Simplified fault-tolerant topologically optimized version control system

Lazar Jelić

May 21, 2021

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 Kademlia [2] DHT protocol.

2 Identification

Since this system works with objects that are serialized and sent over the network, local pointers are out of consideration. This section explains how entities are distinguished from each other using the SHA-1 [1] hashing algorithm.

Apart from the clearly defined protocol in the following sections, every node in the network must use Java serialization before sending an object over the network. This requirement represents a limitation in a domain of available languages and technologies for creating new nodes from scratch. However, there are libraries specifically made to solve the compatibility problem between Java serialization and other programming languages (e.g. python-javaobj).

2.1 Node identification

Every node in a network is identified by the following parameters:

- Universally unique identifier (UUID)*
- IPv4 address
- Port number
- Team name*

2.2 Message identification

Depending on its type, a message can have additional parameters, but at a bare minimum every message is identified by the following parameters:

- Universally unique identifier (UUID)
- Text body*
- Sender node
- Receiver node*
- Node route*

2.3 File identification

Every file in a network is identified by the following parameters:

- Relative path
- Current version (first 0, second 1, . . . , latest -1)*
- Version history*
- Current content*
- Replica indicator*

Parameters marked with * are not used when comparing entities. These parameters are listed to provide a reader with an idea about what information regarding entities in the system is important to store.

3 Communication protocol

The complexity of this system lies in its communication protocol. This section explains the most important algorithms and procedures that every node in the network must follow to function.

3.1 System configuration

At its initialization stage, a node reads a local configuration file, which has the following properties:

- **Storage path** - Directory of files for which this node is responsible
- **Workspace path** - Directory of recently used files
- **Kademlia constant (k)** - Size of the routing table, Total file replicas
- **Node count** - Total number of nodes in the system
- **Team limit** - Maximum number of nodes before topology is opimized

- **Failure soft interval** - Time interval before a node is checked for failure
- **Failure hard interval** - Time interval before a node is declared failed
- **Replicate interval** - Time interval at which file replication is done
- **Bootstrap IPv4 address** - Predefined bootstrap process address
- **Bootstrap port number** - Predefined bootstrap process socket port
- **Node IPv4 address** - Guarantee that node is on the predefined address
- **Node port number** - Node process socket port
- **Node team name** - Node owner team

3.2 Joining a network

Every node, which wants to join a network, must know predefined parameters for setting up a connection with the bootstrap server. These parameters are given in the local configuration file described above. The bootstrap server is a special-purpose node, which is used only to connect new nodes to the nodes that are already in the network.

When the bootstrap server receives a JOIN_ASK message from a new node, it sorts a list of active nodes by XOR distance to the new node. After checking node collision, the bootstrap sends a JOIN_TELL message with k closest nodes to the new node.

This way, connected nodes form a directed non-complete graph, in which the number of edges adjacent to each node is logarithmically dependent on the total number of nodes in the network. This behaviour is a result of using the Kademlia [2] DHT protocol, although the implementation described in this document follows a slightly different but simpler approach.

3.3 Version control

The core purpose of the system is to provide version control for user-generated files. This task is achieved with special CLI commands, all of which are listed below.

- **add FILE** - Add FILE to the system

  Load the given relative file path from the workspace, initialize its version to 0, and using an ADD message send it to the node that is closest to the key associated with that file.

  Upon receiving this message, a node will check if it is responsible for storing this key and save it to the disk and in-memory lookup table. If the receiving node is not responsible for storing this key, it will recursively redirect the message to the closest node it knows of.

- **remove FILE** - Remove FILE from the system

  Safely remove the given file from the in-memory workspace while keeping the actual file on disk and send it to the node that is closest to the key associated with that file using a REMOVE message. Safe removal is utilized because a malicious attacker can freely perform a mass removal on all available files in the system by automatically entering the remove command for a list of files that are present in the system.

  Upon receiving this message, a node will check if it is responsible for storing this key and remove it only from the in-memory lookup table. If this node is not responsible for storing this key, it will recursively redirect the message to the closest node it knows of.

- **pull FILE VERSION** - Pull VERSION of FILE from the system to the local workspace or pull the latest version if VERSION is not provided

  Retrieve the given file by sending a PULL_ASK message to the node that is closest to the key associated with that file.

