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 system using Akka. The actors taking part in the system are called Akka actors and they are able to exchange of messages between each other under some conditions. Each peer is both able to send and deliver multicast messages and the overall system is fault tolerant in case of silent crash from 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 can be of two types:

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

The group manager 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 first that the communication between actors is consistent and secondly that the system is fault tolerant in case of silent crash of one or more actors taking part in the message exchange.

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
- **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 implemented by `public static classes` implementing the `Serializable` interface.
To maintain information about the view and all the actors taking part in that specific view, we defined a data structure called `Groups`. Every Groups instance is composed by the view, 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 give the main idea.
Figure 1 shows whenever a partecipant wants to join a group. First, it contacts the manager. The latter elaborates the request and, eventually, it adds the new peer to the group by sending

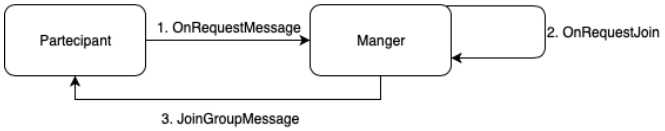

Figure 1. The graps shows the flow of execution when a participant wants to join an existing group

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 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. Only after having received the flush message from all the peers, the group members are allowed to install the new view.
In order to exchange messages, a peer has to have installed

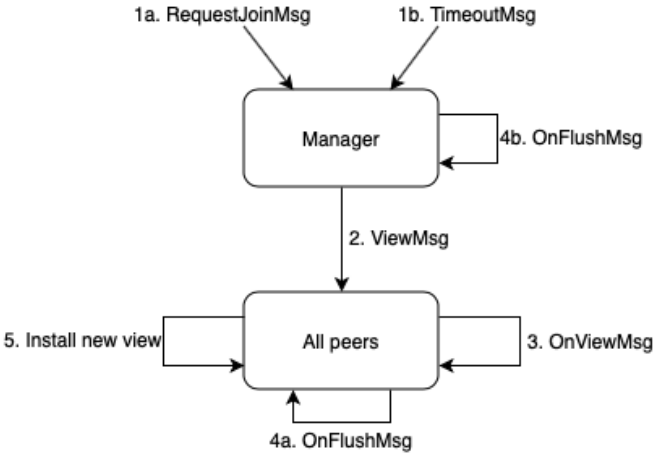

Figure 2. The graph shows the flow of execution when a new view has to be instantiated

the most recent view. Very simply, in the code this is shown by its `inhibit_sends` field, which has to be zero in order to allow the exchange.
Another important scenario to be described is the silent crash detection mechanics. The implementation occurs through a

simple beacon mechanism: every five seconds each peer has to send a beacon to the manager, indicating that 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 peer is crashed. Afterwords, it instantiate a new view without the crashed node and instantiates a view change.

III. Evaluation

IV. Conclusion

Again for very short documents the conclusions can be embedded with the last section, but this depends on the structure you give to your document.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks," *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–116, August 2002.

[2] I. Dietrich and F. Dressler, "On the Lifetime of Wireless Sensor Networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 5, no. 1, pp. 1–39, January 2009.

V. Appendix: LaTeX Tips

A. Cross References

B. References

If you want to cite something proved in another work, use citations. Search for the `bibtex` entry of the article you want to cite on *ieeexplore* or *google scholar*. Put your `bibtex` code into `references.bib` and cite them using the `\cite` command. For example: According to [1], $1 + 1 = 2$. The entries you cite will **automatically** appear in your references.

When citing more than a few pages worth of text, point the reader to the specific part you are referring to in your citation, like so: [2, Table IV].

Do not cite URLs. For pointing a reader to interesting websites, use footnotes.