ECE454

Project 2

Design Document

S12-G035
Ravil Baizhiyenov
Robin Vierich

July 4, 2012

Table of Contents

List of Tables

Table of Figures

1. Introduction

Today, people use a number of personal computing devices in their everyday lives, including desktops, laptops, tablets and smartphones. This introduces a problem of synchronising files between multiple devices in a simple, reliable and efficient way. The project focuses on implementing a distributed file system to solve this problem.

The project addresses issues that other widespread synchronisation techniques such as using a centralized server face, including limited control over the data, high dependence on a single machine - central server, limited reliability and potentially low performance. The system uses a distributed approach to storage of files, where no single device has a complete list of files. Instead, the files are replicated on different devices to introduce a degree of redundancy and eliminate performance bottlenecks.

The project assumes a limited scope related to the intended usage of the system as a synchronization tool for a single user to replicate files among personal devices. The system is not intended to be used in an enterprise environment, or as a tool for collaboration between multiple users. As such, the following is a list of constraints and limitations of the system:

- There exists only a single user of the system.
- Concurrent changes to any given file will not occur.
- One device is able to provide high availability such that it is possible to use it as a system tracker. New devices cannot be added to the system nor can new devices read or write files replicated in the system until the tracker goes online. However, limited read and write access to existing files is provided to the devices that were online before the tracker disconnected from the system.
- The tracker has a permanent (static) network address which is known to all devices.
- The network allows every device registered in the system to directly connect to any other device.
- The system does not provide complete redundancy. In a situation where all devices that store a certain file are offline, read and write access to such file from other online devices is not possible.
- The system allows storing multiple versions of the file, however, the system only allows read access to the earlier versions. Only the latest version is modifiable.

The hybrid architecture was chosen for this project because it combines the benefits of the centralized and decentralized architectures and addresses some of the issues of both.

In this implementation of the project, a single device that is able to provide high availability acts as a tracker. The tracker stores such data as the mapping between devices and files they host, the list of online devices and their network addresses.

The hybrid architecture has the simplicity of the centralized architecture. Having a single device (tracker) that is responsible for some operations simplifies lookup, device join and leave. The tracker particularly simplifies the task of maintaining the list of online and offline devices as well as looking up a certain file. For example, if a new device wants to join the system, it only needs to be aware about a single network address: that of the tracker. File lookups are fast and simple: a device just needs to query the tracker for network addresses of other devices that host the file of interest.

The two important drawbacks of a centralized architecture: reliability and performance are addressed in the hybrid architecture by utilizing elements of a distributed architecture. In a centralized architecture, if the central server goes offline, then the system become not operational. However, in hybrid architecture, if the tracker goes offline, the operation of the system continues normally with the exception that new devices cannot join the system. In the project implementation, files are hosted in a decentralized (distributed) fashion. Every device connected to the system acts as a peer and is able to serve the files it hosts to other peers. The tracker is only used to find which peer hosts a certain file so that a lookup is fast and efficient. However, if the tracker goes offline, a peer can still lookup a file by broadcasting a file request message to all peers. This method is slower than querying a tracker, but provides a good solution during the rare occasions when the tracker being down. The performance issues of the centralized architecture where the central server is a potential performance bottleneck is addressed by hosting files on all peers.

Figure 1 shows a block diagram of the system implementing the architecture described above. Here, the system contains 6 peers, of which one is the tracker. The system contains 4 files, each replicated twice for reliability. All peers are connected together through a network connection.

