Syncframe: A Multi-Peer Synchronization Framework

December 5, 2002

Abstract

Distributed computing and mobile computing have created significant interest in systems that provide for the coherence of user data across multiple machines. While several such systems exist, none is able to cope well with network topologies that are complex, changing, and contain more than a few nodes.

To overcome these problems, we present Syncframe, a framework for creating small-to-medium-scale peer-to-peer data coherence systems. Syncframe performs optimistic replication for hosts with intermittent connectivity. Each node in the network has a complete copy of the working set; changes in working set are automatically detected and propagated to other nodes in the network.

Syncframe differs from other synchronization systems because of its ability to synchronize multiple peers, support for intermittently connected and mobile nodes, and segregation of its synchronization and data management algorithms. Segregation allows Syncframe to synchronize arbitrary data types, e.g., file systems, address books, and databases.

We have implemented the Syncframe framework and a file system synchronizer. Preliminary results verify the feasibility and efficiency of our design.

1 Introduction

Despite the growth of a high-speed wired and wireless networks, many computer users wish to keep multiple copies of their data on multiple computers. Keeping each of these copies in synchronization[1] with the others is a growing problem. Consider these examples:

- John has a working set of 77GB of data that he is analyzing for his thesis. To protect against disk failure and fire, and to allow him to work from home and office, he wants to keep a copy of his dataset on his home computer and one on his office PC. John would like for changes at one location to quickly and automatically reflected at the other.
- Jane's sister collects public-domain music files on her home computer. Jane wants to have these music files automatically copied to her laptop whenever she visits her sister's home. Jane would then like to have these files automatically copied from her laptop to both Jane's home and work computers.
- Sam has two laptops, a computer at home, and a computer at work. On each of these computers he has a directory called "current" which contains the Microsoft Word

[1]Since the introduction of the Palm computer with its *hotsync* technology, the term *synchronization* has increasingly been used to describe a variety of data coherence activities. In this paper, we use the term *synchronization* to refer to the process of maintaining data coherence between multiple computers that may or may not be simultaneously connected to a network and/or each other.

files that he is currently working on. Sam would like to be able to sit down at any computer and start working on these Word files, and then have the files automatically copied to all of the other computers without any intervention on his part. He would like for the laptops to automatically figure out whether they have better connectivity to the home or work machines and take their updates from those computers. He wants the laptops to be able to access the files in the "current" directory even if he is on an aircraft. Finally, he would like his work computer to automatically archive every version of every Word file without any intervention on his part.

- Mimi shares an address book with four other people working on her project. Each person is occasionally connected and occasionally disconnected from the network. They all wish to be able to make additions and deletions to the data set, and have the database automatically synchronized whenever possible.

2 Related Work

There are many systems, both free and commercial, for performing data synchronization. Some of the more common and relevant systems include:

- **rdist** [2], the file distribution program that was included with Berkeley Unix BSD 4.3. The purpose of rdist is to distribute updates from a central machine(s) to client machines. It is used in a master-slave configuration, with updates only being propagated from the master to the slaves. Rdist uses a push method of updating clients. Rdist runs simultaneously on the server and the client, comparing each file on the server with the corresponding file on the client and pushing updates where the client does not match. It does not keep a metadata database. Because the server pushes updates to the clients, it has to keep track of all the clients. This can be a problem if clients are mobile or have dynamic IP addresses. Rdist is capable of updating client machines in parallel, but performance still suffers because it compares new and old versions of files byte-by-byte instead of using hashing.

- **rsync** [9], a file synchronization utility that was developed for use by the Samba file server's development team. Rsync is similar to rdist in purpose and that the program runs on both the client and the server, but updates can be initiated by either side. (Rsync can also be used entirely on one computer to synchronize two directories.) When a client connects to a server, the files are compared to files on the server and only the differences are transfered. Rsync's comparison algorithm is much more efficient than rdist's algorithm, in that it compares files in blocks and only sends over blocks that have changed. Rsync further gains over rdist from pipelining the transfer of files. Also, because clients can initiate the connections, the server does not have to keep track of the clients' address. This is beneficial to having client machines with dynamic addresses.

- **unison** [6], is a two-way file-synchronization tool for Unix and Windows. It has cross platform capability. Synchronizations happens on user request. When one machine requests synchronization, updates are performed using the rsync algorithm. A metadata database is used to detect file deletion and changes. Unison creates a new metadata database for each pair of synchronization roots. Unison is capable of being run on multiple pairs of nodes in order to form a graph (even graphs with cycles) of two-way links between many machines. It is capable of synchronizing between many machines this way, but is inefficient and has problems. The following problem may occur in a graph with a cycle: A file is changed on machine A. A synchronizes with B. A changes the file again. A synchronizes with C. An error occurs when B and C try to sync because neither has the previous version that the other thinks it has. This occurs because each update to a given file from machine A is not guaranteed to be sent to other machines before succeeding updates.

- **Microsoft Briefcase** [4], introduced with Windows 95, allows for bi-directional file synchronization between a Windows file server and a Windows desktop computer. Multiple clients can use briefcase to synchronize from a single server if the clients are arranged in a star about that server, but clients cannot synchronize with each other.

- **CVS** [1], is a concurrency control system that uses a single central repository (master-slave configuration) to ensure that all clients have access to the most recent version of files. Users manually update their files from the repository and manually commit their changes to the repository. Unlike most systems, CVS has the ability to merge conflicting versions of files line by line, instead of file by file. The design requires each client to have direct access the server in order to make or receive updates.

- **Ficus and Rumor** [8] [3] Are systems that have similar functional requirements as Syncframe, specifically performing peer-to-peer optimistic replication for systems with intermittent connectivity. The approach Ficus uses is to keep track of all changes to a file using vector timestamps. Vector timestamps follow a file from one machine to the next. Using the timestamps, Ficus is capable of determining if and when two different copies of a file diverged, and will deal with the divergence. Peers are able to synchronize with any other peers even when disconnected from all other machines on the synchronization network. Rumor is a further adoption of Ficus to mobile systems running the FreeBSD operating system, with some additional refinements.

2.1 Data Publishing Systems

Rdist and rsync can be thought of as either *data publishing systems* or *one-way synchronizers.* Specifically, both rdist and rsync are designed for replicating a directory of files from one master location to one or more slave locations. All files that are present on the master are copied to the slave locations; all files that are present on the slaves but not on the master are deleted.

Data publishing systems easily scale to multiple nodes; in these cases, a single master server can publish to multiple slaves. Large ($n > 50$) *synchronization networks*[2] can be built using multi-level distribution (e.g., a single master distributes to multiple sub-masters, each of which distribute to multiple slaves).

