Virtual Synchrony - Project Report

Distributed Systems 1, 2017 - 2018

Daniele Bissoli - 197810, Andrea Zampieri - 198762

1 Introduction

In this project, we have implemented a simple group communication service that permits different members of a group to communicate through messages exchange. The management of the group is performed in a centralized fashion through a special node, named *Group Manager*. As design requirement, the system is able to guarantee the Virtual Synchrony, providing a reliable communication between members.

1.1 View and Virtual Synchrony

A *view* is a local representation of the system, that is the set of group members that were still operational when last group update (join/crash) has been performed. Each member of the group has its own view and they have to globally agree on the same view in order to consistently communicate. Moreover, a view can be seen as a system epoch, since it changes every time a new participant join the group or when a cohort is detected to be crashed.

Under the assumptions of reliable links and FIFO channels, *Virtual Synchrony* is the mechanism which guarantees that messages sent in a specific epoch to be delivered to all the group members and just those participant in that epoch.

2 Project Architecture

The project is divided into three main classes (*Node*, *Participant* and *Group Manager*). In addition, it contains the definition of all exchanged messages and the class in charge of managing the creation of the system. In the following paragraphs, the three main classes will be explained.

2.1 Node

This is the main class, which represents a common group member (also called *participant*). It defines the main functionalities of a group member which communicates under the Virtual Synchrony. These are:

- **receipt of a message** - when a data message is received, it is delivered to the application, but a copy of it is kept until it becomes *stable*, which means the sender was able to send that message to everybody in the group.
- **receipt of a stable message** - upon this message is received, the corresponding copy of the data message can be safely deleted since it is confirmed that the message has been delivered to everybody in the group.

- **view change mechanism** - it is based on the steps described by Virtual Synchrony. For implementation details refer to section 3.

Moreover, each node exploits a View object to keep track of which messages have been delivered, which are still unstable and to recognize whether all the flush messages have been received.

2.2 Participant

It represents a member of the group which continuously sends and receives data messages to/from others. It extends the *Node* class providing more functionalities such as send of data messages and the mechanisms to set it as crashed. In addition, it provides a finer implementation of some functionalities as flush messages management and view change.

It inherits all the attributes of *Node*, such as the list of received messages, the unstable ones (with respect to each view) and the queue for messages received too early. It also adds some new properties to control the flow and the behaviour of the actors. Other than trivial ones (messageID, MIN_DELAY, MAX_DELAY for handling the transmission of new messages) there are some status variables that enforce specific behaviours according to the protocol:

- ***justEntered***: it is used just once to trigger a new *SendDatamessage* that makes the process send a new multicast, and reissues itself with a random delay $d \in$ [MIN_DELAY, MAX_DELAY]

- ***allowSending***: as its name tells, when its value is **false**, the process will not send any new message to the others in the system

- ***crashed***: tells whether the process has failed or not (its objective is to simulate the crash of a given node); this variable can be set in different manners

As described in the Virtual Synchrony protocol, once a node receives a message that notifies a view change, all the unstable messages for the previous epoch are sent to all the participants in the new view, ensuring that all the operational processes receive all the messages sent from everybody within its epoch. This avoids the situation where a node does not receive certain messages from a crashed one, catching up thanks to the non-faulty participants still running. This kind of message is differentiated from the *standard* one since, when this kind of messages is received, no copy of it occurs because it can be seen as *already stable*.

2.3 Group Manager

It is a *reliable* group participant responsible to coordinate view updates between group members. It is built on top of the *Node* class and therefore it shares most of the characteristics of a group participant. Indeed, it also implements Virtual Syncrony logics, such as reaction to View Change and flush messages, although it is only able to receive messages. Besides, it is in charge of detecting crashes, which are recognized through timeout mechanism based on:

- **missing heartbeat message** - it is a message sent continuously by each cohort to the Group Manager to notify that it is still operational. Group manager detects that a participant has crashed when its heartbeat is not received within a predefined timeout

- **missing flush message** - it is a message sent by all the group members when a view change is triggered. Group Manager detects if a participant is crashed when it does not receive its flush message within the expected predefined timeout

In both cases, the group manager will trigger a view change, creating a new view and sending it to every group member through multicast exchange.

Apart from previous tasks, Group manager also manages requests from new participants to join the group. To respect the Virtual Synchrony, a new view is generated to keep track of the new member acquisition.

3 Implementation Details

3.1 Management of flushes and view-changes

The implementation in our project is slightly different from the one later proposed by Timofei Istomin. The following pseudo-code and the comment should clarify the process.

Require: flush message $f_a(v_i)$ from node a for new view v_i
  if all nodes in v_{i-1} sent the flush for v_i **then**
    for each v_k with $i < k \leq j$ **do**
      install v_k
      deliver all messages m that were sent in v_k
    end for
    return $\nexists v_l,\ l > i,\ s.t.\ v_l$ is still missing flushes
  end if
  return false

When a flush message is received, the process checks whether all the flushes have been collected from the other participants for the installation of view v_j (a successive view with respect to the current one v_i). If so, all the successive (and pending) views to the current v_i are installed in order up to the new one v_j. This can be the case since faulty nodes may not send the flush message and leave some views pending, thus these ones need to be cascaded installed once a more recent one meets the requisites.
Each view v_k s.t. $i < k \leq j$ gets installed and messages queued waiting for the right view are delivered.

The boolean returned valued is then exploited to check if a new view was installed and no other view change are still pending. Only in that case `allowSending` can be set back to `true` assuring the desired behaviour. Starting sending while the view change process is still running can lead to delivery of messages in wrong epochs from other participants. This happens because the process of inferring from which view data messages have been sent concurrently access the same data structure that the view change process is updating.

3.2 Heartbeat

Each participant constantly sends a *HeartbeatMessage* to the Group Manager: its function is just to prove the liveness of the process. This message does not get interrupted nor interferes with the behaviour and evolution of the system and it is just a trick to detect silent crashes of the nodes.

4 Crashes and Joining

The system is able to tolerate silent cohort crashes by means of the flush protocol. To control cohorts state, it is possible to send them special messages that make them crash according to predefined behaviour. Crashing modalities are specified below:

- **crash at any time** - immediately set the chosen node as crashed
- **crash on sending** - set the node crashed when it will perform next multicast to the group. As a result, only half of the group members will receive the sent message. It is up to the *all-to-all* exchange, performed during view change, to make all the other operational nodes delivering the sent message.
- **crash on receiving** - set the node crashed when it will receive next message. In this way the node is not able to perform the delivery of received message before crashing.
- **crash on view change** - set the node crashed when next view change is triggered. Therefore, the chosen node will not be able to perform the all-to-all message exchange and to send flush messages.

The system also allows to add further members. To join a new participant, a request must be sent to the group manager. Then the Group Manager will generate a new *id* for the new member and it will introduce the new participant into the group, triggering a view change.