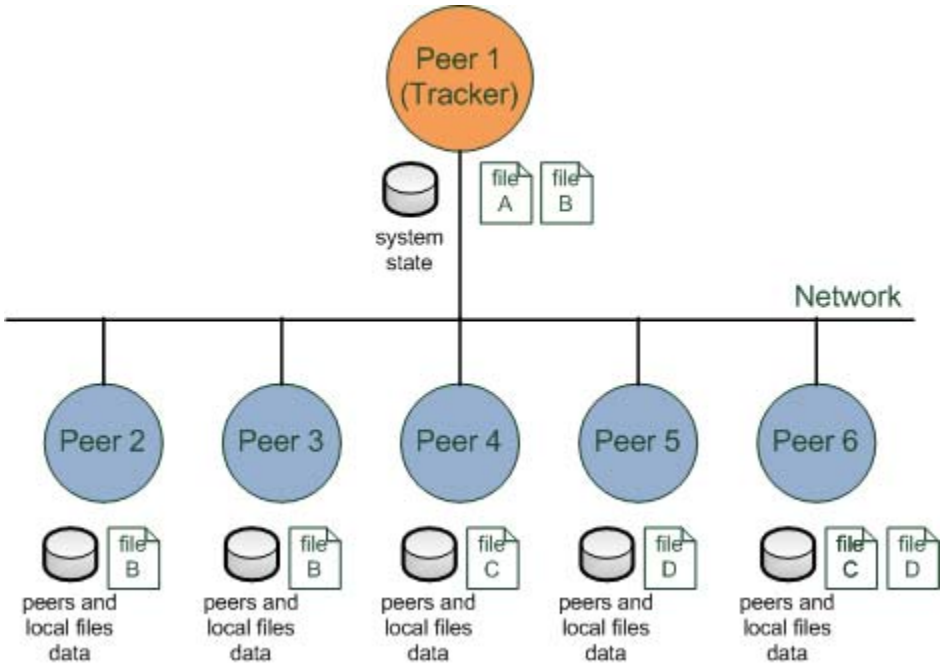

Figure 1: Block diagram of the system

2. File Distribution

The system contains a tracker node and other peers, as specified in Introduction. The tracker node is used for file lookup and for handling system join and leave.

The files in the virtual file system are replicated across a number of nodes, following a hybrid architecture approach. This allows for optimal performance by ensuring that many of the files are located on the local system (thus allowing quick reads and writes). The other advantage of this method is the redundancy through file replication which increases reliability. This ensures that most files are made available in the system even if a number of nodes go offline. Other benefits of this approach are discussed in Introduction.

The virtual file system allows to configure the level of replication all files should achieve. It is defaulted to 2, but can be increased as needed. The level of replication is the number of nodes that should be used to store copies of a file.

There are cases where certain nodes do not participate in replication of some files. The first case is based on the size of the file. Each node has a max file size associated with it. If the file is larger than the max file size, the file is not loaded locally. This means all requests to access the file will be remote and possibly slower. At the same time, certain devices have limited storage space (such as a mobile phone) and thus it is desirable to impose an upper limit on the file size for these devices.

The second case is where a node is at its full capacity. In this case, it cannot replicate any new files.

The third case is where a file is not replicated if it is explicitly excluded from a node. Each node holds a list of excluded files and it does not download any files that are in this list. This may be desirable to save space on devices where the file is not readable or interpretable (such as certain formats of video files on a mobile phone).

The tracker is assumed to be online and available most of the time during the file system operation. Using the tracker allows for the most efficient way of doing file operations. However, when the tracker is down, the system is still able to support most of its functionality, albeit with a degradation in performance. Table 1 lists the major file operations related to file distribution and specifies the system operation when the tracker is present and when it is not present. This is a high level overview of the system functionality. A more detailed specification is provided in the subsequent sections.

Table 1: System operation when tracker is online/offline

File	Tracker Online	Tracker Offline

Operation		
add new file	1. The node notifies the tracker that it has a new file and provides the file info. 2. The node queries the tracker for a list of nodes that are able to replicate the file. 3. The node pushes the file to the other nodes using the list received from the tracker. If the size of the list is larger than the maximum file replication level, the node uses a random subset of nodes from the list. 4. If pushing the file to a certain node fails, the node with the new file tries a different node from the list and notifies the tracker about the event. 5. When a node receives a complete file, it notifies the tracker that it has the file.	1. The node records the event in a backlog. 2. The node broadcasts that it wants to add a new file to the system to all other nodes it has a knowledge of. 3. As other nodes receive a request, they send a reply back to the original node if they can replicate the file. 4. When the original node receives replies from other nodes, it start pushing files to them. The number of nodes used is constrained by the setting of file replication level. 5. When a node receives a complete file, it records the event in a backlog.
read file	If the node has the requested file, the read is performed and operation is complete. Else, the following steps are executed: 1. The node queries the tracker for a list of nodes that have the file of interest. 2. If the file can be replicated on this node, it is downloaded from one of the other nodes returned in a list from tracker and read is performed. Else, the file is not saved locally as it is read from one of the other nodes. 3. If read errors occur, the node notifies the tracker about the event. 4. If the node downloaded the file, it notifies the tracker that it now has this file.	If the node has the requested file, the read is performed and operation is complete. Else, the following steps are executed: 1. The node broadcasts a request to other nodes that it is looking for a particular file. 2. As other nodes receive the request, they reply back if they have the particular file. 3. As the original node receives the responses, it initiates a read from the node for which the response was received first. If the file can be replicated on this node, it is downloaded completely first before it is read. 4. If the file was saved in this node, the appropriate event

