A Peer-to-Peer Approach to Distributed Document Management

Eric Kennedy Advisor: Arvind Krishnamurthy

May 6, 2002

1 Introduction

Over the last decade, computer prices have plummeted to the point where many people now work with multiple computing devices. While it is common to use one computer at home and another at work, PCs are only one of the many computing devices that users will come to rely on in the years ahead. The latest hand-held devices offer operating systems like Windows CE that can run simplified versions of popular PC applications. As the capabilities of these devices rise to meet those of laptops, the need for a scalable document synchronization system across many computing devices becomes readily apparent. Such a system could provide consistency for one user's files, or allow distributed sharing and version control across groups of users. For this project, I investigated a number of the existing implementations and created a proof-of-concept system that synthesizes the best elements to make distributed document management useful and accessible to end users.

Distributed document management is a product category that leverages the technology of replication for the specific problem of maintaining document synchronization across a network of computing devices. The problem that it attempts to solve is one that I experienced personally. During the summer of 2001, I created a home network for two desktops and and one laptop. The laptop was equipped with an 802.11b wireless network card, and I frequently used it in rooms other than the office. I needed to have access to all of my personal files on each of the machines, but if I added or modified a file on any machine, I quickly ran into the issue of replicating the change to the other two. The idea of using one machine as a dedicated server was

vetoed by my mother, who had been inspired by the California power crisis to cut back on our energy consumption. Thus the laptop was usually in a mode of disconnected operation, where access to the other computers was not available because those computers were not running. Disconnected operation is a frequent occurrence for mobile devices, and thus must be designed into any future synchronization system.

Although there are a number of distributed UNIX file systems that accommodate disconnected operation, including Coda, Ficus, and Rumor, I could not find a similar system for Windows [4, 8, 2, 3]. All of the existing commercial Windows software products were designed for synchronization between two computers, and could not handle synchronization between three or more computers. Inspired by the promise of peer-to-peer (P2P) file sharing, I decided to investigate many-to-many replication and then create a proof-of-concept P2P replication system. (While this proof-of-concept system is implemented on Linux, I intend to port it over to Windows after graduation.) This paper describes the system that I designed, and explains how that system can be extended to meet the needs of a wide audience of users.

2 Proposed System

2.1 Background

As with most new product categories in the computer software industry, distributed document management combines the ideas of existing products in new ways. There are many groupware products, including Lotus Notes and Microsoft Exchange Server, that replicate data across multiple computers. Those products were designed to facilitate communication within an organization, ignoring the significant productivity gains that could be offered by improving communication between organizations. With origins in the late 1980s and early 1990s, groupware products were designed around an Internet that was primarily used to transfer email.

While email is still a powerful application a decade later, the Internet has spawned many new ways to communicate: the World Wide Web allows people and organizations to offer information and services online; supply chain management allows companies to efficiently communication with their suppliers; instant messaging allows people to chat one-on-one with others;

online learning and presentation software allows for moderated, one-to-many communication; peer-to-peer file sharing enables highly-scalable, low-cost file distribution; and software like Groove allows ad hoc group collaboration for users within and outside of an organization.

These new uses for the Internet show a trend toward communication applications that end users can easily install and maintain on their own. PC-based communication applications are accessible to a much wider audience of users than server-based applications, and yet can leverage the speed of modern PCs to offer comparable functionality. Consequently, I adopted the approach of the Rumor P2P synchronization system, creating this proof-of-concept system at the application level[3].

2.2 Related Work

A plethora of Computer Science research papers have been written on the topic of data replication over the last decade and a half. There are two types of replication algorithms: *pessimistic* and *optimistic*[7]. Traditional pessimistic algorithms, such as those used by banks to manage transactions, offer single-copy semantics[7]. A pessimistic algorithm will allocate the right to access files (for both reads and writes) to individual clients[8]. These restrictive access provisions ensure that there are never any update conflicts, but at the cost of preventing a user from accessing a file that is cached locally if the system cannot guarantee that no other users have updated the file[8]. As this is often the case during disconnected operation, pessimistic algorithms are not very useful for systems where disconnected operation occurs frequently.

