COMP2207: Distributed File System coursework

Leonardo Aniello, Kirk Martinez

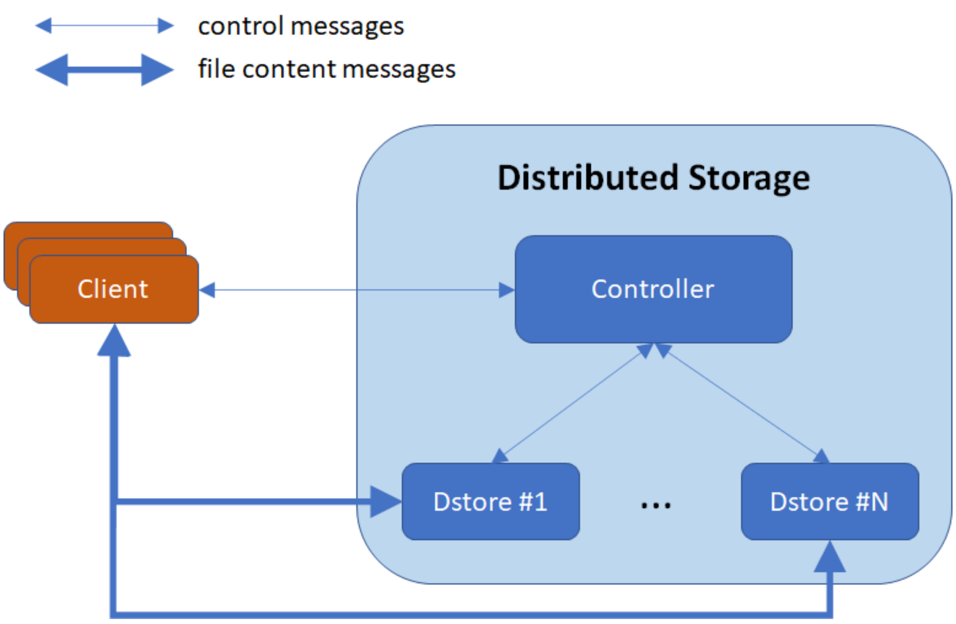
Course:	COMP2207
Document version:	1.1 – April 26, 2023

1 Introduction

In this coursework you will build a distributed storage system. This will involve knowledge of Java, networking and distributed systems. The system has one Controller and N Data Stores (Dstores). It supports multiple concurrent clients sending store, load, list, remove requests. You will implement Controller and Dstores; the client will be provided. Each file is replicated R times over different Dstores. Files are stored by the Dstores, the Controller orchestrates client requests and maintains an index with the allocation of files to Dstores, as well as the size of each stored file. The client actually gets the files directly from Dstores – which improves scalability. For simplicity, all these processes will be on the same machine, but the principles are similar to a system distributed over several servers. Files in the distributed storage are not organised in folders and sub-folders. Filenames do not contain spaces.

The Controller is started first, with R as an argument. It waits for Dstores to join the storage system (see Rebalance operation). The Controller does not serve any client request until at least R Dstores have joined the system.

As Dstores may fail and new Dstores can join the storage system at runtime, rebalance operations are required to make sure each file is replicated R times and files are distributed evenly over the Dstores.



2 Networking

Controller, Dstores and Clients will communicate with each other via TCP connections. Because they will be on the same machine, the Dstores will listen on different ports.

Each client will submit requests to the Controller sequentially over a separate TCP connection.

The Dstores will establish connections with the Controller as soon as they start. These connections will be persistent (i.e., they are expected to be kept alive for as long as the Dstore is running). All the communications between a Dstore and the Controller must take place over that connection; no further connections must be established between a Dstore and the Controller. If the Controller detects that the connection with one of the Dstores dropped, then such a Dstore will be removed from the set of Dstores that are part of the storage system.

Processes should send textual messages (e.g., LIST – see below) using the *println()* method of *PrintWriter* class, and receive using the *readLine()* method of *BufferedReader* class. For data messages (i.e., file content), processes should send using the *write()* method of *OutputStream* class and receive using the *readNBytes()* method of *InputStream* class.