		is stored in the backlog.
write file	1. If the node has the requested file, the file is modified locally. 2. The node notifies the tracker that it wants to perform a write. If it has written changes to the local copy, it also sends the tracker the checksum of the updated file. 3. The tracker finds what other nodes have the file and marks them as pending to be updated. If it also receives the checksum of the updated file, it sets the golden checksum of the file to the new value. The tracker replies back to the node with the list of nodes that have the file of interest. 4. The node pushes changes to all other nodes as received in the list from the tracker. 5. As other nodes receive changes, they modify their file and report the new checksum to the tracker. If the reported checksum matches the golden one, the node is unmarked as pending to be updated meaning that it has the up-to-date version of the file. If it doesn't match, the file on this particular peer is marked erroneous, and the peer is instructed to re-download the entire file from the source.	1. If the node has the requested file, the file is modified locally. 2. The node generates a checksum for the updated file and stores it in its local state as the golden checksum of this file and the event is written to the backlog. 3. The node broadcasts a request to other nodes that it is looking for a particular file. It also broadcasts the golden checksum if it has updated the local file. 4. As other nodes receive the request, they reply back if they have the particular file. If the golden checksum is received, it is updated in the local state and the event recorded in the backlog. 5. As the original node receives the responses, it pushes the changed data to the other nodes. 6. As other nodes finish receiving the new data, they compare their resulting checksums with the golden one. If they match, then the backlog is updated. If they don't match, the node re-downloads the entire file from source.

The efficient operation of the system is dependant on tracker having a complete and accurate information. Therefore, it is essential that the tracker is updated about all changes in the system when it comes back online. Each node keeps a backlog of significant events that have occurred while the tracker was offline. When the tracker comes back online, it collects information from all nodes about occurred events and updates its local state.

3. Version Management

The system handles versioning in a straightforward way. When a user wants to create a new version of a file, the user needs to specifically request this.

The version information is kept both on the tracker and on all nodes that host the file. The relevant information is kept in an embedded database to allow speedy lookups and insertions of new data. This option also allows for a reliable persistent storage. If a node or a tracker experiences a fault or is shut down by the user, it is able to load the previously stored data as soon as it restarts. Figure 2 provides the overview of the database.

Every version is stored as a separate file that follows the naming scheme as follows: file_name.version_number. All versions of the file are stored on the same node as the latest version of the file. If a certain node decides to distribute a file, it also has a responsibility to retrieve all versions. It is assumed that if the latest version of the file (aka. the “working copy”) is present on the node, it will have the file version history as well.