Optimistic algorithms avoid this problem by allowing file access even if the system cannot guarantee that no other updates have occurred to the requested file[8]. Update conflicts, where objects in two or more replicas are each modified during the same interval of disconnected operation, can occur with optimistic algorithms[7]. However, the authors of the Coda replication system have found that update conflicts are extremely rare in practice, and can often be easily resolved[8]. File changes can be determined by creating file metadata and then comparing modification times from the system against the metadata[7]. Because update conflicts can occur, computers using an optimistic replication algorithm must track the complete history of changes to a file to correctly determine changes and conflicts between computers[7]. (Note that the changes themselves do not have to be recorded.) This history

information, created with timestamps from synchronized wall-clocks or with non-decreasing counters, is stored in *version vectors*[7].

While optimistic replication systems can resolve update conflicts by simply discarding all but one conflicting update to a file, that is much less useful than a system that prompts users to reconcile the differences manually[3]. Reconciliation, where replicas contact each other and exchange file version information, can either occur automatically or manually[3]. Automatic reconciliation requires integration into the kernel, and thus is not possible on systems where users lack root access or do not want to modify their operating system[3]. Further, automatic reconciliation may occur when a user does not want it to and consequently copy files that should not be replicated. For example, temporary save files and core files would be copied under a replication system which automatically replicated every new, changed, or deleted file[3]. Manual replication allows users to determine when and what files get replicated, resulting in a system where the default state is that of disconnected operation.

Optimistic replication systems are implemented as either single-master or multi-master systems[7]. A single-master system is a client-server hub-and-spoke architecture, where all of the clients synchronize their local copies with the master server[7]. The total number of synchronizations required in a single-master system is $O(N)$, where N is the number of client replicas[7]. A multi-master system uses a peer-to-peer replication approach, where all replicas must synchronize their local copies with every other replica that has write-access[7]. Consequently, the total number of synchronizations required in a multi-master system is $O(M * N)$, where N is the number of both read-only and write-access replicas and M is the number of write-access replicas[7]. To improve the scalability of multi-master systems while avoiding the single point-of-failure present in single-master systems, hybrid systems have been created where individual write-access replicas are in a hub-and-spoke configuration with relay servers at the hub[7]. A relay server will synchronize with other relay servers, reducing the total number of synchronizations to $O(R * N)$, where R is the number of relay servers. The new distributed document management system from XDegrees uses this hybrid approach[10]. In general, while single-master systems are simple and easy to implement, multi-master and hybrid systems are more versatile and fault-tolerant[7].

Optimistic replication systems can be implemented as either log-transfer or contents-transfer systems. Log-transfer systems only transfer the incremental changes between replicas, while contents-transfer systems transfer the

entire changed object. Contents-transfer systems are simple yet transfer unchanged portions of local files. Log-transfer systems require file deltas to be created from remote files and applied to local files, but use the minimum amount of bandwidth.

2.3 System Objectives

The proposed distributed document management system is based on the following objectives:

- file access must be supported even during disconnected operation
- when connected, users should be able to allow others to access files in the system without any specialized client software
- if the indexing service is managed by another organization, then the system must not require that any peer be available at all times for the system to function correctly
- to maximize privacy and scalability, the indexing service should track only enough information to provide a client with the addresses of the other clients in the channel and to ensure serializability of the version numbers

To avoid the network overhead of pure peer-to-peer file sharing systems like Gnutella, which ping the network to look for peers, I chose to implement a hybrid peer-to-peer system[1]. This hybrid system has an indexing service like Napster, but unlike Napster, the indexing service does not have to track the file information on individual clients because searching across clients is unnecessary[5]. (After reconciliation, each client has all of the files being shared in the channel, so searches can be conducted on the local disk.)

There are two types of programs involved in the proposed distributed document management system: replicating client programs and an indexing service program. While a single computer must run the indexing service for the proposed system, the proof-of-concept system demonstrates that the indexing service can run on a highly-available peer. (To insure fault tolerance for the indexing service, fail-over servers could be implemented by leveraging the fact that the list of peers kept by the indexing service is in the same format as the file data exchanged between reconciling client programs.)

The indexing service has two responsibilities: to insure that any client can receive a monotonically increasing, globally unique identifier (GUID) to use as a version number, and to track and return lists of the peers that are connected to the channel. This list could be kept either with hard-state, where reconciling clients must notify the indexing service before they are disconnected from the network, or using soft-state, where the indexing service will assume that a client has been disconnected if a keep-alive message has not been received within a timeout interval. For simplicity, peers in the proof-of-concept system are not removed from the list as long as the indexing service is running. (If the indexing service is shut down, the list will be flushed.)

