Report DS1 Project

Alessandro Torresani 181677
Emanuele Viglianisi 187270

Introduction

Structure
- Actors
  - Actor
  - GroupManager
  - GroupMember
- Messages
  - ChatMessage
  - HeartBeat
  - ViewChangeMessage
  - RequestJoinMessage
  - FlushMessage
  - Other Messages

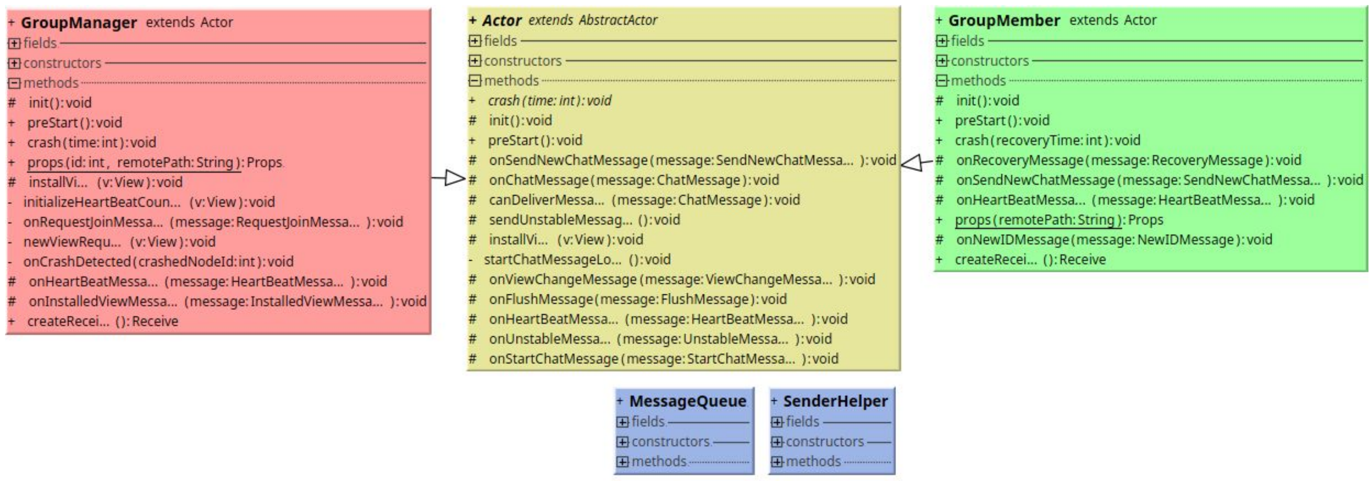
Logic
- Actor
  - Queue Multicast FIFO and scheduling
  - Delivery and unstable messages
- GroupManager
  - View Change
    - Node Join
    - Heartbeat and crash detection
- GroupMember
    - ID Request

Introduction

In this document we describe our implementation of a peer-to-peer group communication service providing the virtual synchrony guarantees. The system is developed using the Akka framework and, it is composed of multiple remote Actors communicating over the network. In the first part of the document we describe the architecture of the project, specifying the actors and the messages. In the second part we analyze how the system handles some of the main and critical events such as: joining a new node, crash detection, view change and the multicast of chat messages.