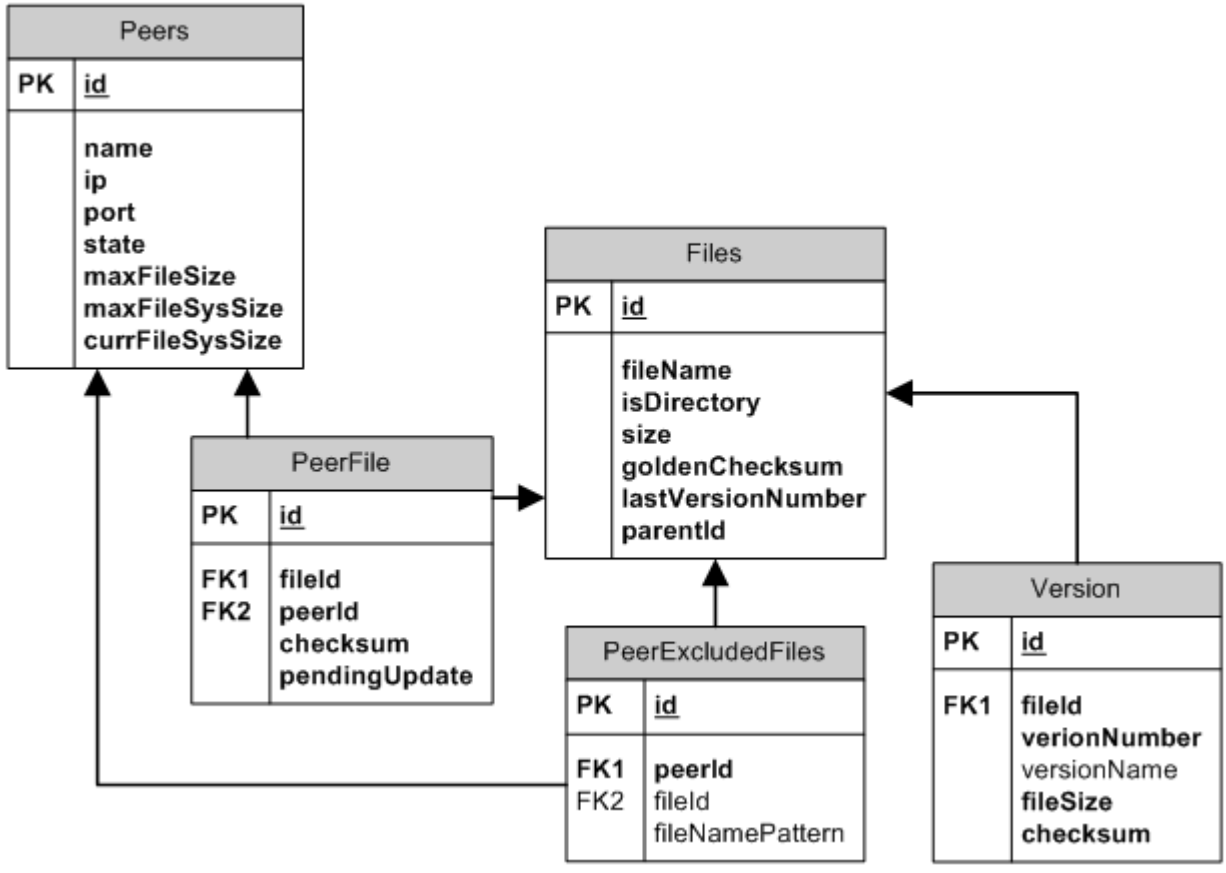

Figure 2: Structure of the tracker’s embedded database

If a request to create a new version is made by the user, the node sends a request to the tracker to update the version of the file. After the tracker receives this request, it updates the Version and the Files tables of its embedded database. It then notifies every node that has the file about the new version. At the point when the request to create a new version is just made, the file representing the latest version will be identical in content to the file representing to previous version on all nodes. However, as the file is modified, only the latest version is modified, while the old version remains the same.

Previous versions of files cannot be written to, but only read. This is a design choice, as versioned files are meant to be stable archives of a file. New changes must be made to the working copy which is not considered stable.

4. Synchronization

The tracker has an embedded database to track all of the relevant system state and synchronise it among the nodes. Every node also has an embedded database to reflect the local state. The structure of the tracker's embedded database was shown in the previous section. Figure 3 shows the structure of the local node embedded database.

