Syncframe: A Multi-Peer Synchronization Framework

Chris Elledge, Simson L. Garfinkel, Jason Levine and Hans Robertson

Abstract

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 all other nodes in the network.

Syncframe differs from other synchronization systems because of its ability to synchronize complex network topologies, support for intermittently connected and mobile nodes, and the segregation of its synchronization and data management algorithms.

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

I. 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 local copy in synchronization[1] is a challenging problem. Consider these examples:

- John has 77GB of data that he is analyzing for his thesis. To protect against disk failure 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.
- Jane's sister collects music on her home computer. Jane wants to have these music files automatically copied to her laptop whenever she visits her sister's home, and then have them automatically copied from her laptop to her home and work computers when she is in those locations.
- Sam has two laptops and two computers — one at home and one at work. Sam would like to be able to work at any computer and have any files that he modifies on any machine automatically copied to the others. He would also 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.

II. Related Work

There are many systems for performing data synchronization, including:

- **rdist** [1] distributes (pushes) updates from a master machine to client machines. It compares files on the client and server and issues an update when files do not match. 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.
- **rsync** [8] is similar to rdist, but updates can be initiated by either party. 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 since it only sends over blocks that have been added or changed.
- **unison** [5] is a two-way file-synchronization tool for synchronizing a pair of Unix or Windows computers. Synchronizations may be initiated either side. A metadata database is used to detect file deletion and changes. Unison creates a new metadata database for each pair of synchronization roots. Although

[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.

Unison can run on multiple pairs of nodes in order to form more complex topologies, such operation is inefficient and problematic.

- **Microsoft Briefcase** [3], 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 topology around the server.
- **Ficus and Rumor** [7] [2], like Syncframe, perform peer-to-peer optimistic replication for systems with intermittent connectivity. Ficus keeps track of all changes to a file using vector timestamps. Vector timestamps follow a file from one machine to the next, allowing Ficus to determine if and when two different copies of a file diverge. Rumor extends Ficus to mobile systems.

A. *Data Publishing Systems*

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.

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 data publishing systems for two-way synchronization by first replicating in one direction and then in the other, this method fails if modifications are made simultaneously at each location.

B. *Pairwise Synchronizers*

Unison and Microsoft Briefcase are examples of *pairwise synchronizers*. To perform pairwise synchronization, both programs maintain metadata for each root. A metadata database is created 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.

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 $A \leftrightarrow C$.

Pairwise synchronizers are unsatisfactory for synchronizing more than two nodes 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. One way of configuring this network would be to have both HOME and LAPTOP synchronize with WORK. But this approach fails when LAPTOP is co-located with HOME and the connection between HOME and WORK fails. Faced with such a network topology, a user might be tempted to create a new synchronization association between HOME and LAPTOP. But that doesn't work: Unison used in this situation would generate conflicts, as it would be the first two times that the two "roots" would be synchronized with each other. Briefcase requires that a directory be manually copied from the server into the briefcase before it can be synchronized.

Another approaching for synchronizing HOME, WORK and LAPTOP would be to establish three sets of pairwise relations. This approach works, but is awkward: unless all pairwise relations are continually updated, the system will report conflicts when they do not exist.

C. *Rumor*

Rumor is an optimistically replicated file system designed for use in mobile computers. Syncframe resembles Rumor, but improves upon it in several key ways: Syncframe replicates arbitrary objects, while

[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.

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 Rumor only runs on an obsolete version of FreeBSD.

III. DESIGN

Syncframe is designed to efficiently maintain a replicated object set across multiple computers (or nodes). Syncframe's most important features are: support for arbitrary topologies, support for arbitrary objects, and support mobile nodes.

A. *Architecture Overview*

All Syncframe nodes are equal. Nodes are linked together using peering relationships. All links are bi-directional; connections can be *initiated* by either peer. The topology is determined before Syncframe is started, although it is easy to add and remove peering relationships or nodes.

