GROUP MEMBERSHIP MANAGEMENT SERVICE

Distributed Systems - Exercise 2

JEANCARLO ARGUELLO CALVO

STUDENT ID; 17307291

Trinity College Dublin

Table of Contents

Requirements

The application must support a Group Membership Management Service which allows:

- A process can create one or more groups by specifying an address.
- Any process can join to one or more existing groups by specifying the respective address of the desired group to join to.
- Any process in a group can left this group.
- Every node must report a alive status to the other members of the group(s) it belongs.
- A process group must be able to detect if another node in the group has failed, and therefore communicate this failure to the remaining processes.
- All the nodes of the group must have a consistent view of its membership at any time. As a minimum, it is required that all members 'see' the same sequence of membership changes over time. Each group view will differ from the previous group view by the addition or deletion of exactly one member and the same sequence of group views should be available to each process over time.
- When a node fails, this failure must be detected, and as a consequence, the node must be removed from the group. This is, the node must be removed from the group view.
- The group membership service must be fault tolerance in the presence of unordered messages, duplicated message, lost messages, latency, node failure and network partition.
- Any node must be able to return the current view of the group when required.
- The application must have the capability of injecting the failures mentioned previously to the group membership service.
- Every process must be executed independently through a User Interface application.

Specification

Every node process is a desktop application which allows the user to specify the identifier of the node as first mandatory step. After this, the user can join the node to any desired group or create a new group. This user interface also allows to visualize all the groups the node belongs to by using tabs. On each tab, the current view and a history of the views of that group can be visualized as list. Besides that, Every tab contains controls to either leave the group or inject failures from and for this node. This UI uses the group membership service API.

The Group Membership Service allows to join to a group or to create a group where none nodes have joined by specifying a multicast address, to leave the group, to obtain the current view, to subscribe to failures handlers notifications, to inject failures such as network partition, duplicate messages, lose messages and fails the node. This API is used for every node independently.

The Group Membership Service is built on multicast IP as Internet protocol. This allows to create groups by specifying a multicast address. The communication of the nodes in each group is done by multicasting messages to the whole group.

The service keeps a consistent view by using versioning of the view, which allows to identify the last/current view. Every view version differs by one deletion or addition of a process group. For example, when two of these events happen, two or more view versions must be created. Every node possesses a group view which is updated over the time when events happen.

The processes can join or leave a group by using the proper method of the Java Multicast API [1]. When a node joins and no other node is in the group, this will be the creation of a new group with the starting view version ‘1’, where the only node in the view is the node that just joined/created the group. If at least one node is already in the group that the node is joining, this existing node or nodes respond to the new node with the current view of the group where this new node is part of it.

The service is fault tolerance to different scenarios. A detailed explanation can be found in the implementation section. Messages lost and latency are managed by retransmitting the message after timeouts of 2 seconds. Duplicated and unordered message are detected by using a sequential identifier to each view message, so every receiver only processes the message with identifier *i* once. The failure detection and monitoring of members in the group is done by using a heartbeat that is sent by every node to the whole group every second. The network partition are handled naturally by the failure detection, because when the group is partitioned on two or more subgroups, the nodes in each of these subgroups cannot communicate with nodes in other subgroups. Therefore, the nodes will be detected as failure nodes and they will be removed from the current view of the subgroup.

Architecture

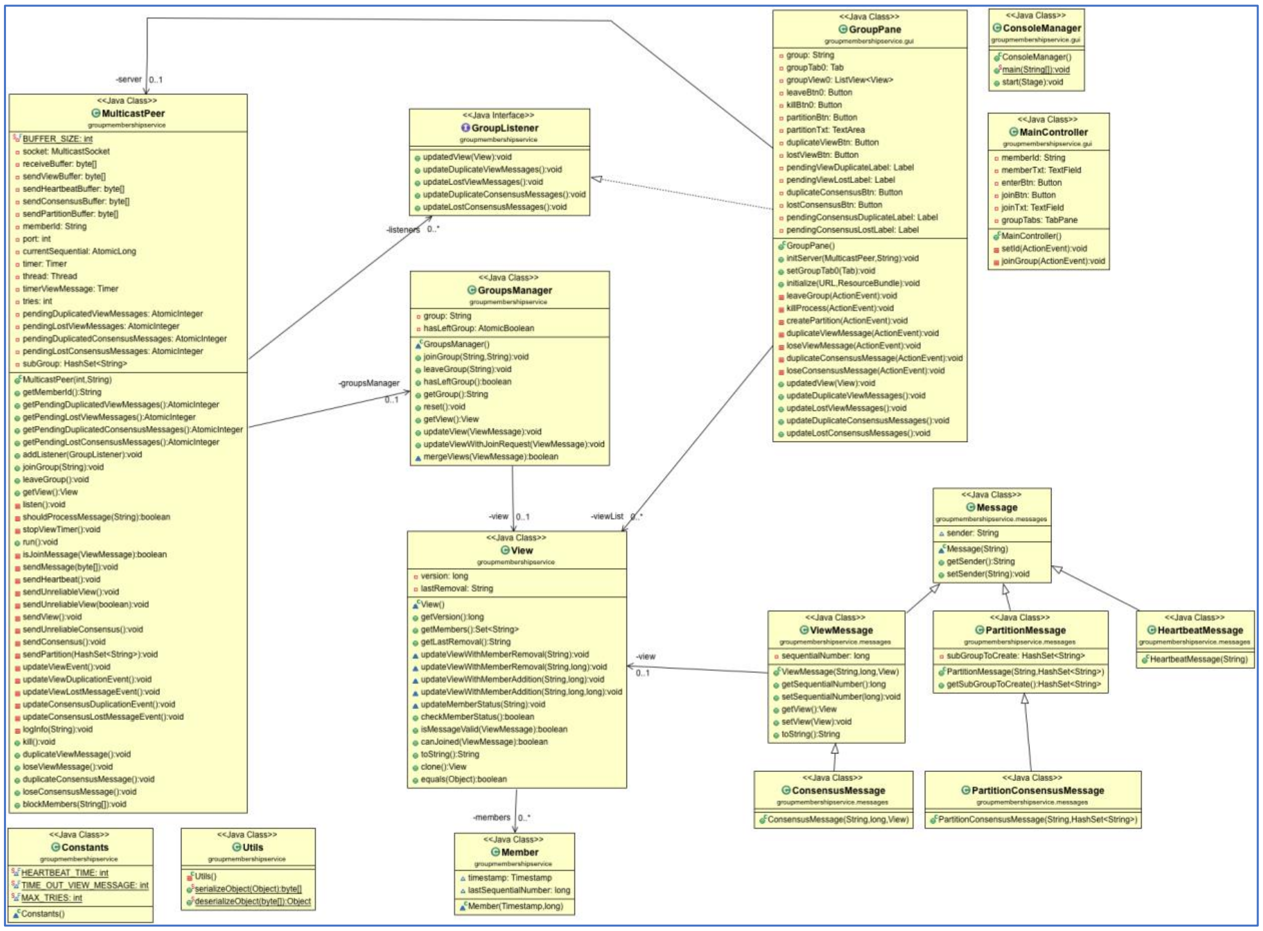


Figure 1. Class diagram for the whole system

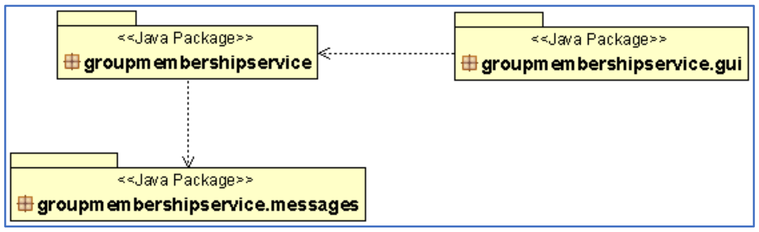


Figure 2. Package diagram

Figure 1 shows the classes implemented for the whole system, this includes the UI and the API. The figure 2 shows the relationships among the packages.

