempt to synchronize their replicas. Pastwatch ensures that neither modification is lost and that all replicas eventually reflect both modifications. That is, Pastwatch applies an optimistic replication strategy [25] to the repository.

Construction: The fundamental task of a repository is to store past revisions of the project files. Pastwatch stores these revisions in a *revtree* data structure that exploits the inheritance between immutable revisions to help it provide optimistic replication.

Each revision logically contains *one* version of *every* file in the project. Revisions are related through inheritance: normally a project member starts with an existing revision, edits some of the files, and then commits a new revision to the repository. This means each revision except the first one is a descendant of an earlier revision. Figure 2a illustrates this inheritance relationship between revisions A through E. The dashed arrow from A to B indicates that a user modified some files from revision A to produce revision B.

Pastwatch stores the repository as a revtree modeled after the inheritance graph. A revtree is a directed acyclic graph, where each node contains a revision. Each revision is immutable and has a unique revision identifier called an RID. Each revision contains a *parent* pointer: the RID of the revision from which it was derived (see Figure 2b).

When a user commits a change to the repository, Pastwatch creates a new revision, adds it to the revtree in the user's local repository replica and finally synchronizes with the other replicas to share the new revision. If users commit new revisions one at a time, each based on the latest revision acquired via synchronization, then the revtree will be a linear revision history.

Handling Network Partitions: Users may not be able to synchronize their replicas due to lack of network connectivity. They may still commit new revisions, but these revisions will often not be derived from the globally most recent revision. These concurrent updates pose two problems: the overall revision history will no longer be linear, and the various repository replicas will diverge in a way that leaves no single most up-to-date replica.

When two users with divergent repositories finally synchronize, Pastwatch must reconcile their differences. Its goal is to produce a new revtree that reflects all changes in both users' revtrees, and to ensure that, after sufficient pair-wise synchronizations, all replicas end up identical. Each repository can be viewed as a set of revisions, each named by an RID. Revisions are immutable, so two divergent revtrees can only differ in new revisions. This rule holds even for new revisions that share the same parent, since the parent revision is not modified when a new child is added. Pastwatch chooses RIDs that are guaranteed to be globally unique, so parent references cannot be ambiguous and two copies of the same revision will always have the same RID no matter how many times the replicas are synchronized. These properties allow Pastwatch to synchronize two revtrees simply by forming the union of their revisions. Any synchronization topology, as long as it connects all users, will eventually result in identical repository replicas.

Revtrees gain several advantages by using the union operation to synchronize replicas. First, partially damaged revtrees can be synchronized to reconstruct a valid and complete replica. Second, the synchronization process can be interrupted and restarted without harming the revtrees. Finally, the synchronization system does not need internal knowledge of the revtree data structure; Section 5.2 describes how Pastwatch uses this property to store a replica in a distributed hash table.

Managing Forks: The usual result of commits while disconnected is that multiple users create revisions with the same parent revision. After synchronization, users will see a *fork* in the revtree: a non-linear revision history in which one revision has multiple successors. Figure 3 illustrates the formation of a fork, caused by two disconnected users both creating revisions (C and D) based on revision B. Now the revtree has two leaves; the path from each leaf to the root node is called a *branch*.

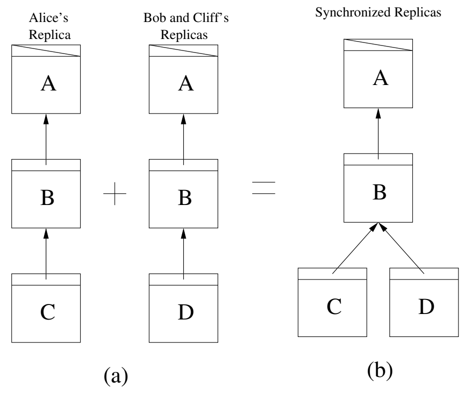

Figure 3: Forking Examples. (a) Two divergent revtree replicas. (b) The two divergent replicas from (a) are synchronized and the resulting revtree contains a fork.

A fork correctly reflects the existence of potentially incompatible updates to the project files, which can only be be resolved by user intervention. If nothing is done, the repository will remain forked, and users will have to decide which branch they wish to follow. This may be appropriate if the concurrent updates reflect some deeper divergence in the evolution of the project. However, it will often be the case that the users will wish to return to a state in which there is a single most recent revision. To reconcile two branches, a user creates a new revision, with the help of Pastwatch, that incorporates the changes in both branches and contains two parent pointers, referring to each of the two branch leaves. Ideally, the user should reconcile when he is connected to the network so that the reconcile is available to other users immediately; this avoids having other users repeat the reconcile unnecessarily. Figure 4a illustrates two branches, C and D, that are reconciled by revision E.

As with any commit, a disconnected user may commit a new child to revision C before he sees E. The resulting revtree is illustrated in Figure 4b. Once again, the revtree has two leaves: F and E. To reconcile these two branches a user proceeds as before. He commits a new revision G with parents E and F. The final branch tree is shown in Figure 4c. Two members can also reconcile the same two branches concurrently, but this is unlikely because Pastwatch will detect a fork when the diverging replicas first synchronize and suggest that the user reconcile it immediately.

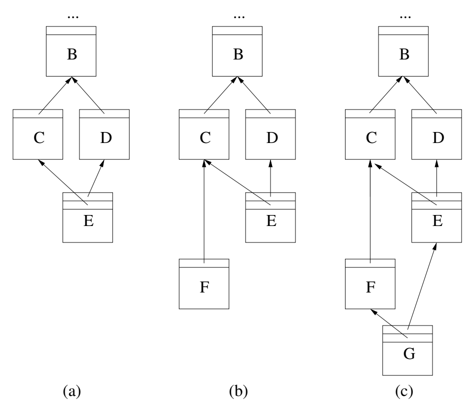

Figure 4: Reconciling Branches. (a) Revision E reconciles the fork and joins C and D. (b) Revision F creates a new fork and reuses C. (c) Revision G reconciles the new fork.

