WashuDCFS

Distributed Consistent File Sharing across Peer Server Systems

Beracah Yankama, Edmund Chou, Mingyang Xu
{beracah, naptime, xumy}@mit.edu

Abstract

WashuDCFS proposes a distributed file system without locking – instead with optimistic, realtime, modification level consistency. Finely grained file/dir changes are effected *immediately* locally, then all participating servers and applications are queried for conflicts. Conflicts are detected and resolved instantly, if they occur, by the user & application – though the application is allowed to go on and continue effecting edits in the meantime. WashuDCFS clients however, *strongly* expect not to conflict – clients' optimism is an *informed* one – clients receive enough information about all other servers and clients to know where conflicts are likely to occur – conversely, where they are less likely to occur. When a conflict does occur, the overlaying application has enough information to decide how to proceed – whether to retry the operation, to abort it, or to perform a different operation.

Preliminary evaluations show that WashuDCFS maintains consistency for multiple clients and single server scenario. Multiple server multiple client development is not yet complete. Though not known for certain, it is expected that the local performance of the client using WashuDCFS will be comparable to other network file systems (NFS/AFS/etc) for read operations, but because WashuDCFS only transmits modifications, it will far outstrip them for conglomerated write operations and for consistency operations (if they exist in other systems).

1. Introduction

This paper proposes a new method to maintain consistency across a distributed filesystem. This paper steers away from a 'versioning' and 'file level locking' method of consistency maintenance among distributed files by moving to a high granularity of modification messages. It does not try to avoid file level or directory conflicts with locking mechanisms – rather, it provides enough information for applications to *predict* where conflicts are likely to occur *within files or directories*, and to resolve conflicts instantly, if [as] they occur. This differs from existing systems, which resolve conflicts reactively. The side effect of this is that a much higher level of integration is required between the application layer and the filesystem.

It is our philosophy that in distributed files, where conflicts are bound to occur, that the component best suited to resolving conflicts is not the file system or the file servers, but rather it is the originating application and the user.

To facilititate this, all modifications to any file or directory inside the distributed system, and current modification pointers will be sent to the other servers – as they happen. Applications using this system will have to use, interpret, and reflect a *constant* stream of file change and modification pointer messages – constantly deciding whether or not such messages initiate a conflicts with, and reflecting them to the user.

In traditional filesystem-application services, files are whole entities that multiple applications operate on independently of each other – in effect creating conditions by which the latest changes to a file may be clobbered. Superficially, some locking can be imposed (at a file level), but user-version conflicts can still go unresolved in some filesystems (NFS). Some consistent distributed filesystems Ficus [1], etc change this slightly, providing an intermediate solution – they attempt to maintain versioning information of files, and to resolve conflicts between versions after they have occured.

Unfortunately, this requires the file-system to have *file specific information* on how to resolve conflicts. In a mail program, this may be simple – just append the new message – but in other file examples, for instance bitmapped images, how to resolve differences may be more obscure. This is commonly done with a standard merge algorithm, but conflict resolution may potentially arise and the file system has no way of fairly committing one version over another. This paper counters that the file system should not be the level at which a change conflict is resolved, but rather the file system should only concern itself with facilitating consistency and imposing consistency semantics – that the client applications using the file system should be the arbiters of conflict resolution. This amenable to the point of view that the originating client application is the only member of the system containing sufficient information to make that decision appropriately.

This paper proposes a model by which several peer servers are connected together, exporting a replicated filesystem, several client computers mount the filesystem from any known server (running our application-fs emulation client), make changes both to the directory structure and to the files, and those changes are accurately represented to all relevant users, along with conflicts. This system does not try to handle failed servers or unreliable networks, instead it focuses on a design and protocol that could provide distributed, consistent, filesystem access at nearly local performance.

2. Motivation

The desire to maintain *realtime* file & directory consistency across multiple computers and across multiple servers is growing. File systems & sharing has become more widespread and more distributed in nature – it is increasingly common to find that a single file is being accessed and modified by multiple clients, or that a single file exists in several places on a network – all of which need to be kept updated. The current solutions – to lock the file, force reloads, or overwrite is unacceptable at best – intermediate changes are not transmitted or are overwritten, requiring significant mechanisms to be recovered. Even some cache consistency solutions (e.g. Sprite [2]), which insure that multiple replicated caches are kept consistent, rely on blocking caching for sources opened in r/w mode & flushing/reloading portions of stale cache, thus eliminating the realtime component to concurrent cache changes.

We want to make a system where local changes can be reflected immediately to the user, but are propagated around the network in realtime, so that all servers and participants have the *absolute* latest set of changes. We want the decision of how to resolve conflicts to lay in the hands of the user or the application. Relative to the file system, this means that we want the application/user to be able to operate on files with local performance, but with the added information of conflicts, external changes, and consistency.