3 The Index

The index refers to the data structure used by the Controller to keep track of stored files. As Store and Remove operations involve a number of messages to be completed (see Section 4), it is important to ensure that other possibly conflicting concurrent operations are served properly. To achieve that, the index data structure should include a dedicated field for each file to record its current state.

For example, while a file F is being stored (i.e., corresponding index entry updated with state set to “store in progress”), we do not want the storage system to serve any Load or Remove operations on F, nor to include F when List operations are invoked. In this sense, it should be as if F does not exist yet. However, if another concurrent Store operation is requested for another file with the same name of F, then we want to reply with an ERROR ALREADY_EXISTS message. Handling this kind of situations requires to explicitly manage the lifecycle of files, e.g., from "store in progress" to "store complete" to "remove in progress" to "remove complete". The expected behaviour of the storage system in these situations is defined at the end of Section 4.

4 Code development

Only use Java openjdk-17-jdk, on Linux/Unix. Do not use Windows. The code must be testable and not depend on any IDE directory structure/config files.

Command line parameters to start up the system:

```
Controller:   java Controller cport R timeout rebalance_period
A Dstore:     java Dstore port cport timeout file_folder
A client:     java Client cport timeout
```

The Controller is given a port to listen on (`cport`), a replication factor (`R`), a timeout in milliseconds (`timeout`) and how long to wait (in seconds) to start the next rebalance operation (`rebalance_period`).
A Dstore is started with the port to listen on (`port`) and the controller's port to talk to (`cport`), timeout in milliseconds (`timeout`) and where to store the data locally (`file_folder`). Each Dstore should use a different path and port, so they don't clash.
The client is started with the controller port to communicate with it (`cport`) and a timeout in milliseconds (`timeout`).

Controller and Dstores do not need to keep any state between different executions. This means the Controller does not need to save/load the index to/from disk, and Dstores should empty their file folder at start up.

The timeout should be used when a process expects a response from another process; for example, when the Controller waits for a Dstore to send a STORE_ACK message (see below Store operation). Timeouts should not be used in other circumstances; for example, when the Controller waits for a Client to send a request.

Store operation

- Client -> Controller: `STORE filename filesize`
- Controller
  - updates index, “store in progress”
  - Selects R Dstores, their endpoints are port1, port2, …, portR
  - Controller -> Client: `STORE_TO port1 port2 … portR`
- For each Dstore i
  - Client->Dstore i: `STORE filename filesize`
  - Dstore i -> Client: `ACK`
  - Client->Dstore i: `file_content`
  - Once Dstore i finishes storing the file,
    Dstore i -> Controller: `STORE_ACK filename`
- Once Controller received all acks
  - updates index, “store complete”
  - Controller -> Client: `STORE_COMPLETE`

Dstores might be terminated during this operation. You can assume all files are not empty (i.e., file size is always greater than zero) and their size is lower than 100KB.

Failure Handling

- Malformed message received by Controller/Client/Dstore
  - Ignore message (it would be good practice to log it)
- If not enough Dstores have joined
  - Controller->Client: `ERROR_NOT_ENOUGH_DSTORES`
- If filename already exists in the index
  - Controller->Client: `ERROR_FILE_ALREADY_EXISTS`
- **Please ignore this bullet point** ~~Client cannot connect or send data to all R Dstores~~
  - ~~No further action, the state of the file in the index will remain “store in progress”; future rebalances will try to sort things out by ensuring the file is replicated to R Dstores~~
- If the Controller does not receive all the acks (e.g., because the timeout expires), the `STORE_COMPLETE` message should not be sent to the Client, and `filename` should be removed from the index

Load operation

- Client -> Controller: `LOAD filename`
- Controller selects one the R Dstores that stores that file, let port be its endpoint
- Controller->Client: `LOAD_FROM port filesize`
- Client -> Dstore: `LOAD_DATA filename`
- Dstore -> Client: `file_content`

Dstores might be terminated during this operation.