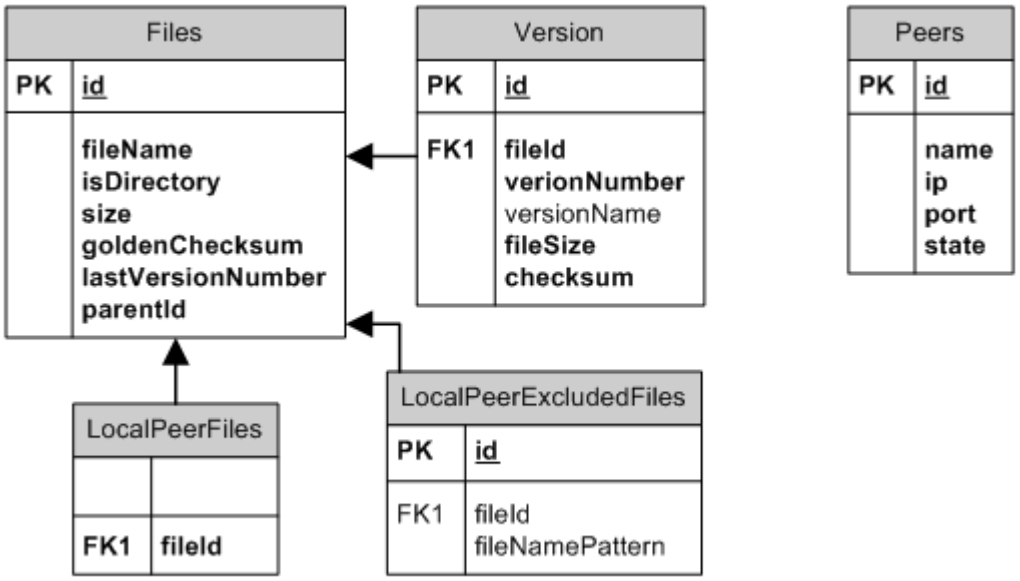

Figure 3: Structure of the peer embedded database

During the typical system operation, synchronisation is achieved by using the tracker. All needed data in nodes can be synchronised by querying the tracker. When the tracker is offline, synchronisation is achieved by broadcasting messages to all nodes and retrieving applicable responses. Table 2 lists synchronisation tasks and events and provides descriptions how the system handles them.

Table 2: Synchronisation mechanisms in the system

Task / Event	Tracker Online	Tracker Offline
Change of state of the node	When a node changes state (i.e. joins or leaves the system), it notifies the tracker which tracks the state of all nodes. After updating the local state, the tracker then notifies the rest of the nodes in the system about the new state of a particular node, or notifies that the new node is now in the system. It also broadcasts the node's network address in the latter case. Every node tracks the state of all nodes in the system so that the system can continue operation when the tracker is offline.	When the tracker is offline, new nodes cannot be added. Existing nodes that change their state broadcast it to all other nodes.
Addition of a new file	Every file has a unique ID associated with it. Every node has a knowledge of all files in the system and their unique IDs. When a new file is added to a node, the node notifies the tracker about the event. The tracker generates a new file ID and notifies the node back as well as broadcasts the unique ID to the rest of the nodes so that they become aware of the new file in the system. The node is responsible for pushing the file to other nodes. In which case it notifies other nodes about all relevant file information, including the unique ID.	Every file has a unique ID associated with it. Every node has a knowledge of all files in the system and their unique IDs. When a new file is added to the node, it generates a new unique ID and broadcasts it to the rest of the nodes in the system so that they become aware of the new file.

Lookup a node that has a particular file	The tracker tracks nodes and files, and is able to do a quick lookup and send a list of nodes back to the requesting peer. The unique file ID is used in the lookup operation.	The peer broadcasts a message to all nodes. A node that replicates a certain file replies back the the node that sent the request.
File change and latest version	Every file has an associated checksum. The tracker also tracks the golden checksum for every file as well as a flag if the file on a particular node is pending to be updated. Any node that has a file checksum not matching the golden one is subject to have the file updated at earliest opportunity.	Every peer also stores the golden checksum information locally. When the file gets updated on a peer it computes the golden checksum and notifies all other peers through a broadcast of the change. Having the golden checksum information, a peer can decide whether it has the most up-to-date version of the file or not.

It was decided against the use of any sort of time stamps within the system to synchronise data such as file versions. Time synchronisation is a difficult and fragile task which should be avoided, if possible. Instead, as it is shown in the table above, as well as in the diagrams of embedded database structures, a checksum is used to synchronise version of file. It provides a robust and an efficient way to achieve synchronisation among the peers.

In an event when the tracker goes back online after being offline for some time, it has to synchronise its state with that of the rest of the peers. Every peer keeps a backlog of events that have occurred while the tracker was offline. When the tracker reconnects, it fetches all of the events from all peers' backlogs and updates its internal state accordingly. After this, the tracker has a synchronised and up-to-date view of the system.

