Distributed system for storing files

Sava Ivković

June 12th, 2024

1. Introduction

This document describes a TCP/IP distributed system with adaptive architecture which provides fault-tolerant version control of files. This document gives a detailed overview of the system as well as guidelines for its implementation and usage. Implementation of this system is heavily based on the Chord protocol.

2. Node configuration

Before joining the system, node needs to set up its parameters, which consist of:

- root: Absolute path to the storage of the node
- ip: Ip address of the node
- port: port of the node
- bs.port: port of the Bootstrap server
- weak_failure_limit: time limit to mark a node suspicious
- strong_failure_limit: time limit to remove a node from the system

Note: all nodes need to have the same weak and strong failure limit, and bootstrap ip address is currently set to "localhost" as default (subject to change). All files in the root ***WILL BE DELETED*** *during startup!*

3. Starting the system

To start the system, bootstrap server needs to be up and running before adding any nodes. Currently, bootstrap is started using the MultipleBootstrapServer.java class, and you need a special servent_list.properties file that contains:

- servent_count: number of servents that are started with the system at the start
- bs.port: bootstrap port

You will also need a special serventX.properties files, where X is the index of the servent (index goes from 0 to Servent_count-1)

4. Token system

For a node to be added to the system, or when a node wants to gracefully leave the system, it will first need a token to do so. Token is unique for the whole system, and only one node can have the token at a time. In case a node doesn't have a token, and wants to enter/exit the system, it will send a broadcast message requesting the token to each node it is aware of.

For the token system, the Suzuki-Kasami algorithm is implemented

5. Suzuki-Kasami algorithm

Each node contains versions of every node in the system. Version is a number that says how many times a node has requested a token.

Token contains in itself a queue of which nodes are currently waiting to get the token, and a version map of all nodes (how many times a node entered a critical section/requested a token).

Every time a node needs a token to enter a critical section, it takes its current version, increments it by one, and sends this new number to all other nodes. When a node receives a token request, two options are possible:

1) Node contains the token
2) Node doesn't contain the token

In case node doesn't contain the token, it will just update its local counter to the new value.

If a node contains a token, first it will check whether or not it is currently in a critical section. If it is, it will wait to exit the critical section before sending the token.

Once when a node contains a token, and is not in a critical section, it will compare its local version to the tokens, and will add to the queue only ones that are one behind in the token.

Node that is on top of the queue will get the token.

6. System organization

System is organized using the Chord algorithm. Every node gets a unique servant id, which is calculated by taking a string value ipAddress:Port, and is hashed by using the SHA-1 algorithm. Every node is aware of every node in the system, but nodes won't communicate directly because we want to avoid heatmaps in the system.

Chord functions by sorting all nodes by chord Id, and then starting from node X, it calculates which node is next to us, 2 steps, 4 steps, etc. This way, we can ensure every node is seperated in $\log_2 n$ steps, where n is the total number of nodes.

When data is requested from node X to node Y, node X will try to jump as close as possible to node Y, but not overshooting it.

7. Friend system

Any two nodes can become friends.

Node X that wants to be a friend with node Y will send a NewFriend message to the address of node Y. In case there is a node on the given address, node X and Y will become friends (both ways). Friend system is used for private sharing of files (explained in section 8).

8. Adding and removing files from the system

To add a new file, user needs to specify file path (absolute file path) which will be added. This can be started from any node.

Once when node receives a new file message, it will check whether or not it belongs to him (using hashed name of the file). If it does, node will check whether or not this file already exists, and will store it if the name is unique. If it does not belong to him, node will try to send it to the node closest to the hashed name of the file.

File can be shared in a private or a public mode, when file is public anyone can see it, if its private, then only the node that initiated the creation of the file and his friends can access it.

Similarly, the system works with removing files, if the hash belongs to him, node will remove it from the system if it exists, if not, node will send it to the node closest possible. In case a file is private, then only the node that created it and his friends can remove it.

9. Getting files from a node

Any node can request to get files from any node.

If a file is public it will be added regardless, if it is private, information about it will only be sent if the node requesting data is a friend with the node that created it.

When a node sends a request for files, it also sends his list of friends, so the node that is being asked can know which files to hide from it.

10. Adding a new node to the system

New node can request to join the system at any time, but the process won't start until the node gets the token. Workflow of adding a new node (in case it is not a first node in the system):

- New node contacts bootstrap
- Bootstrap returns address of a random node in the system
- New node sends a PING message to the given node
- Node in the system returns a PONG message, containing all nodes
- New node sends a broadcast TOKEN_REQUEST to all the nodes in the system
- New node gets the token from a node in the system
- New node sends a request to get put into the system
- New node is inserted into the system, with the information about all the nodes
- New node request files that belong to him from his successor

When a new node starts the process of inserting, first the system finds the node the new node belongs to, after which a circular UPDATE message is sent between nodes, which contains info about the new node, and every node in the system (when a existing node gets a UPDATE message, it puts his info before sending it to the next node).

There is a special case where multiple nodes want to join the system at the same time, and are not aware of each other. To make sure a new node is aware of nodes that requested nodes in the meantime (but did not send a request message to the new node), during the UPDATE phase token version maps are updated as well (in case a node does not contain a key that is present in UPDATE message, it will add it). This way we make sure new nodes are aware of each other.

If a node doesn't receve a PONG message within STRONG_FAILURE_LIMIT + WEAK_FAILURE_LIMIT * 3/2, it will request a new node from bootstrap, and bootstrap will pronounce this node as dead (will remove it from list of nodes).

