Course Project: Virtual Synchrony

Distributed Systems 1, 2017 – 2018

rev. 2

In this project, you are requested to implement a simple peer-to-peer group communication service providing the *virtual synchrony* guarantees, following the algorithm outlined in class.

1 General description

The project should be implemented in Akka with the group members being Akka actors that send multicast messages to each other. The system should allow adding new participants to the group as well as tolerate silent crashes of some existing participants at any time.

Multicast properties. The reliable multicast implementation should assure that a multicast sent within a group view V_i is either delivered to all **operational** actors of V_i or to none of them, with the *none* option allowed only in the case when the initiator crashes while sending the multicast. Within a single group view, concurrent multicast messages might be delivered in any order.

Group manager. Designing a distributed group membership management subsystem is non-trivial in a general case, requiring distributed locks or a form of consensus algorithm. Therefore, in this project, we introduce a set of simplifying assumptions. We assume that there is a dedicated *reliable* group member (with ID=0) called the *group manager*, responsible for serialising group view changes and sending view update messages to the group. Other participants are not allowed to send view update messages but they might notify the manager if they notice a crash.

Joining. To join the group, a new participant (actor) contacts the group manager by sending a join request. Eventually, after receiving the request, the group manager should initiate a view change, including the new participant to the group. After the newly joining node has learnt that it is a part of the new group view, it should start sending multicasts to the group. Each group member should have a unique integer ID assigned by the group manager.

Data traffic. All group members should be sending multicast messages continuously, with randomly varying intervals that must not exceed a predefined constant T_d. The system **must** detect duplicated messages and never deliver a message more than once.

Detection of stable messages. According to the protocol, upon receiving a message, the receiver should keep a copy of it until it learns that the message is *stable*. For simplicity, we assume that a group member may send a new multicast only after it completed sending the previous one, which effectively means that when a participant receives a multicast from an initiator P, all previous multicasts from P are stable and their copies might be safely dropped.

Crash detection. The system should implement a simple crash detection algorithm based on timeouts. The group manager might detect a crash of process P when no messages arrive from P within a predefined timeout $T_{timeout}$. This mechanism might be based only on the data multicasts or, optionally, use special heartbeat messages sent periodically by all participants to the manager to keep it up-to-date even when no data messages are being sent.

Optionally, participants might timeout as well, when expecting a *Flush* control message from a participant P and notify the group manager that P might have crashed.

2 Implementation-related assumptions and requirements

- We assume that the unicast message exchange between actors is reliable.
- To emulate network propagation delays, you are requested to insert small random intervals between the unicast transmissions of the multicast implementation (as it was done in the examples seen in class), for both data and control multicasts.
- The program might be implemented either as a single Akka system with multiple local actors or as a group of Akka systems communicating over the network. The latter will be considered a plus at the evaluation.
- Ensure proper actor encapsulation. Avoid using shared objects unless they are immutable. For example, do not use static properties in actor classes.
- The program should generate a log file (or multiple log files) recording the key steps of the protocol. To automate testing, in addition to arbitrary logging, the program must record the following log messages:

  - `<ID> install view <view_seqnum> <participant_list>` — with comma-separated IDs
  - `<ID> send multicast <seqnum> within <view_seqnum>`
  - `<ID> deliver multicast <seqnum> from <ID> within <view_seqnum>`
- To emulate crashes, a participant should be able to enter the "crashed mode" in which it ignores all incoming messages and stops sending anything. In the case of the networked implementation, the whole Akka system of the crashed node might be shut down.
- During the evaluation it should be easy, with a simple instrumentation of the code, to emulate a crash or to join a new member (local actor) at the key points of the protocol, e.g., during the sending of a multicast, after receiving a multicast, after receiving a view change message, etc. In addition to that, in the case of the networked implementation, the user should be able to add a new group member by running a new instance of the program and specifying the IP address and the port of the group manager.

3 Grading

You are responsible to show that your project works. The project will be evaluated for its technical content (algorithm correctness). *Do not* spend time implementing unrequested features — focus on doing the core functionality, and doing it well.

A correct implementation of the whole requested functionality is worth 6 points. It is possible to submit programs implementing a subset of the requested features or systems requiring stronger assumptions. In these cases, lower marks will be awarded.

You are expected to implement this project with exactly one other student, however the marks will be individual, based on your understanding of the program and the concepts used.

4 Presenting the project

- You MUST contact through e-mail the instructor (gianpietro.picco@unitn.it) AND the teaching assistant (timofei.istomin@unitn.it), well in advance, i.e., at least a couple of weeks before the presentation.
- You can present the project at any time, also outside of exam sessions. In the latter case, the mark will be "frozen" until you can enrol in the next exam session.
- The code must be properly formatted otherwise it will not be accepted (e.g., follow style guidelines).
- Provide a short document (1-3 pages) in English explaining the main architectural choices.
- Both the code and documentation must be submitted in electronic format via email at least one day before the meeting. The documentation must be a single self-contained pdf/txt. All code must be sent in a single tarball consisting of a single folder (named after your surnames) containing all your source files. For instance, if your surnames are Rossi and Russo, put your source files and the documentation in a directory called `RossiRusso`, compress it with zip or tar ("`tar -czvf RossiRusso.tgz RossiRusso`") and submit the resulting archive.
- The project is demonstrated in front of the instructor and/or assistant.

Plagiarism is not tolerated. Do not hesitate to ask questions if you run into troubles with your implementation. We are here to help.