Synchronization Patterns: Pastwatch users may synchronize pairs of replicas in whatever patterns they prefer. One reasonable pattern is to mimic a centralized system: for every replica to synchronize against the same designated "rendezvous" replica. This pattern makes it easy for all users to keep up to date with the latest generally-available revision. Another pattern is ad-hoc synchronization which helps when users are isolated from the Internet but can talk to each other. Figure 5 illustrates both rendezvous and ad-hoc synchronization.

Revtree Benefits: Revtrees provide a number of key benefits to Pastwatch. First, revtrees provide flexibility in creating and maintaining replicas because they guarantee that the replicas will converge to be identical. For example, if a project's rendezvous service is not reliable, its users can fall back to ad-hoc mode. Alternatively, the users could also start or find a replacement rendezvous service and synchronize one of the user's local replicas with it, immediately producing a new working rendezvous replica.

Revtrees also aid with data corruption and data transfer. If two replicas are missing a disjoint set of revisions, they can synchronize with each other to produce a complete replica. Also, the new revisions are always easy to identify in a revtree, so synchronization uses very little bandwidth.

Revtrees handle several types of failure, using the fork mechanism for all of them. For example, if a rendezvous loses a leaf revision due to a disk failure, then another user could inadvertently commit without seeing the lost revision. After repairing the rendezvous, the visible evidence of the failure would be an implicit fork. Similarly, network partitions and network failures can result

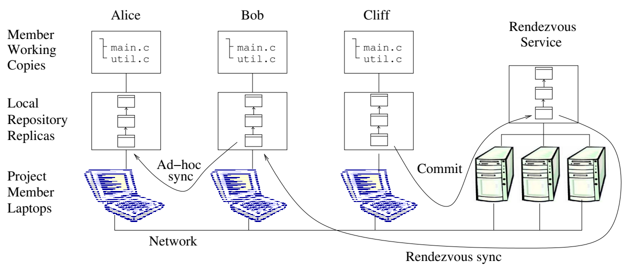

Figure 5: User-visible Model. Local repository replicas, rendezvous replica and working copies.

in forks. The revtree's eventual consistency ensures that the only impact of such failures is a fork. Users only need to learn one technique (reconciling forks) in order to deal with a wide range of underlying problems and as we show in Section 4, reconciling forks is not difficult, so using forks to handle failures is convenient for users.

4 User-Visible Semantics

This section explains the user's view of how Pastwatch works. To the extent possible, Pastwatch's behavior mimics that of CVS.

Working Copy: A Pastwatch user edits a *working copy* of the project files, stored as ordinary files in the user's directory. A user creates a working copy by *checking out* a *base* revision from the repository. The `checkout` command copies the files from the base revision into the working copy and remembers the working copy's base revision.

Tracking New Revisions: In order to see other users' new revisions, a user will periodically *update* her working copy. The `update` command first fetches new revisions from the rendezvous replica. It then checks if the working copy's base revision has any new children. If the base revision has just one child, Pastwatch will apply changes from the child to the working directory. Pastwatch will follow single children, merging them into the working directory with 3-way diff[6], until it reaches a revision with either zero or more than one child. Pastwatch changes the working directory's record of the base revision to reflect this last revision.

Committing New Revisions: In most cases, a linear history of changes is desirable, so Pastwatch will not create a fork if it can avoid it. When a user tries to commit new changes stored in the working copy, Pastwatch first tries to synchronize the local revtree against the rendezvous. It then checks whether the working copy's base revision has any descendants. If the base revision does have new descendants, Pastwatch will refuse to create a new revision until the user updates his working copy.

There is a potential race between reading the base revision and appending a new revision. As an optimization, Pastwatch uses a best-effort leasing scheme to prevent this race from causing unnecessary forks. Pastwatch tries to acquire a lease on the repository before fetching the base revision and releases it after synchronizing the new revision with the rendezvous replica. When the user can contact the rendezvous service, Pastwatch uses the rendezvous service to store the lease. The lease is only an optimization. If Pastwatch did not implement the lease, the worst case outcome is an unnecessary fork when two connected users commit at exactly the same time. If the rendezvous is unavailable, Pastwatch proceeds without a lease.

Implicit Forks: If two disconnected users independently commit new revisions, an implicit fork will appear when synchronization first brings their revisions together. A user will typically encounter an unreconciled fork when updating her working copy. If there is an unreconciled fork below the user's base revision, Pastwatch warns the user and asks her to specify which of the fork's branches to follow. Pastwatch allows the user to continue working along one branch and does not force her to resolve the fork. This allows project members to continue working without interruption until someone reconciles the fork.

Explicit Forks: Pastwatch users can fork explicitly to create a new branch so that they can keep their changes separate from other members of the project. To explicitly fork, a user commits a new revision in the revtree with an explicit branch tag. Pastwatch ignores any explicitly tagged revisions when other users update.

Reconciling Forks: Both implicit and explicit branches can be reconciled in the same way. Reconciling forks is no more difficult than updating and committing in CVS. Figures 6 and 7 illustrate the process.

Forks first appear after two divergent replicas synchronize. In the examples, Alice synchronizes her local replica during an `update` and Pastwatch reports a new fork because both Alice and Bob made changes while Alice was disconnected from the network. To reconcile the fork, Alice first issues a `reconcile` command which applies the changes from Bob's branch into Alice's working copy.