The description of the most relevant classes is the following:

- **Member:** It represents the tracking information about a node in the group. This includes the last sequential identifier received from this nodes and the last timestamp when a heartbeat from the given member was received.
- **View:** it represents the group view. It contains the version of the current view and a set of member in that view version.

- **GroupManager:** It controls the changes in the group. It controls the changes in the view. It controls the removal and addition of new members as updating and merging views. It controls the joining and leaving capabilities for the node in a group.
- **MulticastPeer:** This class is the public interface. It handles the interaction with the client by allowing join and leave functionalities, and also the subscription to failures and the injection of failures. It also controls the communication of the node. This class is always listening to new communications of the group such as view, heartbeat and partition messages. *Note: This class is big and have several responsibilities, in a good design this class should be refactored on several classes where every class is decoupled and has only one responsibility.*
- **ViewMessage:** It is the type of message that is transmitted for view updates. It contains the view information.
- **PartitionMesage:** It is used to inject a network partition to the system. It contains the subgroup that is going to be created.
- **HeartbeatMessage:** it is the message used to report the a node is alive. It is used for monitoring and node failure detection.
- **ConsensusMessage:** View and Partition consensus messages are used as part of the protocol to sync the respective change.
- **GroupPane:** It is a component of the User Interface, specifically a tab. As every tab is a group interaction, every tab represents and isolates this communication to only one group per tab. It contains the logic to handle to incoming and outgoing interaction with the MulticastPeer class. It sends the proper request to the API when an event is triggered from the interface such as when a “Lost Message”button is pressed. Also, it subscribes to failures handling of the MulticastPeer, so it receives notifications when a failure was handled in order to reflect this in the interfaces. The main notification subscription is done to update in the interface the current group view.
- **MainController:** It is the main UI component. It handles the events triggered from the UI controls outside the tabs.

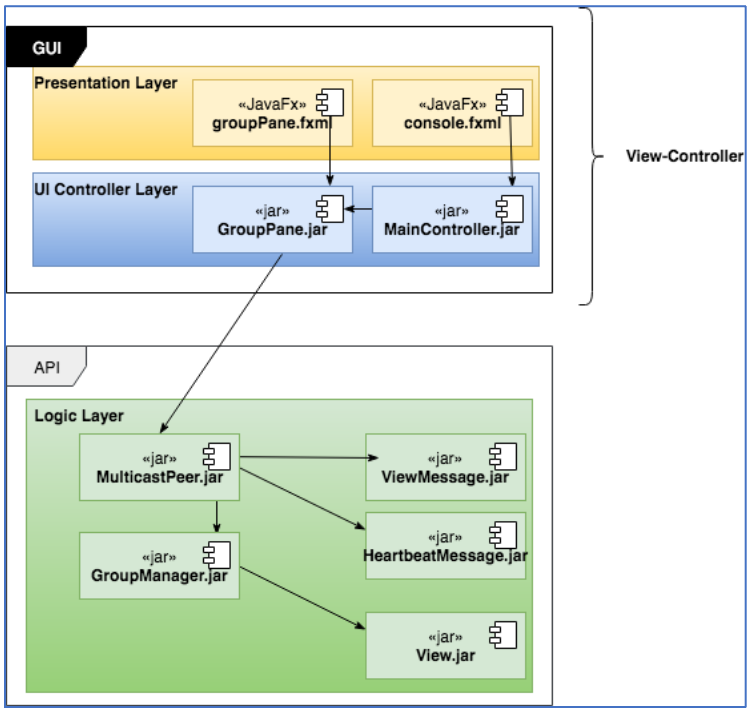


Figure 3. Block Diagram with main components

The figure 3 shows how the main components interacts. The application consists of two main modules, the Graphical User Interface (GUI) and the Application Programming Interface (API). The API manages all the logic related to the Group Membership Service. The GUI handles the events from the user interfaces to either updates UI components when a notification from the API is sent or when the user interacts with any UI component. The application was built on Java 8. The user interface was built with JavaFx [2], which allows to implement a View-Controller Design. The presentation layer is specified in the fxml files and the control of events from the presentation layer is managed by java files in the controller layer. The API layer is pure Java without interaction with any third party libraries.

Implementation: Protocol

Communication Mechanism

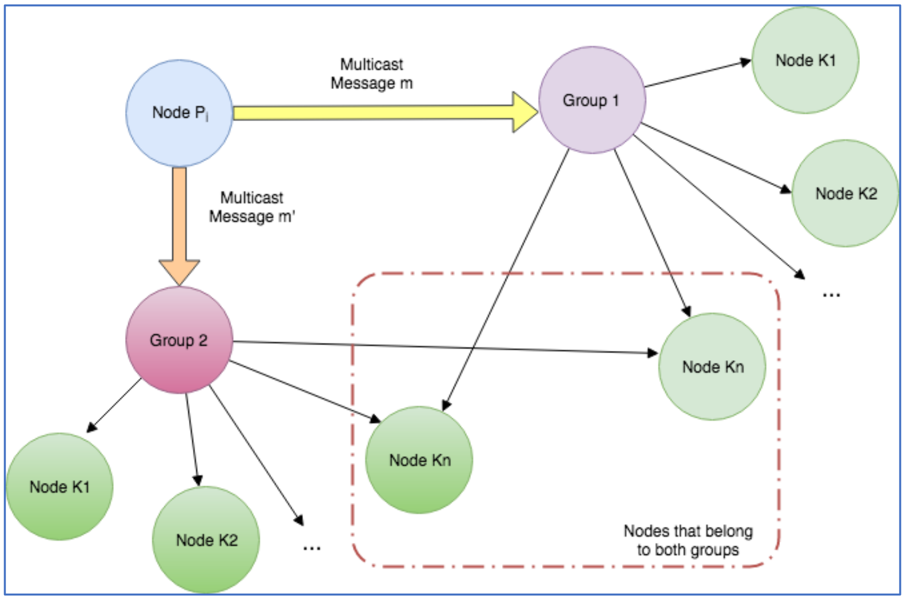


Figure 4. IP multicast as communication internet protocol

The group uses multicast IP to send messages, therefore every message is sent to all the members of the group including to the sender process. Nonetheless, the sender ignores its messages. This is expensive for large groups, so this implementation is not intended to be scalable. If one wants to create a scalable Group Membership Service, this can be implemented by using dissemination algorithms such as Gossip. For matter of this assignment the decision of using multicast was based on simplicity and that the specification does not state the need of a scalable protocol. This relies on the principle of 'Keep It Simple'. The communication uses a FIFO ordering to synchronize the group view.