2.4 File Transfer Mechanism

There are two primary considerations when selecting a file transfer mechanism:

- available bandwidth in the worst, average, and best case
- installed base and usability of the file transfer client

While the Rsync system (or another system that sends only incremental changes) has the advantage of requiring the least bandwidth, it has the fewest installed clients relative to FTP and HTTP[9]. While there are easy-to-use FTP clients, the average computer user has a far higher level of familiarity with HTTP. Thus a system that uses HTTP has the advantage of allowing any web browser-equipped individual to receive a copy of a file under synchronized distribution, fulfilling the objectives of the proposed system. Further, a log-transfer system could use HTTP/1.1 to exchange only the file deltas, while the backward compatibility for HTTP/1.0 would still allow any user with only a browser to access the complete file contents.

Many of the new XML Web Services initiatives utilize the Simple Object Access Protocol (SOAP) encapsulated in HTTP requests. The disadvantage of building SOAP into the transfer mechanism is that it obviates the advantage of HTTP – users with only a web browser cannot communicate using SOAP, and thus could not access files being shared. Equally important, HTTP's header model already offers all of the extensibility required for synchronized file distribution, so SOAP is unnecessary. XML documents can be exchanged over the system, but manual reconciliation makes real-time exchange impossible.

2.5 Security Considerations

While this implementation does not offer any security precautions, security is imperative for any viable system for replication of personal files. There are three main security concerns:

- who has access to files being shared
- encrypted communication over the network
- encrypted storage of files

Note that a user who uses a utility to encrypt her files could solve all three security concerns and yet still use this proof-of-concept implementation.

3 Implementation

This proof-of-concept system is implemented using optimistic replication with manual reconciliation and contents-transfer. Version vectors are stored using GUIDs obtained from the indexing server. The indexing service can generate the GUIDs with any mechanism that returns monotonically increasing, globally unique identifiers. In this implementation, when the indexing service first begins executing, it acquires the number of seconds since 00:00:00 1/1/1970 GMT to use as the GUID seed value. It increments this value by 1 before each valid HTTP request, and then returns that to the requesting client. A production system would need to save and reload this value from a file to avoid possible duplication if the indexing service was killed and then restarted while the number of seconds used for the seed value was less than any previously used GUID values.

The system will communicate using HTTP to allow any individual with a web browser to access files under synchronization. (XDegrees uses the same approach[10].) The HTTP server is a simple single-process event-driven architecture, which is known to perform well relative to other HTTP servers on un-cached files, yet does not spawn additional threads or processes. The system uses a pull-based file transfer approach(HTTP GET), meaning that a remote peer involved in reconciliation will only send its file metadata; the local peer will not send any of its files to the remote peer.

During the reconciliation process, the reconciling peer will contact the indexing server for a new GUID and a list of all of the other peers available

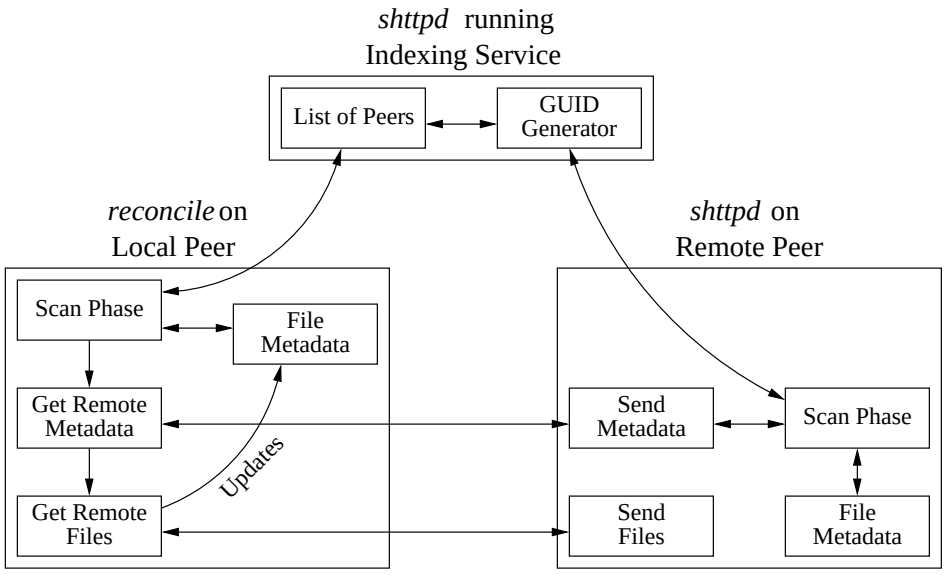

Figure 1: The indexing service for this proof-of-concept system can be run from an always-available server, as this figure shows, or from one of the peers, as shown in the next figure. *shttpd* will not run the indexing service unless it receives the -localIS command-line argument.