Consider a load balanced system with only 1 physical storage device, but many clients. It is desirable to have multiple servers that replicate the storage in memory and serve to balance filesystem load to clients. In such a system, it is possible for the individual replicated systems to become inconsistent – it is our desire to produce a system that relays changes in realtime, while maintaining both local consistency and local performance.

3. Design

The design discussion will focus on three components. First we will provide an overview of the system design & present some basic implementation limitations/assumptions. Then Washu protocol and the role it plays will be discussed. Finally we will discuss the separate responsibilities of the file servers and the FS clients of our file system.

While our system is designed as multiple peer servers (Figure 1), exporting replicated file systems of **text** files, we first implemented a single server (Figure 8 – '4. Results') maintaining consistent, realtime transactions among multiple clients. Technically, operation is similar to peer servers and this provided us with some basic understanding that could be used to further peer server development. Following this, we began work on implementing the distributed peer server model.

We mentioned that we will not handle network failures between servers or clients, or server failures. This because generically, all of those failures induce version inconsistencies that are not immediately protocol related – handling requires other semantics that (of course a well designed protocol could facilitate) are more similar to algorithms for file level consistency/versioning systems like Ficus or CVS than they are to WashuDCFS.

All files and directories are 'text files'. This assumption allows directory structure modification to proceed in realtime, with real user input, the same as with files. Text changes are easily visible, and text is the primary way in which directory and file listings are currently represented.

“Locking” of directories does not need to be imposed (even if a file is open within), as all the clients are receiving realtime information about changes to that directory. The only conflicts that can occur are 2+ name changing conflicts, or deleting of an open file conflict. Proper method would be to query the client to see if it is alright to delete its open file, but we give the server the presumption to deny the delete of an already open file.

We do not handle security, user rights, or nonreplicated file structures.

Clients may connect to any particular peer server for any particular file or directory – their changes to that file are reflected across all servers immediately, and across Clients if the client is editing that file/dir.

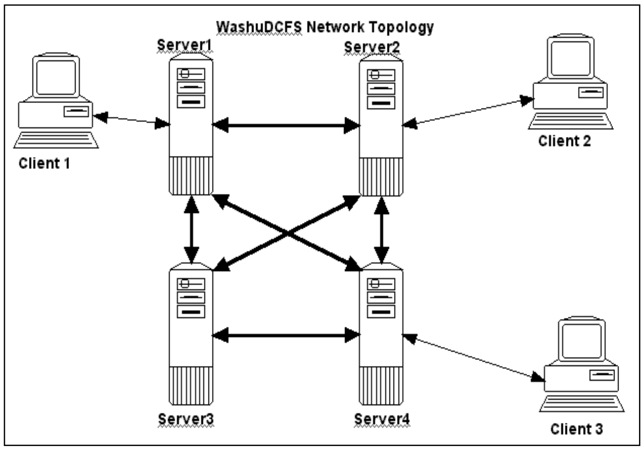


Figure 1.
The WashuDCFS Network Topology shows multiple interconnected servers relaying information to maintain consistency. Clients then connect to servers to access and modify files on the network.

Multiple servers communicate changes amongst themselves and work with clients individually to resolve conflicts.

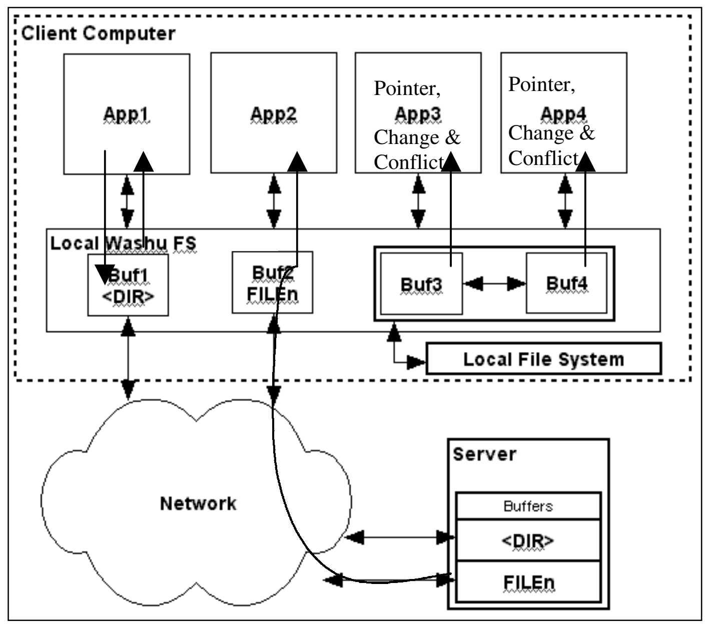


Figure 2.
A representation of the application integration necessary within the client to edit a consistent file. Files and directories are replicated locally to yield local access performance. App1 is modifying the directory, which is reflected on the server. App2 is modifying a file, which is reflected on the server. App3 & App4 are modifying a local file simultaneously – their changes are reflected to each other. Transmitted between the App layers and the Local Washu buffers is change information, edit pointer information, and conflict information

Conceptually there is really little difference between the local Washu FS and the Remote server – however, for the point of this project we are using a client application that plays the part of the local FS and Application layer. We have concentrated our design efforts into building the Remote Servers.