Although it is possible to use these programs for two-way synchronization by first replicating in one direction and then in the other, this method fails if modifications are being made simultaneously at each location.

2.2 Pairwise Synchronizers

Unison and Microsoft Briefcase are examples of *pairwise synchronizers.* Both programs are designed to synchronize a single root directory among multiple computers. (Multiple roots are synchronized through repeated invocations of the programs.) To perform pairwise synchronization, both programs need to maintain metadata for each root. A metadata database is cre-

[2]In this paper, we use the term *synchronization network* to denote a collection of computers, or *nodes*, that work together to assure that a single data set replicated on multiple machines remains consistent. Changes made in one locations should be detected and automatically propagated to the other nodes.

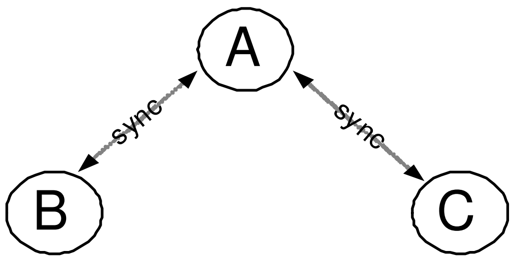

Figure 1: Using pairwise synchronizers to synchronize three nodes.

ated when a synchronization pair is first set up: the database consists of a list of every file under control of the synchronization system. Metadata is needed when a file exists on host A but not on host B: if the metadata reveals that the file was added to host A since the last synchronization, the file is copied to host B. If the metadata reveals that the file was deleted from host B since the last synchronization, the file is deleted from host A. Metadata can also be used to detect conflicts: if a file is changed on both computers between synchronization, or if a file is changed on one computer and deleted on another, these programs will report a *conflict* to the user. The user must then decide upon the appropriate action.

Pairwise synchronizers can be used to synchronize more than two hosts by setting up multiple pairwise relationships. For example, if A, B and C are all to be synchronized, one approach is to set up synchronization relationships $A \leftrightarrow B$ and $A \leftrightarrow C$, as shown in Figure 1). Such usage is common today. As we shall see, this approach is unsatisfactory for a number of reasons.

Pairwise synchronizers do not work well when nodes are mobile or when nodes or connections fail. Consider the case of three machines, HOME, WORK, and LAPTOP, that seek to maintain a shared set of synchronized files. A synchronization network can be set up using pairwise synchronizers by having both HOME and LAPTOP synchronize with WORK.

Although this approach normally works, using it can be awkward in practice. For example, in order to move a synchronized file from LAPTOP to HOME, LAPTOP needs to be synchronized with WORK, then WORK needs to be synchronized with HOME. This pairwise synchronization network cannot synchronize LAPTOP and HOME directly. Even worse, if the network connection between HOME and WORK is temporarily unavailable, there is no way to synchronize HOME and LAPTOP, even if the computers are physically connected together.

Faced with such a network topology, a user might be tempted to create a new synchronization association between HOME and LAPTOP. Alas, the synchronization tools mentioned above do not conveniently handle synchronization of directories that contain different files. Specifically, Unison will generate conflicts that must be manually resolved while Microsoft Briefcase, will require that a directory be manually copied from the server into the briefcase before it can be synchronized. In other words, there is no way to apply Briefcase to a set of files that are already resident on a workstation. This is a fundamental limitation of the way that Microsoft Briefcase is implemented.

2.3 Rumor

Rumor is an optimistically replicated file system designed for use in mobile computers. While Syncframe closely resembles Rumor, it improves upon it in several key ways:

- Syncframe replicates arbitrary objects, while Rumor only synchronizes files.
- Syncframe has higher tolerance of node and link failures
- Syncframe exploit fasts paths through the network
- Syncframe runs on FreeBSD 4.8 and MacOS X, while development on Rumor has been terminated and the system only runs on an obsolete version of FreeBSD. (Although it might be possible to port Rumor to a current system.)

3 Design

Syncframe is designed to efficiently maintain a replicated object set across multiple computers (or nodes). Syncframe's most important features are its support for:

- Arbitrary topologies
- Arbitrary objects
- Mobile nodes

Syncframe also provides the following features:

- Use efficient network paths (Fast Paths)
- No kernel modifications
- OS agnostic
- Archiving
- Firewall friendly
- On-Demand or Automatic Synchronization

3.1 Architecture Overview

All Syncframe nodes are created equal. Nodes are linked together using peering relationships. All links are bi-directional, although sometimes links are only *initiated* by one peer. (For example, if one peer is mobile and the other peer is not, only the mobile peer will initiate connections, but once a connection established, data may move in either direction.) The topology is determined before Syncframe is started, although it is easy to add and remove nodes. For example, Syncframe could be used to synchronize an office machine, a home machine, and a laptop, and then a user could later add another laptop.

Each Syncframe node periodically detects changes to objects in its object store. These changes are encapsulated in a self-contained update and propagates updates to all of the node's peers. Each of these tasks is done asynchronously; this allows updates to be created and stored while a node is disconnected from the network. If conflicts are detected upon propagation, the object is forked and the user notified.

3.2 Feature Comparison

3.2.1 Arbitrary Topologies

Syncframe allows nodes to be arranged in arbitrary topologies, e.g., trees, chains, and meshes. Such flexibility makes Syncframe useful in a wide range of usage, redundancy, and scalability scenarios. For example, if a small number of nodes are updating files for a large audience, a hierarchical tree is appropriate because it scales well. If a small number of users are all updating files and it is important to tolerate node and link failures, a complete mesh is appropriate.

3.2.2 Arbitrary Objects

While most synchronization systems are designed to handle a specific data type, such as a file system or database, Syncframe can be used to rapidly build synchronizers for arbitrary data types (objects). We have built such a synchronizer for file systems; it would require only a small amount of work to build an address book or email repository synchronizer.

3.2.3 Mobile Nodes

Nodes may become disconnected or connect to the network at different places (e.g. home and office). Syncframe allows nodes to continue to work on their local repository while disconnected.

3.2.4 Fast Paths

If there are multiple paths through which an update could propagate through the network, Syncframe will dynamically use the fastest one.

3.2.5 No Kernel Modification

Syncframe runs entirely in user mode, making it highly portable and easy to install.

3.2.6 OS Agnostic

We have cautiously avoided using OS-specific functionality in our design and implementation. For example, we do not assume that file names are case sensitive or that files have users and groups.