Synchronize View

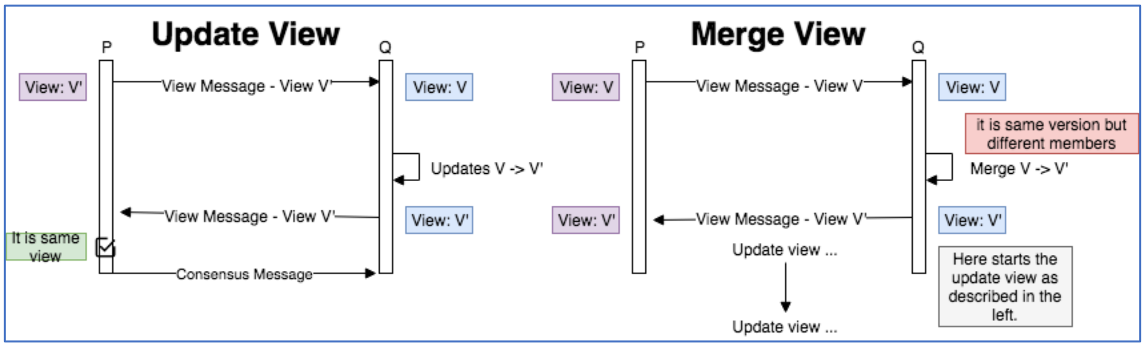


Figure 5. Update and merge views algorithms

As can be seen from figure 5, the synchronization of the view can happen in two manners, by updating or merging. Firstly, an update view starts with P sending a newer version of the view (V’) that Q receives and it applies the changes to the members collection and it updates the version. Then, Q send back the new version (V’) to the group to ratify the version. When a node receives a view with the same version as it possesses, it can send a Consensus messages to notify that this is the most recent view it knows. The consensus message allows Q and P processes to stop waiting new versions of the view and to confirm that it has the last version. The alternative is to merge the views, this happen when the sender and the receiver processes has the same versions, so the receiver needs to merge both views and to start a update view way as described earlier.

Joining and Leaving Groups

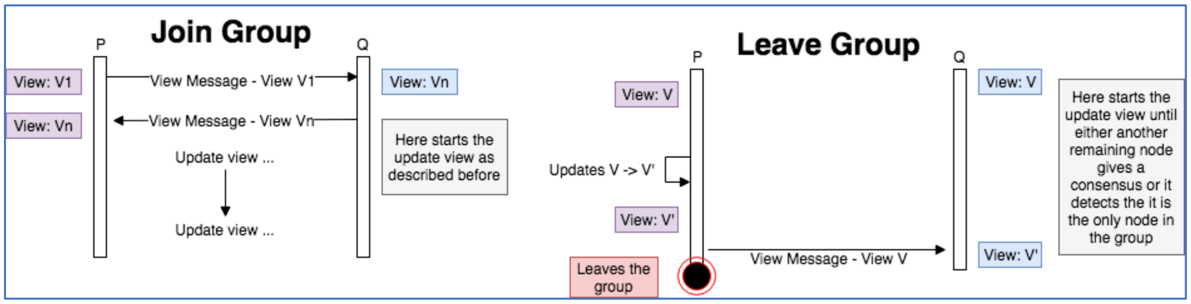


Figure 6. Join and Leave algorithms

The joining process P will always send a view with version 1 (V1). If there is another process Q in the group, this latter will send the current view of the group (Vn). So, P can start an update view protocol.

The leaving process P updates the view from V to V’, where V’ does not contain P. This view V’ is sent to the group. Before gets any another message, P leaves the group. Hence, the remaining nodes will get a consensus without P. If Q is the only process in the node, it will notice this by using the heartbeat mechanism, and therefore it will get a consensus by itself.

Implementation: Handling Failures

Message Lost and Latency

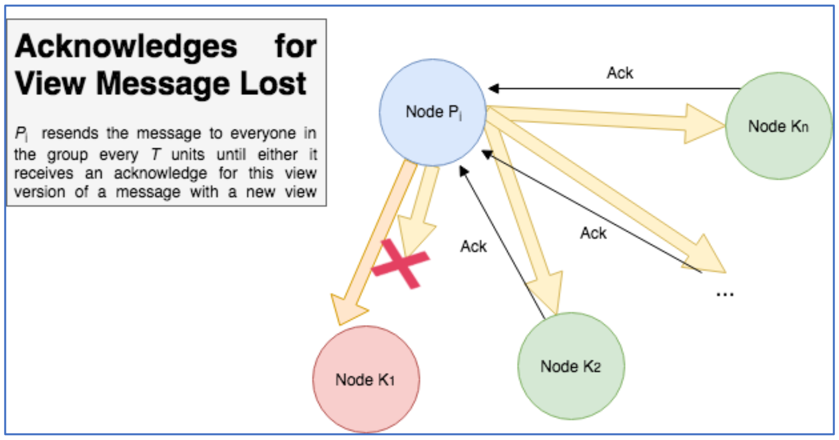

Acknowledges for View Message Lost
P_i resends the message to everyone in the group every T units until either it receives an acknowledge for this view version of a message with a new view
Node P_i
Node Kn
Node K1
Node K2
Ack
Ack
Ack
...

Figure 7. Lost messages handling

As can be seen in figure 7, P_i resends the message to everyone in the group every T units until either it receives an acknowledge for this view version of a message with a new view version. In this implementation *T* units is 2 seconds. It stops resending the message when it receives one of these two kind of messages. If Node P_i stops resending the message before Node K_n receive it, there is a possibility Node K_n will receive the view message from K – 1 nodes. In the case where only P_i and K_1 are in the group, the view is going to be certainly delivered to K_1 because P_i cannot receive notification of any other node. If K_n fails, P_i will know by using monitoring mechanism and it will stop the resending message.

In the case of the consensus message is lost the same case solve the problem. Given that P_i has not received a consensus message, it will resend the same message, therefore Node K_n will receive this view and then send the consensus again. This is a consequence of the message view replication. In conclusion, the protocol does not need to implement a resending mechanism for the consensus because it is given by the view resending implicitly.

Duplicated and Unordered Messages

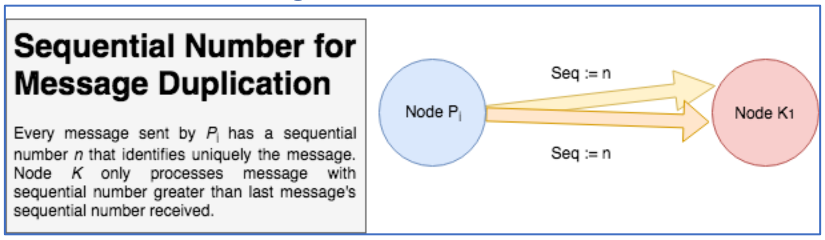

Sequential Number for Message Duplication
Every message sent by P_i has a sequential number n that identifies uniquely the message. Node K only processes message with sequential number greater than last message's sequential number received.
Node P_i
Node K1
Seq := n
Seq := n

Figure 8. Sequential identifiers to detect unordered and duplicated messages

In order to avoid duplicated messages, every message sent by P_i has a sequential number *n* that identifies uniquely the message. This number is incremented by one on every message. Node *K* only processes messages with sequential numbers greater than last message's sequential number received. Therefore, old messages and same messages are ignored. Old messages can be delivered if a newer message is delivered first because of UDP does not guarantee ordering, however due to the nature of the application old messages can be forgotten without consequences because the application only cares about the most recent view, not on every view version.