Process:
1. Client 1 opens file A on server 1.
2. Server 1 opens the file on servers 2, 3, and 4.
3. Client 2 opens file A on server 2
4. Server 2 opens connections between 3 & 4 – 1 is already open
5. Client 1 makes a change to file A; effective immediately.
6. Client 1 sends the change to Server 1, “can I make this change?”. – Client 1 continues editing
7. Server 1 sends the change request to 2, 3, and 4. Both 3 and 4 reply “OK”, but Server 2 holds it and asks client 2 if it is ok.
8. Client 2 reponds “OK”, at which time, Server 2 response “OK” to servers 3, 4 and 1.
9. Servers 1, 3, and 4, having received a full complement of “OK”s, implement the change in their local buffers.
10. Server 1 responds back to Client 1, “OK, no conflict”, at which point in time Client 1 knows that its past change transaction is complete.

Had one of the Clients REJECTed the message, then the Servers would all have aborted the transaction, and the Client would be notified that its past modification was rejected. It becomes the Client or User's decision at that point, how to proceed.

What is noticeable here is 2 things – that will be elaborated in Washu Protocol.
1. Servers communicate with all other active servers
2. Clients continue operating as though they never needed to know if a transaction went through.

3.1 {Washu} Protocol, in development

Relevant to this current DCFS design, Washu Protocol provides *nonblocking*, consistent transactions across network replicated data buffers. This is so that local modifications to the buffer can proceed at high speeds, have little expectation of conflict other modifications, and multiple modifications can proceed in parallel. This is important for a file system as, standard file system semantics aside, if file operations block on the network, then each transaction will incur a significant local performance penalty.

Some other systems also provide realtime, distributed modification : REDUCE [3], for example. Unlike REDUCE, Washu does not require an external Transaction Server to order transactions.

Washu constantly maintains all participants' edit location pointers separately from the edit commands (to add or remove elements). The commands to MOVE the pointer are separate from the commands to EDIT the buffer for important reasons:
1. The pointers allow Washu participants to operate on their buffers with *informed optimism*. Location pointers establish 'independency' of edits, and the locations from which external edits will proceed – that is, edits in separate locations, can proceed forward in parallel with few ordering semantics.
2. If the position of the edit was contained within the edit command itself, then Washu would need complete ordering (or else the position would become invalid) – and hence would need to block on the network for commands to be commited before reflecting them to the user. (Figure 6)

But because the pointers are separate, the only ordering semantics necessary are those to needed protect the relative pointer positions. Any number of edit transactions can be outstanding on a Washu buffer, but all transactions *must* have a properly positioned edit pointer. If all the pointers are consistent, any participant can send or remove data from his/her pointer with no concern about message ordering. In fact, the only thing the participant has to be concerned with, is the chance that another Washu changed his state or buffer in a way that conflicts with his pointer & edits during transmission. This condition is handled with Accept/Reject semantics on a second phase of the transaction.

It is often observed that the pointers provide information to the user for the user to 'be careful', and this is certainly possible, but the pointers are technically necessary to allow independent edits to proceed without ordering constraints.

The single case that breaks pointer coherence is MOVE pointer -> MODIFY buffer ordering between participants. TCP will guarantee ordering *from* a host. But if a MOVE arrives out of order with an edit (Figure 3 – red MOVE) between participant, then the pointers will become desynchronous – entirely breaking consistency.