5. Fault Tolerance

The system attempts to provide an efficient and reliable operation. Multiple faults are handled and operation is continued, if possible. The various fault recovery scenarios are discussed below.

Tracker goes offline

If the tracker goes offline, the system is able to sustain the normal operation with the exception of adding new peers. The communication between peers and synchronisation is achieved through direct peer-to-peer connections and message broadcasting, as discussed in the previous two sections.

Tracker crashes

Since the tracker uses a persistent embedded database, it is able to reload its state when it restarts. Any events that have occurred while the tracker was offline are fetched from the nodes. After this, the tracker is synchronised and fully operational again.

The source peer crashes while uploading file to another peer

The receiving peer is able to detect a crash by a broken network connection. If the source of the file cannot be reached after several attempts, the peer deletes any received data. The tracker monitors the level of replication per file within the system, and prompts nodes to replicate certain files if the replication level is low. In this case, the tracker may prompt another peer to upload the file to this node. If the file is not available anywhere else in the system, then the only option is to wait until the source peer comes back online.

The receiving peer crashes while downloading file from another peer

The source peer detects the event by the broken network connection. The source peer tries to connect to the crashed peer several time. If the peer is not available, it is the responsibility of the crashed peer to re-request the file again when it comes back online.

The file becomes not available when all replicating peers are offline

The system proactively monitors the situation to prevent the case where the level of replication becomes dangerously low such that there is an imminent possibility of losing the file if some nodes go offline. In this case the tracker prompts the nodes that have the file to push it to other nodes, and hence increase the level of replication. Additionally, a node may push out non-replicated portions of the file before it disconnects.

The source node crashed during the file read

The receiving node attempts to read the file from another node transparently from the user. If there are no other nodes available, an error is returned.

6. System Interface

The system interface is designed to be simple and straightforward. This system is attempting to offer the highest amount of transparency that is possible. The goal is that an application developer is offered the same interface to this distributed filesystem as an interface to a local filesystem. Due to the complexities associated with distributed computing, and the design of this system, there are a few minor changes.

The connection and disconnection interfaces deal with adding and removing a node to or from the system:

connect(string password)

password: an encrypted password string.

The connect function allows a node to connect to the system. Connecting to the system implies that the user will be able to perform file operations. When this function is run, a connection request is sent to the tracker node.

If the tracker is offline, the connecting node will not be able to connect to the system. If the tracker is online, the tracker will check the supplied password.

If the password is correct, the tracker will add the node to its internal state, and respond to the node to let it know that it is now included in the system. If the password is incorrect, the tracker will return a response indicating that the connection failed.

disconnect(boolean checkForUnreplicatedFiles=True)

The disconnect function allows an application to disconnect from the system. This function sends a request to the tracker that the node wishes to disconnect. The tracker then makes a choice depending on the checkForUnreplicatedFiles parameter.

If checkForUnreplicatedFiles is true, the tracker then checks to see if there are any unreplicated files on the node. If there are unreplicated files or parts of files, the tracker sends a response indicating this. The node then pushes any unreplicated files to another peer that is connected.

If checkForUnreplicatedFiles is false, then the tracker disconnects the node regardless of whether or not there are replicated files.

The function also handles the case where a tracker is offline. In this case, the node broadcasts the change of state message to all other nodes.

Once connected, a node can perform file operations using the following functions:

read(string filePath, long startOffset=None, long length=None)

The read function allows an application to read the contents of a file. First, the read operation checks if the file is currently stored locally. If it is, the file is opened using standard operating system methods, read, then returned to the user as a string.

The logic of the write operation is as described in Table 1.

The startOffset and endOffset parameters define which part of the file to read. startOffset refers to the number of characters from the beginning the file to skip before starting to read. length is the total number of characters to read.

By default, these parameters are set to None which means that the full file will be read.

write(string filePath, string data, long startOffset=None)

filePath: The path of the file.
data: The data to write to the file.
startOffset: the number of characters from the beginning the file to skip before starting to write.

The write operation allows writing to a file. First, the read operation checks if the file is currently stored locally. If it is, the file is simply opened using standard operating system methods, and written to.

The logic of the write operation is as described in Table 1.