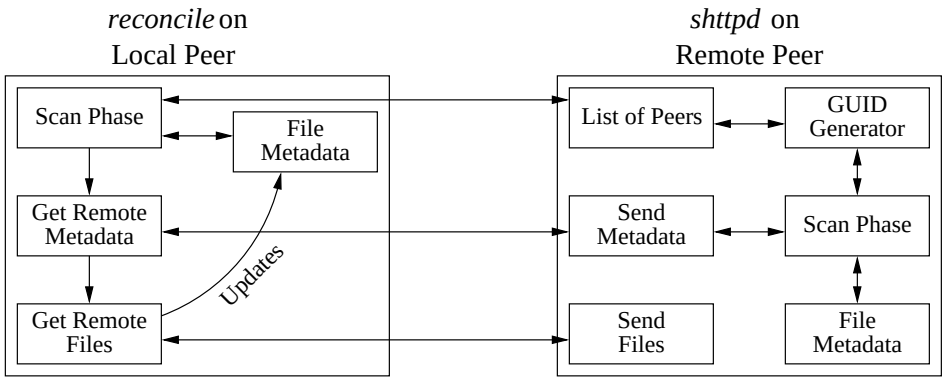

Figure 2: The indexing service is run on one of the peer computers. This approach is only advisable for a personal file system, since reconciliation cannot occur if the indexing service is not running. This is the configuration used for the demonstration session provided in Appendix A.

to synchronize with. The client application will then identify new, changed or deleted files by comparing the current directory structure of the local files against its metadata. It will update the version vector for each file with a new version, using the GUID as the new version number. (While the version numbers will not be unique across *files*, they will be unique for any given file across all *peers*.) Then this peer will contact all of the other peers in the list provided by the server. For each peer it contacts, it will request that peer's metadata. Then it will compare the version vectors in the remote peer's metadata against those in its own, identifying file updates, deletes, and additions that have occurred since the last common GUID. Remote versions will be downloaded to the local peer over HTTP.

The remote versions of files with conflicts, where the two peers have modified the file during disconnected operation, will also be downloaded. However, the local copy will first be renamed with a ".conflict" extension. The reconcile program will print a message to the user telling them to compare the two files, delete the one they don't want to keep, and then insure that the final copy does not have the ".conflict" extension. (If the user forgets to follow those instructions, then the ".conflict" file will simply be added to the replication system as would any other new file. The user can follow the instructions later, delete the file, and the ".conflict" file would then be deleted from the system.) In the case of delete conflicts, where the remote peer has deleted the file and the local peer has modified it during disconnected operation, the remote version no longer exists and thus cannot be downloaded. So the local peer needs only to print out a message noting the conflict and telling the user to determine whether to keep the file or not. For examples of both of these cases, see Appendix A.

To insure that a conflict only needs to be resolved once, the client will check each file with an apparent disordering of version numbers. If either the remote or local version vector is a subset of the other, then the client will know that the version that is the superset corresponds to the file where the conflict has been resolved. If this is the remote file, the client will download it. If it is the local file, then the remote peer will download that file when it runs the pull-based reconciliation process. To insure that all peers learn about file deletions, *tombstones* are permanently kept for deleted files.

3.1 Usage

The programs for this project are distributed as Linux x86 binaries. They have only been tested at the Yale Computer Science Department computer cluster. Source code is available upon request. There are two programs in this system, *shttpd* and *reconcile*.