In Figure 6, there were no textual conflicts while applying Bob's changes to Alice's working copy, so Alice can just commit a new revision that is a child of both Alice's and Bob's revisions as shown in Figure 4a. In contrast, Figure 7 shows what Alice must do if the fork created a source code conflict. Pastwatch notifies Alice during the `reconcile` and inserts both conflicting lines into her working copy the way CVS reports conflicts during an `update`. After Alice resolves the conflict she can commit the final revision.

```
alice% past update
  Tracking branch: init, alice:3
  Branch "init" has forked.
  current branches are:
    branch "init": head is alice:3
    branch "init": head is bob:2

alice% past -i reconcile -t bob:2
  Tracking branch: init, alice:3
  updating .
  Reconciling main.c
  M main.c: different from alice:3

alice% past -i -k bob:2 commit -m "Reconcile branches"
  Tracking branch: init, alice:3
  checking for updates and conflicts
  updating .
  M main.c
  committing in .
  committing main.c
  Built snapshot for revision: alice:4
```

Figure 6: Reconciling a fork without source code conflicts.

5 Implementation

The Pastwatch software is written in C++ and runs on Linux, FreeBSD and MacOS X. It uses the SFS tool-kit[18] for event-driven programming and RPC libraries. It uses the GNU diff and patch libraries to compare different revisions of a file and perform three-way reconciliation. Pastwatch is available at: `http://pdos.csail.mit.edu/pastwatch`.

5.1 Storage Formats

Pastwatch stores the entire local replica in a key-value store implemented by a BerkeleyDB database for convenience. All the replica data structures are composed of key-value pairs or *blocks*. Immutable blocks are keyed by the SHA-1[11] hash of their content.

For the sake of storage and communication efficiency, each revision in the revtree only contains the difference from the parent revision rather than an entire copy of the source code files. The internal representation of a revision is a combination of a *revision record* and *delta blocks*, all of which are immutable blocks. Delta blocks contain the changes made to the parent revision in the GNU diff format. Figure 8 illustrates the structure of a revision record. The RID of a revision equals the SHA-1

```
alice% past update
  Tracking branch: init, alice:3
  Branch "init" has forked.
  current branches are:
    branch "init": head is alice:3
    branch "init": head is bob:2

alice% past -i reconcile -t bob:2
  Tracking branch: init, alice:3
  updating .
  Reconciling main.c
  C main.c: conflicts with alice:3

alice% grep -A4 "<<<" main.c
  <<<<<<< alice:3
      int increase (int x) { return x + 1; }
  =======
      void increase (int &x) { x++; }
  >>>>>>> bob:2

< Alice reconciles conflicting edits with a text editor >

alice% past -i -k bob:2 commit -m "Reconcile branches"
  Tracking branch: init, alice:3
  checking for updates and conflicts
  updating .
  M main.c
  committing in .
  committing main.c
  Built snapshot for revision: alice:4
```

Figure 7: Reconciling a fork with a source code conflict.

hash of the revision record block. `parent` contains the RID of the parent revision. `previous` contains the key of the previous entry in the member log described in Section 5.2. The remainder of the revision record contains references to delta blocks. The revision record includes the first few delta blocks; if there are more deltas, Pastwatch will use single and double indirect blocks to reference the deltas. The arrangement of delta blocks was inspired by the UNIX file system's[19] handling of file blocks.

Pastwatch keeps a local snapshot of each revision's files and directories so that it can retrieve old revisions quickly. Pastwatch saves the snapshots locally in a CFS[9] like file system that reuses unchanged blocks to conserve storage space. Since revision records only contain deltas, Pastwatch constructs the snapshots by applying deltas starting at the root of the revtree. Since Pastwatch keeps all the snapshots, it only needs to construct snapshots incrementally when it retrieves new revisions. Snapshots are stored in the local key-value store.

5.2 Rendezvous Services

We have implemented two different rendezvous services for Pastwatch. First, we implemented a single server rendezvous service where all users synchronize their replicas with the single server. This service is fully functional, but if the server becomes unavailable, the users will probably need to use ad hoc synchronization to share changes which can be slow to propagate new changes. It is possible to implement a hot spare replica for the single server but instead, we constructed a resilient rendezvous service using a distributed hash table(DHT)[27][10][17][33].

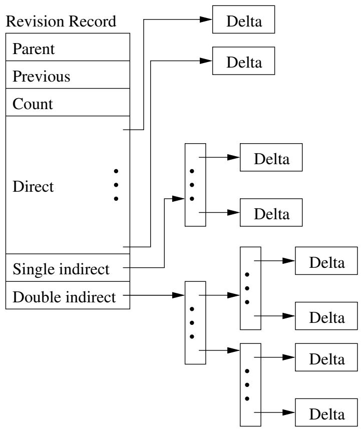

Figure 8: Revision record data structure with delta blocks.

DHTs promise abundant, reliable storage and the arrival of public storage DHTs like OpenDHT[13][26] make them an attractive choice for a Pastwatch rendezvous service. Many different projects can all share the same DHT as their rendezvous service and since DHTs are highly scalable, one could build a large repository hosting service like Sourceforge[5] based on a DHT.

Revtrees are compatible with DHTs because a DHT is a key-value storage service and revtrees can tolerate the imperfect consistency guarantees of DHT storage. As shown in Section 3, revtrees handle network partitions, missing blocks and slow update propagation, so a storage inconsistency in a rendezvous DHT will at worst cause a fork in the revtree. The only additional requirement of the DHT is that it must support mutable data blocks so that Pastwatch can discover new revisions.

