Distributed Systems 1

Course Project: Virtual Synchrony

Students:
GENNARO CIRILLO – 198760
GIULIA ZANELLA – 195385

In this project we implemented a simple peer-to-peer group communication service providing the virtual synchrony guarantees. This project has been implemented in Akka with the group members being Akka actors that send multicast messages to each other. The system allows adding new participants to the group as well as tolerates silent crashes of some existing participants at any time. The program has been implemented as a single Akka system with multiple local actors.
The program generates a log file (*"project.log"*) recording the key steps of the protocol (install view, send multicast, deliver multicast).

Assumptions:
-We assume that the unicast message exchanged between actors is reliable.
-To emulate network propagation delays, we inserted small random intervals between the unicast transmissions of the multicast implementation, for both data and control multicasts.
-Multicast do not cross epoch boundaries: if there are multicasts and a view change occurs among them, these multicasts complete before the new view is installed (to satisfy virtual synchrony). This means that each multicast is delivered within the same view by all participants (except in case of crashes).
-There is a dedicated reliable group member (group manager) with ID=0, called "coordinator", responsible for serialising group view changes and sending view update messages to the group.
-To join the group, a participant contacts the group manager by sending a join request. Then, the coordinator initiates a view change, including the new participant to the group and sends in multicasts to the group that new view.
-A group member sends a new multicast only after it completed sending the previous one, otherwise the multicast is "unstable".
-The coordinator detects a crash of process P when no messages arrived from P within a predefined timeout T.
-The system detects duplicated messages and never delivers a message more than once.

Architecture:

- The program is composed of a Coordinator (with ID=0) and 3 Participants: p1, p2, p3.
- Class "VirtualSynchrony" contains classes "JoinMessage", "CrashedNodes", "Crash", "Message" which are used in the function "Main" and access to variables shared by all participants (global lists).
- Class "Node" is a common function for both Coordinator and Participants, implementing messages "View", "ViewUpdate", "StartChatMsg" and all methods regarding sending, receiving, delivering, multicasting of a message and the view updating.
- Class "Coordinator" extends class "Node"; it is specific for the coordinator and implements methods for the crash detection, setting the new view after a crash happens, updating the view, managing a request for joining of a participant.
- Class "Participants" extends class "Node"; it is specific for the participants and implements methods for init the view of the participant, init the id of participant, managing the crash (and emulate it), emulating the request for joining.
- In the "Main" function we configure the logger and, after creating the actor system (and coordinator, participant1, participant2), participants 1 and 2 request to the coordinator to joining. So it starts sending messages.

- We emulate the first crash after 10 seconds from the beginning of the execution and the second crash after 30 seconds from the first one. Both of them try to rejoin after a delay of 20 seconds.
- We emulate the request for join of the participant3 after 10 seconds of the beginning.
- Function "onStartChatMsg" (in the class "StartChatMsg") implements the random scheduling of multicast messages.

In order to test crash and the request for join, in the "main" function we prepare 2 pieces of code that could be uncommented in order to test both or only one of the two parts. Here below the 2 pieces:

```
//Function to check join and crash functionality (to test remove/add comments)
checkjoin(coordinator,system);
checkcrash(participant2,participant1,system);
```

In the "*checkcrash*" function we emulates 2 crashes: node p2 at first, then node p1 (after a while).
In the "*checkjoin*" function we emulate the request for joining of the node p3.
So you can run the program with 4 modalities:

1. without the request for joining of node p3 nor any crashes;
2. with the request for joining of node p3, but any crashes;
3. with any requests for joining, but crashes of node p2 and node p1;
4. with both the request for joining of node p3 and crashes of node p2 and node p1.

In order to test that duplicated messages are not sent, we can uncomment this piece of code:

```
//uncomment this row to test duplicated message (here we force the message counter to a number that is surely already been sent)
numberMessage = 0;
```

Running steps:
-p1 asks for joining
-p2 asks for joining
-coordinator assigns ids to p1 and p2 and updates the view, including also the 2 participants
-p1 e p2 installs the new view
-the multicast communication runs in a "shuffled" modality
-Messages arrived
-Messages are delivered
-p3 asks for joining
- coordinator assigns the id to p3 and updates the view, including also the p3
-All participants installs the new view
-p3 crashes
-Unless the crash is detected, the multicast keep on going
-p2 do not send the "Arrived" message to the sender, so that message can't be delivered
-the next multicast, within the same view, from the same sender, is unstable.
-At the crash detection, the new view is installed (first by the coordinator, then by the other participants, as usual). It does not contain the crashed node.
-now multicast proceeds stable

-after 20 seconds from the crash, nodes 2 asks for re-joining to the coordinator
-p2 is added to the coordinator’s view and then all participants install the new view
-p1 crashes (and the execution proceeds as occurred for the previous crash of p3).

