A Series of Indicative Votes

Kirk Martinez Leonardo Aniello Corina Cîrstea

April 2, 2020

Course:	COMP2207
A.Y.:	2019-20
Document version:	1.3

1 Introduction

The purpose of this assignment is to implement a consensus protocol that tolerates participant failures. The protocol involves two types of processes: a **coordinator**, whose role is to initiate a run of the consensus algorithm and collect the outcome of the vote; and a **participant**, which contributes a vote and communicates with the other participants to agree on an outcome. Your application should consist of 1 coordinator process and N participant processes, out of which any number of participants may fail during the run of the consensus algorithm. The actual consensus algorithm is run among the participant processes, with the coordinator only collecting outcomes from the participants. The behaviour of the two types of processes is described below. Furthermore, your application should include a **logger server** where the other processes send log messages over the UDP protocol.

Note that the IRC (Internet Relay Chat) example provided may prove useful in parsing messages and managing sockets.

2 Protocol for Participant

1. Register with coordinator. The participant establishes a TCP connection with the coordinator and sends the following byte stream:

   `JOIN ⟨port⟩`

   Where `⟨port⟩` is the port number that this partcipant is listening on. This will be treated as the identifier of the participant. For example, the participant listening on port 12346 will send the message:

   `JOIN 12346`

2. Get details of other participants from coordinator. The participant should wait to receive a message from the coordinator with the details of all other participants (i.e. read from the same socket connection):

   `DETAILS [⟨port⟩]`

   Where `[⟨port⟩]` is a list of the port numbers (aka. identifiers) of all **other** particpants. (Note that we do not want a participant sending its vote to itself.) For example, a participant with identifier/port 12346 may receive information about two other participants:

`DETAILS 12347 12348`

3. Get vote options from coordinator. The participant should wait again to receive a message from the coordinator with the details of the options for voting:

   `VOTE_OPTIONS [⟨option⟩]`

   Where `[⟨option⟩]` is the list of voting options for the consensus protocol. For example, there may be two options, A and B:

   `VOTE_OPTIONS A B`

   Then decide on own vote, from the options received.

4. Execute a number of rounds by exchanging messages **directly with the other participants** using the TCP protocol.

   Round 1 The participant will send and receive messages of the following structure in the first round:

   `VOTE ⟨port⟩ ⟨vote⟩`

   Where `⟨port⟩` is the sender's port number/identifier, and `⟨vote⟩` is one of the vote options (i.e. that agent's vote).

   For example, if we have 3 participants listening on ports `12346`, `12347` and `12348`, and their votes are A, B and A respectively, and there are **no failures**, then the messages passed between participants will be:

   - **12346 to 12347**: `VOTE 12346 A`
   - **12346 to 12348**: `VOTE 12346 A`
   - **12347 to 12346**: `VOTE 12347 B`
   - **12347 to 12348**: `VOTE 12347 B`
   - **12348 to 12346**: `VOTE 12348 A`
   - **12348 to 12347**: `VOTE 12348 A`

   Round j > 1 The participant will send and receive messages of the following structure in all subsequent rounds:

   `VOTE ⟨port 1⟩ ⟨vote 1⟩ ⟨port 2⟩ ⟨vote 2⟩ ...⟨port n⟩ ⟨vote n⟩`

   Where `⟨port i⟩` and `⟨vote i⟩` are the port (identifier) and vote of **any new votes received in the previous round**.

5. Decide vote outcome using majority. In case of a tie, pick the first option according to an ascendant lexicographic order of the options with the majority of votes.

   For example, if there are three options A, B and C with 2, 2 and 1 votes, respectively, then there is a tie between A and B; this tie should be resolved by sorting the majority options (i.e. A and B) by lexicographic order (i.e. [A,B]) and picking the first one (i.e. A).

6. Inform coordinator of the outcome. The following message should be sent to the coordinator on the **same connection** established during the intial stage:

   `OUTCOME ⟨outcome⟩ [⟨port⟩]`

   Where `⟨outcome⟩` is the option that this participant has decided is the outcome of the vote, and `[⟨port⟩]` is the list of participants that were taken into account in settling the vote. For example, participant `12346` in the above example should send this message to the coordinator:

   `OUTCOME A 12346 12347 12348`

   In other words, agent `12346` has taken into account its own vote and those of `12347` and `12348` and come to the conclusion that `A` is the outcome by majority vote.

3 Protocol for the Coordinator

1. Wait for the number of participants specified to join. The number of participants should be given as a parameter to the `main` method of the coordinator (see below).

   `JOIN` ⟨*port*⟩

   Where ⟨*port*⟩ is the port number of the participant.

2. Send participant details to each participant once all participants have joined:

   `DETAILS` [⟨`port`⟩]

   Where [⟨`port`⟩] is a list of the port numbers (aka. identifiers) of all **other** particpants.

