A virtual synchronicity implementation with the Akka framework

Michael Giambi
University of Trento, Italy
michael.giambi@studenti.unitn.it

Diego Lobba
University of Trento, Italy
diego.lobba@studenti.unitn.it

ABSTRACT

In this report it is described the implementation of virtual synchronicity developed to have reliable multicasts among a set of dynamic partipants.

1. INTRODUCTION

The goal is to implement a reliable multicast algorithm among a set of participants. Participants are part of a group in which a message generated by one of the members is sent to every other actor within the group. Participants have a dynamic behavior, meaning that they can join or leave the group. We assume that a member leaves the group only in the case it is effectively crashed.

To simplify the algorithm we impose the presence of a special actor, the group manager, which cannot crash and is in charge to manage changes within the group.

The set of working participants known to every member of the group at a given time is called a *view*. Each actor (group manager included) has two views. One is the last *stable* view, correctly installed by all working participants. The other is a *temporary* view, generated when the group has been changed and every working participant has not the same group view yet. So the stable view is the same among all participants in the group.

A member has an identifier which is assigned by the group manager when joining the group as a participant. Once the id is assigned to an actor it cannot be assigned to other participants. Note that this means that the same actor is assigned different ids upon multiple join operations.

Multicast is implemented in a way that every working participant within the view receives the message and *delivers* it upon receiving. The message is saved by the actor until a separate message (the *stable* message sent by the multicast sender) is received. To avoid the possibility of delivering the same message twice each member records the last stable message it has received from a given process and checks whether a message next to be delivered was delivered previously (see question 1 in section 4).

To detect crashes within the current view the group manager schedules periodically a series of control messages (*AliveMessage*) to every member of the temporary view (see question 2 in section 4) and lists all actors the group manager expects to receive a reply from. Upon receiving the alive message a working participant replies to the group manager which marks the actor as alive. When a new alive check is triggered the group manager is able to determine whether some node crashed by checking if some actor in the list has not replied within a given time slot.

Finally, to provide a way to control the behavior of the system during the evaluation, an event-based system has been developed. Events are handled by the group manager. Whenever the group manager receives a message it computes the message label made of the sender process id and the multicast id. Messages labels are used to track the ongoing state of the system, so they are effectively used to track events. In case a message label has an associated event this is triggered.

2. ARCHITECTURE

The modelling of actors involved in the implementation is contained mainly in three classes (see appendix A for details): BaseParticipant, GroupManager and Participant.

BaseParticipant provides common attributes and methods shared both by the group manager and by a normal participant. The normal behavior for handling the receiving, the multicasting and the view changing is in this class. By default a base participant starts with both views identified by the -1 view-number. Also the default participant identifier is set to -1.

The *Participant* class provides a constructor that sends a join message to the group manager asking for an id. Additionally, since a normal participant can crash, a flag notifying this status is present. In "crashed" mode the participant is neither able to send or receive any message.

The class provides also methods to let the node crash under different scenarios. The following cases are covered:

- the participant sends a multicast and then it crashes without sending any stable message;
- the participant, in the process of sending a multicast, is able to send successfully the message to just one other member (with the exception of the group manager); the participant crashes thereafter;
- the participant receives a data message and then crashes;
- the participant crashes upon receiving a view-change message.

The *GroupManager* is the actor in charge of assigning id to new participants joining the group. It is the first actor to be started and **initially it does not generate any multicast**. The actor is the only one responsible to manage view changes triggered when a new node joins or when a node is crashed.

Crashed nodes are detected by the group manager by a simple crash detection algorithm that makes use of an ad hoc heartbeat message called *AliveMsg*.

Being the only participant that cannot crash, the group manager has been chosen to implement an event-based system to track the state of the protocol and to trigger events.

3. PROTOCOL BEHAVIOR

In this section details of the protocol in critical steps are discussed. In particular the focus is placed in the way participants act when receiving a view-change and when receiving a message. Finally the crash detection algorithm adopted by the group manager is analysed.

3.1 View changing

The BaseParticipant class implements core methods that effectively implement the protocol. Of major interest are actions performed when the group manager issues a view change. When a view change is in place, the group manager instructs every participant to stop sending new multicasts, then sends a message containing the new view to be installed. Therefore the participant interrupts the generation of any new multicast upon receiving a stop-multicast message.