Rules for consistency:
1. Edit commands have to be accepted by all parties to insure that the conditions prior to the insert are true. [i.e. nothing changed remotely that the local doesn't know about yet]
2. All relevant pointers must be fully consistent before an edit is effected

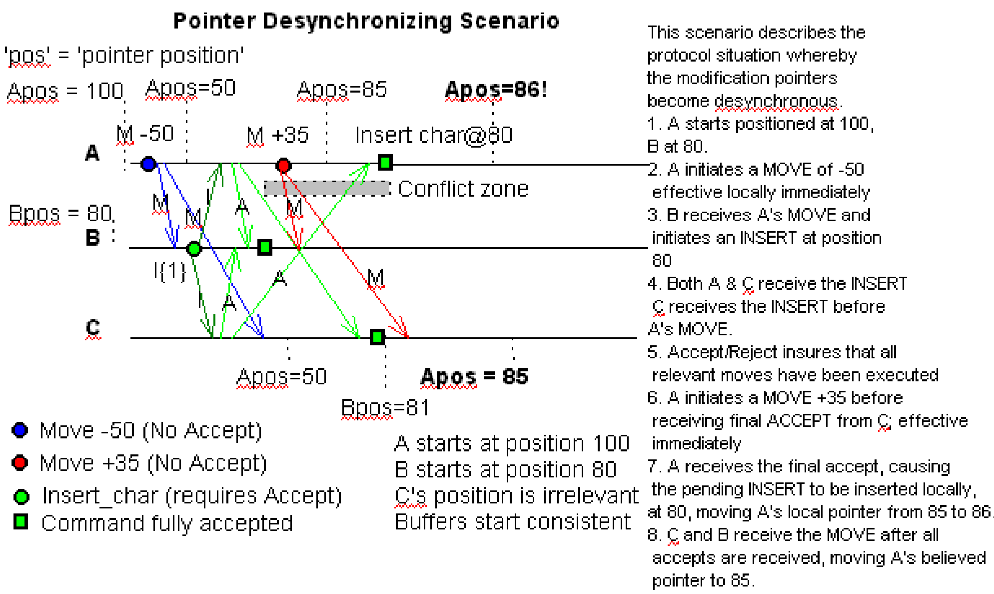
To keep pointers synchronized, yet steer away from blocking on the network, Washu uses a hybrid topology.
1. EDIT commands are broadcast to all participants, where they are ACCEPTED or REJECTED based upon conflicts.
2. A REJECT decision is made by default if the EDIT orderings would cause the buffer pointers to dissociate and become incoherent. For instance, if two participants both performed 'insert' at the same location, simultaneously – both participants would received inverted orderings.

3. MOVE commands are effective immediately, and are not ACCEPTED or REJECTED.
4. Washu is fully connected to minimize the time that the transaction is outstanding. The current solution to the one ordering problem is to stall transmission of commands whos position will be interfered with. Fully connected allows that stall to be minimized.

Accept-Reject Semantics achieve 2 needs:
1. Provide a mechanism for rejecting of edits that conflict – this is possible due to network delay. i.e. not all participants may have the latest edit position.
2. Insure all pre-edit move commands to be received before a relevant modify command (Figure 3)

By receiving A/R responses from all participants, Washu insures that all MOVE commands have been received by all participants. The only problem condition is that a MOVE command is effected before an EDIT is fully accepted -- most participants have already accepted it. This is a situation where the edit pointers become desynchronous. (Figure 3)

Figure 3



B's pointer position is nonrecoverable because insufficient information about state exists.

However, a possible solution can be achieved by stalling the transmission of the offending MOVE in the Conflict Zone from Figure 3 until all relevant ACCEPTS/REJECTS are received. This allows the transmission of the move command to be made with determinant information. (Figure 4)

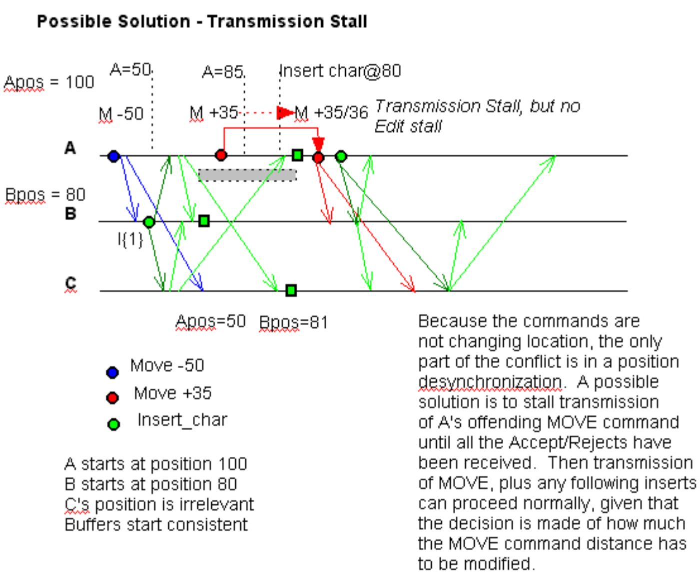


Figure 4

The blue & red lines represent the MOVE commands, which take place without any response. Their synchrony is guaranteed by the EDIT elements, which insure that all relevant MOVE commands have been received long before the modification is committed. Another possible conflict is if the blue move arrives after the insert is effected locally on B. Fortunately, there is enough state in B to know that no accepts have been received, so the move was applied from A before B's insert.