3. Send request for votes to all participants:

   `VOTE_OPTIONS` [⟨`option`⟩]

   Where [⟨`option`⟩] is the list of voting options for the consensus protocol. The voting options should be given as a parameter to the `main` method of the coordinator (see below).

4. Receive votes from participants:

   `OUTCOME` ⟨`outcome`⟩ [⟨`port`⟩]

   Where ⟨`outcome`⟩ is the option that this participant has decided is the outcome of the vote, and [⟨`port`⟩] is the list of participants that were taken into account in settling the vote.

4 Logging

Participant and Coordinator processes should log their operations in two different ways. They should (i) use two ad-hoc classes provided by us to log to a file and (ii) implement a **logger** process that receives messages via UPD and stores them in a file. The following two subsections describe these two logging approaches in more detail, and a third subsection explains how to use the provided logger classes.

4.1 Logger classes

You should use the public methods provided by the classes `ParticipantLogger` and `CoordinatorLogger` to log the main actions that coordinator and participant processes take.

For example, right after the coordinator accepted a connection from a participant, the method `connectionAccepted()` of `CoordinatorLogger` should be invoked. As another example, as soon as a participant detects the failure of another participant (i.e. the corresponding connection has been interrupted), the method `participantCrashed()` of `ParticipantLogger` should be invoked.

4.2 Logger server

You should implement a `UDPLoggerServer` class that starts a server process which listens for UPD log messages on a specific port, given as input parameter when the server process is started. Whenever a new message is received by the server, an acknowledgement is sent back to the sender process and the message is stored in a log file by the server. The acknowledge message is the string "ACK", without quotes. As this is using UDP the client should timeout if it did not receive an ACK

and resend up to three times. The log messages sent by the coordinator and participant processes have the following format:

`id msg`

where `id` is the TCP port where the sender process is listening on and `msg` is the log message.

For each log message `id msg` sent by a process `id` at time `t`, the log file should include an entry formatted in this way:

`id t msg`

The method `System.currentTimeMillis()` should be used to obtain the current time. For example, if log message "12346 [P12346] JOIN sent to Coordinator on port 12345" is sent by the participant listening on port `12346` and is received by the Logger server at time `1583508972566`, then the log file should include the following entry:

`12346 1583508972566 [P12346] JOIN sent to Coordinator on port 12345`

The name of the log file must be `logger_server_`⟨t⟩`.log`, where ⟨t⟩ is to be replaced with the time the Logger server was started. For example, if the Logger server was launched at time `1583508972566`, then the name of the log file should be `logger_server_1583508972566.log`.

You should also implement a `UDPLoggerClient` class that coordinator and participants will use to send log messages to the logger server using UDP protocol. In the specific, this class will provide a single method `logToServer(String message)` that will be invoked internally by all the public methods of `ParticipantLogger` and `CoordinatorLogger` classes. This means that you do not need to invoke `logToServer()` explicitly in your code. The implementation of `logToServer()` method should retransmit the message to the Logger Server if an acknowledge is not received within a specific timeout, given as input parameter to coordinator and participant processes when they are launched. If no acknowledge is received after three retransmissions, then the `logToServer()` method should throw an `IOException`. A skeleton of `UDPLoggerClient` is provided, with a constructor, three final attributes (i.e. `loggerServerPort`, `processId` and `timeout`) and their corresponding getter methods. Note that the `processId` attribute represents the TCP port that the Coordinator/Participant is listening on.

4.3 How to Use Logger Classes

`ParticipantLogger` (`CoordinatorLogger`) class is implemented as a singleton, i.e. there is at most one instance for each Participant (Coordinator) process, and that instance can be conveniently retrieved by invoking the `ParticipantLogger.getLogger()` (`CoordinatorLogger.getLogger()`) static method. Before retrieving the singleton and invoking its public methods, the `ParticipantLogger` (`CoordinatorLogger`) must be initialised. The initialisation requires three parameters, namely the UDP port that the Logger Server is listening on, the TCP port that the Participant (Coordinator) process is listening on and the socket timeout (in milliseconds).

For example, if the Participant (Coordinator) process is listening on TCP port `12346`, the Logger server is listening on UDP port `12345` and the timeout is `500` milliseconds, then the `ParticipantLogger.initLogger(12345, 12346, 500)` (`CoordinatorLogger.initLogger(12345, 12346, 500)`) static method should be invoked. Note that either using the public methods of `ParticipantLogger` (`CoordinatorLogger`) before initialising it, or initialising it more than once will throw a `RuntimeException`.

`ParticipantLogger` (`CoordinatorLogger`) instantiates a `UDPLoggerClient` object when initialised, and uses it to forward log messages to the Logger Server by invoking its `logToServer()` method.