3.1.1 shttpd

This program can be run in three modes: as an indexing service (-p, -q, -localIS, and -log arguments specified), as a file sharing peer (-p, -q, -remoteIS, -wwwroot, -m, and -log arguments specified), or as a combination of both (all arguments specified). If the -localIS flag is provided, then it runs the indexing service. This entails creating GUID values and tracking file sharing-only *shttpd* peers that request GUIDs. If the -remoteIS flag is provided, then it requests GUID values from a remote indexing service to update the metafile when responding to metafile requests from *reconcile* programs on peers. The calling conventions of this program are:

```
-p port                 the port to use
-q qsize                the size of the request queue
(-remoteIS IP port      use remote indexing service at specified IP, port
        | -localIS IP)  run the index service locally on the specified IP
[-wwwroot dir           directory of the files under replication
-m metafile]            filename of the metadata for files in wwwroot dir
-log file               a file for logging of requests
```

3.1.2 reconcile

This program creates metadata, contacts *shttpd* programs running as the indexing service and/or as the file sharing service on peers, and downloads updated files from *shttpd* programs on peers. The calling conventions of this program are:

```
-is IP port             IP and port of the indexing service
-m metadata             filename of the metadata for files in channeldir
[-dir channeldir]       if the metafile exists, channeldir is optional
```

4 Conclusion and Potential Applications

Replication will be increasingly important as the number of computing devices that each person interacts with increases. Optimistic replication offers a very usable system for devices which will be operated disconnected from the network or over low-bandwidth links. This proof-of-concept system demonstrates that a peer-to-peer system using optimistic replication with manual-reconciliation can maintain document consistency across multiple peers. The following applications can all utilize this distributed document management system.

4.1 Source control

Source control (without preservation of historical versions) could be implemented by adding an optimistic form of locking to this system. In addition to tracking which peers were available, the indexing service would also track the files that each user had locks on. The semantics of optimistic replication would mean that update conflicts could still occur to locked files, but the likelihood of those update conflicts would be reduced. An additional utility would have to be provided to allow users to lock and unlock files. Note that changing the implementation of the replication system from contents-transfer to log-transfer would allow preservation of historical versions.

4.2 Content distribution network

This system effectively functions as a content distribution network, where the clients propagate additions, modifications, and deletions across the network. To improve the scalability of the system, computers which only receive distributions (and thus do not make them) should not run the shttpd service, and thus would not be registered with the indexing server as peers capable of file sharing. If those read-only computers just run *reconcile* via chron, and some other means is used to prevent local changes, then no user intervention will be required because no update conflicts can occur.

4.3 Personal file system

See Appendix A for an example of a personal file system in action. For this functionality, choose a computer in the file system that you want to function

as the local indexing server. (As long as the computer clocks are reasonably-well synchronized, different computers can run the indexing service within minutes of each other.) Then run *shttpd* with the local indexing server flag (-localIS) set to the computer's IP, with -wwwroot set to the value of the local files on that machine, and with -m set to the location of the metafile.

5 Appendix A: Demonstration

The following is a transcript of a real session using the proof-of-concept distributed document management system. The left column contains the transcript from the remote peer, which is running the *shttpd* program as both an indexing service and as a file sharing peer. The right column contains the transcript from the local peer, which is running the reconciliation program *reconcile*. While this demonstration only shows synchronization occurring between two peers, *reconcile* will contact as many peers as are currently registered with the indexing service. (Because *shttpd* is running as a file serving peer, it is automatically added to the list of peers.) Initially, both peers have no files in their channel directories ('remote/' and 'local/').

The following important events occur during this session at the noted alphanumeric line values:

4 A file 'test.txt' is created on the remote peer.

5 The local peer runs *reconcile*, contacts the indexing service (running on remote), and receives a new GUID value and a list of all of the peers (only 128.36:232:27:8080) currently available. It then requests the remote peer's metadata. The remote peer responds first by initiating a scan of its local files to update its metadata. Because the remote peer is both a file sharing client and the indexing service, it can generate the GUID required for any new versions found during this filesystem scan. (If it were not running the indexing service, it would have to contact the indexing service for the GUID.) Once the scan is complete, the remote peer replies by sending its metadata to the local peer.

6 The local peer compares its metadata against the remote peer's metadata, and determines that the file 'test.txt' has been added on the remote peer. The local peer downloads 'test.txt' from the remote peer, sets the modification date on the file to match the value in the remote peer's metadata, updates its local metadata, and then prints out a confirmation message.

7, 8 The local peer deletes the new file, then recreates it with different contents. (Note that these commands are equivalent to changing the file, but are more easily demonstrated in a transcript.)

9 The local peer runs *reconcile*, and the process described for line 5 is repeated. This time, the only change has occurred locally, so the local peer simply updates its metadata. In a pull-based system, the remote peer will not be updated until it runs *reconcile* on its own files.