When the view-change message is received the participant starts sending buffered messages to all members in the new view, followed by *FLUSH* messages. When flushes related to the view being installed are sent by all members in the new view, the step is finished and the view can be installed (fig. 1).

3.2 Message Reception

When receiving a data message the participant check whether the sender is in the temporary view members. If it is not, then the message will be ignored. The second check is related to the view, the message is received only in the case the view the message was sent in is the same as the current temporary view. If this is the case then the message is correctly received and delivered if it has not been delivered previously. To check the message is not a duplicate the participant delivers the message *iff* the message has a multicast id greater than the last message delivered coming from the same sender (in a similar fashion to vector clocks). This makes the check to avoid double delivery particularly robust (as explained in section 4).

Finally the participant stores the message in the buffer in the case the message is not stable, otherwise the actor will try to remove (if effectively saved in the buffer) the corresponding unstable message from the buffer.

3.3 Detecting crashes

To detect crashed nodes the group manager implements a simple algorithm which uses a special message. Participants have to reply to this message within a given time in order for actors to be considered alive.

The group manager stores a set of actors (*alivesReceived*) representing participants that have not replied to the alive message yet. Initially the set contains all members of the temporary view, with the exception of the group manager. Heartbeat messages are then sent to all actors in the set. When a participant receives an alive message it simply replies to the group manager with a message of the same type. Upon receiving the reply, the group manager removes the sender from the set of actors previously created. In this way the set ends up containing actors that have not replied to the alive request. The check process is scheduled on a given time basis. When the next check operation is triggered the group manager checks whether the set contains any actors. If this is case then those actors have not replied to the alive request within the established time slot, so it is assumed they crashed. The group manager will therefore start the view change process to install a new view (fig. 3).

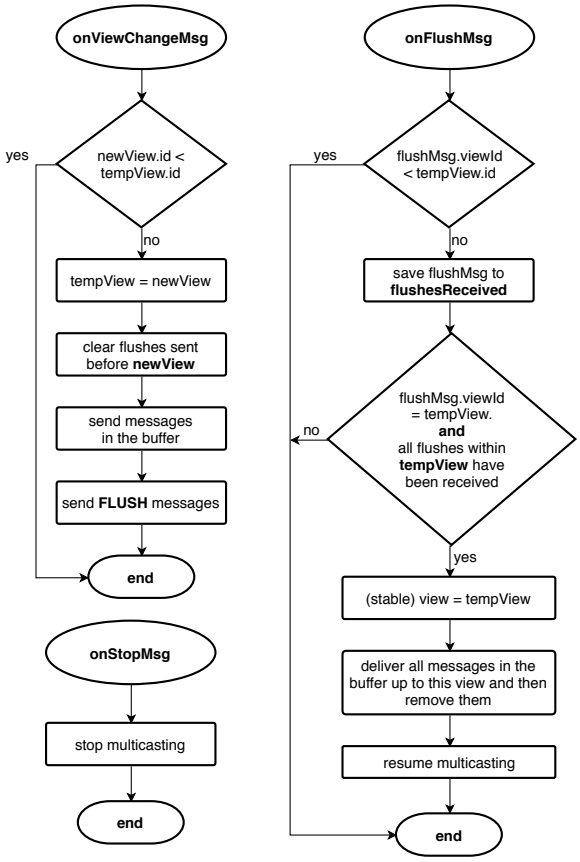

Figure 1: Events involved in the view-change steps.

4. Q&A

In this section a small set of questions that may arise are answered.

1 Why participants need to store the last message delivered from every other member?

This is required to avoid scenarios that could potentially lead to a double delivery of the same message. The message is kept into the buffer until a *STABLE* message is received.

Consider now the case where a message is received by three processes (one of them being the sender). Suppose now that the sender crashed while sending stable messages. Before crashing, only one actor is able to effectively receive the stable message and remove the message from the buffer.