Other Transactional Semantics & Washu

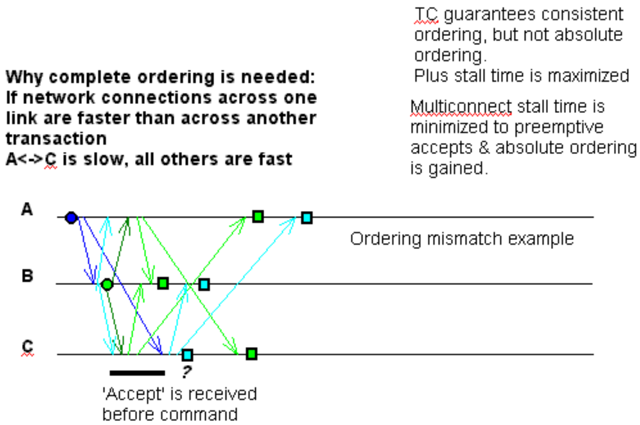


Figure 5
Pure Accept-Reject does not guarantee ordering, it merely guarantees detection of ordering mismatches. This can be seen in that the light blue ACCEPT is received ahead of the blue command, with the dark green command falling in-between.
However, the total time any transaction is outstanding is extremely small.

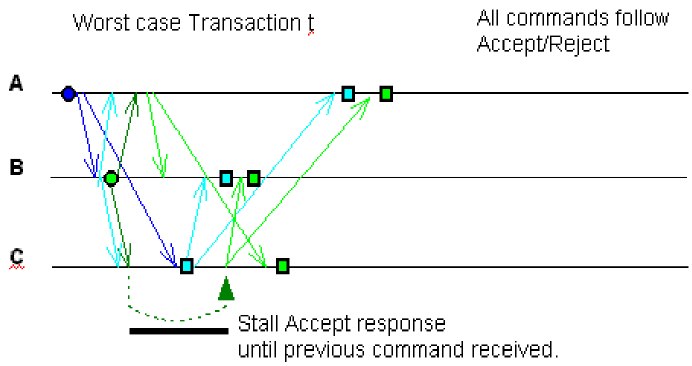


Figure 6
A 'stall' can be added into pure Accept-Reject, where C does not respond to any command until the previous Accept's Command comes in. This guarantees absolute ordering. Transaction t is also reasonably small. Unfortunately, this also forces transactions to wait *and* forces the client application (on top of A/B/C) to block on the network. It cannot reasonably effect a buffer modification in a system where ordering is important. This becomes important to the protocol, though ordering is unimportant to Washu.

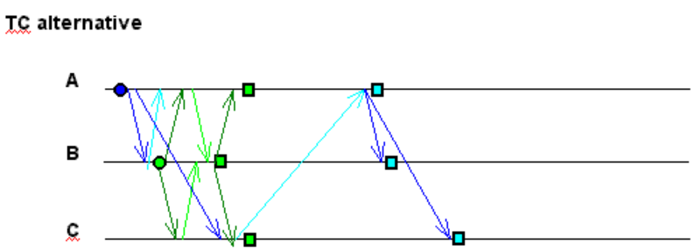


Figure 7
Two Phase Transaction Coordination; Coordination insures ordering. However, it also maximizes the time in which a transaction is outstanding. Since ordering is unimportant to Washu, and transaction completion time also important, Two Phase Commit is not a good solution.

Washu does not concern itself with absolute ordering, because that would necessitate local blocking. Also Washu replies to every participant to insure that the transaction is left outstanding for the minimum possible time. The only parts that need ordering are MOVE to EDIT command transitions. Other than this, ordering does not achieve anything.

Washu is designed so that edits to a replicated buffer proceed in parallel, without blocking. That is, edits are highly localized operations that have little to no bearing on edits in other locations. Provided that all participants in a Washu session maintain fully synchronized pointers of where the other is editing, no ordering of edits is necessary. And because each client knows the location of other edits; local edits will rarely conflict remotely.

Washu accomplishes this by maintaining 2 transactional topologies for the two kinds of transactions. Single broadcast is used for pointer information and other non-modification related data. Two phase Accept-Reject is used to discover conflicts & to insure all single broadcast information has been effected appropriately. Accept or Reject specifically is not necessary for to insure single broadcast has been received, just a second phase of the transaction is necessary

3.2 Servers

To simplify our system, we assume that crashed servers to not rejoin the system. Furthermore, during the initialization phase, we require that the root directory for each server is identical. We expect that most systems will begin with an empty root directory and allow user-created files to populate the root. The goal of the servers is to maintain the consistency of all files within the root directory so that each server's root directory is identical in terms of file structure and contents. This must be accomplished while accepting modifications to the files or directory structure supplied by connected clients.