C, D Remote peer deletes its local copy of 'test.txt', then recreates it with new contents.

E Local peer runs *reconcile*, and the process described for line 5 is repeated. Now, the versions on the remote and local peers conflict, so the local file is renamed and the remote file is downloaded. The user is alerted to merge and delete the unwanted file manually.

P Remote peer deletes its local copy 'test.txt' but does not create a new file.

Q, R Local peer deletes its copy of 'test.txt', and renames 'test.txt.conflict' to 'test.txt', effectively modifying 'test.txt'.

S Local peer runs *reconcile*, and the process described for line 5 is repeated. The versions on the remote and local peers conflict, but the remote file has been deleted, so only the local peer's metadata is updated.

```
1 remote% shttpd                                                    1 local% reconcile
2 usage: shttpd -p port -q qsize (-remoteIS IP port | -localIS IP) 2 usage: reconcile -is IP port -m metadata [-dir channeldir]
       [-wwwroot dir] [-m metafile] -log file
3 remote% shttpd -p 8080 -q 10 -localIS 128.36.232.27               3 local% reconcile -is 128.36.232.27 8080 -m localfile -dir local
        -wwwroot remote -m remotefile -log Log &
4 remote% echo "A test file" > remote/test.txt                      4 local% ls local
                                                                    5 local% reconcile -is 128.36.232.27 8080 -m localfile -dir local
                                                                    6 Downloaded changed file [test.txt] from peer 128.36.232.27:8080
                                                                    7 local% rm -f local/test.txt
                                                                    8 local% echo "A test file - changed locally" > local/test.txt
                                                                    9 local% reconcile -is 128.36.232.27 8080 -m localfile -dir local
A remote% more remote/test.txt                                      A local% more local/test.txt
B test file                                                         B test file - changed locally
C remote% rm -f remote/test.txt                                     C
D remote% echo "A test file - changed remotely" > remote/test.txt D
                                                                    E local% reconcile -is 128.36.232.27 8080 -m localfile -dir local
                                                                    F A conflict has been detected in test.txt.  The local version
                                                                    G has been renamed test.txt.conflict and the remote version has
                                                                    H been downloaded.  To resolve the conflict, delete the file you
                                                                    I don't want to keep and name the other file test.txt.
                                                                    J Downloaded changed file [test.txt] from peer 128.36.232.27:8080
                                                                    K local% ls local
                                                                    L test.txt  test.txt.conflict
                                                                    M local% more local/test.txt
                                                                    N A test file - changed remotely
                                                                    O local% more local/test.txt.conflict
P remote% rm -f remote/test.txt                                     P A test file - changed locally
                                                                    Q local% rm -f local/test.txt
                                                                    R local% mv local/test.txt.conflict local/test.txt
                                                                    S local% reconcile -is 128.36.232.27 8080 -m localfile -dir local
                                                                    T A conflict has been detected in test.txt, but the remote
                                                                    U version was already deleted.  If you don't want to keep this
                                                                    V file, you must manually delete your local copy.
```

References

[1] Gnutella Development Home Page. http://gnutelladev.com/.

[2] R. G. Guy, J. S. Heidemann, W. Mak, T. Page Jr., G. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer Usenix Conference*, pages 63-71, Anaheim, CA, June 1990.

[3] R. G. Guy, P. Reiher, D. Ratner, M. Gunter, W. Ma, and G. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *ER'98 Workshop on Mobile Data Access*, 1998.

[4] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3, February 1992.

[5] Napster Home Page. http://www.napster.com/.

[6] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *USENIX Conference Proceedings*, pages 183-195, Boston, MA, June 1994. USENIX.

[7] Y. Saito. Consistency Management in Optimistic Replication Algorithms. Available at http://www.hpl.hp.com/personal/Yasushi_Saito/replica.pdf

[8] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experiences with Disconnected Operation in a Mobile Computing Environment. *USENIX Symposium on Mobile and Location Independent Computing*, pages 11-28, 1993.

[9] A. Tridgell. Efficient Algorithms for Sorting and Synchronization. PhD thesis. Available at http://samba.org/ tridge/phd_thesis.pdf

[10] XDegrees Home Page. http://www.xdegrees.com/.

[11] B. Yang and H. Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. In *VLDB 2001*, September, 2001. Available at http://www.dia.uniroma3.it/ vldbproc/060_561.pdf