Structure

Actors


+ GroupManager extends Actor
fields
constructors
methods
# init():void
+ preStart():void
+ crash(time:int):void
+ props(id:int, remotePath:String):Props
# installVi... (v:View):void
- initializeHeartBeatCoun... (v:View):void
- onRequestJoinMessa... (message:RequestJoinMessa...):void
- newViewRequ... (v:View):void
- onCrashDetected(crashedNodeId:int):void
# onHeartBeatMessa... (message:HeartBeatMessa...):void
# onInstalledViewMessa... (message:InstalledViewMessa...):void
+ createRecei... ():Receive
+ Actor extends AbstractActor
fields
constructors
methods
+ crash(time:int):void
# init():void
+ preStart():void
# onSendNewChatMessage(message:SendNewChatMessa...):void
# onChatMessage(message:ChatMessage):void
# canDeliverMessa... (message:ChatMessage):void
# sendUnstableMessag... ():void
# installVi... (v:View):void
- startChatMessageLo... ():void
# onViewChangeMessage(message:ViewChangeMessa...):void
# onFlushMessage(message:FlushMessage):void
# onHeartBeatMessa... (message:HeartBeatMessa...):void
# onUnstableMessa... (message:UnstableMessa...):void
# onStartChatMessage(message:StartChatMessa...):void
+ GroupMember extends Actor
fields
constructors
methods
# init():void
+ preStart():void
+ crash(recoveryTime:int):void
# onRecoveryMessage(message:RecoveryMessage):void
# onSendNewChatMessage(message:SendNewChatMessa...):void
# onHeartBeatMessa... (message:HeartBeatMessa...):void
+ props(remotePath:String):Props
# onNewIDMessage(message:NewIDMessage):void
+ createRecei... ():Receive
+ MessageQueue
fields
constructors
methods
+ SenderHelper
fields
constructors
methods

Actor

Actor is an abstract class implementing methods that each node must have. For instance, all the nodes are able to send peer-to-peer messages, install a new view, deliver messages and so on. Some of the methods are overwritten in the two child classes: GroupManager and GroupMember.

GroupManager

The GroupMember class extends the abstract class Actor. The system contains exactly one GroupMember actor which is the only reliable node in the system. The GroupManager is in charge of detecting a node crash or the joining of a new node and, consequently, requests a new view change.

GroupMember

Unlike GroupManager, the system contains multiple GroupMembers. A new node can join to the group and may crash at any time. GroupMember implements additional methods for the ID request and to simulate the crashing and recovery.

Messages

Every message described in this section extends the abstract class *Message*. They inherit the attributes *senderId* and *viewId,* which represent the id of the actor sending the message and the id of the view in which the message is sent.

ChatMessage

ChatMessage is the type of the data messages sent by all the actors of the peer-to-peer system. Each ChatMessage contains an integer counter “*content”* which, together with the *senderId*, represent a unique identifier for the ChatMessage. ChatMessages are always sent in multicast to all the actors participating in the current view. There is a delay between a ChatMessage and the next one, given by the schedule of the message *SendNewChatMessage* which logically implements the *sending loop*.

HeartBeatMessage

This message is used by the *GroupManager* to check the status of the nodes inside the current view. The GroupManager sends the HeartBeatMessages in multicast and uses the replies to determine which of the current participants are still alive.

ViewChangeMessage

The *GroupManager* sends this message to notify a view change. It contains the attribute *View* which stores the id and the participants of the proposed view.