If a node already exists with the chordId the new node is requesting (same Ip and port), new node will get a Sorry message, and will not be put in the system.

11. Removing a node from the system gracefully

Node can request to be removed from the system at any time, however to actually start the process of removing itself, it will need a token. Workflow of removing a existing node:

- Node sends a TOKEN_REQUEST message to each node
- Node receives the token
- Node sends a STOPPED message to its neighbor
- Node receives STOPPED message from its predecessor
- Node sends his files and token to his neighbor
- Node shuts down

When a node that is not shutting down receives a STOPPED message, it will remove the node that originally sent the message, and will send the same message to his neighbor.

12. Failure detection

Node sends to every node a HEARTBEAT message every WEAK_FAILURE_LIMIT/2 milliseconds. In case a node hasn't received a HEARTBEAT message from another node for WEAK_FAILURE_LIMIT milliseconds, we will flag this node as suspicious. However, this node is still not removed from the system.

Node will send a NodeCheck message to another healthy node, and will not remove it from the system until he gets a confirmation from another node that the node is dead and until STRONG_FAILURE_LIMIT has passed.

When a node finds out another node has died, there is a special process that happens to determine whether or not the dead node had the token.

Whenever a node sends a token, he keeps a local copy of how it looked like before he sent it, as well as who he sent it to.

Node sends a request to every node to send its data about token. If any token sends that he has the token, this process is canceled. If no one has a token, node will compare its last version compared to everyone else. In case it has the latest version, it will check whether or not he sent it to the dead node. If he did send it to a dead node, he will be granted the token, which will have versions like his last version.

13. Backup of the system

For every node, there is a backup on its predecesor and successor. Whenever a node receives a new File, or Delete message, node sends it also to its backups to keep up-to-date info. Whenever a message is sent, node expects a OK message in return to confirm a message is processsed

(Note that OK messages are sent only for messages that change state of the system, or getting information from a node). In case a node dies, every message that has not been confirmed is sent again (to the node next to the one that died).

When a node dies, its successor will take its data, and will send new backups to the new predecesor and successor.

In a special case, where two nodes in a row die, there is a special message type sent from the predecesor of a dead node, which will send files from the backup two nodes forward. When a node receives this message, it will sleep for WEAK_FAILURE_LIMIT + STRONG_FAILURE_LIMIT, after which it will check whether or not those files belong to him (they will only belong if two nodes in a row died).

14. Model (special data structures used in the system)

This section will cover some data structures that are used later in messages

- Servant Info
    - ipAddress (String)
    - listenerPort (Integer)
    - chordId (String)
- Network file
    - Name (String) – file name
    - Owner (String) – used for private sharing
    - hashedName (String) - used for chord
- Data file
    - NetworkFile (NetworkFile) – file that is being transferred
    - Bytes (byte[]) – data of file being transferred
- Token
    - VersionVector (Map<ServentInfo, Integer>) – contains versions of each node
    - Queue (Queue<ServentInfo>) – nodes that are currently waiting for the token
    - inUse (boolean) – whether or not we are in a critical section

15. Messages in the system

This section contains all the messages that are sent through the system. Note that all messages require a sender ip address and port, as well as receiver ip address and port

- Backup (13)
    - networkFile (NetworkFile) – metadata of the file being backed up
    - bytes (byte[]) – bytes of the file
- Delete (8)
    - FileName (String) – name of the file that is being removed from the system

- Error – used when file that is being added already exists, file that is being removed doesn't exist, and when someone wants to add a friend that doesn't exist
    - errorText (String) – description of the error

- File (8)
    - networkFile (NetworkFile) – metadata of the file being sent
    - bytes (byte[]) – bytes of the file

- GetLastToken (12)
    - lastToken (Token) – last version of the token
    - iOwnIt (boolean) – true if the node owns the token
    - dead (SeventInfo) – which node died

- Heartbeat (12)

- Info (9)
    - Files (List<NetworkFile)
    - Friends (List<String>)

- NewFriend (7)
    - Text (String) – used only when two friends are confirmed, to send to the original sender that it exists

- NewNode (10)

- NodeCheck (12)
    - isDead (boolean) – true if the node is dead, false otherwise
    - nodeToCheck(ServentInfo) – node that is suspicious

- OK (13)

- Ping (10)

- Pong (10)
    - versionMap (Map<SeventInfo, Integer>) – version map of the node that got the ping, used only to get addresses of all the nodes to send the broadcast token request message

- RemoveFromBackup (13)
    - networkFiles (List<NetworkFile>) – files that no longer belong to the original node

- RequestMyFiles (10)

- Sorry (10)

- Stopped (11)

- Token (5)
    - Token (Token) – token that is being sent

- TokenRequest (5)
    - serventInfo (ServentInfo) – node that is requesting the token
    - version (Integer) – token version number of the node requesting

- TwoNodeBackUp (13)
    - networkFile (NetworkFile) – metadata of the file being sent
    - bytes (byte[]) – bytes of the file

- Update (10)
    - Text (String) – text containing all node addresses, seperated by a comma

16. Node commands

This section contains all the commands that can be requested from a node

Parameter in [] brackets is required, () is optional

- add_friend [node_address] – adds the specified node as a friend
- add_file [path] (private/public) – adds a new file to the system, default is public
- view_files [node_address] – returns all files on the specified address
- remove_file [file_name] – removes the specified file from the system
- stop – stops the node from working, this is graceful and won't be instant