Pastwatch uses mutable blocks and one extra data structure when using a DHT in order to discover new revisions; this is because the `put`/`get` DHT interface requires a client to present a key to get the corresponding data block. Each revtree arc point upwards, towards a revision's parent; the revtree does not contain pointers to the newest revisions, so Pastwatch must provide a way to discover the keys for new revisions. Pastwatch accomplishes this by storing the revisions in a per-user log structure that coexists with the revtree; the structure is rooted by a mutable DHT block. The address of the mutable block is an unchanging repository ID. Pastwatch can find the new revisions as long as it has the repository ID, thus it can find all revisions in the revtree.

Figure 9 illustrates the revtree DHT structures. In this example, the project has two developers, Alice and Bob.

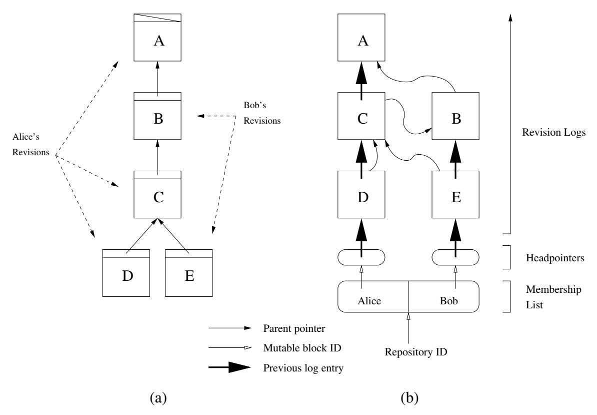

Figure 9: DHT storage structures. (a) shows an example revtree. (b) shows the same revtree, as stored in the DHT. Rounded rectangles are mutable blocks and square rectangles are immutable blocks.

Figure 9a shows the repository's revtree including who created each revision. Figure 9b shows how the revtree is stored in the DHT.

Each user has their own log that contains every revision they created. Each user maintains a pointer to their newest revision in a mutable block called a *headpointer*. The project's membership list contains a pointer to each of the users' headpointers. All the blocks in the DHT structure are immutable except the headpointers and the membership list.

It is efficient to synchronize replicas because finding the newest revisions is efficient. Pastwatch only needs to scan the membership list and traverse each user's log until it encounters an RID it has encountered in the past because the newest revisions are at the front of the log. Since the revisions are immutable, Pastwatch can be sure that the remainder of the log has been processed at an earlier time.

5.3 DHT Implementation

At the time of writing, OpenDHT is the only public DHT storage service we are aware of that implements the `put/get` interface. Although OpenDHT provides the correct interface, it will purge a data block after storing it for a week unless the block is inserted again. Pastwatch cannot use OpenDHT because Pastwatch reads old blocks during a fresh checkout and a checkout from the DHT replica will fail if a block is unavailable. We implemented our own DHT that provides long-term storage, but Pastwatch can be modified to use a suitable public storage DHT if one becomes available.

The Pastwatch DHT rendezvous service is derived from Dhash[9][10]. Immutable blocks are stored under the SHA-1 hash of their content. Each mutable block has a single writer and the DHT only allows mutations that are signed by a private key owned by the writer. Each mutable block has a constant identifier equal to the hash of the owner's public key. Each mutable block contains a version number and the owner's public key along with the block's payload. Each time an owner updates his mutable block, he increases the version number and signs the block with his private key. The DHT stores the block along with the signature and will only overwrite an existing mutable block if the new block's version number is higher than the existing block and the signature is correct.

5.4 Data Durability

The Pastwatch DHT uses the IDA coding algorithm [23] to provide data durability. For each block, the DHT stores 5 fragments on different physical nodes and requires 2 fragments to reconstruct the block. The DHT also actively re-replicates blocks if 2 of the fragments become unavailable. Data loss is unlikely because the nodes are well maintained server machines, but if the DHT does experience a catastrophic, corollated failure, any user with an up-to-date local replica can perform a repair by synchronizing his local replica with the rendezvous service. Alternatively, he could easily create a new single server rendezvous service. In either case, synchronizing his local replica will completely repopulate the empty rendezvous service. A corrupt replica on the rendezvous services can also be repaired by synchronizing with a valid local replica and in some cases, two corrupt replicas can repair each other simply by synchronizing with each other.

In practice, each Pastwatch project must evaluate its own data durability requirements. If a project has many active members who keep their local replicas up-to-date, then the members may elect to forgo any additional backup strategy. On the other hand, a project with only one member may choose to keep regular backups of the member's local replica.

6 Evaluation

This section evaluates the usability and performance of Pastwatch. First, we analyze a number of open-source projects and find that real users frequently commit 5 or more times a day, enough that they would want disconnected commits during a long plane flight. We also find that in a real 26-person team, 5 or fewer team members commit in the same day 97% of the time which suggests that even a day-long network partition will not overwhelm a Pastwatch project with implicit forks. We then share experiences from a small initial user community which has been using Pastwatch for more than a year. In that time, Pastwatch has been easy to use and survived a number of network and storage failures. In the same time period our CVS server experienced significant down-time.

We then show that Pastwatch has reasonable performance. Common operations in our experimental workload, like `commit`, take 1.1 seconds with CVS and 3.6 seconds with Pastwatch. Pastwatch can also support many members per project; increasing the number of members from 2 to 200 increases the update time from 2.3 seconds to 4.2 seconds on a wide-area network. We also show that retrieving many old revisions is not expensive; pulling 40 new revisions from the rendezvous replica and processing the revisions locally takes less than 11 seconds.