3.2.7 Archiving

Every node in the synchronization network is guaranteed to see every synchronization update. As a result, any node may be configured as an *archiving node* which stories copies of files to a permanent store, in addition to storing files in the file system.

3.2.8 Firewall Friendly

A Syncframe node may be operating behind a firewall, prohibiting it from receiving inbound connections. We support such operation by allowing a single TCP connection to be used for both incoming and outgoing Syncframe links.

3.2.9 On-Demand or Automatic Synchronization

Synchronization depends on three steps: the creation of synchronization updates, the propagation of these updates to other nodes in the network, and application of the updates. Each step can be done on a periodic basis or can be initiated at user request. Some people may like to have Syncframe always running in the background, whereas others may wish to have Syncframe only collect updates when requested.

3.2.10 Storage Requirements

Because Syncframe propagates updates which need to be created and stored on each node before they are transmitted to other nodes, Syncframe doubles the amount of storage required for the data under synchronization control.

4 Implementation

We have implemented the Syncframe framework and a single Syncframe module, Filesync, which synchronizes file systems. Unless otherwise specified, Syncframe refers to the framework alone.

Syncframe comprises the following modules:

1. **The configuration file**, which specifies the synchronization parameters.
2. **The metadata database**, which is used to track changes made to objects currently under synchronization.
3. **The Hunter**, which iterates through all of the objects under synchronization control and creates updates associated with object creation, modification, and deletion.
4. **The Update Database**, which stores all updates that need to be sent out to the node's peers.
5. **The Publisher**, which publishes updates to the node's synchronization peers.
6. **The Committer**, which applies updates that are received from other synchronization peers to the system.
7. **The Cleaner**, which erases updates on this node after they have been either sent to all other peers or seen by those peers, and committed.

In the Filesync system objects are files that are stored in a single *synchronization root directory* and subdirectories of that directories. Each computer that maintains a synchronization directory is called a *node.* We call the entire set of nodes a *synchronization community.* Each node communicates with a subset of the community, each member of which we call a *synchronization peer.*

The Syncframe Hunter and Committer are implemented as abstract C++ class libraries; these libraries are subclassed to create the FileHunter and FileCommitter used by the Filesync system. Cleaner is also a C++ class, although it does

Feature	Syncframe	Rumor	Pairwise	Publishers
Arbitrary Topologies	yes	partial	no	no
Arbitrary Objects	yes	no	no	no
Mobile Nodes	yes	yes	partial	partial
Fast Paths	yes	no	no	no
No Kernel Mods	yes	yes	varies	varies
OS Agnostic	yes	partial	varies	varies
Archiving	yes	no	no	no
Firewall friendly	yes	no	no	no

Table 1: Comparison of Syncframe features with other systems

not need to subclassed for different Syncframe applications as the garbage collection logic is independent of the type of objects being synchronized. The Publisher is implemented as a Java application; like the Cleaner, this application has no dependencies on the particular type of object being synchronized.

Under normal circumstances, each node has a TCP connection open to all of its peers. Updates are automatically sent to peers as they created by the Hunter or received from other peers. If a peer is unavailable, either because the peer is down or because of a network partition, updates are kept in the update database until connectivity is reestablished.

The peers for each synchronization node must be manually configured in a configuration file that appears on each peer. Nodes in the network do not need to be explicitly configured: they are discovered as the network runs.

4.1 Configuration File

Each synchronization node must be preconfigured with the node's name and its peers' names. In our implementation changes to this configuration require a Syncframe restart. However, this is not a fundamental design limitation. It is important to note that a node does not need to know about all other nodes, only its peers.

Each synchronization node has a name that must be unique within the community. Nodes can be located at fixed IP addresses (or IP host names), in which case they can either receive TCP connections from their peers or initiate connections to their peers. Alternatively, nodes can have dynamic addresses or hostnames (as is the case with mobile nodes). Because the IP addresses of mobile nodes are not known, mobile nodes cannot receive connections from their peers, they can only open new connections.

No special initialization phase is required. When Syncframe starts for the first time, all objects found in the object store are considered "new" objects and published to the node's peers.

4.2 Objects, Metadata and the Metadata Database

An object comprises data and metadata. The data is an ordered sequence of bytes. The metadata is an extensible set of name/value pairs associated with each object.

Each Syncframe node has a metadata database that is used to store the metadata for each object within the synchronization root. We implement this database with a persistent B-Tree using the Sleepycat Berkeley DB package [7]. The object name is used as the key of the B-Tree.

Syncframe requires the following metadata for each object:

- Object name
- Length
- Creation time

- Modification time

Filesync adds the following metadata:

- File username
- File groupname
- File permissions
- File type (e.g. regular file or symbolic link)

Filesync uses filenames as object names and stores data in files on the file system. An address book synchronizer might used a unique object identifier as an object name and store the address book entries in a special database.

4.3 Hunter

Hunter is a program that runs on the Syncframe node. Hunter scans, or hunts, for changes to stored objects. A change is detected when an object's metadata does not match the metadata stored in the metadata database. Hunter can be run periodically or on-demand, as desired by the user.

When a change is detected, Hunter creates one or more updates that encapsulate the change and adds the update to the update database. These updates will then be distributed to each of the node's peers by the Syncframe Publisher, described below.

Hunter's logic is straightforward:

1. At startup, Hunter makes a list of every object currently in the metadata database.
2. Next, Hunter iterates through every object in the object store. In the case of the File-Hunter, iteration uses a recursive directory list.
3. For each object in the object store:
   (a) If the object does not appear in the metadata database, Hunter creates a **WRITE** update that contains the file name, modification times, and file contents. Large files may generate multiple updates.
   (b) If the object appears in the metadata database, the object's current metadata is compared with the metadata stored in the database.
      i. If the two metadatas are same, no action is performed.
      ii. If the file's metadata does not match the metadata in the database, Hunter creates a **WRITE** update that sets the object's data and metadata to reflect the data in the object store. These updates are designed to be as small as possible: if the object's data has changed, the update contains the new data and the new metadata; if only the metadata has changed, the update contains only the new metadata.
      iii. In either case, the object is removed from the list that Hunter created in the first step.
4. Finally, for every object that was in the database but was not found in the directory, Hunter creates a **DELETE** update.

Each update contains the name of the Syncframe node on which it was created, a monotonically increasing update number, an object name, the object's metadata before and after the update would be applied, and, optionally, update-specific data.

Note that only two updates - WRITE and DELETE - are sufficient for synchronization of arbitrary objects.