Monitoring members and detecting member failures

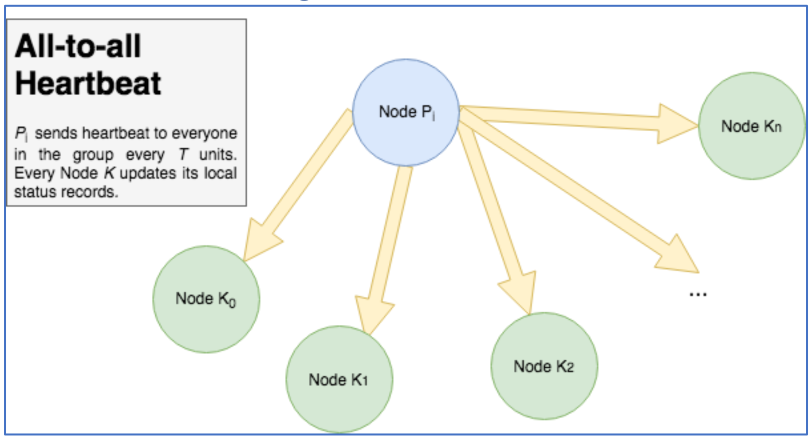

All-to-all Heartbeat
P_i sends heartbeat to everyone in the group every T units. Every Node K updates its local status records.
Node P_i
Node Kn
Node K_0
Node K1
Node K2
...

Figure 9. Heartbeat to detect node failures

As shown in figure 9, each process P_i sends out heartbeats every *T* units to all the other processes in the group. In this implementation *T* units is 1 second. However, if a node *K does* not receive a heartbeat in *T* about node P_i , this latter is not reported as failed until *T * 2.5* units. This is with the purpose of dealing with latency and messages lost. So, this allows to not report a node as failed when it just had a delay or the heartbeat message was lost.

This approach allows to distribute the load equally across all the members. The protocol is complete, if P_i fails, as long as there is at least one other non-faulty process in the group it will detect P_i as having failed. If the only node in the group fails, none needs to be notified and the group is dissolved.

Network Partition

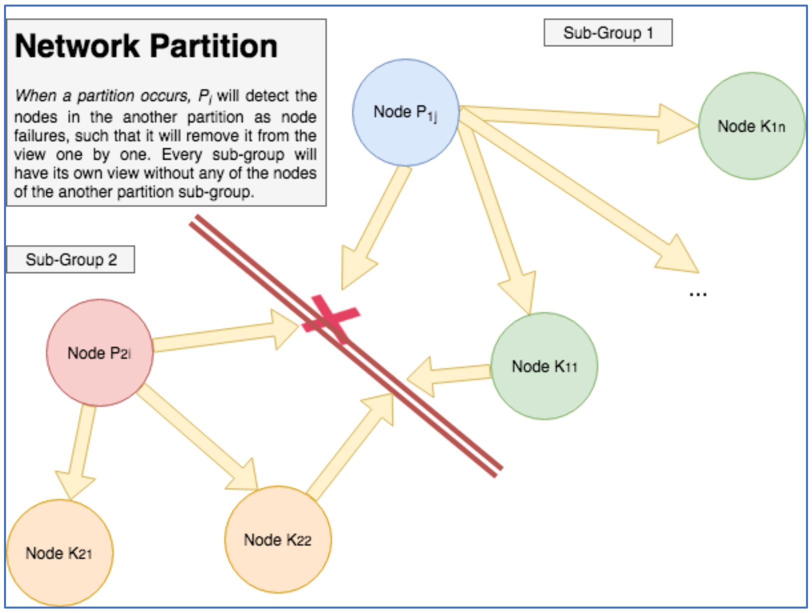
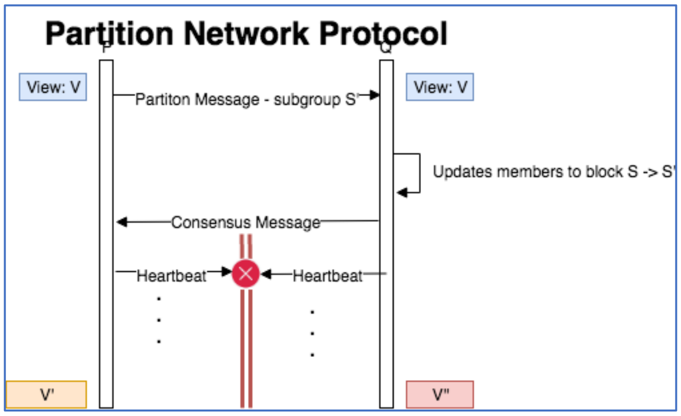

Network Partition
When a partition occurs, P_i will detect the nodes in the another partition as node failures, such that it will remove it from the view one by one. Every sub-group will have its own view without any of the nodes of the another partition sub-group.
Sub-Group 1
Sub-Group 2
Node P_{1j}
Node K1n
Node K11
Node P2i
Node K21
Node K22
...

Figure 10. Network Partition handling

Network partition can happen at any time. The way how this service handles it is by leveraging the monitoring and detection of members mechanism. When a network partition occurs, it creates implicitly two subgroups for the same group, this is, both groups has the same multicast address but the nodes on each group cannot communicate with each other. Because of this, the heartbeat mechanism will detect as failure nodes the nodes in the another subgroup. Therefore, each subgroup will remove the nodes of the another group. Hence, if the view of the subgroup 1 is the set S_1 which contains the nodes $K_{11},\dots, K_{1n}$, and the view of the subgroup 2 is the set S_2 which contains the nodes $K_{21},\dots, K_{2n}$, then the intersection of both view is an empty Set ($S_1 \cap S_2 = \emptyset$). This means, that any node in a subgroup is not part of the another subgroup. The union of these two sets will create the previous group S ($S_1 \cup S_2 = S$).