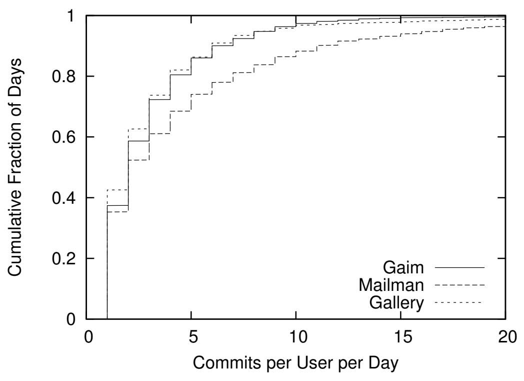

Figure 10: Cumulative distribution of per user, daily commit counts. In 18% of the active days, a single Gallery member made 5 or more commits in a single day. In 22% of the active days, a single Mailman members made 7 or more commits in a single day.

6.1 Usability Evaluation

6.1.1 Disconnected Operations:

To evaluate the usefulness of the ability to commit while disconnected, we analyze the per-member commit frequency of real open-source projects. We find that it is common for a single project member to commit several new revisions in a single day and conclude that the ability to commit while disconnected more than a few hours would be useful.

We analyzed the CVS commit history from three of the more active open source projects hosted on the Sourceforge[5] repository service: Gaim, Mailman and Gallery. Figure 10 characterizes the daily commit activity for all members in each project for days that contain commits. The plot shows that the median number of commits is relatively low at only 2 commits, but there is a significant fraction of days in which a single user commits 5 or more times. In 18% of the active days, a single Gallery member made 5 or more commits in a single day. In 22% of the active days, a single Mailman member made 7 or more commits in a single day.

Considering that most users will be programming fewer than 16 hours in a day, the high daily commit counts suggest that even a disconnection period of 3-5 hours would interrupt a user's normal work-flow and so disconnected commits could be useful for these projects.

6.1.2 Commit Concurrency:

Pastwatch users are able to commit while disconnected or partitioned so there is a risk that many project members will commit concurrently and create a large number of implicit forks. To evaluate how often disconnected commits would actually result in an implicit fork, we analyzed the temporal proximity of commits by different project members in real open-source projects. We found that different project members do commit at similar times, but the level of concurrency should not cause a large number of forks.

The number of forks that may result from a network partition is limited to the number of partitions because replicas in the same partition can always synchronize with each other and they should not accidentally create a fork within their partition. The worst case occurs when every member in a project commits a new revision while they are all partitioned from each other. This scenario results in a separate branch for each member. To evaluate the likelihood of the worst case forking scenario, we analyzed the CVS logs for the same three open-source projects used in Section 6.1.1.

The Gaim, Mailman and Gallery projects have 31, 26 and 21 active members respectively, so the worst case number of branches is quite high. The highest number of unique committers in a single day, however, was only 9, 6 and 5 respectively. Even if all the members in each project were partitioned into individual partitions for a 24 hour period and they made the same commits they made while connected, the number of resulting forks in each project would still be quite low and significantly fewer than the total number of members in the project.

The low number of concurrent commits on the highest concurrency day already suggests that the number of implicit forks will be manageable, but to better understand the common case, we consider the distribution of unique committers. Figure 11 shows the distribution of the number of unique users who commit in a calendar day. Mailman sees three or fewer unique committers 99% of the time and Gaim sees five or fewer unique com-

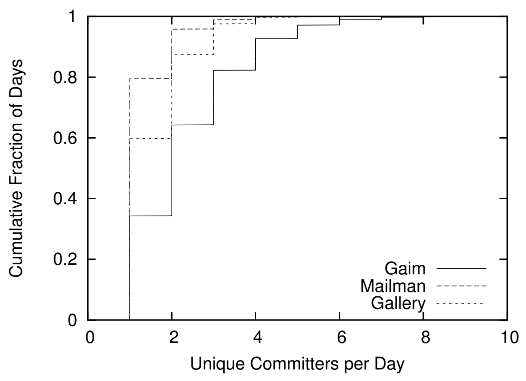

Figure 11: Cumulative distribution of unique committers per day. Mailman sees three or fewer unique committers 99% of the time and Gaim sees five or fewer unique committers 97% of the time.

mitters 97% of the time.

The distribution suggests that the number of concurrent committers is normally low with respect to the number of project members. The low frequency of concurrent commits combined with the ease of reconciling forks described in Figure 6 suggests that implicit forks will be manageable in practice.

6.1.3 Practical Experience:

Pastwatch currently has a user community of five people and six projects: three documents and three software projects including this research paper and the Pastwatch software itself. All Pastwatch users agree that the system is as usable and convenient as CVS.

The Pastwatch users primarily use connected mode and the system has behaved like a traditional centralized VCS. On occasion, the users also use disconnected reads and commits. For example, this paper's repository has been active for 202 days. During that time, it has served 816 repository operations including updates, commits, checkouts and diffs; 25 of those operations were performed while the client was disconnected from the network. Out of the 816 operations, there were 181 commits; seven of those commits were performed while the client was disconnected from the network.

All Pastwatch projects use the Pastwatch DHT as their rendezvous service, and it has proven to be robust. In the past year, our research group's main file server experienced three days with extended down-time. Since many people store their CVS repositories on the file server, they could not commit changes or read old revisions from the repository during the down-time. Pastwatch users were able to read and write to their local replicas while the file server was unavailable.

6.2 Performance Evaluation

6.2.1 Experiment Setup:

The following experiments though Section 6.2.5 are based on the following setup: CVS is configured with a single CVS server in Cambridge, Massachusetts. It has two different client hosts; one is in New York, New York and the other is in Salt Lake City, Utah. The client in New York has a 1.1 GHz CPU and a 4.3 MB/s bi-directional bottleneck bandwidth to the CVS server with a 6ms round trip latency. The host in Utah has a 1.7 GHz CPU and 0.5 MB/s bottleneck bandwidth to the CVS server and a 55ms round trip latency.

