IN4150 Distributed Algorithms, exercise 3

Koen Boes (1314785) and Zmitser Zhaleznichenka (4134575)

June 8, 2012

1 Project setup

The code is written in Java and consists from server and client parts. The server part can be deployed at a number of different physical machines in a fully connected network. The communication between the server nodes is managed via Java RMI.

To work with the project, one has to adjust the network configuration, start servers at all the allocated machines and connect to any of the servers with a client that issues the instructions to the servers.

The network configuration should be located in `network.cfg` file in `resources/` directory. Each line in this file provides a unique URL of a server process. Several processes may be located at one physical machine but have to have different names. If file `network.cfg` is not found in the directory, a default file will be used instead, but it is recommended to create `network.cfg` file. For local processes, "localhost" and "127.0.0.1" values can be used as a host name part of a process URL.

Build process is organised with Maven build tool. To compile source files and make an executable .jar file, one has to run '`mvn clean install -DskipTests`' command in the terminal from the project root. After the execution of this command, a newly created .jar file will be placed in target/ directory with its dependencies in target/lib/.

Before starting the server, it is needed to copy `.java.policy` file to the home folder and issue the following terminal command from the project root.

```
java -Djava.security.policy=java.policy -jar target/DA-3.1.0.jar
```

If using a distributed setup, all the instances should be started within a frame of 5 seconds as during this time servers instantiate its local processes and start to look for the remote ones afterwards. If any remote process is not resolved, the server will stop.

Client connection to the server is emulated with JUnit tests. Once the servers are started, one has to run JUnit tests from `test\` directory to connect to the servers and issue the first order to the commander process. The correctness of the protocol can be verified from the server and client logs as well as from the assertions in JUnit files.

2 Algortihm implementation

The main class `DA_Byzantine_Main` is used to start the RMI registry and initiate the process manager which resides in `ProcessManager` class. The manager is responsible for the proper instantiation of the processing nodes. It processes the network configuration, starts the local processes and locates the remote ones.

The processing nodes implement `DA_Byzantine_RMI` interface. The implementation file is `DA_Byzantine`.

The main methods of the RMI interface are the following:

- `receiveOrder(OrderMessage message)` is used for the exchange of orders. When process A wants to send an order to process B, it forms an `OrderMessage` m and invokes `B.receiveOrder(m);`.
- `receiveAck(AckMessage message)` is used to confirm the receipt of the order message. The introduction of acknowledgements allows to simulate the synchronous operations in an asynchronous environment. If a process sends an order to another process and does not receive an acknowledgement, it retries the delivery. If it fails the second time, the process gives up. In our

setup, only the delivery of the order messages is affected by faulty behavior, but the acknowledgements are always delivered.

- `getfault()` and `setFault(Afault fault)` are used to apply the faulty behavior to the process. We have a number of implementation classes extending the abstract `AFault` class which introduce various faulty patterns. Each class implements `applyFaultyBehavior(Order order, int iteration)` method that takes an order, changes it according to the fault rules in the given iteration and returns the result - either the same order, or reversed one, or nothing at all. This method is used while broadcasting the received order to the other lieutenants.

- `decide()` is used to ask the process to make a final decision when it is ready to do so. This method is invoked internally in a separate thread.

Also, the interface contains a number of methods created for logging and debugging purposes, as well as some getters and setters.

Inside `DA_Byzantine` we have a two-step algorithm that is based on the decision tree.

Each process maintains a tree where it keeps the orders received from the other processes. The tree implementation is provided at `Node` class. At the root of the tree the process keeps the order received directly from the commander, at the next level - the commander decision as transmitted by one lieutenant and so on. Upon receipt of the order messages we fill in the receivers' decision tress. This is the first algorithm stage.

After the receipt of a new process message we check whether the tree contains enough messages to make a decision. To do it, at each level of the tree we compare the number of ready nodes with the total number of nodes. If at all the levels the number of ready nodes is enough to make a decision despite of traitors, we continue with the next step.

At the second phase, we start a waiting thread that is needed to let the late messages to be delivered. After a delay it executes the `decide()` method of a process and the process comes to the decision.

Decision is done as following. At each level of the tree we take the orders from the subtree root and its children and apply a majority function to them. The code is executed recursively until we reach the decision at root. This is our final decision.

This algorithm is very simple and efficient as it allows to mitigate all the communication problems easily. We just fill in the tree with a simple code until it is nearly ready and no new messages will affect the decisions in it, and then decide based on the orders in the tree.

3 Test results