These goals are related since servers will only wish to modify files or directory structure as a result of client requests. Thus the servers behave similarly to the client peers handled by the Washu protocol, but with a few slight modifications. There are two cases to consider for handling a request. First, the modification request may come from a client. As a result, the server then communicates with all the other servers to verify that the modification should be accepted. Thus a server may also receive a request from another server.

In the case where a server receives a request from a client, the server must query all the other servers to see if the transaction can be globally accepted. When the server receives confirmation from all the other servers, the transaction is committed and echoed to all the clients connected to it. Otherwise, the transaction is aborted and the requesting client is notified of the rejection. A pseudo-code implementation of this scenario would be:

```
On server S:
1. Receive client C request to perform transaction t
2. Ask all other servers, "Can I perform t?"
3. If all servers reply "ok"
     then: commit t to each client connected to S
     else: notify C that t has been aborted
```

Because servers need to notify all other servers in step 2 of the pseudo-code above, each server must also know how to handle server requests to perform a transaction. A transaction can be approved if it causes no conflicts with any of the current client operations. According to the Washu protocol, the server request is acknowledged by responding to each other server which are presumably also considering the transaction. The server broadcasts either an accept or reject response to a transaction, then waits to hear the broadcasts from all the other servers. If all the responses are positive, then the server commits the modification to the local file system. Each server will also be essentially running the following pseudo-code to handle server requests:

```
On server S:
1. Receive server request to perform transaction t
2. Check client activity to verify t causes no conflict
3. If t causes conflict
     then: send "no" message to each server and exit
     else: send "ok" message to each server and continue
4. If all clients replied "ok"
     then: commit t to local file system
     else: abort t
```

By handling client and server requests as described above, we expect each server to successfully respond to client requests in such a way that the root directories are consistent across the entire system.

3.3 Clients

As mentioned, the necessary functionality of a client machine comprises not only a filesystem that can partake in the consistency layer, but also an application that can interface and understand the messages it is receiving. Rather than build this system entirely from scratch, we connect a customized client text editing application to a Washu server to emulate this functionality. Also, clients are singly connected to servers – not only do we believe this to be a real-world consideration, but also to connect the clients together with all other servers at runtime will require advancements to the protocol we are not prepared to handle.

Clients connect to a known server through a text editing program supporting the Washu protocol. Directory listings are reported as text strings to the text editing program – changes to the directory structure (copy/move/del/rename) proceed in realtime, as will directory browsing. For the purpose of this project, the text editing functionality will be limited to basic entry, delete, copy and paste operations. The directory semantics will be shared as consistent text buffers – changes happen immediately unless rejected by servers. For instance, renaming a file (or directory) would consist of changing the name in realtime, with all other directory listings changing simultaneously in realtime. It would proceed normally unless one of the servers REJECTED the edits either because it is already changing the name, or because the file/dir is currently open.

4. Results

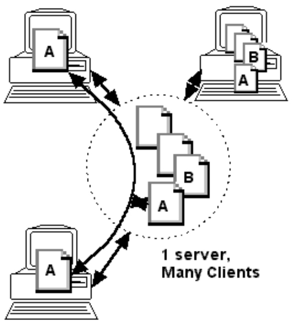


Figure 8
Single Server, Multiple replicated clients

A prototype system, written in perl consisting of a single server and multiple clients (Figure 5) successfully maintained file level synchrony for clients and server under a variety of tests (Appendix A). We have not yet tested directory operation.

It is too early to tell if distribution of servers (written in C) will be successful. Preliminary results are promising, however, as the pointers remain positioned correctly. Since it is technically not much different than multiple clients connected to one server (remember that the clients have local replications), we do not expect that it should proceed with any problems.

WashuDCFS will also need to be tested for a 'stop condition', that is, if there is a case and a situation where the number of peers operating on a buffer automatically destroys synchronization.

5. Conclusion

WashuDCFS is a system to provide distributed, consistent, realtime file system r/w access without sacrificing performance due to blocking. Currently, insufficient information is present to be conclusive, but given the success of the single server-multi client model; successful peer-peer server is expected.

If the overall WashuDCFS system is successful, both on an application and user scale, it has wider implications than just as a distributed file system. Imagine that two or more different applications can simultaneously apply changes to a file and have them reflected in the other! The model of application integration with the file system will have to change accordingly to acknowledge [conflict/change/position] messages at the application level. This will allow more powerful IPCs (Interprocess communications) and mechanisms by which separate applications can simultaneously read, write, and be aprised of changes to any file that they have open.

Imagine that the line between local and remote file systems is blurred with DCFS! The same applications could perform operations over the network just as readily as on local files!