When the crash is detected and a view change is issued the node still having the message in the buffer sends

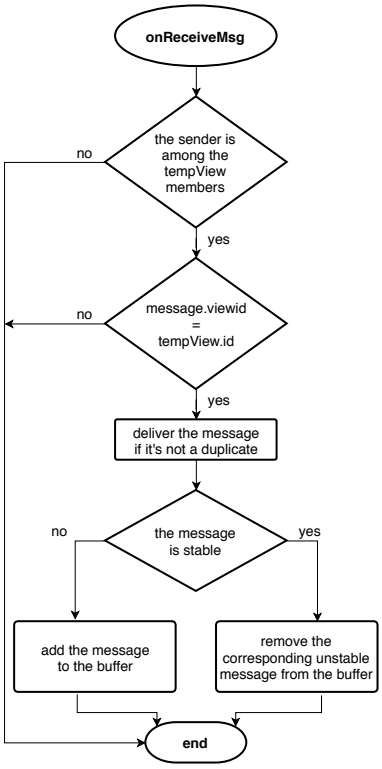

Figure 2: Flowchart for the reception of a message containing data.

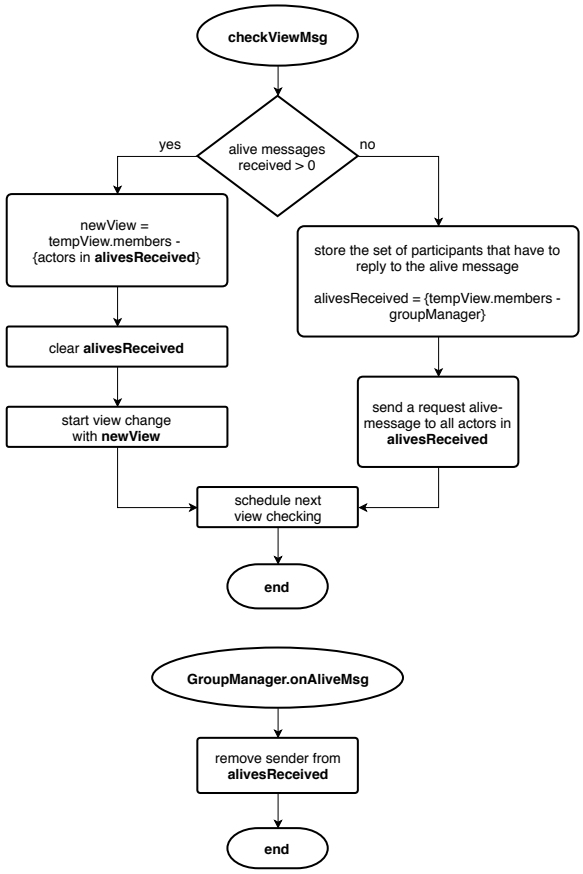

Figure 3: Flowchart for the crash detection scheme used by the group manager.

it to all participants in the current view, included the node that removed the message from the buffer. If the node did not save messages delivered in would receive the message as it was the first time (since it is not in the buffer now) and would proceed with its delivery. This means the message would be delivered twice, causing a misbehavior in the system.

2 Why the temporary view is considered in the crash detection algorithm?
The crash detection algorithm determines the status of nodes within a view. If the stable view was considered then it would not be possible to detect changes during the installation process for new views and could potentially lead to scenarios where the process just stalls. If the node is crashed and the group manager keeps on checking the status for the stable view then the crashed process is always asked for its status, resulting in a loop.

3 How delay in multicasts is implemented?
The implementation did not reproduce the use of the *thread.sleep* method showed in the laboratory classes since we believe it could interfere with the reception of messages. Since the use of the sleep method blocks the participant messages reception we thought it was inconvenient for the purpose.

We tried instead to achieve the same goal by using the *schedule* method provided by the Akka framework. Thus, messages are sent with a one second delay between each sending. Due to the fact that some process required a two series of multicasts performed sequentially, an estimation of the time required for the first multicast operation is computed and it is then added as a wait time for the second multicast.

As an example, let us consider the multicast operation. A first multicast is performed to send unstable messages to operational participants. An estimate T is obtained from this process, being T the result of a one second delay times the number of operational participants that will receive the message.

When the second multicast is performed the required time T previously estimated is added as the required time the second multicast has to wait in order to guarantee the FIFO properties provided by Akka.