RequestJoinMessage

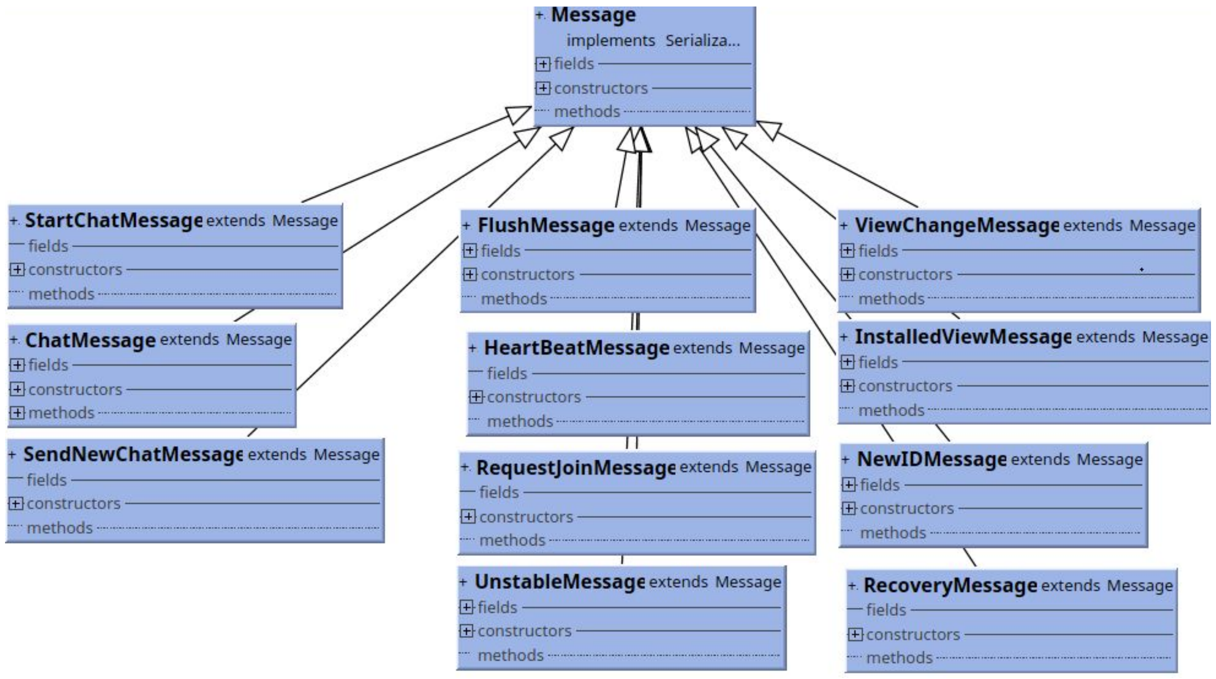
Every new node sends this message to the *GroupManager* to request the access to the “chat” session.

FlushMessage

This message is sent by all the participants of the conversation, after having received a view change message, to notify the others when they have finished sending all unstable messages and they are ready to install a new view.

Other Messages

In addition to the messages described above, which represent the core of the system, we defined some other types of messages. *NewIDMessage* is used by the *GroupManager* to assign an id to a new node.. *RecoveryMessage* is used to simulate a node recovery after a crash. *InstalledViewMessage* is used by the *GroupMembers* to notify the GroupManager when they have completed a new view installation. Finally *UnstableMessage* is used to send all the unstable messages in a single message.

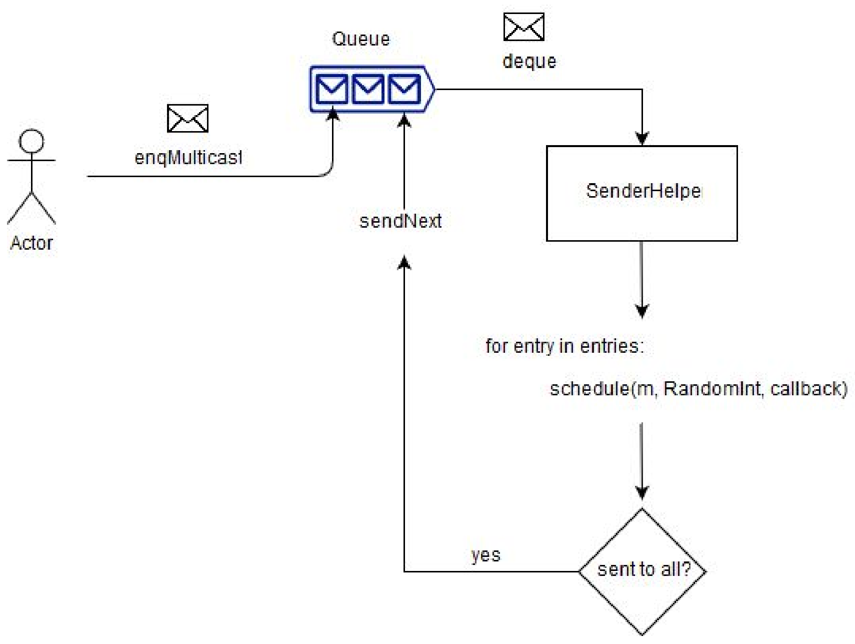

+ Message
implements Serializa...
fields
constructors
methods
+ StartChatMessage extends Message
+ ChatMessage extends Message
+ SendNewChatMessage extends Message
+ FlushMessage extends Message
+ HeartBeatMessage extends Message
+ RequestJoinMessage extends Message
+ UnstableMessage extends Message
+ ViewChangeMessage extends Message
+ InstalledViewMessage extends Message
+ NewIDMessage extends Message
+ RecoveryMessage extends Message