In a distributed shared memory system, cache consistency could be maintained in the same manner as WashuDCFS.

6. Future Work

Since our system was designed with a number of simplifying assumptions, we present a number of future modifications to our system that would successfully deal with the problems for which we avoided consideration.

6.1 Connectivity Issues

One of the assumptions that we made implicitly was that network allows all the servers to be fully connected and that no connectivity problems will ensue.

For example, in the event that Client B does not receive a response to a transaction [operating on memory location X] from client A, then Client B will ask either Clients C or D for the transaction state. If Client C is able to answer this question, then C constitutes (as an intermediary) a more reliable connection from B to A, than the direct one. B will officially disconnect from A and notify the network that it is now C is relaying from A to B. But until that point, B will keep memory X protected, REJECTING any attempted edits (because it has not received all replies). A, if still alive, will ACCEPT edits to that location, but until it receives an ACCEPT from B, will not commit them.

6.2 Synchronization Procedures

WashuDCFS relies on the consistency of root folders. Most noticeably lacking in our current design is the inability to handle server crashes or network failures. Servers with obsolete data should be able to connect with known good servers to receive the latest approved transactions past the crash point. While a naive implementation would simply copy all the files under the root directory of a server, this could be an expensive operation for large file systems. Therefore obtaining only modification data would be much more efficient.

The synchronization idea does not offer a reasonable solution to adding new servers to WashuDCFS. Synchronizing a new server requires a copy of each file through the network, which would be inefficient for the reasons given above. Instead, we can consider a more dramatic and fundamental modification to WashuDCFS of decreasing the number of redundant files to achieve a good mechanism for adding server nodes.

6.3 WashuDCFS with Less Redundancy

Rather than maintaining an exact replica of the root directory at each server, we borrow ideas from more traditional distributed file systems and use all the servers collectively to agglomerate the file system. The benefit is that data is only replicated a few times, rather than at each server node, resulting in less network traffic – as a smaller number of servers would require coordination. Furthermore, server nodes joining the system would be handled simply by having servers recognize the presence of the new node and integrating its system.

There are a few disadvantages to this as well. Directory structure becomes significantly more difficult to maintain as querying the files in a directory requires searching for all the files in a virtual directory. This would require communication with each server node, opposed to our original design which was capable of supplying directory information from any given server. We also decided not to pursue this design since it would require several complicated algorithms to ascertain that server crashes would not result in missing files from the system and that files, when added, really are the same.

6.4 Security

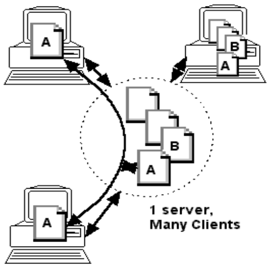
We realize that our system lacks any form of security. We had no intention of solving this problem, but here we explain how a secure version of WashuDCFS could be achieved. First, each participant of our system, both clients and servers, require public and private key pairs to encrypt and decrypt network messages. Our communication design would also require the use of a time stamps or incremental counters to prevent replay attacks.

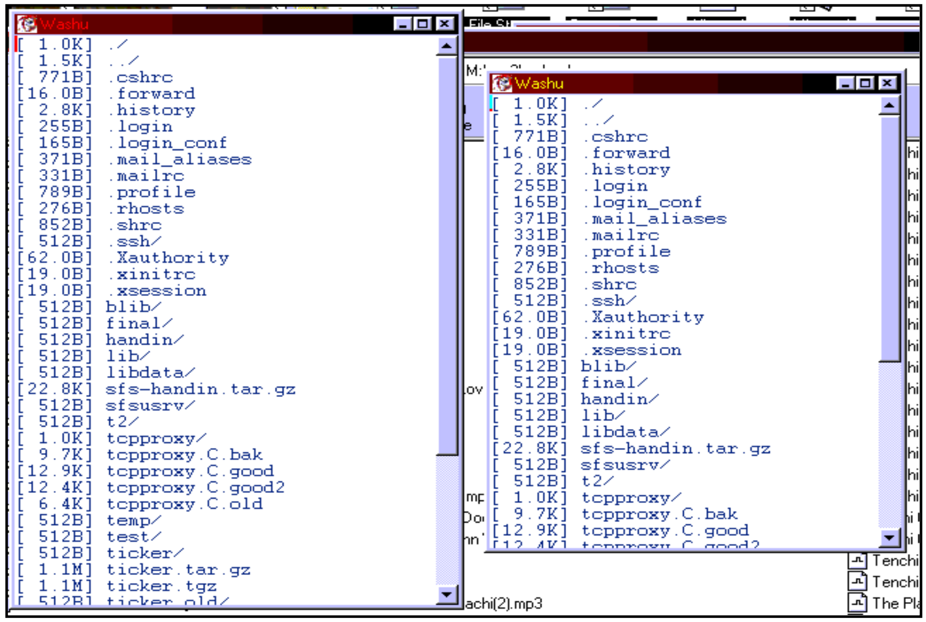
Furthermore, we would need a user authorization and rights system. Since the servers already have a mechanism for replicating files, we could extend the system to replicate lists of known users and password hashes. User administration functionality such as adding users, removing users, and changing passwords, would all be achieved by updating the user listing file. Users log-in by sending their usernames and hashed passwords to the server. This opens its own set of security concerns. It assumes that users do not have access to the user listing file. It also assumes that servers do not maliciously use the list of user password hashes to imitate other users.

