COMP2207: Distributed file system coursework

Leonardo Aniello, Kirk Martinez, Temitope Omitola

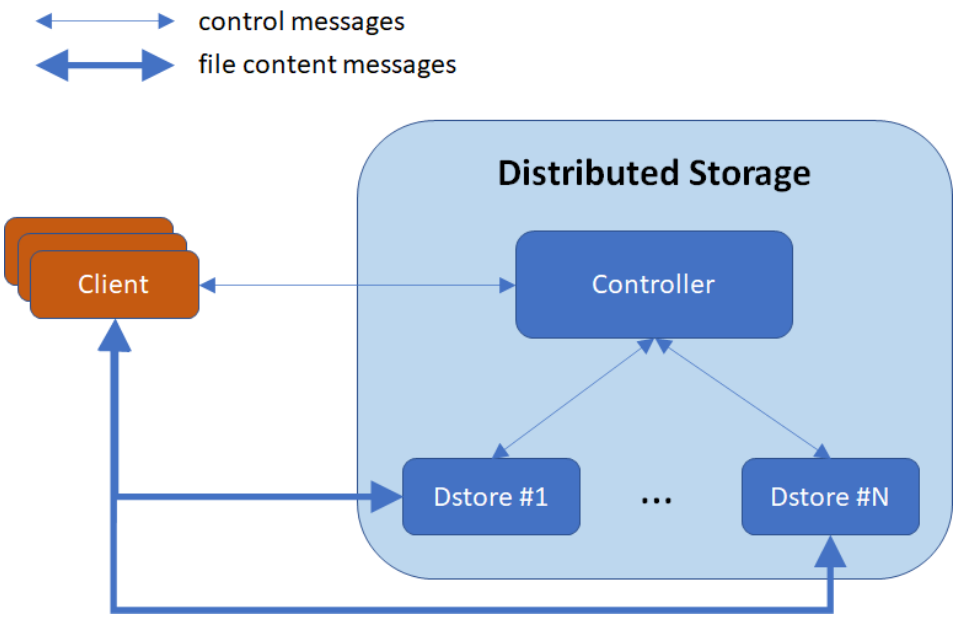
Course:	COMP2207
Document version:	3.0 Apr 28, 2021

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. 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 makes it very scalable. 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 datastore (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 be restarted, and new Dstores can join the datastore 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

The modules will communicate with each other via TCP connections. Because they could be on the same machine for testing, the datastores will listen on different ports.

The Dstores will establish connections with the Controller as soon as they start. These connections will be persistent. 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 joined the data store.

Processes should send textual messages (e.g., LIST) 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 Code development

Only use Java openjdk-14-jdk-headless, 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`

Where 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 milliseconds) to start the next rebalance operation (`rebalance_period`).
The 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`).

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`

Failure Handling

- Malformed message received by Controller
  - Log the error and continue
- Malformed message received by Client
  - Log the error and continue
- Malformed message received by Dstore
  - Log the error and continue
- 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`
- Client cannot connect or send data to all R Dstores
  - Log the error and continue; future rebalances will sort things out
- 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`

Failure Handling

- Malformed message received by Controller
  - Log the error and continue
- Malformed message received by Client
  - Log the error and continue
- Malformed message received by Dstore
  - Log the error and continue
- 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
  - Dstore -> Client: 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`

Failure Handling

- Malformed message received by Controller
  - Log the error and continue
- Malformed message received by Client
  - Log the error and continue
- Malformed message received by Dstore
  - Log the error and continue
- 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
  - Controller logs error and the protocol continues; future rebalances will sort things out
- If Dstore does not have the requested file
  - Dstore -> Controller: `ERROR_FILE_DOES_NOT_EXIST filename`

List operation

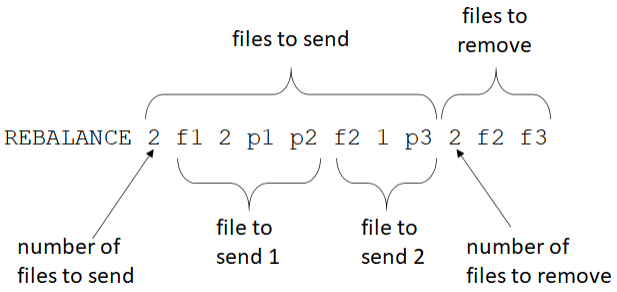
- Client->Controller: `LIST`
- Controller->Client: `LIST file_list`
  - `file_list` is a space-separated list of filenames

Failure Handling

- Malformed message received by Controller
  - Log the error and continue
- Malformed message received by Client
  - Log the error and continue
- If not enough Dstores have joined
  - Controller->Client: `ERROR_NOT_ENOUGH_DSTORES`

Storage Rebalance operation

This operation is started periodically by the Controller (e.g., based on the `rebalance_period` argument) and when a new Dstore joins the data store. 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`
    - `file_list` is a space-separated list of filenames
- 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)
- 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_send1 file_to_send2 ... file_to_sendN`
  - and `file_to_sendi` 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
        - 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`

Failure Handling

- Malformed message received by Controller
    - Log the error and continue
- Malformed message received by Dstore
    - Log the error and continue
- Controller does not receive REBALANCE COMPLETE from a Dstore within a timeout
    - Log the error and continue, future rebalance operations will sort things out

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
- 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

4 The Index

The index refers to the data structure used by the controller to keep track of files. As Store and Remove operations involve a number of messages to be completed, it is important to ensure that other possibly conflicting concurrent operations are served properly. For example, while a file F is being stored (i.e., updates index, “store in progress”), we do not want the data store to serve 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".