Updates are uniquely identified (named) by the originating node ID and the update number. Because of this unique naming, update receipt and write-to-disk are idempotent operations. Note, however, that an update may be "committed" only once.

Because the Hunter runs periodically, it may miss some file updates. For example, if a file foo is created and then deleted between runs, other

WRITE *node number* M_0 M_1 *data*	Writes *data* into the object specified by metadata M_0. At the end of the update, the object's metadata is set to be M_1. In Filesync, WRITE updates can be used to write file contents, change file ownerships, truncate files, and even rename files.
DELETE *node num* M_0	Deletes the object specified by M_0.
TWRITE *node number* M_0 M_1	This is a special WRITE update that is tagged so that it only needs to be sent from one node to a second specified peer, rather than be distributed throughout the entire synchronization network. The TAGGED WRITE update is used when nodes wish to join the synchronization network. [note: Joining is not currently implemented.]
SEEN *node node_list update_list*	Informs other nodes that this node has seen the updates in *update_list* from the nodes in *node_list*. The SEEN update is used by the Cleaner to determine when updates on a node can be garbage collected.

Table 2: Updates supported by the Syncframe system.

nodes will never learn about foo. This is not a problem, however, since we are aiming to synchronize nodes, not guarantee that all changes are propagated to all nodes.

4.4 Update Database

In the present implementation each update is stored as a single file. While this is useful for debugging purposes and keeps our code simple, it does exact a performance penalty. A more efficient implementation would store updates in a database.

4.5 Publisher

Publisher is a program that keeps a node in sync with its community. It interfaces with the Hunter and Committer programs through the update database; in our initial implementation, this database is implemented as a directory of files, each file representing a single update.

The nodes with which a given node communicates are termed "peers" and are enumerated in each node's configuration file. To accomplish its synchronization tasks, Publisher sends updates to its peers and receives updates from other nodes.

Publisher attempts to keep a network connection (socket) open to each of its peers at all times. Communication over a socket is bi-directional, which is useful when a node cannot receive incoming connections (which is often the case for nodes behind firewalls or for mobile nodes that do not have a fixed IP address or hostname).

Publisher uses the Syncframe protocol (SFP) to communicate with Publishers running on other nodes. SFP is simple (about 10 messages and 10 states) and efficient (it keeps the pipe full and does not transfer updates to a node if that node has already received the update in question.) The protocol's principle drawback is that it does not support simultaneous bi-directional transfers.

The Syncframe protocol runs on on top of TCP or, optionally, SSL. Authorization is accomplished by means of a shared secret that is stored in the configuration file.

While it is essential for correctness that every node in the community sees every update, it is equally essential for efficiency that each node see each update only once. Publisher accomplishes this by keeping track of which peers have seen which updates. When all peers have seen an update, that update is deleted by the Cleaner (described below). Publisher never sends updates to a peer that the peer has already seen.

The order in which Publisher sends out updates is important for correctness. Specifically, for all updates originating at a given node, the Publisher sends out the updates in the order that it has received them. There is no notion of a serialized order for the whole network. If a conflict occurs as a result of this policy, it is detected and scheduled for manual correction.

The Syncframe Protocol is described in more detail in Appendix A.

4.6 Committer

Committer is responsible for committing updates received from Publisher. The program periodically polls the update database for new updates, checks for conflicts, and applies updates.

Committer detects conflicts by examining the updates "old metadata" and the metadata of the object on disk. If the metadatas are not equal there is a conflict because every node always sends out updates in the order that they are received. When a conflict occurs the object is then forked and the user altered.[3]

4.7 Cleaner

After an update has been sent to all of the node's peers by the Publisher and applied to the local object store by the Committer, it is deleted by the Cleaner. An archiving node could instead choose to save updates in a long-term persistent store.

5 Security

The current Syncframe implementation provides basic authentication and encrypted network communication. We have not rigorously tested the security system and have identified a man in the middle vulnerability; however, we believe the system is adequate for research use. We have not implemented an authorization or access control system.

5.1 Authentication

We will refer to the peer initiating a network connection as the client and the peer accepting the network connection as the server. Each configuration file stores the community name and the community key. In our model the client and the server must prove to each other that they know the secret community key without revealing the key. This exchange occurs through a challenge/response process issued from both sides.

An encryption cipher is setup using DES and a key derived from the community key using MD5. Each side then randomly generates a 64-bit number, encrypts it, and sends it to the other as a challenge. The server must reply to the client's challenge with the client's random number minus one encrypted using the same key. The client must reply to the server's challenge with the server's random number plus one in the same manner. If both sides accept the other's responses, then authentication has completed and the protocol can continue.

This challenge/response method prevents any eavesdroppers from obtaining the key directly by listening to this conversation over an unencrypted link. It is also very difficult to determine the appropriate response to the challenge without knowing the symmetric secret key used in the encryption or having already seen the correct response to that challenge. Unfortunately this means that an eavesdropper who is lucky or has a very large storage space can eventually authenticate with one of our community nodes without knowing the community key.

[3]Note that our current Syncframe implementation does not fork the object, but only alerts the user. We did not have the time to implement and test the code for conflict management.

5.2 Encryption

By running the Syncframe protocol on top of the Secure Socket Layer with Diffie Hellman anonymous key exchange, we gain protection from eavesdroppers. The synchronized data that we transport maintains its privacy, and eavesdroppers cannot listen to the challenge/response in order to record valid responses to challenges. Our current implementation of Syncframe has this level of security available.

5.3 Vulnerabilities

The system is vulnerable to an IPspoofing man in the middle attack. By intercepting a client's connection to the server and then connecting to that server itself, the man in the middle can allow the two to authenticate each other through his connection to each side. Once authentication has been performed, he can listen to their updates and issue fake updates. The current implementation of Syncframe has no defense against this, but we feel that IPspoofing is sufficiently difficult that we can accept the risk of a man in the middle attack until such a time that we can implement an improved security layer.

There are a few options for defending against the man in the middle attack. Authentication can be moved to the SSL layer by using server and client certificates. This would ensure that the SSL connection would be formed between two hosts with properly signed public keys. The other option would be to provide a layer similar to SSL that performed the key exchange under the protection of the secret keys. Both of these methods would ensure that only the intended client and server would know the encryption and signature keys for the session.

6 Performance

To test the performance of the Filesync system, we created a test directory with 100 files of 1 megabyte each. Table 3 indicates the time for Hunter to scan all 100 files when all files were new, after 50% of the files had been modified, and after 0% of the files have been modified. For comparison, we indicate the time for the Unix "tar" program to read through the contents of these files.