7. References

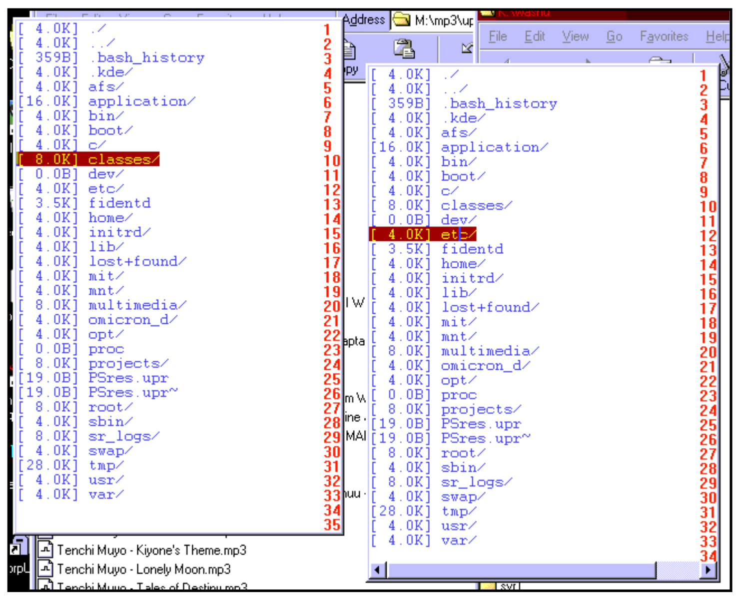
[1] Page, et. al. *Perspectives on Optimistically Replicated, Peer-Peer Filing*
[2] Nelso, et. al. *Caching in the Sprite Network File System*
[3] Chengzheng Sun, et al. *REDUCE: A prototypical cooperative editing system*, 1997.

Appendix A

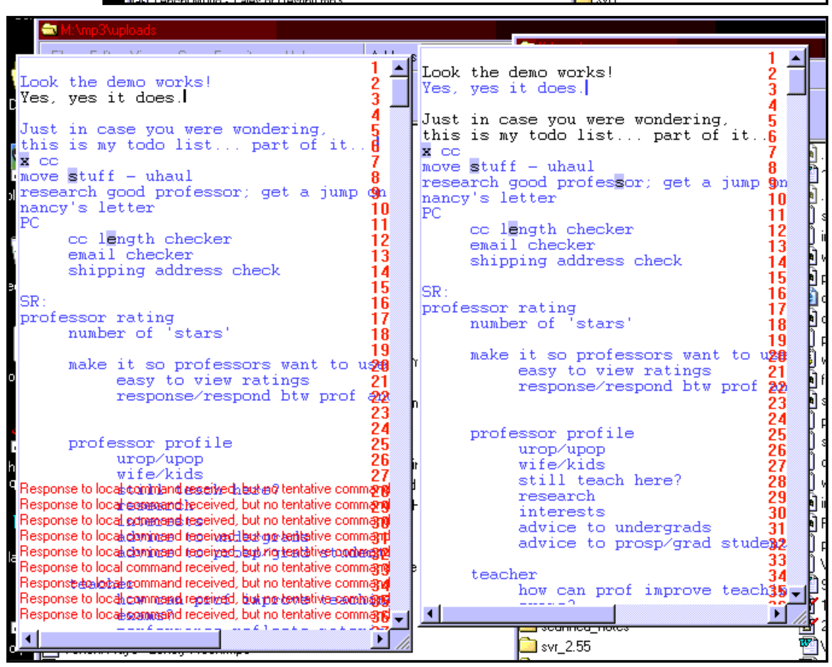




Directory Listings from my home directory, replicated across client applications.



Root Directory Listing from my home linux server, replicated across 2 Client Applications



File open (todolist) and replicated across two Client Applications, one Server intermediary. All three have cohesive buffers even after inserts of text. Remote text is appearing in black, locally inserted text appearing in blue.

The red notes at the bottom are the set of reponses the application received to its text insertions. This is from the second phase of edit transaction.