Pastwatch uses the same two client hosts and an 8 node DHT. The client host in New York accesses the DHT through a node with a 6ms round trip latency. The client host in Utah connects to a nearby DHT node with a 13ms round trip latency. Four of the DHT nodes are spread over North America and the other four are located in Cambridge. The New York client and many of the DHT nodes are on the Internet2 research network but the Utah client is not, so the New York client has higher throughput links to the DHT than the Utah client.

The base workload for each experiment is a trace from the CVS log of the SFS open-source software project. The trace begins with 681 files and directories and includes 40 commit operations. On average, each commit changes 4.8 files, the median is 3, and the highest is 51. Together, the 40 commit operations modify roughly 4330 lines in the source-code and add 6 new files. Each data point is the median of 10 trials and for each trial, Pastwatch used a different repository ID and different head-pointer blocks.

6.2.2 Basic Performance:

This section compares the performance of basic VCS operations like `import`, `checkout`, `update` and `commit` in Pastwatch and CVS. Their times are comparable, but round trip times and bottleneck bandwidths affect them differently.

In each experiment, the primary client host creates the project and imports the project files. Each client then checks out the project. Afterwards, the primary client performs the 40 commits. After each commit, the secondary client updates its replica and working copy to retrieve the new changes. The experiment was run once with the New York client as primary and again with the Utah client as primary. The Pastwatch project has two members.

Table 1 reports the costs (in seconds) of the import operation, the checkout operation, and the average costs of the commit and update operations for each client running the workload.

	New York Client				Utah Client			
	import	checkout	mean commit	mean update	import	checkout	mean commit	mean update
CVS	5.4	5.8	1.1	2.9	13.0	10.5	2.2	3.8
Pastwatch	167.4	16.3	3.6	3.0	161.4	25.9	3.9	2.4

Table 1: Runtime, in seconds, of Pastwatch and CVS import, checkout, commit, and update commands. Each value is the median of running the workload 10 times. The update and commit times are the median over 10 trials of the mean time for the 40 operations in each workload.

Since Pastwatch creates a local repository replica during import and checking out a working copy from a complete replica is trivial, the checkout time for the client that imported is not reported here. Instead, we report the checkout time on the client that did not import.

Initially importing a large project into CVS takes much less time than with Pastwatch because CVS stores a single copy of the data while the Pastwatch DHT replicates each data block on 5 different DHT nodes. In practice, a project is only imported once, so import performance is not very significant.

Pastwatch has a slower checkout time than CVS because it must process the repository files twice. Once to create the replica snapshot and once to update the working directory. The Utah Pastwatch client has a slower checkout time than the New York Pastwatch client because it has lower bottleneck bandwidths to many of the DHT nodes.

Commit performance is comparable for Pastwatch and CVS. The difference is at most 2.5 seconds per operation. Pastwatch commits are slower than CVS because inserting data into the DHT replica is more expensive than into a single server and acquiring the lease takes additional time.

Update performance for the New York client is similar for CVS and Pastwatch. CVS update is slower at the Utah client than the New York client because the Utah client has a longer round trip time to the server and CVS uses many round trips during an update. Pastwatch updates at the Utah client are faster than at the New York client because the update operation is CPU intensive and the Utah client has a faster CPU.

6.2.3 Storage Cost:

A revtree contains every historical revision of a project; this could have resulted in a heavy storage burden, but Pastwatch stores revisions efficiently by only storing the modifications rather than entire file revisions, so the storage burden on the client replica is manageable.

After running the workload, each client database contained 7.7 megabytes of data in 4,534 blocks. 3,192 of the blocks were used to store the revtree replica. The remaining blocks were used to store the snapshots. On disk, the BerkeleyDB database was 31 megabytes, because BerkeleyDB adds overhead for tables and a transaction log. The transaction log makes up most of the overhead but its size is bounded. In comparison, the CVS repository was 5.2 megabytes not including any replication.

The storage burden on the DHT is not very high. After running the workload described in Section 6.2.1, the resulting revtree was 4.7 megabytes in size. This means that the DHT was storing 24 megabytes of revtree data because each mutable blocks in the DHT is replicated 5 times and immutable blocks are split into 5 fragments (most of the immutable blocks are small, so each fragment is roughly the same size as the original block). Each of the 8 DHT nodes held 3 megabytes each. Again, the BerkeleyDB database adds storage overhead, so the size of the entire database on each node was 15 megabytes.

6.2.4 Many Project Members:

This section examines how Pastwatch scales with the number of project members. Pastwatch checks for new revisions at the rendezvous before most operations, so it regularly fetches each member's headpointer. This imposes an $O(n)$ cost per project operation where n is the number of project members. This experiment uses the same setup and workload as Section 6.2.1, except the number of project members increases for each experiment. In this experiment, the New York client performs the commits and the Utah client performs an update after each commit.