Hunter Benchmark	Clock	User	System
all new files	17.01	5.943	4.833
50% modified	8.931	4.346	2.991
0% modified	4.153	2.936	1.118
tar	9.142	0.076	3.803

Table 3: Hunter time tests: time for Hunter to process 100 files of 1 megabyte each on a 400MHz Apple PowerBook G4 with 384M of RAM. Times reported are clock time, user CPU time, and system CPU time. Each test was repeated 10 times and an average presented. All times in seconds.

Hunter always takes more user time than tar because it needs to calculate the MD5 sums for every file that it processes. However, hunter takes noticeably less clock time time than tar when 0% of the files have been modified, even though it needs to compute the MD5 codes for each file: this is because Hunter is only reading the files, not writing transactions. When Hunter *does* need to write transactions, it takes more than twice the lock time as tar, even though the amount of data that it is writing is comparable. This indicates that it might be possible to tune Hunter's output routines to improve overall performance.

To test end-to-end performance, we set up the synchronization network described by Figure 2 and started the system with empty root and scratch directories. Each system was run with a SYNCFRAME_SCAN set at 2 seconds, meaning that Hunter ran every 2 seconds and Publisher scanned for new update every 2 seconds.

We then created a single small file on the computer named *ni*. Hunter running on *ni* detected this file and created a update. That update was sent to *r2*, which deposited the update into the update database. The update was sent to *lap1*

Committer Benchmark	Clock	User	System
all new files	5.326	2.682	1.571
50% modified	5.696	1.857	2.553
0% modified	0.022	0.005	0.011
untar	10.508	0.113	3.538

Table 4: Committer time tests: time for Committer to process transactions created by Hunter for tests in Table 3. Times reported are clock time, user CPU time, and system CPU time. Each test was repeated 10 times and an average presented. All times in seconds.

and *v*; After *lap1* received the update, it also attempted to send the update to *v*, but *v* refused the update, telling *lap1* that it already had the update. The update was then sent by *v* to the computer affectionately known as *pain*.

Each of these computers, in turn, executed the update and created a SEEN update. These SEEN records are then propagated to every other host in the network. Total time to send the data through the network was negligible; the longest delays (2–4 seconds) were delays caused by the *sleep* instructions in the Committer and Publisher components. This is clearly an opportunity to employ the techniques in libasync[5] to improve performance.

We then copied 20 files ranging in size from 230 bytes to 18,428 bytes into a directory under synchronization control on the *ni* computer. Hunter running on *ni* created the transactions 1 second after the files were created. Unfortunately, at this point the Sun JDK 1.4.1 runtime generated a runtime error and requested that we reported the bug to Sun with Error ID 4F530E43505002E6.

The Java errors that we experienced on this project were rarely repeatable. Re-running the system and repeating the test, the batch of 20 files appeared within seconds on all of the computers within the test network. The time it took for the files to appear was entirely dependent upon the frequency that Hunter and Committer scanned for updates and the speed of the network connection.

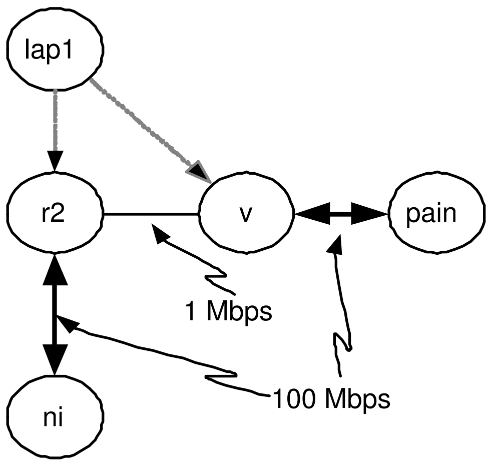

Figure 2: A simple synchronization network set up for performance measurements. The computers ni, r2, v and pain are all workstations running the FreeBSD operating system; r2 and ni are connected with a 100 Mbps network, as are the computers v and pain. The computers lap1 and lap2 are laptop computers that have intermittent connections to r2 and v. All synchronization was done on port 9011 using TCP/IP with encryption disabled.

7 Subtleties

This section of the paper addresses several subtle issues with Syncframe.

7.1 What is Correct Synchronization?

Correctness means that:

- all updates are eventually seen by all nodes
- every conflict is detected by at least one node

Note that we do not guarantee, and that it not essential for correctness, that:

- there is a global ordering of updates
- every change to an object will be captured by an update

In the event of a permanent partition, the system is not guaranteed to work correctly.

7.2 Allowed Topologies

A properly formed Syncframe community is a strongly-connected directed graph in which vertices are hosts and edges are peer relationships (defined in the configuration file). A strongly connected graph is one in which for all pairs of vertices, both vertices are reachable from the other.

7.3 Conflict Detection and Resolution

As described above, when conflicting updates are received the object is forked. The user is alerted via a message in the log and an error message to standard error. A conflict will be detected on at least one node, at which point further updates to the object are stalled until the conflict is resolved. Users resolve conflicts by deleting one object and renaming the correct one to be the original name (after possibly manually merging conflicting updates). Conflict resolution is in turn treated as an update and propagated to the network.

Note that if two identical writes to the same object are done on two different hosts, then both writes will result in the same metadata M_1 being created on each host. When the committer goes to commit the second write, it will discover that the object's metadata has already been updated from $M_0 \rightarrow M_1$ and the update will be dropped, as it has already taken place.

Put another way, bundling both before and after metadata with each transaction makes them idempotent, which make the system robust with respect to certain kinds of conflicts.

7.4 Limits to Network Size

We have not done enough testing to know where our bottlenecks are. However, since we allow arbitrary network topologies it should be possible to build fairly large networks. For example, a tree (in which updates flowed from the root down) of five levels with a branching factor of 10 would not be unreasonable since it would only take 4 hops for any update to reach a leaf and every node would only have to have 10 network connections open.

7.5 Joining the Network

To add a node to an existing network at least one existing configuration file will need to be modified to add the new node as a peer (to meet the strongly connected requirement). The new node can designate any existing nodes as peers.

Any node that designates the new node as a peer will create special write transactions to get the new node synchronized initially.

7.6 The SEEN update

After each run of the Committer, a Syncframe node creates a SEEN update that is sent to its peers, which in turn sent the update to their peers. Alas, in our design the SEEN update needs to be sent to *every node*. The seen update is used by peers

7.7 Object Renaming

Renames are currently implemented as a pair of updates: the old object name is removed with a DELETE update and a new object with a new name is created with a WRITE update. While this is correct, it is inefficient; to improve efficiency we could implement a separate RENAME transaction.