5. CONCLUSIONS

This report described the architectural choices behind our implementation of an algorithm for reliable multicast. Core ideas and behaviors of the algorithm have been exposed, detailing its behavior in the core steps.

APPENDIX

A. CLASS DIAGRAM

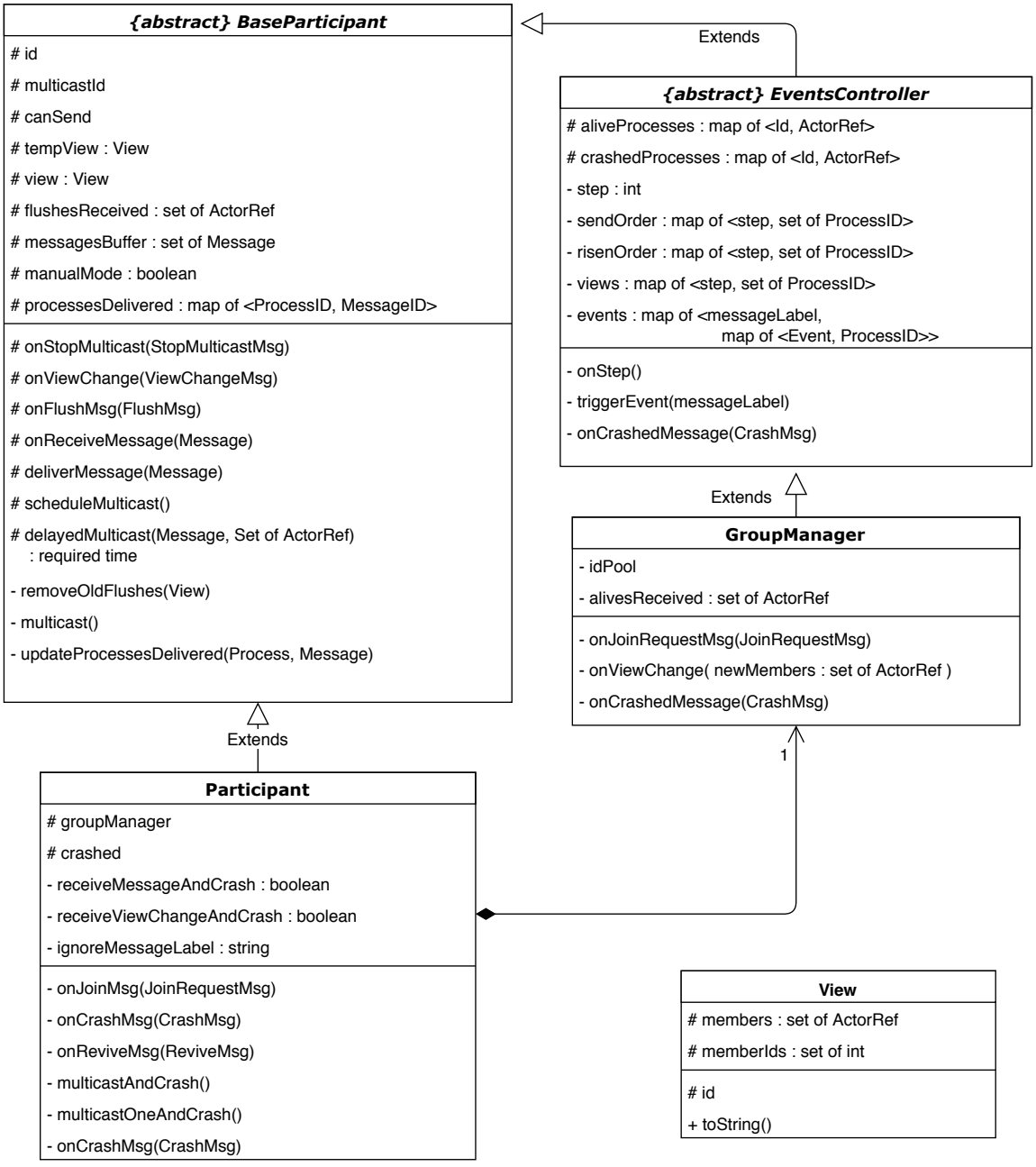

Figure 4: Class diagram of the main classes involved.