Pastwatch can fetch the headpointers in parallel because it has all the headpointer addresses after retrieving the member list. Since the headpointers are small and the number of network round trips necessary to retrieve them does not depend on the number of project members, large numbers of members do not greatly affect Pastwatch operation times. Figure 12 shows that the median costs of commit and update operations increase as the number of project members increases but even at 200 members, twice as large as the most active project in Sourceforge, commits take only 1.7 seconds more and updates take 1.9 seconds more than a 2 member project. The standard deviation is between 0.4 and 0.9 seconds

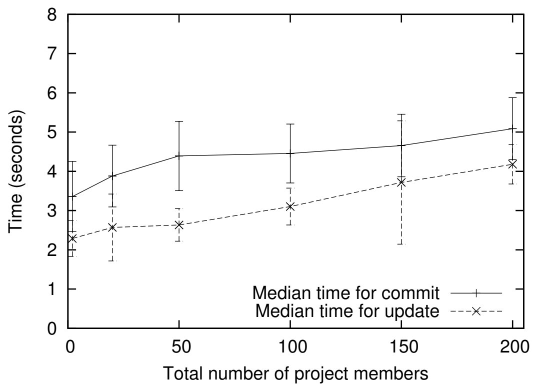

Figure 12: Median costs of commit and update in a workload for each user as the number of project members increases. Each value is the median of running the workload 10 times. The error bars are standard deviations.

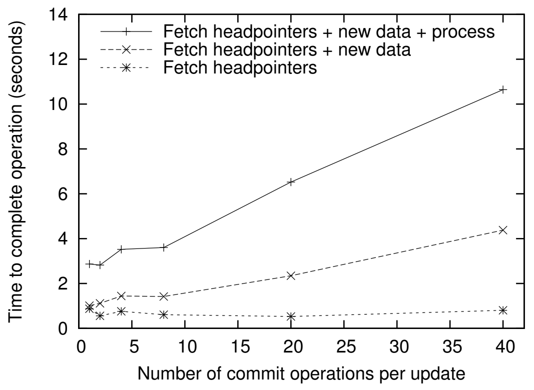

Figure 13: Median time to complete one update operation as the number of commits per update operation increases.

and is due to varying network conditions. Ultimately, scanning the membership list is not a significant expense.

For a fixed number of new revisions, increasing the number of members who committed a revision reduces the time to retrieve the revisions because Pastwatch can retrieve revisions from different members in parallel. The worst case performance for retrieving a fixed number of new revisions occurs when a single member commits all the new revisions because Pastwatch must request them sequentially.

6.2.5 Retrieving Many Changes:

This section examines the cost of updating a user's working copy after another user has committed many new revisions. To bring a revtree up-to-date, Pastwatch needs to fetch all the new revisions which could be expensive in cases where a user has not updated her replica for some time.

These experiments use the same setup and workload as Section 6.2.1, except that only the New York client commits changes and it commits several changes before the Utah client updates its revtree and working copy. The number of commits per update varies for each experiment.

Figure 13 reports the cost of one update operation as the number of commits per update increases. The bottom curve in the figure shows only the time spent fetching headpointers. The middle curve adds the time spent fetching new revisions and delta blocks. Finally, the top curve adds in the cost of local processing to build snapshots and modify the working copy.

The top curve shows that the total cost of an update operation increases linearly with the number of revision records it needs to fetch. Decomposing the update time reveals that the linear increase is due to Pastwatch spending more time fetching revision records and delta blocks and building new snapshots. The widening gaps between the three plots illustrates that these two operations increase the runtime linearly.

7 Related Work

Version Control Systems: There are many existing VCSs but most do not attempt to support disconnected commits and ad-hoc synchronization.

Most existing VCSs are based on a client-server architecture. CVS[7], Subversion[22], Perforce[32] and Clearcase[31] all rely on a single repository server to store and manage different revisions of the project data. They do not support disconnected commits. Similarly, if the server becomes unavailable, no user can access the repository. Two users who are connected to each other cannot share changes with each other through the system when the server is unavailable.

Bitkeeper[1] uses a hierarchy of repositories to cope with server failures; it divides users into subgroups. Each subgroup commits changes to a sub-repository and propagates changes to a parent repository when they are ready to share them. A user may also have her own sub-repository, so she can read and write the repository while disconnected from the other repositories. After she reconnects, she commits locally saved changes to the parent repository. The local repository supports disconnected commits, but users in different groups cannot share changes if a parent is unavailable.

Coven[8] uses lightweight forks to support disconnected commits. When Coven users cannot contact their repository, they commit to a local lightweight fork which resembles a log. Later, when they can communicate with the repository, they commit the lightweight fork back into the repository. Coven can support a disconnected user's commits, but directly connected users cannot share changes if the repository is unreachable.

The Monotone[4] repository resembles a revtree internally. Its repository tolerates the same kinds of inconsistencies that Pastwatch does. Monotone provides authentication by having each committer sign each revision separately whereas Pastwatch authenticates revisions with a hash tree based on a single signed reference for each writer. The hash tree makes it possible for Pastwatch to find the newest revisions when storing data in a DHT. Monotone was developed concurrently and independently from Pastwatch. In the past year, Mercurial[3] and GIT[2], have been developed based on the ideas found in Monotone and Pastwatch. We are encouraged by the use of revtree concepts in these systems.

Optimistic Replication: In addition to Pastwatch, there are many other optimistic concurrency systems that use a variety of techniques for detecting write conflicts. Using version vectors[21][24] is one common technique along with its newer variant, concise version vectors[16]. These techniques use logical clocks on each replica to impose a partial order on shared object modifications. The systems tag the shared objects with logical timestamps, which allow the systems to detect when a write-write conflict appears. Systems like Locus[30], Pangaea[28] and Ficus[20][25] use these optimistic concurrency techniques to implement optimistically replicated file systems.

Other systems, such as Bayou[29], use application specific checker functions to detect write-write conflicts. For every write, a checker verifies that a specific precondition holds before modifying the object. This ensures that the write will not damage or create a conflict with an existing object.