Failure Handling

- Malformed message received by Controller/Client/Dstore
    - Ignore message (it would be good practice to log it)
- If not enough Dstores have joined
    - Controller->Client: `ERROR_NOT_ENOUGH_DSTORES`
- If file does not exist in the index
    - Controller -> Client: `ERROR_FILE_DOES_NOT_EXIST`
- If Client cannot connect to or receive data from Dstore
    - Client -> Controller: `RELOAD filename`
    - Controller selects a **different** Dstore with endpoint port'
    - Controller->Client: `LOAD_FROM port' filesize`
    - If Client cannot connect to or receive data from any of the R Dstores
        - Controller->Client: `ERROR_LOAD`
- If Dstore does not have the requested file
    - Simply close the socket with the Client

Remove operation

- Client -> Controller: `REMOVE filename`
- Controller updates index, “remove in progress”
- For each Dstore i storing filename
  - Controller->Dstore i: `REMOVE filename`
  - Once Dstore i finishes removing the file,
    Dstore i -> Controller: `REMOVE_ACK filename`
- Once Controller received all acks
  - updates index, “remove complete”
  - Controller -> Client: `REMOVE_COMPLETE`

Dstores might be terminated during this operation.

Failure Handling

- Malformed message received by Controller/Client/Dstore
  - Ignore message (it would be good practice to log it)
- If not enough Dstores have joined
  - Controller->Client: `ERROR_NOT_ENOUGH_DSTORES`
- If filename does not exist in the index
  - Controller->Client: `ERROR_FILE_DOES_NOT_EXIST`
- Controller cannot connect to some Dstore, or does not receive all the ACKs within the timeout
  - No further action, the state of the file in the index will remain "remove in progress"; future rebalances will try to sort things out by ensuring that no Dstore stores that file
- If Dstore does not have the requested file
  - Dstore -> Controller: `ERROR_FILE_DOES_NOT_EXIST filename`

List operation

- Client->Controller: `LIST`
- Controller->Client: `LIST file_list`
  - `file_list` is a space-separated list of filenames

Dstores might be terminated during this operation.

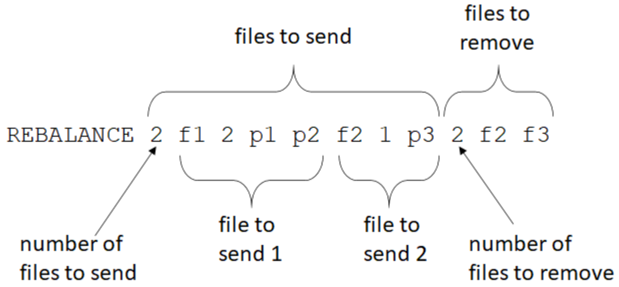
Failure Handling

- Malformed message received by Controller/Client
  - Ignore message (it would be good practice to log it)
- If not enough Dstores have joined
  - Controller->Client: `ERROR_NOT_ENOUGH_DSTORES`

Storage Rebalance operation

This operation is started periodically by the Controller (i.e., based on the `rebalance_period` argument) **and** when a new Dstore joins the storage system. In the latter case, this is the message (where *port* is the endpoint of the new Dstore)
Dstore -> Controller: `JOIN port`

- For each Dstore i
    - Controller -> Dstore i: `LIST`
    - Dstore i -> Controller: `LIST file_list`
- Controller revises file allocation to ensure (i) each file is replicated over R Dstores, and (ii) files are evenly stored among Dstores
    - With N Dstores, replication factor R, and F files, each Dstore should store between floor(RF/N) and ceil(RF/N) files, inclusive
- Controller produces for each Dstore i a pair (`files_to_send`, `files_to_remove`), where
    - `files_to_send` is the list of files to send and is in the form `number_of_files_to_send file_to_send_1 file_to_send_2` … `file_to_send_N`
    - and `file_to_send_i` is in the form `filename number_of_dstores dstore1 dstore2` … `dstoreM`
    - `files_to_remove` is the list of filenames to remove and is in the form `number_of_files_to_remove filename1 filename2` … `filenameL`
- For each Dstore i
    - Controller->Dstore i: `REBALANCE files_to_send files_to_remove`
        - Example
            - Assume that (where pi is the port where Dstore i is listening on)
                - file f1 needs to be sent to Dstores p1 and p2
                - file f2 needs to be sent to Dstore p3
                - file f2 needs to be removed
                - file f3 needs to be removed
            - `REBALANCE 2 f1 2 p1 p2 f2 1 p3 2 f2 f3`