Logic

Actor

Queue Multicast FIFO and scheduling

One of the most challenging requirements of the system was adding a delay between each individual unicasts transmission of a multicast message. This delay is added to *simulate* a network delay between the transmission of the multicast message to each of the participant of the system. We tried differents techniques, one of these was adding a *Thread.sleep* between the unicast transmissions; however, this causes the complete block of the node, that was unable to receive and deliver messages in the meanwhile. The alternative, which is currently implemented, is the use of Akka Scheduler. A multicast message is instantly considered as sent, but the individual unicast transmissions are scheduled at a randomly chosen (bounded) time in the future. However, the scheduler introduced more difficulties to ensure the FIFO ordering among messages sent by the the same actor; the actor, for instance, could start sending a new message without waiting for all the previous message to be sent to all recipients. To solve this problem we implemented an Actor's Queue of MessageRequest and a SenderHelper class which deques and sends the next message of the queue only after the previous message is effectively sent to all the recipients.



Message delivery

The Actor class defines two sets: msg*Buffer* (contains unstable messages) and msg*Delivered* (contains messages already delivered). Upon receiving a ChatMessage, the callback function *onChatMessage* calls the function *canDeliverMessage* passing the received message as parameter. The received message is first added to the *msgBuffer* which will be cleared once that we are sure that all the messages that it contains are stable. To avoid delivering a message twice, we check whether the message is in the set *msgDelivered and,* If not, the message is delivered and added to the set.

GroupManager

View Change

The view change procedure is always initiated by the *GroupManager* when it receives a *RequestJoinMessage* or when it detects a node failure. In any of the two cases the *GroupManager* sends the new proposed view *v* through a *ViewChangeMessage* to all the members of the *proposed view v.* Upon receiving a *ViewChangeMessage,* every actor of the network, including the *GroupManager*, will stop sending chat messages. Moreover, each actor sends in multicast to every node of the proposed view its unstable messages inside *msgBuffer* (wrapping all the messages inside a single message called *unstableMessage)* and then a flush message (*FlushMessage*) which confirms the successful sent of all the *unstableMessages*. When an actor receive all the flush message from every actors in the proposed view, it will install the new view and send a confirmation to the GroupManager using the *InstalledViewMessage*. The groupManager, upon receiving all the *InstalledViewMessage* will send to all the participants a *StartChatMessage* to effectively start the message exchange within the new view. The install view procedure could fail if a node join or a crash is detected in the meanwhile; if so, a new install view procedure will start.

Heartbeat and crash detection

The *GroupManager* checks the aliveness of every node in the network using the *HeartBeatMessages*. It keeps a separate time counter for every node in the list: each specific node counter is reset every time that node replies to the HeartBeatMessage. If the counter of a node reaches a threshold value, that means this node didn’t reply to any of the recent *HearthBeatMessages* and therefore the *GroupManager will* consider it crashed. Additionally a *GroupMember* can crash intentionally when it has not received *HeartBeatMessages* for a long time.

GroupMember

New ID request

Using of the networked implementation it is possible to run nodes in separated machines. To simplify the process of configuration and execution of a new node we developed a python script *run_node.py <port-of-the-new-node> <address-of-the-node>* which create a new gradle configuration file specifying the address and the port of the new node and the group manager’s address (for debug, it is hard-coded with the address localhost:10000). The group member starts with a random negative ID and sends a RequestJoinMessage message to the manager asking for a valid ID for joining the group. On receiving the request, the manager will assign the next valid ID (implemented using an incremental integer) and it will start the view change pipeline, adding the new node to the set nodes in the next view.