  When the remote node receives this message, it will check if it is responsible for storing this key, load the file from the disk, and return a PULL_TELL message to the sender directly, i.e. without intermediate nodes that propagated this message. If this node is not responsible for storing this key, it will recursively redirect the message to the closest node it knows of.

  Upon receiving a response in form of a PULL_TELL message, pulled file will be saved to the disk and in-memory lookup table.

  If the network topology is optimized, the cached version from a team node that contains its copy (3.6) is referenced.

- **push FILE** - Push FILE from the local workspace to the system

  Load the given file from the workspace, check if there are any changes made by comparing the file on disk to its in-memory version from the lookup table, increment file version and send it to the node that is closest to the key associated with that file using a PUSH_ASK message.

  When the remote node receives this message, it will check if it is responsible for storing this key, check for conflicting versions of the file, and return a PUSH_TELL message to the sender directly, i.e. without intermediate nodes that propagated this message. If there is no conflict, the file will be saved to the disk and in-memory lookup table. If this node is not responsible for storing this key, it will recursively redirect the message to the closest node it knows of.

  A response in form of a PUSH_TELL message will be received and if there was a conflict between local and remote file versions, a prompt described below will be shown to the user.

If the network topology is optimized, the cached version from a team node that contains its copy (3.6) is referenced.

If a directory path is passed to the above commands, procedures are done recursively to all files and directories in the given directory.

In case of a push conflict, the following options are available:

- **view FILE** - Show local and remote FILE content
- **pull FILE** - Overwrite local changes with remote FILE content
- **push FILE** - Overwrite remote changes with local FILE content using a PUSH_FORCE message
- **cancel FILE** - Cancel the conflicting operation

3.4 Fault detection

Node failure detection is done in two phases using parameters α and β, which are defined in the configuration file. Firstly, node a performs a ping operation in the background by sending a PING_ASK message and waits α milliseconds, which represents failure soft interval. Secondly, If node b does not respond by sending a PING_TELL message, node a checks to see if node b truly failed. Other possibilities include the PING_ASK message arriving late, the PING_ASK message not arriving at all due to problems in the channel, node b failing and coming back again, etc.

To perform this check, node a sends a CHECK_ASK message to node c, which is one of the nodes that responded to the initial PING_ASK message and is the closest node to node b, and waits β milliseconds, which represents failure hard interval. Node c, upon receiving the CHECK_ASK message, sends a PING_ASK message to node b and waits α milliseconds.

If node b responds with a PING_TELL message, failure is not detected and node a is notified with a CHECK_TELL message. If node b does not respond in the given time frame, node c sends also a CHECK_TELL message but with a certain indicator, signalling that node b failed. When node a receives this message, it removes node b from the list of active nodes to which node a is connected and sends a FAIL message to the bootstrap server, which also removes node b from the list of active nodes.

One important to mention is that node c upon receiving the PING_ASK message will add node a to its list of active nodes. This will ensure that all connections in the network are two-directional, i.e. symmetrical. However, this does not violate the above-mentioned logarithmic dependence.

3.5 Fault tolerance

Fault tolerance is achieved through background data replication running on each node every γ milliseconds, which is a parameter defined in the configuration file. Data replication is necessary for at least two reasons. Firstly, data loss can occur in case of a node failure. Secondly, data needs to be redistributed when new nodes join the network.

To illustrate the second scenario, let's suppose that there is a node a in the network that holds a file associated with a key κ. This is because node a is at the moment the closest node to the given key based on the XOR distance. At any given moment in the future, node b, which is even closer to the key κ then node a, may join the network. If this scenario holds, node a should replicate the file associated with the key κ on node b.

The data replication procedure is fairly simple. Node a has a special thread running in the background that loops through all files that are closest to node a and sends REPLICATE messages to other $k - 1$ closest nodes. Upon receiving a REPLICATE message, node b saves the file to the disk and in-memory lookup table.

3.6 Topology optimization

The system adapts through time as nodes tend to work more frequently with a certain subset of data. Local nodes that are constantly pulling the same data from remote nodes are going to receive their copy of that data, while all future pull/push operations are directed only to the main node in the team.

