

Project Report

Distributed System 1- 2017/18

Cheema Danish Asghar (196267) (danishasghar.cheema@student.unitn.it)

Khan Yasir (190465) (yasir.khan@studenti.unitn.it)

Introduction

This project is based on virtual synchrony which is an interprocess message passing technology that allows reliable multicasting in a group. The group is dynamic means that processes can join, crash or leave the group. Each message is delivered to everyone in the group member even if two messages are transmitted simultaneously.

For the implementation of the project we have added different cases. Each case is discussed in the implementation down below:

Implementation

We implemented the project in Akka in which group members are Akka Actors that send multicast messages to each other. One member of a group is considered as a General Manager that take care of crashes of members or joining of new members in the group. In the implementation we have consider different cases where a participant might crash or join the group. Each case is discussed below:

Normal Multicast “Case 1”

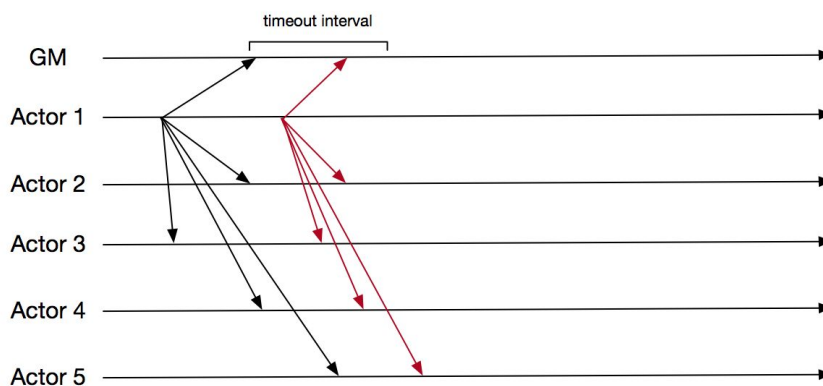


Figure 1: unstable message multicast (black), stable message multicast (red)

In case of normal multicasting we are telling every member of a group except General Manager to start sending multicast messages with a boolean **sendStable** to allow each member to send stable messages too as in this case no joining or crashes are allowed. Then every member goes to the **onStartMsg** where first it check the if condition that how many messages it has send already as every member is allowed to send maximum of two messages. Then we check another if condition for **inhibit_sends**. If inhibit_sends is greater than zero it means that there is a request for view change from General Manager so we are not allowed to multicast a message in this case and then it waits until the inhibit_sends equal to zero.

After both conditions are satisfied then each member create two messages, one for unstable and one for stable:

```
ChatMsg um = new ChatMsg(sendCount, this.id, false, false, this.view); //unstable msg
ChatMsg sm = new ChatMsg(sendCount, this.id, true, false, this.view); //stable msg
```

- The parameter of message **sendCount** identifies the index of the message.
- The parameter **this.id** tells us the id of the sender to know who is sending the message.
- The **first boolean** parameter shows us that the message is stable or not.
- The **second boolean** parameter identifies that the message is a normal message or a flush message.
- The last parameter **this.view** identifies that in which view the message is send.

After the creation of stable and unstable messages each member start multicasting unstable messages. In multicasting each member after sending a unicast message add a random network delay using Thread.sleep() around 10 millisecond.

Then other member of the group receive the message and in the deliver function he checks three conditions for different kinds of messages which are explain below:

1. For unstable message group member first add it to the buffer and then set the timeout for stable message using the scheduler. If the member did not receive the stable in that timeout then it goes to the onTimeout function where it check that the stable message is received or not. If the stable message is not received it tells the General Manager to install new view as the actor is crashed after sending unstable message.

2. For receiving flush message he adds the sender in the flush group list to know up to now how many members have sent the flush message.
3. For receiving stable message and if the message view is not the same as the receiver view then stable message will be discarded as according to flush protocol else we will iterate all the messages in the buffer to find the unstable message of the sender to remove it as it has received the stable message from the sender.

As unstable messages have been sent then the group member check the boolean `sendStable` to send the stable messages or not depending on different cases.

