A Series of Indicative Votes

Leonardo Aniello Corina Cîrstea Timothy J. Norman

May 7, 2019

Course:	COMP2207
Coursework:	2
Document version:	1.0

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 at most 1 participant 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.

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, 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**.

   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 n > 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 (null if no majority)

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 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. ~~If not all participants join, then the coordinator should abort.~~

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.

5. Print out the results in a meaningful way.

4 Submission Requirements

Submission checklist:

- You should write your solution using **Java SE 12** (released March 2019).
- You should submit a one-page PDF of a document describing your approach. Minimum text size 11pt. The document must include your name and student ID. It should also include details of:
  - how the inter-process communication was implemented (sockets/RMI/...),
  - whether your solution implements the basic setup (no failures), or some/all of the extensions,
  - key design decisions that we should be aware of when testing/evaluating your code.
- Your submission should also include the following files:
  - `Coordinator.java`
  - `Participant.java`
  - `Coordinator.class`
  - `Participant.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, `Coordinator.class` and `Participant.class` should be located in the current directory.
- The `Coordinator` class file will be executed at the Unix/Linux/DOS command line as follows:

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

  Where ⟨port⟩ is the port number that the coordinator 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. For example, if we want to start the coordinator that is expecting *4* participants listening on port *12345* where the options are *A*, *B* and *C*, then it will be executed as:

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

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

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

  Where ⟨cport⟩ is the port number that the coordinator is listening on, ⟨pport⟩ is the port number that **this** participant will be listening on, ⟨timeout⟩ is a timeout in *milliseconds* and ⟨failurecond⟩ is a *flag* indicating whether and at what stage the participant fails.

  The permitted values of ⟨failurecond⟩ are:

  0 The participant does not fail;

  1 The participant fails during step 4 (i.e. after it has shared its vote with some but not all other participants); and

  2 The participant fails after step 4 and before the end of step 5.

  For example, if we want to start a participant that does not fail, that operates with a timeout of 5000 milliseconds (5 seconds) and that is listening on port 12346 with the coordinator listening on port 12345 as above, this will be executed as:

  ```
  java Participant 12345 12346 5000 0
  ```

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 SE 12), and they will be run with various numbers of participants with various inputs. We may also test your coordinator with participants written by **us** using the same protocol.

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

```
#!/bin/sh

java Coordinator 12345 4 A B &
echo "Waiting for coordinator to start..."
sleep 5
java Participant 12345 12346 5000 0 &
sleep 1
java Participant 12345 12347 5000 0 &
sleep 1
java Participant 12345 12348 5000 0 &
sleep 1
java Participant 12345 12349 5000 0 &
```

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). Your coordinator should write out the result of the consensus protocol to the standard output; i.e. via `System.out`.

It is **your** responsibility to write out a meaningful sequence of messages that indicates what your system is doing.

When you submit your code, you should get an email from `user@svm.tjn1f15-comp2207.ecs.soton.ac.uk` with output from your client and some simple feedback from the processing of your code.

5 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. Any tie (e.g. four participants, two options, and each option with two votes) is reported correctly as a tie.

$>$ 60% Your solution operates as above, but is robust to one participant failure under failure condition **1** (i.e. one participant fails during step 4). This requires a second round of messages in step 4 of the participant protocol.

$>$ 70% Your solution is robust to both types of failure (i.e. one participant fails during either step 4 or step 5) and ties are resolved by the vote being restarted with fewer options.

$>$ 80% Your solution is robust to any number of participant failures.