Suppose that there are three nodes a, b, and c and that node a, which is geographically distanced from both node b and node c, has a key κ. This is a case in the real-world application of this system in which two remote teams work on the same files.

For this reason, let's suppose that nodes b and c are in one team and node a is in its team. Instead of both node b and node c frequently asking node a for the value of the key κ, node a keeps track of the total number of distinct nodes from the same team working on a single key-value pair, i.e. file. If this number becomes greater than the predefined upper bound, node a replicates that file on just one of the nodes from the team.

When node c in the future wants to work with key κ, it will pull from and push to only node b, which now contains a copy of that key-value pair. For this purpose, node c needs to have a cache, which maps keys to nodes on which those keys are copied. This map is queried each time node c wants to pull/push key-value pairs to the network. Synchronization with the remote version of key κ on node a can be done only on node b via pull/push commands.

Let's end this section with a specific communication protocol that follows the above-mentioned algorithm. Node a will keep track of all PULL_ASK messages received for every file in its storage. Because b and c attach their team's name to PULL_ASK messages, node a can easily calculate the distinct number of team members that are accessing the same file. When this number becomes greater than the predefined limit in the configuration file, a random node from that team is selected and named as a team leader node. Node a sends a REPLICATE message to the team leader node, which stores the received file. Other team nodes receive a REDIRECT message from node a containing team leader node information and the actual file that has been referencing. Upon receiving a REDIRECT message, team nodes will add (file, team leader node) pair to the cache map, which will be referenced when team nodes are trying to send PULL_ASK or PUSH_ASK messages to the node a for the specific file. If this map happens to have that file, the message receiver is set to the team leader node instead of node a.

4 Messages

The following messages were used to make a detailed explanation of the communication protocol in the previous section. For details regarding File and Node data types, refer to sections 2.1 and 2.3.

- **ADD** (3.3)
  - File file - File to be added to the network
- **CHECK_ASK** (3.4)
  - Node node - Potentially failed node that need to be checked
- **CHECK_TELL** (3.4)
  - Node node - Potentially failed node that was checked
  - bool active - True if that node did not fail
- **FAIL** (3.4)
  - Node node - Node that failed
- **JOIN_ASK** (3.2)
  - array⟨Node⟩ nodes - Nodes closest to the node joining the system
- **JOIN_TELL** (3.2)
  - bool joined - True if no hash collision is detected between new node and existing nodes in the network
- **PING_ASK** (3.4)
  - bool check - True if it is sent as a result of CHECK_ASK message

- **PING_TELL** (3.4)
  - bool check - True if it is sent as a result of CHECK_ASK message
- **PULL_ASK** (3.3)
  - File file - File to be pulled to the workspace
- **PULL_TELL** (3.3)
  - File file - File to be pulled to the workspace
- **PUSH_ASK** (3.3)
  - File file - File to be pushed to the network
- **PUSH_TELL** (3.3)
  - File local - Local file to be pushed to the network
  - File remote - Remote file that already exists in the network
  - bool conflict - True if remote file version is greater than or equal to the local file version
- **PUSH_FORCE** (3.3)
  - File file - File to be force pushed to the network, overwriting remote changes
- **REDIRECT** (3.6)
  - Node node - Team node that contains a copy of the remote file
  - File file - File that is copied on the team node
- **REMOVE** (3.3)
  - File file - File to be removed from the network
- **REPLICATE** (3.5, 3.6)
  - File file - File to be replicated
  - bool topology - True if topology is being optimized (3.6) and false if data is being replicated (3.5)

5 Usage

1. Grant execution permission to the runner script

   ```
   chmod +x run.sh
   ```

2. Run the runner bash script for bootstrap

   ```
   ./run.sh 0
   ```

3. Run the runner bash script for node

   ```
   ./run.sh NODE_ID
   ```

4. Repeat the previous step for each node in the network

6 Resources

Source code: `https://github.com/jelic98/raf_kids/tree/master/project`

References

[1] D. Eastlake and P. E. Jones. Us secure hash algorithm 1 (sha1), 2001. `https://www.rfc-editor.org/rfc/pdfrfc/rfc3174.txt.pdf`.

[2] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric, 2002. `https://doi.org/10.1007/3-540-45748-8_5`.