    - Dstore i will send required files to other Dstores, e.g., to send a file to Dstore j
        - Dstore i -> Dstore j: `REBALANCE_STORE filename filesize`
        - Dstore j -> Dstore i: `ACK`
        - Dstore i -> Dstore j: `file_content`
    - Dstore i will remove specified files
    - When rebalance is completed
      Dstore i -> Controller: `REBALANCE_COMPLETE`

Additional notes on Rebalance operations

- The first rebalance operation must start `rebalance_period` seconds after the Controller started
- Clients' requests are queued by the Controller during rebalance operations; these requests will be served once the rebalance operation is completed
- A rebalance operation should wait for any pending STORE and REMOVE operation to complete before starting
- At most one rebalance operation should be running at any time
- Dstores will not be terminated during this operation (but might fail)
- If it turns out that the index includes a file that no Dstore included in the list sent to the Controller, then it would be safe to remove this file from the index

Failure Handling

- Malformed message received by Controller/Dstore
  - Ignore message (it would be good practice to log it)
- Controller does not receive REBALANCE COMPLETE from a Dstore within a timeout
  - No further action; future rebalance operations will sort things out

Concurrent Operations

- Ongoing operation: Store file
  - If a concurrent Store operation on the same file is received, then return ERROR_FILE_ALREADY_EXISTS
  - If a concurrent Load operation on the same file is received, then return ERROR_FILE_DOES_NOT_EXIST
  - If a concurrent Remove operation on the same file is received, then return ERROR_FILE_DOES_NOT_EXIST
  - If a concurrent List operation is received, then do not include file in the list to return
- Ongoing operations: Remove file
  - If a concurrent Store operation on the same file is received, then return ERROR_FILE_ALREADY_EXISTS
  - If a concurrent Load operation on the same file is received, then return ERROR_FILE_DOES_NOT_EXIST
  - If a concurrent Remove operation on the same file is received, then return ERROR_FILE_DOES_NOT_EXIST
  - If a concurrent List operation is received, the do not include file in the list to return

5 Submission Requirements

- Your submission should include the following files:
  - Controller.java
  - Dstore.java

  As well as all the additional .java files you developed
- These files should be contained in a single zip file called <your username>.zip
- There should be no package structure to your java code
- When extracted from the zip file, the files should be located in the current directory
- These files will be executed at the Linux command line by us for automatic testing

6 Marking Scheme

You are asked to implement the Controller and Dstores. You will be given the client, as an obfuscated jar. The client allows the execution of operations via a terminal.

- Up to 50 marks are awarded based on whether the storage system works in compliance with the protocol and correctly serves sequential requests from a single client
- Up to 10 marks are awarded based on whether each file is replicated R times and files are evenly spread over the Dstores (only when stored, not when Dstores fail or new Dstores join the storage system)
- Up to 10 marks are awarded based on whether the storage system correctly serves concurrent requests from more clients (up to 10 concurrent clients)
- Up to 10 marks are awarded based on whether the storage system correctly tolerates the failure of one Dstore
- Up to 10 marks are awarded based on whether the storage system correctly tolerates the failure of up to N-R Dstores
- Up to 10 marks are awarded based on whether files are evenly spread over the Dstores despite Dstores failing and new Dstores joining the storage system

7 Code development suggestions

There are various things to develop step-by-step. This includes making TCP connections and passing data to/from, implementing timeouts for when the communication is broken, and so on. This is a good place to start.

- Draw an outline of your system to keep track of the functionality/code structure
- Use techniques you tested from the Java sockets worksheet.
- For the Controller, you can start by making it accept connections
- Avoid multithreading until you are ready for it
- Make sure your Controller and Dstores print detailed log messages to stdout/stderr
- Work with just the Dstore to be able to save and read files
- Progressively add the features such as delete and allocating files to Dstores
- Test progressively so you know each area works and can return errors.
- Finally write the rebalance operations

8 Objectives

This coursework has the following module aims, objectives and learning outcomes:

A5. Client-server applications and programming
D1. Build a client-server solution in Java
D2. Build a distributed objects solution in Java
D3. Build and operate simple data networks
B5. Understand the use and impact of concurrency on the design of distributed systems