Schema:
Here below a schema of a possible execution of the program:

Legenda:

↗ = Ask for join

↗ = Ack

↗ = Message delivered

↗ = Message “crashed”

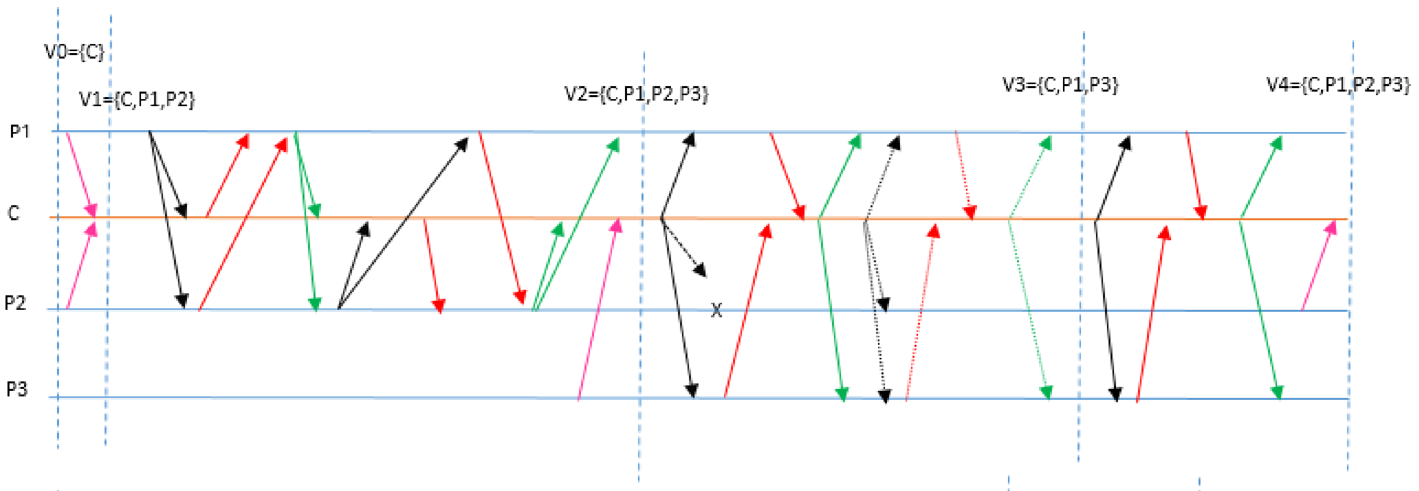
↗ = Multicast

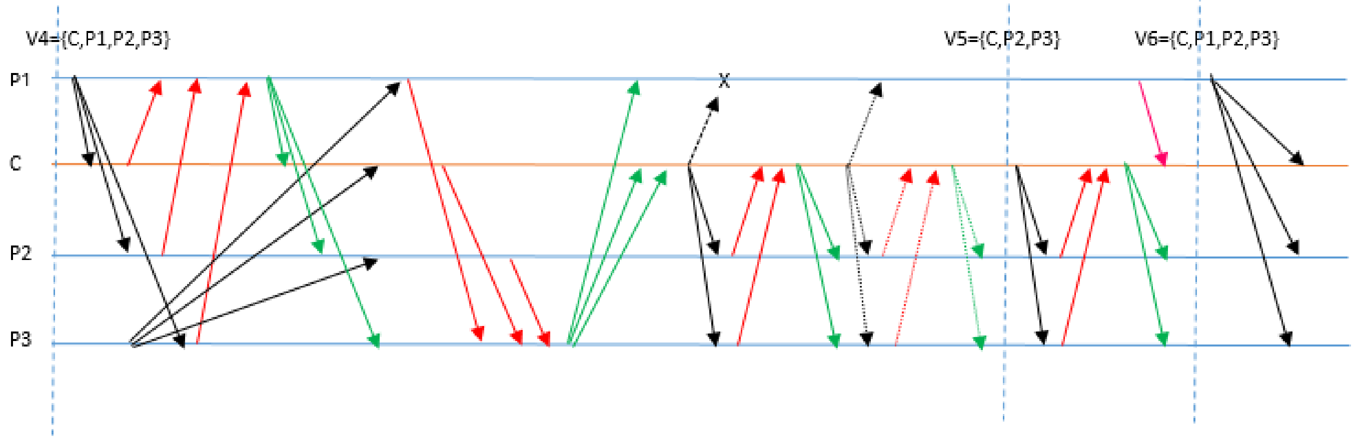
↗ = Unstable

↗ = delivered (unstable)

↗ = Ack (unstable)

┆ = Install view


V0={C}
V1={C,P1,P2}
V2={C,P1,P2,P3}
V3={C,P1,P3}
V4={C,P1,P2,P3}
P1
C
P2
P3
X


V4={C,P1,P2,P3}
V5={C,P2,P3}
V6={C,P1,P2,P3}
P1
C
P2
P3
X