The `ParticipantLogger` methods `votesSent()` and `votesReceived()` expect a `List<Vote> votes` as parameter. A `Vote` class is provided with two final attributes, i.e. `int participantPort` and `String vote`, which can be initialised when the class is instantiated. Hence, before invoking `votesSent()` or `votesReceived()`, you must create a `List<Vote>` object (e.g. `List<Vote> votes = new ArrayList<Vote>();`) and add a `Vote` instance for each vote you need to consider.

Note that `ParticipantLogger`, `CoordinatorLogger` and `Vote` classes must not be modified.

5 Submission Requirements

Submission checklist:

- You should write your solution using **Java Version 8 Update 241** (released January 14, 2020).
- Your submission should include the following files:
  - `Coordinator.java`
  - `Participant.java`
  - `UDPLoggerServer.java`
  - `UDPLoggerClient.java`
  - `Vote.java`
  - `Coordinator.class`
  - `Participant.class`
  - `UDPLoggerServer.class`
  - `UDPLoggerClient.class`
  - `Vote.class`

  Note that you must include any other files in your submission that you wish to rely on, and references to these files must be relative. You may assume that the current directory is in your CLASSPATH.
- These files should be contained in a **single** zip file.
- There should be no package structure to your java code; i.e. no statements in the Java source files such as "`package comp2207.irc`" as in the IRC example.
- When extracted from the zip file, all `.class` files should be located in the current directory.
- The `UDPLoggerServer` class file will be executed at the Unix/Linux/DOS command line as follows:

  `java UDPLoggerServer ⟨port⟩`

  Where ⟨`port`⟩ is the port number that the logger server is listening on. For example, if we want to start the logger server listening on port *12344*, then it will be executed as:

  `java UDPLoggerServer 12344`
- The `Coordinator` class file will be executed at the Unix/Linux/DOS command line as follows:

  `java Coordinator ⟨port⟩ ⟨lport⟩ ⟨parts⟩ ⟨timeout⟩ [⟨option⟩]`

Where ⟨port⟩ is the port number that the coordinator is listening on, and ⟨lport⟩ is the port number that the logger server is listening on, and ⟨parts⟩ is the number of participants that the coordinator expects, and [⟨option⟩] is a set (no duplicates) of options separated by spaces. The timeout should be used by the coordinator when waiting for a message from a participant, in order to decide whether that participant has failed. For example, if we want to start the coordinator listening on port *12345*, expecting the logger server listening on port *12344*, expecting *4* participants, with a timeout of 500 milliseconds and where the options are *A*, *B* and *C*, then it will be executed as:

```
java Coordinator 12345 12344 4 500 A B C
```

- The `Participant` class file will be executed at the Unix/Linux/DOS command line as follows:

  `java Participant ⟨cport⟩ ⟨lport⟩ ⟨pport⟩ ⟨timeout⟩`

  Where ⟨cport⟩ is the port number that the coordinator is listening on, and ⟨lport⟩ is the port number that the logger server is listening on, ⟨pport⟩ is the port number that **this** participant will be listening on, ⟨timeout⟩ is a timeout in *milliseconds*. The timeout should be used by a participant when waiting for a message from another process, in order to decide whether that process has failed.

  For example, if we want to start a participant that operates with a timeout of 500 milliseconds and that is listening on port 12346 with the coordinator listening on port 12345 and the logger server on 12344 as above, this will be executed as:

  ```
  java Participant 12345 12344 12346 500
  ```

When you submit your solution, the zip file will be unpacked, the files will be checked (i.e. location of `.class` files, use of Java Version 8 Update 241), and they will be run with various numbers of participants with various inputs. We will test your solution both with the provided logger classes and with variants of these classes that simulate participant failures.

For example, a Shell script for a simple test where there are no failures may be:

```
#!/bin/sh

java UDPLoggerServer 12344 &
echo "Waiting␣for␣logger␣server␣to␣start..."
sleep 5
java Coordinator 12345 4 500 A B &
echo "Waiting␣for␣coordinator␣to␣start..."
sleep 5
java Participant 12345 12346 500 &
sleep 1
java Participant 12345 12347 500 &
sleep 1
java Participant 12345 12348 500 &
sleep 1
java Participant 12345 12349 500 &
```

The consensus outcome for a majority vote will be one of A or B in this case because with 3 participants there will always be 2 that agree assuming all vote (no failures).

It is **your** responsibility to use available logger classes to accurately indicate what your system is doing.

6 Marking

The marking scheme is:

> 40% The protocol operates as expected, and the coordinator reports some outcome. The outcome is not necessarily correct (i.e. not all participants agree).

> 50% The protocol operates as expected and the coordinator reports the correct outcome in all cases in which there are no participant failures.

> 60% Your solution operates as above, and is robust to one participant failure.

> 70% Your solution operates as above, and is robust to any number of participant failures.

> 80% Your solution operates as above, and is robust to participants and coordinator starting in any order.

up to additional 10% Your solution correctly sends and receives UDP log messages.