Coda[14][15] detects write-write conflicts by tagging each file with a unique identifier every time it is modified. When a disconnected client reconnects and synchronizes a remotely modified file, it will detect a write-write conflict because the file's tag on the server will have changed. Coda can use this technique because its file server is the ultimate authority for the file; all changes must go back to the server. Pastwatch cannot use this method because it has no central authority for its repository.

Hash histories[12] also detect write-write conflicts and resemble revtrees, but their focus is to understand how much history to maintain while still being able to detect conflicts. Pastwatch intentionally keeps all history because the version control application needs it.

All these optimistic concurrency systems provide a way to detect write-write conflicts on a shared object, but the version control application needs more than conflict detection. It also needs the contents of all past revisions and the inheritance links between them.

It may be possible to combine version vectors with write logging to get both conflict detection and revision history, but revtrees perform both tasks simultaneously without the limitations of version vectors; revtrees do not need logical clocks and they readily support adding and removing replicas from the system.

It may also be possible to use an optimistic concurrency system to replicate an entire repository as a single shared object containing all the revision history. This approach is difficult because most existing version control systems are not designed for concurrent access and conflict resolution. The version control system's data structures must be consistent for it to function properly, but the data structures in the repository and working copies often contain interdependencies. This means the conflict resolver will need to repair the repository replicas and the working copies or else the VCS will not function properly. Although it may be possible to construct an automatic conflict resolver for an existing VCS, Pastwatch shows that a separate conflict resolver is unnecessary if the data structures are designed for concurrency. The revtree requires no active conflict resolution for its data structures and the Pastwatch working copies do not need to be repaired after concurrent writes.

8 Conclusion

We have presented Pastwatch, a distributed version control system. Under normal circumstances, Pastwatch appears like a typical client-server VCS, but Pastwatch optimistically replicates its repository on each users' computer so that each user may commit modifications while partitioned from servers and other members of the project. A user can directly synchronize his replica with other user's replicas or a rendezvous service in any pattern. All users in a given network partition can always exchange new modifications with each other.

Pastwatch supports optimistic replication and flexible synchronization between replicas because it represents the repository as a revtree data structure. Revtrees provide eventual consistency regardless of synchronization order and they detect repository level write-write conflicts using forks. Reconciling these forks is easy because they only appear at the source code level, not in the data structures of the repository and working copies.

We analyzed real-world software projects to show that disconnected commits are likely to be useful to their developers. We also showed that handling concurrent commits with forking is not a burden, even for active projects.

We implemented Pastwatch and have an initial user community with more than a year of experience using the system. Although Pastwatch is more complex than a client-server system, the implementation successfully hides those details from the users. The users have found Pastwatch to be very usable and the system has masked a number of actual failures, in contrast to the VCS it replaced.

Acknowledgments

We are grateful to the many people who contributed to this work: Butler Lampson and Barbara Liskov for their help with early revisions of the design; Frank Dabek and Emil Sit for their help with Chord/DHash; David Andersen for his help with the RON testbed; members of the PDOS research group and the anonymous reviewers that helped refine our work; and our shepherd Emin Gün Sirer.

The National Science Foundation supported this work as part of the IRIS project under Cooperative Agreement No. ANI-0225660 and with a Career grant.

References

[1] Bitkeeper. `http://www.bitkeeper.com/`.

[2] GIT. `http://git.or.cz/`.

[3] Mercurial. `http://www.selenic.com/mercurial/wiki/index.cgi`.

[4] Monotone. `http://www.venge.net/monotone/`.

[5] Sourceforge. `http://www.sourceforge.net/`.

[6] UNIX diff3 utility, 1988. `http://www.gnu.org/`.

[7] B. Berliner. CVS II: Parallelizing software development. In *Proc. of the USENIX Winter Conference*, 1990.

[8] Mark C. Chu-Carroll and Sara Sprenkle. Coven: Brewing better collaboration through software configuration management. In *Proc. ACM SIGSOFT Conference*, 2000.

[9] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2001.

[10] F. Dabek, J. Li, E. Sit, J. Robertson, M. Frans Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, March 2004.

[11] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 1995.

[12] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. Hash history approach for reconciling mutual inconsistency in optimistic replication. In *23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, 2003.

[13] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Adoption of DHTs with OpenHash, a public DHT service. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, February 2004.

[14] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the ACM Symposium on Operating System Principles*, 1991.

[15] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proc. of the USENIX Winter Conference*, January 1995.

[16] Dahlia Malkhi and Doug Terry. Concise version vectors in WinFS. In *The 19th Intl. Symposium on Distributed Computing (DISC)*, Cracow, Poland, September 2005.

[17] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st IPTPS*, March 2002.

[18] D. Mazières. A toolkit for user-level file systems. In *Proc. of the USENIX Technical Conference*, June 2001.

[19] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3), 1984.

[20] T. Page, R. Guy, G. Popek, and J. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report UCLA-CSD 910005, 1991.

[21] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. In *IEEE Transactions on Software Engineering*, volume 9(3), 1983.

[22] C. Michael Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., 2004.

[23] Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.

[24] D. Ratner, P. Reiher, G.J. Popek, and R. Guy. Peer replication with selective control. In *Proceedings of the First International Conference on Mobile Data Access*, 1999.

[25] Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *Proc of the USENIX Technical Conference*, 1994.

[26] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of ACM SIGCOMM 2005*, August 2005.

[27] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[28] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[29] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the ACM Symposium on Operating System Principles*, December 1995.

[30] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 1983.

[31] Brian White. *Software Configuration Management Strategies and Rational ClearCase*. Addison-Wesley Professional, 2000.

[32] Laura Wingerd. *Practical Perforce*. O'Reilly & Associates, 2005.

[33] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.