Distributed Systems 1 Project Report

Virtual Synchrony

Daniele Bissoli - 197810, Andrea Zampieri - 198762

1 Introduction

In this project, we have implemented a simple group communication mechanism that permit different members of a group to communicate through messages exchange. The management of the group is done in a centralized fashion through a special node, named *Group Manager* which takes the *id* 0. Moreover, this mechanism provide the notion of Virtual Synchrony, guaranteeing the synchronization between members.

1.1 Virtual Synchrony

Under the assumptions of reliable links and FIFO channels, *Virtual Synchrony* is a mechanism which makes all the member of a group agree on who is currently alive in the group. It is based on the notion of View, which represent all the nodes of the group which are currently alive. A view can be seen as an epoch of the system, so that each epoch represent different state of the system.

2 Project Architecture

The project is divided into three main classes (Node, Participant and Group Manager) and contains also the definition of all messages and the class in charge to manage the creation of the system.

Here maybe we can put a small image?

2.1 Node

This is the main class, which represent a common group member. It has different methods to mainly deal with message receiving, view changes and flush messages.

2.2 Participant Class

The so called *Participant* is the implementation of an active node in the system: it sends messages to the other nodes and delivers the received ones according to the Virtual Synchrony logic.
It inherits all the attributes of *Node* (e.g. the list of received messages, the unstable ones (with respect to each view) and the queue for messages received too early), and adds some of its own in order to be able 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 to enforce the wanted behaviour.

- ***justEntered***: it's 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 won't send any new message to the others in the system

- ***crashed***: tells wether the process has failed or not (it's used to simulate the crash of a given process)

2.3 Group Manager

Stuff to discuss:

- A2A message (state that we decided to distinguish it from common data message) - explain how flush management works (e.g receivedFlush.isEmpty()) - slightly different protocol from the one given by Timofei - how timeouts after flush timeouts are avoided

We need to specify that participant multicast to everybody but itself, but the checker doesn't take into account that. I mean, the checker doesn't take into account the fact that a node can crash silently. So it signals that there are some messages that are not delivered to all the participant in that epoc, but those who're missing are not operational anymore, so it should be fine.

3 Crashes and Joining

The project allows the system to send different messages to make a chosen node crash. The crashing modalities are the following:

- **crash at any time** - immediately set the chosen node as crashed

- **crash on sending** - set the node crashed the next time it will perform a 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 nodes delivering the sent message.

- **crash on receiving** - set the node crashed the next time it will receive a message. In this way the node is not able to perform the delivery of the received message before crashing.

- **crash on view change** - set the node crashed the next time a 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 allow to add further members. To join a new node, a request is sent to the group manager, which will to fulfill it. The group manager will generate a new *id* for the new node and it will introduce the member into the group, triggering a view change.