Partition Network Protocol
P
Q
View: V
View: V
Partiton Message - subgroup S'
Updates members to block S -> S'
Consensus Message
Heartbeat
Heartbeat
V'
V''
Figure 11. Network Partition injection Algorithm

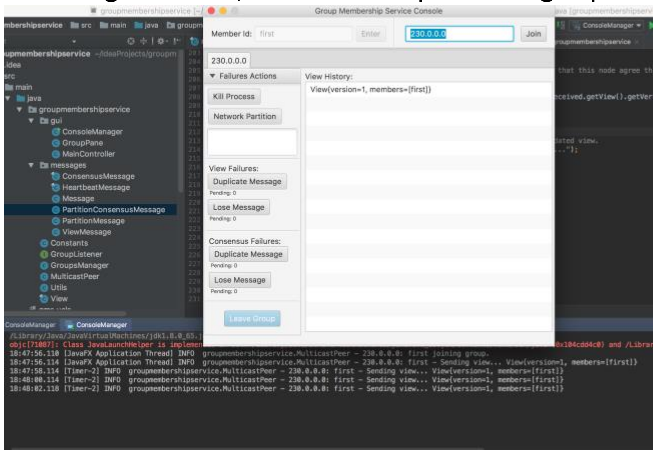
The figure 11 illustrates how the network partition is injected in the system. A node receives a list of nodes to isolate, this list is sent to all the members. They update the list of nodes to block and send a consensus message to notify the it has blocked the members. After this moment, the nodes that receives messages from nodes that belongs to that list are ignore them. The other way around is also implemented in order to allow two subgroups work separately. So, if a node which is in the list, receives a message from another node in the list, this node will process the message, and the messages from nodes that are not included in the list are ignore. This allows to block the communication among the nodes in the list and the nodes that are not in the list creating two subgroups, and simulating effectively a network partition. If a node joins, it will receive the list of members to block if it exists. So, new nodes will respect the partition even when the a node failed and recovered itself.

Testing

All the testing were done using the user interface.

Create Group

Plan: One node join to group where there is no nodes.
Results: The node joined a create the view with version 1 where only the current node is. Starts sending replies to reach existing nodes. After determined retries over the determined time outs, none response is gotten. So, the node stops sending replications.

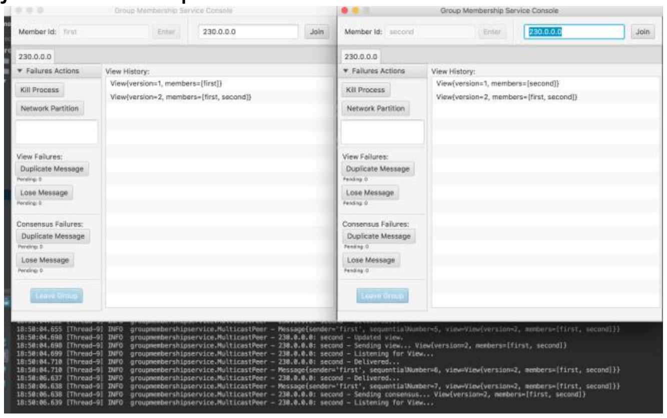


View{version=1, members=[first]}

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_65.j
objc[71087]: Class JavaLaunchHelper is implemen                                                        0x104cdd4c0) and /Librar
18:47:56.110 [JavaFX Application Thread] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first joining group.
18:47:56.114 [JavaFX Application Thread] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Sending view... View{version=1, members=[first]}
18:47:58.114 [Timer-2] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Sending view... View{version=1, members=[first]}
18:48:00.114 [Timer-2] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Sending view... View{version=1, members=[first]}
18:48:02.118 [Timer-2] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Sending view... View{version=1, members=[first]}
```

Join Existing Group

Plan: Second node joins to existing group.
Results: Node joins sending the view of version 1. Existing node responds with a updater view version 2. The joined node update the view.



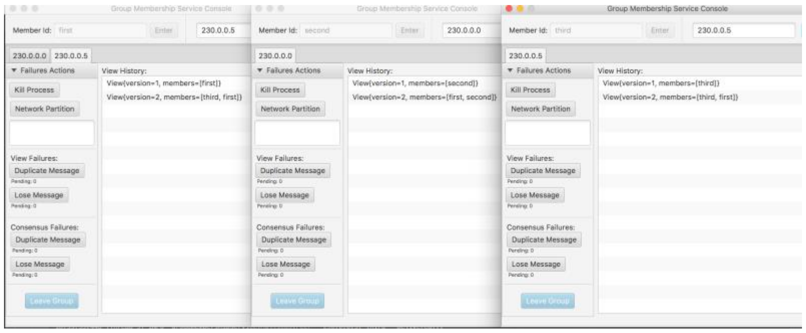
View{version=1, members=[first]}
View{version=2, members=[first, second]}

View{version=1, members=[second]}
View{version=2, members=[first, second]}

```
18:50:04.655 [Thread-9] INFO  groupmembershipservice.MulticastPeer - Message{sender='first', sequentialNumber=5, view=View{version=2, members=[first, second]}}
18:50:04.698 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: second - Updated view.
18:50:04.698 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: second - Sending view... View{version=2, members=[first, second]}
18:50:04.699 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: second - Listening for View...
18:50:04.710 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: second - Delivered...
18:50:04.710 [Thread-9] INFO  groupmembershipservice.MulticastPeer - Message{sender='first', sequentialNumber=6, view=View{version=2, members=[first, second]}}
18:50:06.637 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: second - Delivered...
18:50:06.638 [Thread-9] INFO  groupmembershipservice.MulticastPeer - Message{sender='first', sequentialNumber=7, view=View{version=2, members=[first, second]}}
18:50:06.638 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: second - Sending consensus... View{version=2, members=[first, second]}
18:50:06.639 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: second - Listening for View...
```

Join another Group

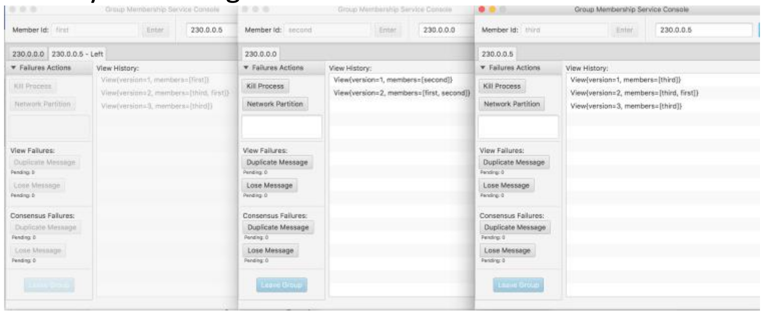
Plan: Node join another group.
Results: The node can be in two groups independently. It has independent group views for every group.



```
Member Id: first    230.0.0.5    Tabs: 230.0.0.0 | 230.0.0.5
View History:
View{version=1, members=[first]}
View{version=2, members=[third, first]}

Member Id: second   230.0.0.0    Tabs: 230.0.0.0
View History:
View{version=1, members=[second]}
View{version=2, members=[first, second]}

Member Id: third    230.0.0.5    Tabs: 230.0.0.5
View History:
View{version=1, members=[third]}
View{version=2, members=[third, first]}
```

Leave Group

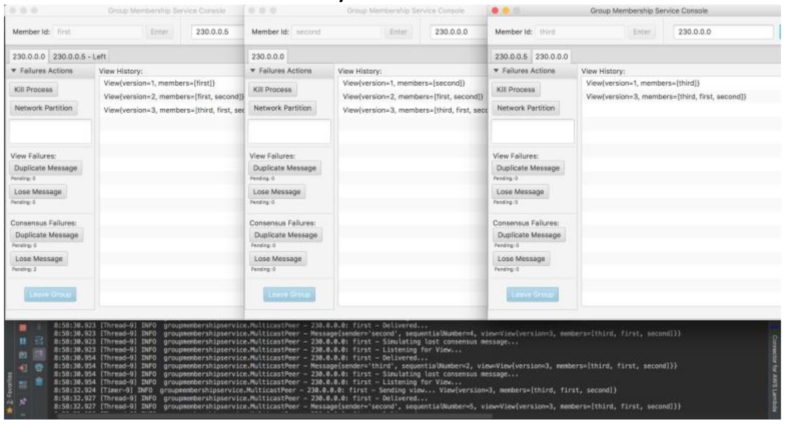
Plan: Node leaves one of the groups.
Results: The node left the group. The view is updated without the node on it. The another group is not affected by this change.