Add New Participant After Stable Multicast “Case 2”


In this case we tell the first participant to multicast unstable and stable message and then we tell the general manager to create new actor. The general manager (GM) now create a new actor by calling the following method:

```
//creating new participant
ActorRef newP= getContext().system().actorOf(Participant.props(this.group.size()), "participant" +this.group.size());
```

After that GM add the new actor to the group and then tell the new actor to join the group. Then the GM use the scheduler to wait for the new actor to update its group list before requesting everyone to change the view because the new actor is also now a part of the group. After the scheduler time GM tells everyone in the group including the new member to change their view.

In the view change method every actor first increment the local counter `inhibit_sends` to stop multicasting new message during the view change. Then we check the buffer of every member of the group that if it is not empty and it contains any unstable message, we try to multicast every unstable message it contains. After that the actor add himself to the flush group list. Flush group list is used to know that every member in the group has send the flush to everyone and every member has received it. After adding himself to the flush group list an actor multicast a flush message to everyone in the group. Then wait for a second by calling the method of a scheduler `scheduleOnce` so that everyone can send the flush message.

After timeout the group member goes to the method which we called flush timeout. On flush timeout first it checks if flush group list is equal to the group of members to be ensure that the actor has received the flush message from everyone. It clears the buffer in the current view so that any message that was sent in this view and has not yet been delivered it simply discards it. Then it changes its view and make the local counter `inhibit_sends` equals to zero to allow multicasting. Instead if the difference between the group and the flush group size is equal to one it means that one member of the group is



crashed before sending the flush message. Then it tells the GM about the crashed member by sending the crashed member of the group to the GM.

Crash First Participant After Stable Multicast “Case 3”

In this case we tell in the main method to the first participant in the group that start multicasting unstable and then stable messages. Then we use Time interval in main so to wait for all the unstable and stable messages to finish their multicasting. Then we tell the GM to crash the first participant in the group. GM in the **onParticipantCrashed** method simply remove the crashed participant from the group and tells every member in the group to change and install the new view. Then tell some other participant in the main after some wait to multicast a message in the new view.

Crash First Participant After Unstable Multicast “Case 4”

In this case we initialize the send stable boolean to false so that the first participant can only send unstable messages. The first participant multicast an unstable message and then any member of the group after receiving the unstable message wait for the stable message. After timeout of any member it will tell the GM that the first participant is crashed. So then the GM will remove the first participant from the group and tell everyone in the group to change and install the new view.

Crash Second Participant After Receiving Message “Case 5”

In this case for crashing the second participant we add in each actor a boolean for crashing that actor after receiving a message from any member of the group. We add another condition in the deliver method for this case. So first we tell the second participant to initialize that boolean to true. Then we tell the first participant to start multicasting messages. Every member in the group check the boolean if it is true then we tell the GM to crash the second participant.

Create New Participant and Crash Third Participant at View Change “Case 6”

In the last case we set the boolean **crashAfterViewChange** for the third participant to true because we want the third participant to get crashed before receiving the view change message. Then we tell the GM to add a new participant and then GM after adding a new participant tells everyone to change the view. Besides the third participant everyone will send the flush message. As no one will receive the flush message from third member so any one will tell GM that third participant is crashed because we have not received the flush message from it. So then GM will say change your view again because the third participant is no more the member of the group.

Commands to run the code:

(run this in the project directory)

You will see different options for different case select one form 1 to 6.

```
gradle run
```

This command will generate a text.log file which can to used to examine the outcome of the final result.

```
python3 parse.py text.log
```

By running the check.r script we can see the performance of the code by giving the text.log file as input generated during the last step.

```
Rscript check.r
```

Conclusion:

Depending on different test cases we had to use different time intervals to let different messages go through before sending new messages to see whether a particular case is doing what it suppose to do. Like in Case 2 we had to use Time interval for 1050 milliseconds before asking a new actor to start broadcasting new messages. Similar to other cases too. But for some cases, we used a scheduler to schedule the message in the future. In conclusion, we had to use the time delay for the dependent messages to different actors, concurrent messages to the same actor work perfectly fine.