5 Logging

The Controller and Dstores must use the provided logger classes to log to file/stdout all the textual messages they send and receive. The Controller must also log when new Dstores join the data store. The Controller must use the `ControllerLogger` class. The Dstores must use the `DstoreLogger` class. Both `ControllerLogger` and `DstoreLogger` extend the `Logger` class.

Initialisation

Once started, the Controller must initialise its logger as follows:

```
ControllerLogger.init(Logger.LoggingType.ON_FILE_AND_TERMINAL);
```

Where the argument defines the type of logging to produce. The argument type is `Logger.LoggingType`, which is an enumeration that can take one the following values.

- `NO_LOG`: no log entry is generated
- `ON_TERMINAL_ONLY`: log entries are printed to stdout only
- `ON_FILE_ONLY`: log entries are printed to file only
- `ON_FILE_AND_TERMINAL`: log entries are printed both to stdout and to file

The name of log files generated by the ControllerLogger is in the form `controller_<ts>.log`, where `<ts>` is the timestamp of log file creation. This means that a new log file is generated every time the Controller is started.

At boot time, the Dstores must initialise their logger as follows:

```
DstoreLogger.init(Logger.LoggingType.ON_FILE_AND_TERMINAL, port);
```

The semantic of the first argument is the same of `ControllerLogger.init()`. The second argument is an int and corresponds to the TCP port where the Dstore accepts connections. The name of log files generated by the ControllerLogger is in the form `dstore_<port>_<ts>.log`, where `<port>` and `<ts>` are self-explanatory.

Logging textual messages sent or received

Right after a textual message is sent, the `messageSent(Socket socket, String message)` method must be invoked, as follows:

```
ControllerLogger.getInstance().messageSent(socket, message);

DstoreLogger.getInstance().messageSent(socket, message);
```

Where `socket` is the Socket instance used to send the message, and `message` is the String sent.

Right after a textual message is received, the `messageReceived(Socket socket, String message)` must be invoked, as follows

```
ControllerLogger.getInstance().messageReceived(socket, message);

DstoreLogger.getInstance().messageReceived(socket, message);
```

Where `socket` is the Socket instance used to receive the message, and `message` is the String received.

Network messages that carry file content must not be logged.

Logging join events

When the Controller receives a JOIN message from a new Dstore that declares to be listening on port `port`, the dstoreJoined(Socket socket, int dstorePort) method must be invoked, as follows

```
ControllerLogger.getInstance().dstoreJoined(socket, port);
```

Where `socket` is the Socket instance used to receive the message, and `port` is the TCP port where the Dstore is accepting connections.

Additional information

- Both `ControllerLogger` and `DstoreLogger` implement the Singleton design pattern, which ensures that a single instance of the class is created in each process. Such an instance can be accessed from anywhere in the process by using the static method `getInstance()`.
- Only the methods explained above should be used. **Any other method provided by `Logger`, `ControllerLogger` and `DstoreLogger` classes must not be used**.
- **The source code of `Logger`, `ControllerLogger` and `DstoreLogger` classes must not be changed**.

Another class is provided, `Protocol`. This class defines public constants (i.e., `public static final` fields) corresponding to the messages that Clients, Controller and Dstores are expected to exchange. In order to avoid problems due to typos, it is highly recommended to use these fields when composing/parsing textual messages.

6 Submission Requirements

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

  As well as all the additional .java files you developed (and corresponding .class files)
- These files should be contained in a single zip file called 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 will be executed at the Linux command line by us for automatic testing

7 Marking Scheme

You are asked to implement a Controller and Dstores. You will be given the client, as an obfuscated jar. The client allows the execution of operations via a terminal. You will also be given some Java classes for the logging (to file).

- >= 40%
  - The datastore works in compliance with the protocol and correctly serves sequential requests from a single client

- >= 50%
  - In addition to the criteria above
  - Each file is replicated R times
  - Files are evenly spread over the Dstores (only when stored, not when Dstores fail or new Dstores join the datastore)
- >= 60%
  - In addition to the criteria above
  - The datastore correctly serves concurrent requests from more clients
- >= 70%
  - In addition to the criteria above
  - The datastore correctly tolerates the failure of one Dstore
- >= 80%
  - In addition to the criteria above
  - The datastore correctly tolerates the failure of up to N-R Dstores
- >= 90%
  - In addition to the criteria above
  - Files are evenly spread over the Dstores despite Dstores failing and new Dstores joining the datastore

8 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
- Write a simple log(message) function which prints the timestamp and message into a logfile.
- Parse incoming commands from the client one at a time, with error codes for failures.
- Send commands to a simplified Dstore (e.g. LIST)
- 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

9 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