```
Member Id: first    230.0.0.5    Tabs: 230.0.0.0 | 230.0.0.5 - Left
View History:
View{version=1, members=[first]}
View{version=2, members=[third, first]}
View{version=3, members=[third]}

Member Id: second   230.0.0.0    Tabs: 230.0.0.0
View History:
View{version=1, members=[second]}
View{version=2, members=[first, second]}

Member Id: third    230.0.0.5    Tabs: 230.0.0.5
View History:
View{version=1, members=[third]}
View{version=2, members=[third, first]}
View{version=3, members=[third]}
```

Consensus Message Lost

Plan: Lost consensus messages are set in the UI. A new node joins to the group.
Results: The consensus messages are lost, so the new node send several view messages to check the view version. The application simulates the loses by not sending them. In the end, the consensus is sent and the views are synchronized across the 3 nodes.



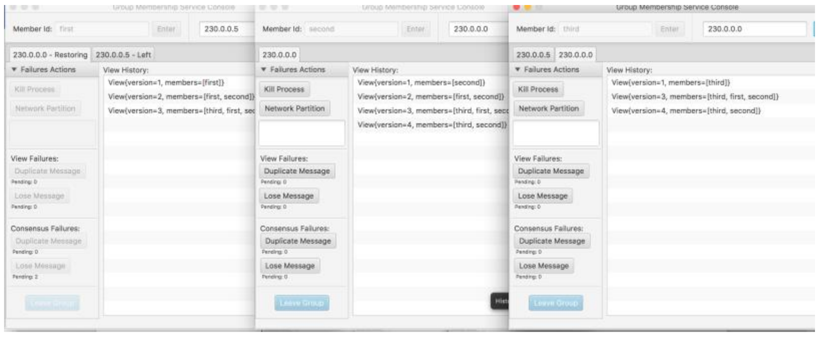
```
Member Id: first    230.0.0.5    Tabs: 230.0.0.0 | 230.0.0.5 - Left
View History:
View{version=1, members=[first]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec

Member Id: second   230.0.0.0    Tabs: 230.0.0.0
View History:
View{version=1, members=[second]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec

Member Id: third    230.0.0.0    Tabs: 230.0.0.5 | 230.0.0.0
View History:
View{version=1, members=[third]}
View{version=3, members=[third, first, second]}
```

```
8:58:30.923 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Delivered...
8:58:30.923 [Thread-9] INFO  groupmembershipservice.MulticastPeer - Message{sender='second', sequentialNumber=4, view=View{version=3, members=[third, first, second]}}
8:58:30.923 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Simulating lost consensus message...
8:58:30.923 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Listening for View...
8:58:30.954 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Delivered...
8:58:30.954 [Thread-9] INFO  groupmembershipservice.MulticastPeer - Message{sender='third', sequentialNumber=2, view=View{version=3, members=[third, first, second]}}
8:58:30.954 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Simulating lost consensus message...
8:58:30.954 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Listening for View...
8:58:32.924 [Timer-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Sending view... View{version=3, members=[third, first, second]}
8:58:32.927 [Thread-9] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Delivered...
8:58:32.927 [Thread-9] INFO  groupmembershipservice.MulticastPeer - Message{sender='second', sequentialNumber=5, view=View{version=3, members=[third, first, second]}}
```

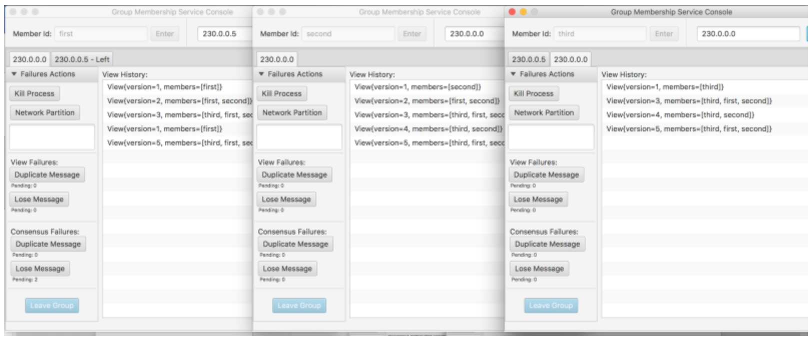
Node Failure

Plan: A node is failed by injecting the error through the UI.
Results: The remaining nodes detect the failure, and they update the view with the removal of the failed node. Then, the node recovers from failure and it joins again to the group. The

other members respond to the join normally ending with a new view with the 3 nodes on it.



```
Member Id: first          230.0.0.5
230.0.0.0 - Restoring | 230.0.0.5 - Left
View History:
View{version=1, members=[first]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec

Member Id: second         230.0.0.0
230.0.0.0
View History:
View{version=1, members=[second]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec
View{version=4, members=[third, second]}

Member Id: third          230.0.0.0
230.0.0.5 | 230.0.0.0
View History:
View{version=1, members=[third]}
View{version=3, members=[third, first, second]}
View{version=4, members=[third, second]}
```

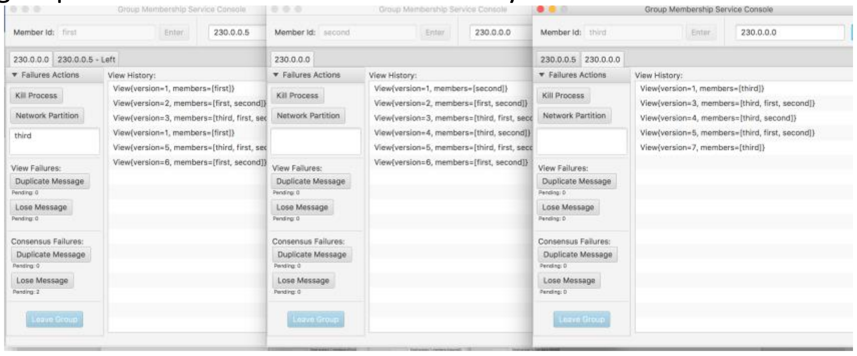


```
Member Id: first          230.0.0.5
230.0.0.0 | 230.0.0.5 - Left
View History:
View{version=1, members=[first]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec
View{version=1, members=[first]}
View{version=5, members=[third, first, sec

Member Id: second         230.0.0.0
230.0.0.0
View History:
View{version=1, members=[second]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec
View{version=4, members=[third, second]}
View{version=5, members=[third, first, sec