If the file path does not currently exist, but the directories in the path are valid, a file will be created.

delete(string filePath)

filePath: The path of the file.

The delete operation allows the deletion of a file from the system. When a user calls delete, the node requests a deletion from the tracker. The tracker checks to ensure that the deletion is valid (ex. the file exists, etc.) and then returns whether or not the deletion is valid to the requesting node.

If the deletion is valid, the node then requests a peer list from the tracker and sends *delete* messages to all peers that have a copy of the file. When a peer receives a delete message, it deletes the file from its local storage.

move(string sourcePath, string destinationPath)

sourcePath: The current path of the file to move.
destinationPath: The desired path of the file after the move operation.

The move operation allows the user to move a file from one directory to another. When a user calls this function, the node sends a *move_request* message to the tracker. The tracker then checks if the move is valid (ex. the source and destination directories exist, etc.). After this, the tracker responds with a *move_response*.

If the move is valid, the node then sends a *move* message to all peers that have a copy of this file. When a peer receives the *move* message, it moves the file on its local file system.

list(string directoryPath=None)

directoryPath: The directory to search. If None, defaults to the root directory

The list operation lists all the files in a single directory. When this function is called, a *list_request* message is sent to the tracker. The tracker then looks in its internal storage and returns the list of files based on the directory specified in the *list_request*. The tracker then responds with a *list* message that contains the path to the root directory as well as a list of files within that directory.

If the tracker is down, *list_request* messages are sent to all peers in the system. The final file list that is returned to the user contains the union of all files that are returned by each peer.

archive(string filePath)

filePath: The path of the file to archive.

The archive operation allows archiving and versioning a file so it cannot be changed. When this method is called, an *archive_request* is sent to the tracker. The tracker then ensures that the request is valid, archives and creates a new version of the file, and sends an *archive_response* to the requesting peer.

If the archiving was successful, the tracker then sends *new_file_available* messages to peers that it determines should replicate the new archive.

Messaging Protocol

Each message type in this section is formatted with ***bold italics*** for clarity.

Common

peer_list_request

A peer list request is sent to the tracker when a node needs to know which peers currently have a specific file.

This message contains a string that represents the file path of interest. If the file path string is empty, the response will contain all peers currently connected to the system.

peer_list

This is often returned from the tracker. It is a message that contains a list of peers. This includes information about the peer number and endpoint address.

file_download_request

This is a request to download a file. The message contains the file id.

file_download_decline

This is a response to a file_download_request in the case that the file cannot be downloaded. This is typically only used in special cases such as the file is not available (i.e. the system is out of sync), the file is currently changing, or in other corner cases. This will not be used often and is meant to provide a fast method of declining a file_download_request as opposed to waiting for a timeout (which can be lengthy depending on the system's setup parameters).

This message will also be used in the case of a broadcasted file_download_request when the tracker is down.

file_data

This is the actual data of a file and is often the response of a file_download_request. This message contains the file's path, checksum, and the entire file itself.

connect()

connect_request

This message is sent when a user calls the connect() function. It is sent to the tracker which handles the authorization. This request contains the password that the user supplied to the connect function.

connect_response

This message is returned by the tracker after a *connect_request* is received. It contains a boolean indicating whether or not the connection was successful.

disconnect()

disconnect_request

This message is sent when a user calls disconnect(). It is sent to the tracker and contains one boolean indicating whether or not the tracker should check for unreplicated files.

disconnect_response

This message is sent by the tracker in response to a *disconnect_request* message. It contains a boolean indicating whether or not the node should wait to replicate files. If the wait value is false, the node can disconnect immediately. If the wait value is true, the node will wait for another *disconnect_response* where the wait value is false.

write()

file_changed

After write() is called (and the peer list is returned from the tracker) a node will send this message to the tracker and all peers that have a copy of this file. The tracker will use this to update its storage and the system state.