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

Updates can be deleted on each node when that node knows that its peers has seen the update, a process called *cleaning*. To facilitate this process, each each Syncframe node creates a special update called SEEN after it applies updates that it has received from any source. The seen is propagated to every other node. The SEEN update tells the nodes peers that other updates can be deleted. This is useful if two nodes A and B have a peering relationship, but the link $A \leftrightarrow B$ is down; as long as there is some other path P that extends $A \leftrightarrow P \leftrightarrow B$, the SEEN update will travel between A and B and each node will know that it is safe to delete updates that would other need to be held.

New nodes can join an existing Syncframe network using a special *joining procedure*. When a new node joins, it receives all the objects under synchronization; the joining process can be interrupted and restarted, which is not possible with Unison.

We define correct synchronization to mean that all updates are eventually seen by all nodes, and that every conflict is detected by at least one node.

Syncframe does not guarantee a a global ordering of updates, or that every change to an object will be captured by an update.

If a Syncframe network is partitioned, updates will be held on both sides of the partition until connectivity is restored. If the partition is permanent, the configuration can be edited to remove the peers that are across the partition, which will allow Syncframe to clean the updates.

B. *Syncframe Features*

Arbitrary Topologies. Syncframe allows nodes to be arranged in arbitrary topologies, e.g., trees, chains, and meshes, allowing Syncframe to be used in a wide range of scenarios.

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 only built such a synchronizer for file systems, but the framework could easily be extended to synchronize an address book or email repository.

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

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

No Kernel Modification. Syncframe runs in user mode, making it portable and easy to install.

OS Agnostic. We have avoided using OS-specific functionality in our design and implementation.

Archiving. Every node in the synchronization network is guaranteed to see every synchronization update. Thus, 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.

On-Demand or Automatic Synchronization. Synchronization depends the creation of synchronization updates, the propagation of these updates to other nodes in the network, and the application of the updates. Each step can be done periodically or on demand.

Firewall Friendly. Syncframe uses a single TCP connection to support sending updates in both directions, allowing nodes to operate behind firewalls where they can initiate but not receive connections.

IV. Implementation

A. Syncframe components

Syncframe comprises the following modules:

1) **The configuration file**, which specifies peers and other synchronization parameters.[3]
2) **The metadata database**, that tracks 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 updates that need to be sent to the node's peers.
5) **The Publisher**, which publishes updates to the node's synchronization peers.
6) **The Committed**, which applies updates that are received from other peers.
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.

B. Filesync

We have used Syncframe to implement implemented Filesync, an application that synchronizes files across multiple computers. Filesync synchronizes files that are stored in a single *synchronization root directory* and subdirectories of that directory.

The Syncframe Hunter and Committed 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; it does not need to subclassed for different Syncframe applications. The Publisher is implemented as a Java application; like Cleaner, Publisher 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, updates are kept in the update database until connectivity is reestablished.

1) Configuration File: Each synchronization node must be pre-configured 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.

[3] It is not necessary to configure each node with the names of all of the other nodes in the network; this information is discovered as Syncframe runs.

2) Objects, Metadata and the Metadata Database: An object consists of 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. Metadata consists of Object name; object Length; object creation time; object modification time. Filesync extends this definition with file username, groupname, permissions, and file type.

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 [6]. The object name is used as the key of the B-Tree.

Filesync uses filenames as object names and stores data in files on the file system. An address book synchronizer might use a unique entry identifier.

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. When a change is detected, Hunter creates one or more updates that encapsulate the change and adds the update to the update database. Hunter can be run periodically or on-demand, as desired by the user.

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 FileHunter, 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) The object is removed from the list that Hunter created in the first step.
4) 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. Updates are uniquely identified (named) by the originating node ID and the update number.

4) Publisher: Publisher is a program that keeps a node in sync with its community. The nodes to which a given node sends updates are termed "peers" and are enumerated in each node's configuration file.

Publisher attempts to keep a 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.

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 if there are updates to send.) The protocol's principle drawback is that it does not support simultaneous bi-directional transfers. SFP runs on on top of TCP or, optionally, SSL.

While it is essential for correctness that every node in the community see 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 and asking each peer permission to send an update before it is sent. SFP never sends updates to a peer that the peer has already seen. When all peers have seen an update, that update is deleted by the Cleaner (described below).