Member Id: third          230.0.0.0
230.0.0.5 | 230.0.0.0
View History:
View{version=1, members=[third]}
View{version=3, members=[third, first, second]}
View{version=4, members=[third, second]}
View{version=5, members=[third, first, second]}
```

Network Partition Failure

Plan: A network partition is injecting through the UI.

Results: The remaining nodes detect the failure that the blocked nodes is not sending more heartbeats, so they update the view with the removal of the failed node. The blocked node also updates its view by removing the nodes that cannot reach. Notice that two nodes were removal from this view, that is the reason the view version went from 5 to 7. While, in the another subgroup the view version is 6 because only one node was removed.

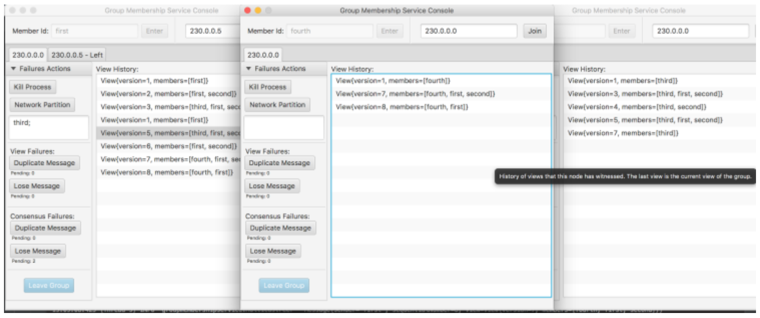


```
Member Id: first          230.0.0.5
230.0.0.0 | 230.0.0.5 - Left
third
View History:
View{version=1, members=[first]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec
View{version=1, members=[first]}
View{version=5, members=[third, first, sec
View{version=6, members=[first, second]}

Member Id: second         230.0.0.0
230.0.0.0
View History:
View{version=1, members=[second]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec
View{version=4, members=[third, second]}
View{version=5, members=[third, first, sec
View{version=6, members=[first, second]}

Member Id: third          230.0.0.0
230.0.0.5 | 230.0.0.0
View History:
View{version=1, members=[third]}
View{version=3, members=[third, first, second]}
View{version=4, members=[third, second]}
View{version=5, members=[third, first, second]}
View{version=7, members=[third]}
```

New Join Node and Node Failure

Plan: A new node joins to the group and an existing node is terminated without recovery.

Results: Only one view is updated because of the addition and removal The another subgroup is isolated by the network partition. This blocked subgroup does not receive any notification.

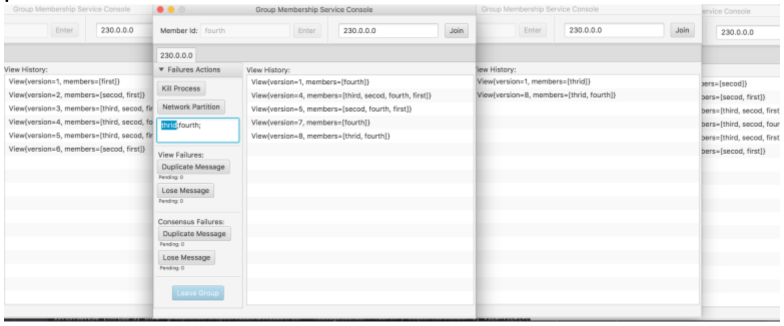


```
Group Membership Service Console
Member Id: first   Enter   230.0.0.5
230.0.0.0   230.0.0.5 - Left
View History:
View{version=1, members=[first]}
View{version=2, members=[first, second]}
View{version=3, members=[third, first, sec
View{version=1, members=[first]}
View{version=5, members=[third, first, sec
View{version=6, members=[first, second]}
View{version=7, members=[fourth, first, se
View{version=8, members=[fourth, first]}
Network Partition: third;
Lose Message  Pending: 2

Group Membership Service Console
Member Id: fourth   Enter   230.0.0.0   Join
230.0.0.0
View History:
View{version=1, members=[fourth]}
View{version=7, members=[fourth, first, second]}
View{version=8, members=[fourth, first]}

Group Membership Service Console
230.0.0.0
View History:
View{version=1, members=[third]}
View{version=3, members=[third, first, second]}
View{version=4, members=[third, second]}
View{version=5, members=[third, first, second]}
View{version=7, members=[third]}
History of views that this node has witnessed. The last view is the current view of the group.
```

New Join Node to the blocked subgroup

Plan: A new which is blocked joins to the group.

Results: The node is joining is only reflected in the specific subgroup that is blocked because of the network partition.

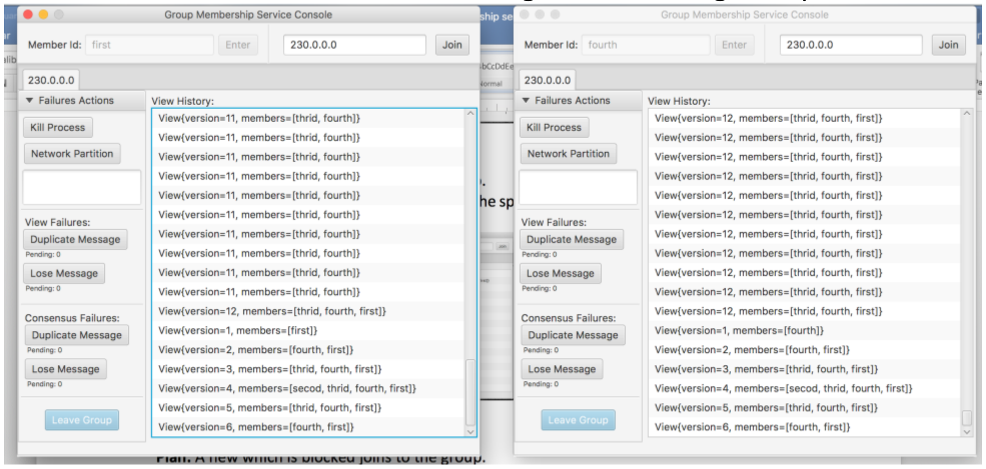


```
View History:
View{version=1, members=[first]}
View{version=2, members=[secod, first]}
View{version=3, members=[third, secod, fi
View{version=4, members=[third, secod, fo
View{version=5, members=[third, secod, fi
View{version=6, members=[secod, first]}

Group Membership Service Console
Member Id: fourth   Enter   230.0.0.0   Join
230.0.0.0
Network Partition: thrid;fourth;
View History:
View{version=1, members=[fourth]}
View{version=4, members=[third, secod, fourth, first]}
View{version=5, members=[secod, fourth, first]}
View{version=7, members=[fourth]}
View{version=8, members=[thrid, fourth]}

