Implementing Virtual Synchrony in a P2P group communication

Piermaria Arvani, S198465, piermaria.arvani@studenti.unitn.it
Lorenzo Brugnera, S197054, lorenzo.brugnera@studenti.unitn.it

***Abstract*—The project consists of developing a distributed application using Akka [1]. The actors taking part in the system are called Akka actors and they are able to exchange messages between each other under some conditions/restrictions. Each peer is both able to send and deliver multicast messages and the overall system is fault tolerant in case of silent crash of a group member.**

I. Introduction

The project aims at showing a simple message exchange mechanism between peers belonging to the same group. The actors taking part in the group are of two types:

- **Manager**
- **Partecipant**

There is olny one group manager and it is reliable, responsible for serializing group view changes and sending view updates to all the participants. Furthermore, it is also able to detect silent crashes and initiate view changes to notify the other members that the group has changed.
The exchange of messages is governed by the `Virtual Synchrony` protocol, which ensures on one hand that the communication between actors is consistent and on the other one that the system is fault tolerant in case of silent crash of one or more actors taking part in the group.

II. Implementation

The overall system has been developed in Akka. Akka provides a higher level of abstraction for writing concurrent and distributed systems.
The code is structured in two java classes:

- **CausalMulticast.java**, which represents the main of the application
- **Chatter.java**, which represents all the logic behind the system

Every message being exchanged in the communication has a specific structure according to the needed information to spread around. All the message types are `public static classes` implementing the `Serializable` interface.
To maintain information about the view and all the actors belonging to that specific view, we have defined a data structure called `Groups`. Every Groups instance is composed by the view ID, IDs and the list of the actors partecipating to that view.
In order to better understand the functioning of the system, the following flow graphs contribute to give the general idea.
Figure 1 shows whenever a partecipant wants to join a group. First, it contacts the manager. Then, the latter elaborates the request and, eventually, it adds the new peer to the group by sending him a new `JoinGroupMsg` containing the unique ID of the new member.
The Figure 2 shows the flow of execution to instantiate a new view. Whenever a new join request or a silent crash occurs, the manager receives respectively either a `RequestJoinMsg` or a `TimeoutMsg`. Afterwords, it instantiates the view change process and, eventually, all the operational peers receives the message and execute the flush protocol. This protocol is needed to make all the participants aware that they have received all unstable messages from all the members in the group: first, every operational peer sends all the unstable messages it owns in the queue to make them stable and second, multicasts a `FlushMsg` object indicating that they have completed the flush protocol. Only after having received the flush message from all the operational peers, a group member is allowed to install the new view.
In order to exchange messages, a peer has to have installed

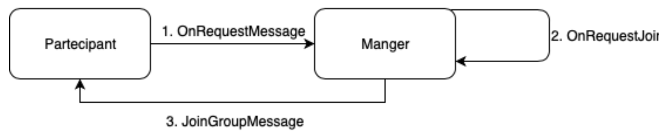

Fig. 1. The graps shows the flow of execution when a participant wants to join an existing group

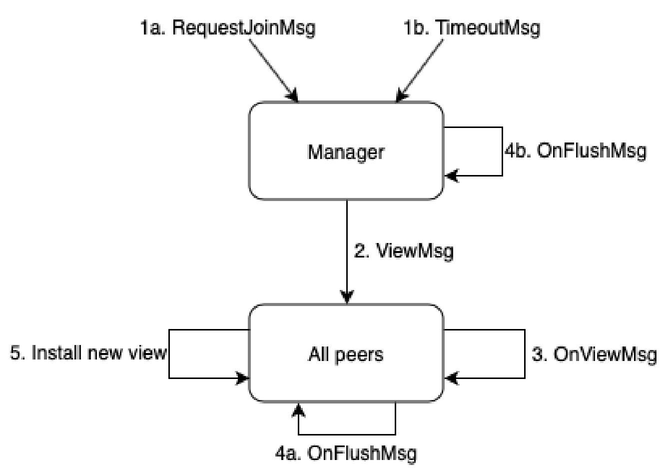

Fig. 2. The graph shows the flow of execution when a new view has to be instantiated

the most recent view. Simply, in the code this mechanism is handled by the `inhibit_sends` field, which has to be zero in order to allow the exchange of messages. Every time a view change has been instantiated the counter is incremented by one; on the other way round, each time a new view is installed, the counter is decremented.

Another important scenario to be described is the silent crash detection. The implementation takes place through a simple beacon mechanism: every five seconds each peer has to send a message (beacon) to the manager, indicating that specific peer is still alive. Since the channel is reliable, as soon as the manager does not receive a beacon from a participant, the group manager receives a `TimeoutMsg` indicating that a silent crash occured. In particular, he is able to understand which peer crashed, thanks to a dedicated HashMap where the key-value pair is represented by the peer and a `Cancellable` object. This data structure is used to have information about the time a peer sent the beacon. By associating to each participant the precise time in which its onw beacon was sent, the group manager is able to keep track about the alive peers. As far as the manager keeps on receiving beacons and updating the time of each peer, everything continues working properly. As soon as a beacon is missing, the manager receives a `TimeoutMsg` and he is able to identify the crashed member. Afterwords, a new view without the crashed node is instantiated.

Furthermore, the overall application tries to take into account possible delays over the channels: every message is propagated with an extremely small and random delay in order to obtain a more realistic scenario. Moreover, there are some supporting functions used to simplify the overall application: e.g., *DeleteOldMessages*, *FindDuplicate*, etc.

III. Conclusion

This project suggests a simple implementation of a distributed application according to the protocol. Everything works properly ad the overall system is very easily tesatble thanks to the **CausalMulticast.java** class.

References

[1] "Akka," https://akka.io.