Publisher guarantees that all updates originating at a given node will be sent out in the order that they

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.
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 but not distributed throughout the entire synchronization network. TWRITE is used when nodes wish to join an existing synchronization network. [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 I

UPDATES SUPPORTED BY THE SYNCFRAME SYSTEM.

were received. 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.

5) 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, causing the object to be forked.[4]

Because each Syncframe peer sends all of its updates to all of its peers — and because updates are not sent to peers that already have those updates — updates flow through fast paths. 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 $A \rightarrow B \rightarrow C$ before they can flow from $A \rightarrow C$.

6) 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 would instead save updates in a long-term persistent store.

C. Implementation Issues

1) 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 update.

2) Conflict Detection and Resolution: Syncframe's use of the metadata database and the tagging of all updates with the object metadata before and after the update is completed guarantees that all updates causing an update will be detected. When conflicting updates are received the object is forked and the user is alerted. (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; Syncframe will detect that the second transaction is redundant and discard it.)

In practice, this is the strategy followed by most other synchronization systems. When systems that Unison or Briefcase detect a conflict, most users will manually make a copy of the two files that caused the conflict and view each file with a text editor. The user will then manually merge the two files. Syncframe assists this process.

More sophisticated conflict management can be found in [7] and [2].

3) 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.

[4]Note that our current Syncframe implementation conflicts are detected but objects are not forked, due to time constraints.

V. 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.

A. 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 prove to each other that they know the secret community key without revealing the key using a challenge/response process.

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 replies 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.

B. Encryption

Setting a flag in the configuration file causes the Syncframe protocol to run on top of the Secure Socket Layer with Diffie Hellman anonymous key exchange. This protects data from eavesdroppers.

C. Vulnerabilities

Even with SSL, Syncframe 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. We accept this vulnerability, although it could be eliminated by using SSL's client and server-side certificates.

VI. Performance

To test the performance of the Filesync system, we created a test directory with 100 files of 1 megabyte each. Table II 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 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 1 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

	Hunter Times			Committer Times		
Benchmark	Clock	User	System	Clock	User	System
all new files	17.01	5.943	4.833	5.326	2.682	1.571
50% modified	8.931	4.346	2.991	5.696	1.857	2.553
0% modified	4.153	2.936	1.118	0.022	0.005	0.011
tar/untar	9.142	0.076	3.803	10.508	0.113	3.538

TABLE II

HUNTER AND COMMITTER TIME TESTS: TIME FOR HUNTER TO PROCESS 100 FILES OF 1 MEGABYTE EACH ON A 400MHZ APPLE POWERBOOK G4 WITH 384M OF RAM (ON LEFT), AND FOR COMMITTER TO COMMIT THESE TRANSACTIONS (ON RIGHT). ALL TIMES IN SECONDS.

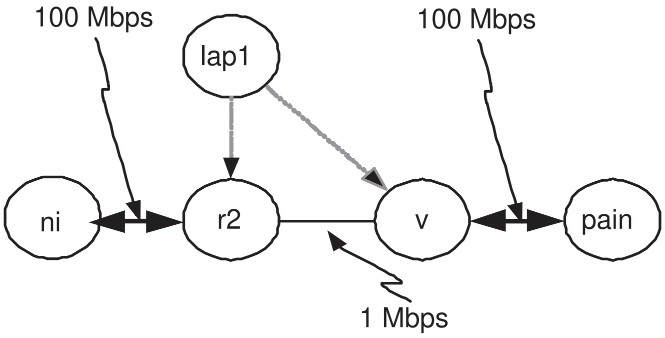

Fig. 1. A simple synchronization network set up for performance measurements. The computers ni, r2, v and pain are all workstations running the FreeBSD operating system; r2 and ni are connected with a 100 Mbps network, as are the computers v and pain. Lap1 is a laptop computer that has intermittent connections to r2 and v.

update was sent to *lap1* 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[4] 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. 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.

VII. CONCLUSION

We have created Syncframe, a framework for building data synchronization tools. Using this framework we built a file synchronizer. We then tested this synchronizer on a small network.

Syncframe makes two import contributions to the field of data synchronize. This first is propagating updates allows for more efficient synchronization than systems that perform synchronization by comparing files on multiple systems. The second contribution is the separation of algorithms that perform synchronization from the algorithms that handle data representation and management.