View History:
View{version=1, members=[thrid]}
View{version=8, members=[thrid, fourth]}

bers=[secod]}
bers=[secod, first]}
bers=[third, secod, first
bers=[third, secod, four
bers=[third, secod, first
bers=[secod, first]}
```

System recovered from network partition

Plan: The partition is removed, the affected nodes are restarted by the UI. Then, two nodes are terminated to check the detection is still working.

Results: The view makes several changes and in the end, it synchronizes the view before and after the nodes termination. The monitoring is still working as expected.

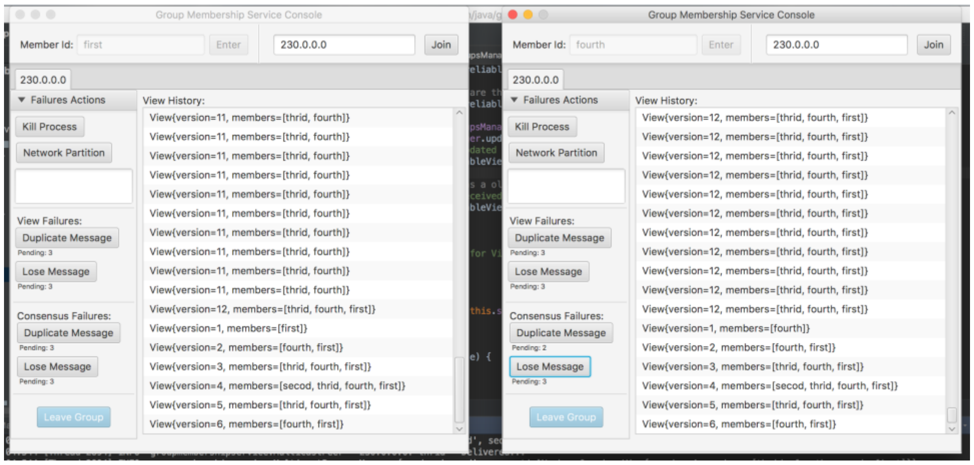


```
Group Membership Service Console
Member Id: first   Enter   230.0.0.0   Join
230.0.0.0
View History:
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=12, members=[thrid, fourth, first]}
View{version=1, members=[first]}
View{version=2, members=[fourth, first]}
View{version=3, members=[thrid, fourth, first]}
View{version=4, members=[secod, thrid, fourth, first]}
View{version=5, members=[thrid, fourth, first]}
View{version=6, members=[fourth, first]}

Group Membership Service Console
Member Id: fourth   Enter   230.0.0.0   Join
230.0.0.0
View History:
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=1, members=[fourth]}
View{version=2, members=[fourth, first]}
View{version=3, members=[thrid, fourth, first]}
View{version=4, members=[secod, thrid, fourth, first]}
View{version=5, members=[thrid, fourth, first]}
View{version=6, members=[fourth, first]}
```

Several Messages are lost and duplicated

Plan: for each process 12 view and consensus messages were set to be lost and duplicated, this is 3 for each case.

Results: The system simulates every duplication and losing of messages. When, the errors reached to 0, the normal protocol completes successfully. The view is synchronize across all the nodes after the several failure scenarios.



230.0.0.0

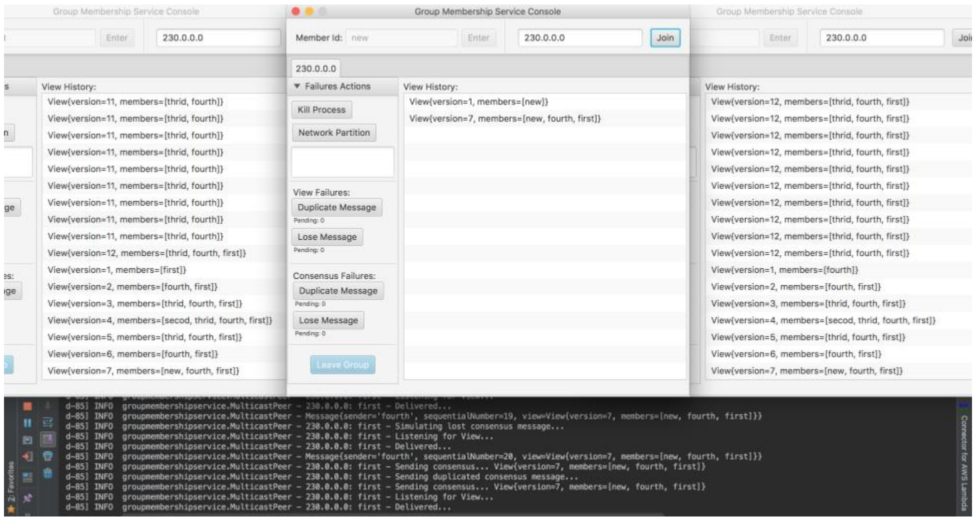
View History:

```
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=12, members=[thrid, fourth, first]}
View{version=1, members=[first]}
View{version=2, members=[fourth, first]}
View{version=3, members=[thrid, fourth, first]}
View{version=4, members=[secod, thrid, fourth, first]}
View{version=5, members=[thrid, fourth, first]}
View{version=6, members=[fourth, first]}
```

230.0.0.0

View History:

```
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=1, members=[fourth]}
View{version=2, members=[fourth, first]}
View{version=3, members=[thrid, fourth, first]}
View{version=4, members=[secod, thrid, fourth, first]}
View{version=5, members=[thrid, fourth, first]}
View{version=6, members=[fourth, first]}
```



230.0.0.0

View History:

```
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=11, members=[thrid, fourth]}
View{version=12, members=[thrid, fourth, first]}
View{version=1, members=[first]}
View{version=2, members=[fourth, first]}
View{version=3, members=[thrid, fourth, first]}
View{version=4, members=[secod, thrid, fourth, first]}
View{version=5, members=[thrid, fourth, first]}
View{version=6, members=[fourth, first]}
View{version=7, members=[new, fourth, first]}
```

View History:

```
View{version=1, members=[new]}
View{version=7, members=[new, fourth, first]}
```

View History:

```
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=12, members=[thrid, fourth, first]}
View{version=1, members=[fourth]}
View{version=2, members=[fourth, first]}
View{version=3, members=[thrid, fourth, first]}
View{version=4, members=[secod, thrid, fourth, first]}
View{version=5, members=[thrid, fourth, first]}
View{version=6, members=[fourth, first]}
View{version=7, members=[new, fourth, first]}
```

```
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Delivered...
d-85] INFO  groupmembershipservice.MulticastPeer - Message{sender='fourth', sequentialNumber=19, view=View{version=7, members=[new, fourth, first]}}
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Simulating lost consensus message...
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Listening for View...
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Delivered...
d-85] INFO  groupmembershipservice.MulticastPeer - Message{sender='fourth', sequentialNumber=20, view=View{version=7, members=[new, fourth, first]}}
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Sending consensus... View{version=7, members=[new, fourth, first]}
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Sending duplicated consensus message...
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Sending consensus... View{version=7, members=[new, fourth, first]}
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Listening for View...
d-85] INFO  groupmembershipservice.MulticastPeer - 230.0.0.0: first - Delivered...
```

Individual Contribution

I implemented the whole system alone.

Summary

A list of achievements is presented as follows:

- Forming, joining and leaving groups.
- Keeping a consistent view of the group in all members.
- Returning the current view when required.
- Monitoring of group members.
- Detection of failures and report of them to remaining members.
- Failure handles for scenarios such as duplicate messages, lost messages, node failures, network partitions and the different inconsistent scenarios as consequence of these failures.
- Framework to introduce failures to the group.
- User Interface for Group Management Service which allows to introduce and to monitor the failures and check the view of the member (Desktop Application).

References

1. Docs.oracle.com. (2018). MulticastSocket (Java Platform SE 7). [online] Available at: https://docs.oracle.com/javase/7/docs/api/java/net/MulticastSocket.html [Accessed 9 Apr. 2018].
2. Docs.oracle.com. (2018). What Is JavaFX? | JavaFX 2 Tutorials and Documentation. [online] Available at: https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm [Accessed 9 Apr. 2018].