This message contains:

the file path
the file checksum
the new file data
the write function's startOffset parameter

validate_checksum_request

After a peer has changed a file in response to a file_changed message, this message is sent to the tracker. This message contains the file path and file checksum.

validate_checksum_response

This message will be sent by the tracker back to a node that sent a validate_checksum_request. This message contains a boolean representing whether or not the checksum matched the golden checksum.

new_file_available

If a call to write() creates a new file, this message will be sent to the tracker. This message contains all the information about the new file including the path, the checksum, and the data. The tracker will broadcast this message to all peers. If the tracker is offline, the node broadcasts the message to all peers.

delete()

delete_request

This message is sent to the tracker when a user calls the delete() function. The message contains the file id.

delete_request

This message is sent from the tracker in response to a delete_request. This response contains a boolean representing whether or not the file can be deleted.

delete

After a delete_response returns that it is ok to delete a file, this message is sent to each peer that has the file. This message contains the id of the file to be deleted

move()

move_request

This message is sent to the tracker when a user wants to move a file. The message contains the file id and the destination path.

move_response

This message is returned from the tracker in response to a move_request. This message contains a boolean representing whether or not the move can be done.

move

After a move_response returns that it is ok to move a file, this message is sent to each peer that has the file. This message contains the file id and the destination path.

list()

list_request

This message is sent to the tracker when a user calls the list() method. This message contains the path of the directory that the user wishes to be listed. If this path is blank, the root directory is assumed.

list

This message is a response to a *list_request* message. This message contains the list of files and the path of the directory in which the files are located.

archive()

archive_request

This message is used when a user calls the archive() function. It is sent from a node to the tracker as the tracker is responsible for handling versioning. This message contains the file path of the file the user wishes to archive.

archive_response

This message is the response to an *archive_request*. It is sent from the tracker back to the node that initiated the *archive_request*. This message contains a boolean representing whether or not the file was archived.

7. Other design issues

Processes

Each node, only needs one process to run. This process handles communication between nodes, as well as accessing the local storage device. The process will handle sending and

receiving messages, and reading and writing from disk. In order to ensure non-blocking performance, the process will have to have at least three threads: one for sending messages, one for receiving and handling incoming messages, and one for reading and writing to the local storage device.

Every node, including the tracker runs an embedded database. However, it runs in the same process, and is just a library that used to handle sql queries to update database files.

Process Communication

Since the main process (that handles communication and file system access) is run in one process, IPC is not a concern.

Replication and File Consistency

The replication technique that is used in this system is based largely on how often a client is connected to the system. The assumption made is that a client that is connected to the system for the most total time will probably continue to be online fairly often. If a peer is online then it can provide files to peers that need to download files. So, replicating files on servers that are often online can ensure that most files are available when they are needed.

A user of the system can also customize the amount of replication in the system. A user can change the desired number of copies of a file.

Security

When joining the system, a node needs to provide the system password. This password is stored on the tracker node. If the password is incorrect, the tracker will not allow the node to connect and will not perform and of the standard connection routines (i.e. adding the node to the list of connected nodes, responding to requests, etc.).

If a node (or malicious user) that is not connected attempts to perform any file operations on any peer, the peer will know that the requester is not connected and will ignore the request.

In order to secure the password on a connection attempt, the connection will use TLS security. The public key will be hosted on the tracker.

8. Others

Limitations

The limitations of the system are described in Section 1: Introduction.

Future improvements

Tracker Migration

In the future, it is desirable to be able to migrate the tracker information to another peer. If the tracker has a problem, the system will still be completely functional and synchronized and new peers will still be able to connect.

Chunking

Currently, the system sends full files as a single message from one peer. For large files, this can be a performance and reliability problem, and does not take full advantage of the distributed system. If a file exists on multiple peers, it could be faster to download different parts of the file from different peers. This bittorrent-style file sharing approach is a desirable performance and reliability improvement for the system.

Resuming Failed Downloads

Currently, if a peer disconnects while downloading/uploading, the file is flushed and it has to be fetched all over again from another node. This has negative performance and bandwidth implications. Allowing a node to finish the download from the point where it was stopped would be a great reliability and performance improvement.

Recursive File Operations

Currently each operation is performed on a single file or directory. If an entire directory with many files needs to be moved, or deleted, this would require many requests and could flood the network. Also, a user may want to move or delete a directory and all files and directories contained within with one function call. Also, listing a directory tree may be desirable. Allowing recursive file operations would solve the network flooding problems, and allow a more simple, yet rich feature set.