REFERENCES

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

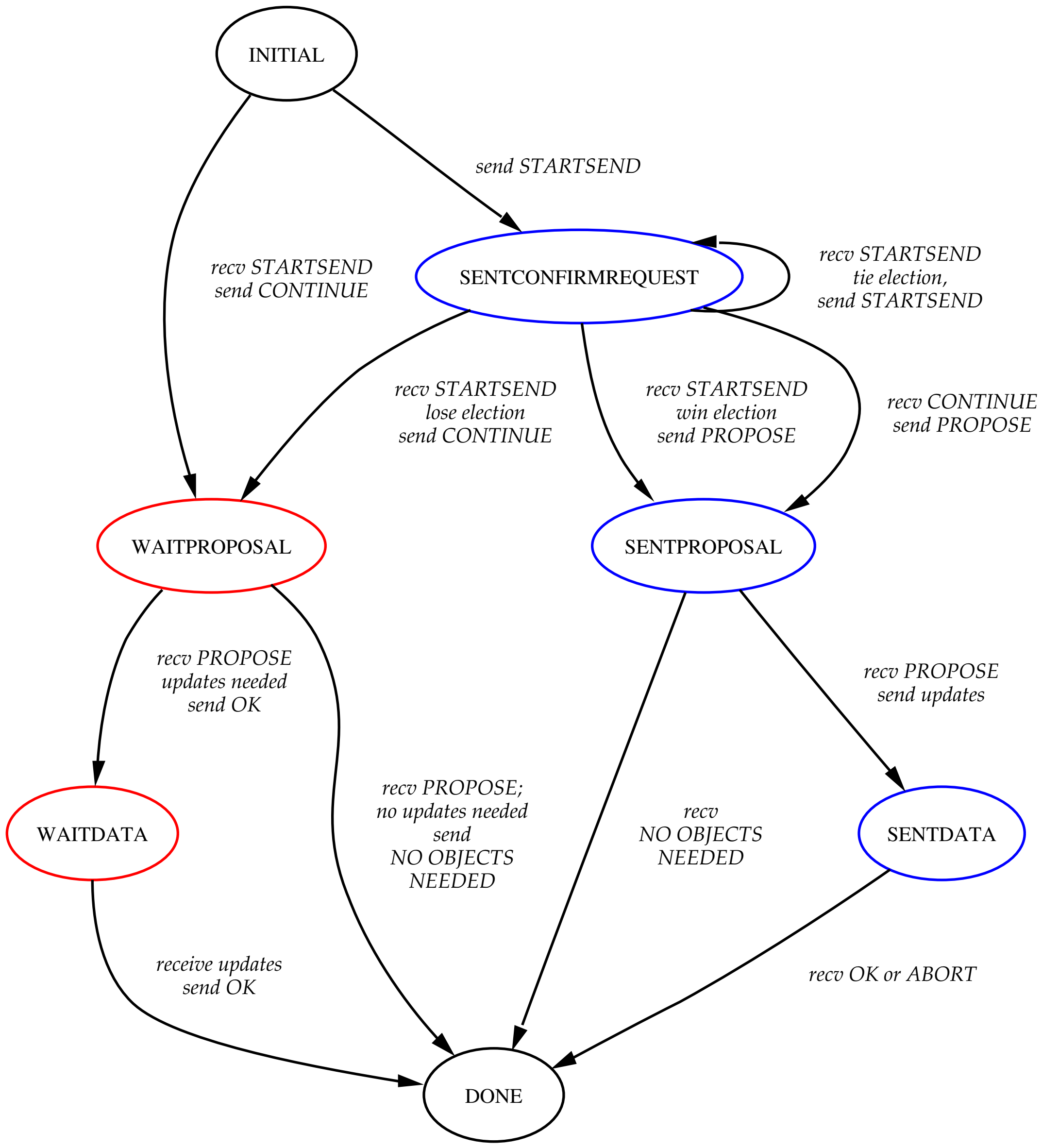

Fig. 2. Syncframe protocol state transition diagram with error states omitted

[2] 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`

[3] 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`

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

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

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

[7] 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.

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