7.8 Permanent Partitions

If a network is permanently partitioned, then the nodes that have peers across the partition will never clean their updates. If the configuration file is edited to remove the peers that are across the partition, the cleaner will then automatically remove the updates.

7.9 Fast Paths

Because each Syncframe peer attempts to send all of its updates to all of its peers — and because updates are not sent to peers that already have those updates — updates implicitly finds fast paths through a synchronization network.

For example, if nodes A, B and C are connected in a triangle, with fast links between A-B and B-C, but a slow link between A and C, updates will flow from $A \rightarrow B$ and then $B \rightarrow C$ before updates can flow from $A \rightarrow C$

8 Detecting Object Changes

Because we wanted to make Syncframe easy to port and debug, we decided early on to keep all modules entirely in user mode. One important effect of this constraint is that we are unable to listen to disk events, such as writes or deletes, that affect files under synchronization. We thus have to do periodic polling via the Hunter to detect changes.

If propagation latency and efficiency became more important, it would be possible to create a "disk event listener" that created updates as they were written to disk. One way to do this would be to patch the Unix file system.

9 Lessons Learned

We learned the following while working on this project.

- **Starting update numbers at 1 may be a mistake.** Our update numbers start at the first ordinal and monotonically increase after that. This produced complications during development and testing, as we frequently needed to restart the ordinal at one. Whenever this happened, we needed to go throughout the network and remove the SEEN metadata files. A better design might incorporate a host-based timestamp in the update number.

- **Separating updating algorithms from data representation was a powerful tool.** The design of Hunter and Committer was significantly simplified by our separation of the abstract algorithms that operated on "objects" from the specific algorithms that dealt with the details of Unix files. Cleaner and Publisher contained no logic having to deal with files at all, simplifying them as well.

- **Separating host-based logic from the network-based logic was a powerful tool** The separation of the Publisher from the rest of the system made it easy to test the individual modules before we put the whole system together.

- **Using the file system as a database was a good debugging tool.** Likewise, our storing of updates in individual files, rather than putting them in a database, eased development. In a future implementation, we will have the decision to store updates in individual files or in a B-Tree as a compile-time option.

- **C++ enabled modularity.** The design of Hunter, Committer and Cleaner as separate C++ classes made it easy to develop them in stand-alone executables, but then it was easy to combine them into a single executable for deployment.

- **It is important to think about logging early on.**

  Good logging makes a system easier to develop, debug, and test. Unfortunately, we didn't specify logging behavior in our initial design, and different team members decided that different information was worthy (or not worthy) of being logged. This created considerable problems later on.

- **It is important to design and implement status commands early on.** Our original design called for the creation of status commands that could be used to rapidly determine the current status of the system. For example, it would be nice to see how many transactions are pending for each peer, whether there is currently a connection with another peer, how many transactions have been sent over the connection, and so on. Unfortunately, for some reason these commands were never developed. They would have been very useful for setup and testing.

- **If you are on a deadline, don't force people to use tools that they are not comfortable using.** One of our project members insisted writing this paper using LaTeX, even though nobody else on the team had experience with the program. This proved to be a mistake.

10 Future Work

10.1 Optimizations

The FileHunter and Committer programs can be modified so that the "object" is actually a 64kilobyte page of a file, rather than an entire file. This will allow the system to perform incremental updates to a large file when only a few bytes change. It will also have the side effect of placing a maximum size limit on updates of just over 64K.

10.2 Enhancements to the Syncframe Server

To support multiple people using Syncframe on the same host, the Syncframe server could use the community that is sent across at the start of the TCP/IP connection to determine which configuration should be used.

10.3 Extending Syncframe to Databases

We have asserted that Syncframe will work with objects other than files, but we haven't proven this. We are considering continuing this project after the termination of the course and creating an address book synchronizer for address books stored in XML.

References

[1] Per Cederqvist. Version management with cvs, 2001.
URL `http://www.cvshome.org/`

[2] Michael A. Cooper. Overhauling rdist for the '90s. In *LISA VI*. October 1992.
URL `http://magnicomp.com/rdist/`

[3] Richard Guy, Peter Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. 1998.
URL `http://lever.cs.ucla.edu/-rumor98/er98wmda.ps`

[4] Susan Hotton. Keep your files in order with my briefcase. *Using Windows 98*, 1999.
URL `http://www.microsoft.com/windows98/-usingwindows/work/articles/906Jun/-briefcase.asp`

[5] David Mazieres, Frank Dabek, Eric Peterson, and Thomer M. Gil. Using libasync.
URL `http://www.pdos.lcs.mit.edu/6.824/doc/libasync.`

[6] Benjamin C. Pierce, 2002.
URL `http://www.cis.upenn.edu/-~bcpierce/unison/`

[7] Sleepycat Software. Berkeley db online documentation, 2002.
URL `http://www.sleepycat.com/`

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

[9] Andrew Tridgell and Martin Pool. rsync – faster, flexible replacement for rcp, 2002. URL http://dp.samba.org/rsync/

A Syncframe Protocol (SFP) 1.1

The Syncframe protocol is a text-based protocol that is used to transfer Syncframe updates between a pair of hosts. The protocol allows a single connection to send updates in both directions.

The protocol runs on a TCP/IP port which can be configured for an entire synchronization community or for any pair of peers. If two users are using Syncframe on a single host, they must each have a dedicated TCP/IP port number for their synchronization activities.

The protocol assumes that only one conversation taking place on a given socket at a time. It handles the cases in which either or both peers have updates that they wish to send. (This is necessary because one Syncframe peer may be behind a firewall or have a changing IP address and thus be unable to receive incoming TCP/IP connections.) In the case that both hosts have updates to send, the protocol arbitrates between the hosts, effectively giving each host an equal chance of sending or receiving updates — without requiring that either party maintain state regarding whose turn it is to send updates.

Performance of the protocol remains to be studied. An advantage that this protocol has is that it allows for the batch transfer of many small objects. The protocol's primary shortcoming is that it does not allow for bi-directional data transfer, so in the case where both parties have updates to send, the protocol is at most 50% efficient.

A.1 Example Conversation

An example SFP conversation illustrates the typical operation of the protocol, while a formal specification using states and state transitions follows.

Nodes A and B have established a connection that is authorized and have agreed on a protocol version (this version). A wants to give B updates from node C numbered 100-200. B has already received updates 100-149 (perhaps from a previous session or from C directly).

The following messages are exchanged:

A→B	`"STARTSEND 3823"` *3823 is a random number - see below*
B→A	`"100 CONTINUE"`
A→B	`"PROPOSE C 100 200"`
B→A	`"PROPOSE C 150 200"`
A→B	*Data C 150-200*
B→A	`"202 CONFIRM C 150 200"`

Note: the Syncframe protocol uses the term "object" to refer to an update that is being sent between peers, rather than an object that is under synchronization control. We apologize for this confusion.

A.2 Protocol Details

The Syncframe protocol relies on the sending of ASCII text messages between the two participants to query for the availability of updates and to initiate their transfer. Updates are transferred simply by sending them down the TCP/IP connection; the receiving side can examine the incoming updates and determine the end of each one.

Syncframe messages are not case sensitive. Each message is terminated by a single newline.

When one Syncframe peer connects to another, both peers are in the INITIAL state. From this state either peer can enter the SENDING state by sending a STARTSEND x command. If the other side acknowledges this with a CONTINUE acknowledgment, then a delegation has been made and one peer will send while the other receives. If both peers send a STARTSEND x command before the other acknowledges, then the peer that sent a larger randomly-chosen integer x will be the sender and the other will be the receiver.

Once a delegation has been made, the sending node will send a *PROPOSAL*, which is a proposed set of updates that it can send. The receiving node will then acknowledge this proposal with a list of which updates it wishes to receive.

```
STARTSEND
PROPOSE
100 CONTINUE
200 OK
201 NO OBJECTS NEEDED
300 ABORT
400 BAD REQUEST
401 NOT ALL OBJECTS RECEIVED
402 UNEXPECTED OBJECT
500 BAD RESPONSE
```

Table 7: Syncframe Protocol Messages

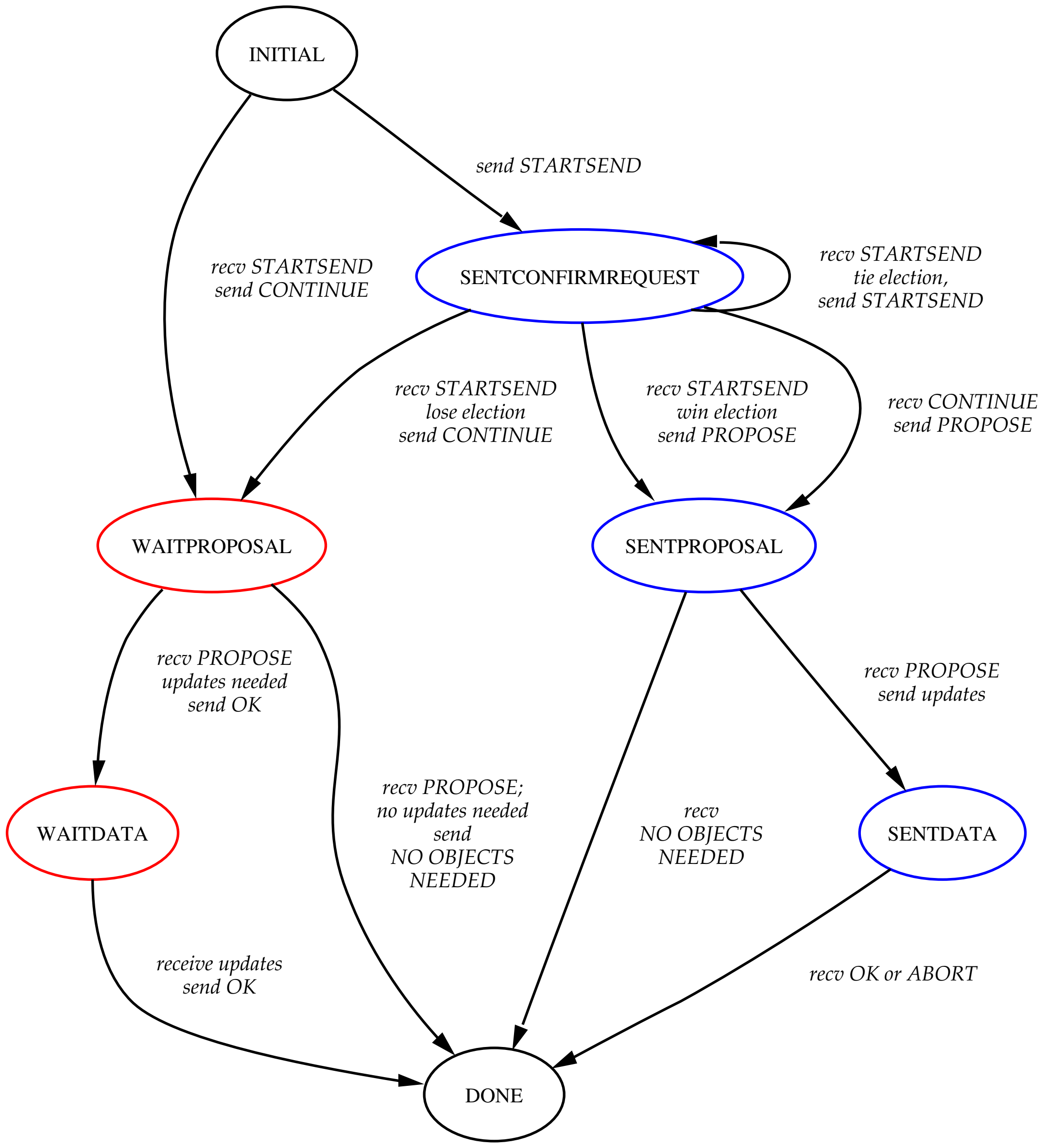
INITIAL
send STARTSEND
recv STARTSEND send CONTINUE
SENTCONFIRMREQUEST
recv STARTSEND tie election, send STARTSEND
recv STARTSEND lose election send CONTINUE
recv STARTSEND win election send PROPOSE
recv CONTINUE send PROPOSE
WAITPROPOSAL
SENTPROPOSAL
recv PROPOSE updates needed send OK
recv PROPOSE send updates
recv PROPOSE; no updates needed send NO OBJECTS NEEDED
recv NO OBJECTS NEEDED
WAITDATA
SENTDATA
receive updates send OK
recv OK or ABORT
DONE

Figure 3: Syncframe protocol state transition diagram with error states omitted

	START STATE - END STATE Input: Text required for transition Output: Text output after transition
R1	INITIAL - WAITPROPOSAL Input: “STARTSEND x” where x is a randomly-chosen integer Output: “100 CONTINUE”
R2	INITIAL - ERROR Input: Not “STARTSEND x” where x is integer Output: “400 BAD REQUEST”
R3	WAITPROPOSAL - WAITDATA Input: “PROPOSE node x y” where node is a node in community, x and y integers, $0 < x \leq y <$ max Output: “PROPOSE node a b” where node is above, $a \ldots b$ is the range receiver wants, $a \geq x, b \leq y, a \leq b$ *Note: Sender MUST send the range receiver wants*
R4	WAITPROPOSAL - DONE Input: “PROPOSE node x y” where node is a node in community, x and y integers, $0 < x \leq y <$ max Output: “201 NO OBJECTS NEEDED” *Note: Receiver already has all the updates (we anticipate this to be a common case)*
R5	WAITPROPOSAL - ERROR Input: Any message not meeting input of R3 or R4 Output: “400 BAD REQUEST”
R6	WAITDATA - DONE Input: All requested updates received or abort Output: “200 DONE” or “300 ABORT” *Note: A “300 ABORT” should be sent when the protocol wants to abort; data received after this point should be ignored*
R7	WAITDATA - ERROR Input: Not all requested updates received or unexpected update Output: “401 NOT ALL OBJECTS RECEIVED” Output: “402 UNEXPECTED OBJECT” *Note: Ignore unwanted updates*

Table 5: Receiving state transition table for Syncframe Protocol

	START STATE - END STATE Input: Text required for transition Output: Text output after transition
S1	INITIAL - SENTCONFIRMREQUEST Input: Output: "STARTSEND x" where x is randomly generated integer
S2	SENTCONFIRMREQUEST - SENTPROPOSAL Input: "100 CONTINUE" or "STARTSEND j" where j is an integer greater than i (above) Output: "PROPOSE node_s x y" where node_s is a node in community, x and y integers, the sender wants to send updates from node x through y (inclusive) *Note: If receives STARTSEND, both nodes are trying to act as senders*
S3	SENTCONFIRMREQUEST - WAITPROPOSAL Input: "STARTSEND j" where j is an integer greater than i (above) Output: "100 CONTINUE" *Note: Sender now becomes a receiver*
S4	SENTCONFIRMREQUEST - SENTCONFIRMREQUEST Input: "STARTSEND j" where j is an integer less than i (above) Output: *nothing*
S4	SENTCONFIRMREQUEST - SENTCONFIRMREQUEST Input: "STARTSEND j" where j is an integer equal to i (above) Output: "STARTSEND i" where i is a newly chosen random number Todo: We may want to but a counter on how many of these transitions we have in case sender and receiver have the same random number generator or have some other problem (avoids infinite loops)
S5	SENTCONFIRMREQUEST - ERROR Input: Not one of the above Output: "500 BAD RESPONSE"
S6	SENTPROPOSAL - SENTDATA Input: "PROPOSE node_r a b" where $a < b, a \geq x, b \leq y$, $\text{node}_r = \text{node}_s$ Output: Send updates [a-b] for node_r
S7	SENTPROPOSAL - DONE Input: "201 NO OBJECTS NEEDED" Output: *Note: The client already had all the proposed updates*
S8	SENTPROPOSAL - ERROR Input: Invalid proposal Output: "500 BAD RESPONSE"
S9	SENTDATA - DONE Input: "200 OK" or "300 ABORT" Output: *Note: In case of a "300 ABORT" the sender should immediately stop sending*
S10	SENTDATA - ERROR Input: Not "200 OK" or "300 ABORT" Output: "500 BAD RESPONSE"

Table 6: Sending state transition table for Syncframe Protocol. States involved in sending updates are in red; states involved in receiving updates are in blue.

```
#
# This is a sample Syncframe configuration file.
# Any line of the file may begin with a hash mark to indicate a comment.
# Blank lines are ignored too.
#
# First we define our community. The community identifies this synchornization network.
# Communities are case-sensetive and may consist of any character other than = and \n.
#
COMMUNITY=Boston3


# Next we define our encryption key. In the initial system, we'll use this for
# access control. They "key" is actually a passphrase that is then hashed with MD5 to create
# a true 128-bit key
#
KEY=gandalf

# Syncframe uses a roll-our-own TCP protocol that runs on a port that we specify.
# Clients and servers use the same port. If multiple people are using Syncframe on a single
# computer, they must choose their own ports.  Different peers in the same community can use
# different ports, but ports other than the standard port need to be clearly indicated
# in the config file
PORT=6533

# This section defines the synchronization root for the COMMUNITY on this host.
# ~ is interperted as the user's $HOME.
# If $HOME is not set, using ~ generates an error
SYNCROOT=~/synctest

# Every Syncframe peer requires a unique ID. You can make this your hostname, but
# beware that DHCP makes hosts change thier hostnames. The name doesn't mean anything, really.
MYNAME=syncmaster1

# Hey, here's who we talk to. It is a set of:
#  *  unique IDs,
#  *  internet addresses, either hostnames or IP addresses.
#  * optional ports (if no port is specified, Syncframe uses the PORT specified above.)
PEER=K1:k1.vineyard.net
PEER=R2:r2.simson.net
PEER=NO:no.simson.net:6500


# These configuration variables are used to specify synchornization behavior.
# IGNORE specifies a pattern that should be ignored if it is found in a filename
IGNORE=*.o
IGNORE=*~
IGNORE=._*
```

```
# Where Syncframe stores the transaction log and other information
# This must be a directory
#
SYNCSCRATCH=~/.syncframe
```

```
# Verbosity [0-5], 0 is default
# 0 is no output
# 1 is errors only
# 2 is minimal informational messages (e.g., "Starting Daemon")
# 5 is full debug

VERBOSITY=0

# Do we hunt on demand? If so, hunter must be called manually
# Default is "no"
HUNT_ON_DEMAND=NO

#
# Logging
#
# Where to log:
#
LOGFILE=~/.syncframe/log.txt
#
# What to log:
#
LOGLEVEL=0x0001

# Note LOGLEVEL can be a decimal number or be preceeded by "0x" to
# be a hexdecimal level.
# The following bit fields are accepted for logging:
# 0x0001 - Program startup/shutdown
# 0x0002 - Thread and task startup/shutdown
# Transactions:
# 0x0010 - Transaction creation
# 0x0020 - Sending transaction to a remote machine
# 0x0040 - Receiving a transaction from a remote machine
# 0x0080 - Committing a transactions
# 0x0100 - Expunging a transaction from the store
#
# Debugging:
# 0x1000 - Updates to metadata database
```

Table 9: A sample Syncframe configuration file, cont.