# KTH H16P01 Distributed Systems Basic Course: Groupy

Thorsteinn Thorri Sigurdsson (ttsi@kth.se)

October 6, 2016

## 1 Introduction

In this assignmnent a group memebership service was created. The group membership service provides atomic multicast (reliable total order multicast). Each node in the service has two processes, an application layer process and a group membership process. The group membership process is either a slave or a leader. If an application layer process wants to multicast a message to its group, it sends a message to its group process. If the group process is a slave, it will forward the message to the leader group process (that lives on another node), which will in turn multicast the message to all slave processes. If the group process is the leader, it will multicast the message to all slave processes directly. If the leader dies a new leader is elected. New nodes can also join the group by contacting any node already in the group.

All group communication is done through the group process and the application layer process does not know if its group process is a leader or a slave.

## 2 gms1: No failure handling

The first implementation is simple, without any failure handling. A leader may fail, and then the system grinds to a halt since a leader election is not performed. The system is unable to perform multicasts and the system will never change state (the remaining slaves stop changing colors).

## 3 gms2: Handling crashes

We refine the solution by making slaves monitor the leader, using erlang's `monitor` feature. This way the slaves will receive a `DOWN` message when the leader dies (we are assuming perfect monitors and that the `DOWN` message will be the last message we ever receive from the dead node). Once a slave

receives the `DOWN` message from the leader, it enters the election state. It will select the first node in the slave list as the leader. If it sees that itself should be the leader, it multicasts an updated view and becomes the leader. Otherwise it just starts monitoring the new leader. We are now able to handle a crashing leader and elect a new one.

Currently, when we receive the updated view message from the new leader we accept him as leader, but do not start monitoring him until we receive the `DOWN` message from the previous leader. Better would be to monitor the new leader immediately as we get the new view, and ignore all trailing `DOWN` messages from the old leader.

But what if the leader crashes while delivering a view (i.e. it sends some message to some nodes, not all of them, before crashing)? We introduce a random crash when the leader is broadcasting, and can then see that the nodes may get out of sync, even though a new leader was successfully elected.

Only some nodes receive the message that the leader is sending. If this is a message about a color change, only some nodes will change their colors. Note that the color change is done in such a way that the R value in RGB is incremented (and then wrapped around). So if only some nodes receive the color change message, only some of them will increment their color. Then, even if following color change messages are received by all nodes, they will still be out of sync, since the color change is incremental.

## 4 gms3: Reliable multicast

To fix this problem of half-finished multicasts we have the slaves keep a copy of the last message that they have seen from the leader. Since the leader is sending out the multicast to the slaves in the order they appear in the slave list, and we are assuming reliable message delivery, then we can be sure that if any node received the message before the leader crashed, then the first node in the slave list received it. Since the first node in the slave list will be the next leader, it is enough to have it re-broadcast the last message it saw to all the remaining slaves.

To avoid slaves receiving duplicate messages, we introduce sequence numbers to the messages that are sent from leader to slaves (`msg` and `view`). The slaves can then discard messages that they have seen before.

Development up until this point went smoothly. However, in the slave process, I was tracking the last received sequence number while the assignment doc is tracking the expected next sequence number. Therefore the given code snippet that discards messages if the `new seqnum < N` didn't work for me, it had to be `=<` instead. This took unnecessarily long to figure out.

# 5 gms4: Unreliable delivery of messages

So far we have been relying on reliable delivery of messages, i.e. if a message is sent it will be delivered. This is not necessarily the case. The implementation was changed to try to handle losing messages in transit.

We track lost messages by introducing acknowledgements (ACKs), where the slaves send an ACK for a message back to the leader to let it know that they have received the message.

The leader maintains a queue og sent-but-not-acked messages, which was called the `OutgoingQueue`. The queue is of the form:

```
[{{RecipientPid, SeqNum}, Message}]
```

When the leader is broadcasting, it adds the sent messages to the queue. When the slaves receive a message, they send an ACK back to the leader of the form:

```
{ack, {Pid, SeqNumToAck}}
```

When the leader receives the ACK, he deletes the corresponding entry from the queue. Before sending out a new broadcast, the leader re-sends the messages that are still in the queue. This was tested by having the slaves discard messages received from the leader randomly.

## 5.1 Issues

However, this is not a complete implementation.

First of all, the leader should monitor the slaves for crashes and remove all entries that the slave owns from the queue when it crashes. Otherwise we might try to resend to it for all eternity since we will never receive an ACK from it.

The sending of a message and receiving of an ACK is asynchronous, we do not wait for an ACK for a message before moving on and sending the next message. Therefore, if we send messages A and B back to back to a slave, message A might get lost, but message B is received, before we have the chance to re-send message A and (hopefully) deliver it to the slave. The slave will then apply the command from message B before A, and go out of sync. In the current implementation, the slave will happily accept message B since it is only checking if the sequence numbers are increasing and doesn't care if there are gaps in them.

To fix this it might be possible to hold off on sending the next message until the `OutgoingQueue` is empty. This way we would be sure that all nodes have received all messages up to a point and the whole system has progressed to the state prescribed in the previous message.

Another and perhaps better way would be to make the slave keep a queue of incoming messages, and only deliver a non-broken succession of incoming

messages (sequence number wise) to the application layer process. So if the slave has received messages `[8, 9, 10, 12, 13]` it could deliver messages `[8, 9]` to the application layer, but wait until it receives message `11` before delivering the rest, [10, 11, 12, 13]. This way we would make sure that the system goes through the same progression of state changes, but some nodes would go through them slower than others.

In the current implementation we might also re-send a message to a slave while the ACK from the slave is in transit. This is OK, since the slaves are able to discard duplicate messages. The slave will re-send its ACK upon receiving the message again.

Another issue is what happens to the `OutgoingQueue` when a leader crashes. The new leader that takes over can't know the crashed leader's `OutgoingQueue` and will start over with an empty queue, and all the messages in the crashed leader's queue may never be delivered.

Introducing the ACK mechanism will obviously affect performance. The message complexity of the system increases, best case scenario the number of messages sent between the leader and slaves is doubled, since there will be an ACK for every sent message. As previously discussed even more messages might also be sent among them, for instance if an ACK is lost in transit (causing a re-send), or if we re-send a message before receiving the ACK for it. The proposed ways to make sure that messages are delivered in order to the slave's application layer process will also lower performance. Either we wait until the `OutgoingQueue` is empty before broadcasting the next state, thereby holding up the entire system, or we make each slave node wait for a non-broken line of sequence numbers before delivering them to the application layer process, thereby holding up individual nodes.

# 6 Other considerations

## 6.1 Imperfect failure detectors

Erlang's failure detectors (`monitors`) are not perfect. What if a node is reported as crashed, when it isn't?

In this case we would end up with two leaders, the correct one and a new one that wrongly believes it should be the leader. A possible fix for this would be to have an ID on all the nodes in the system, and when a new node joins the system it should receive an ID that is `max(IDs already in system) + 1`. By tracking these IDs the other slaves in the system (that did not receive the erroneous `DOWN` message from the leader) could verify that it is in fact their leader that is sending them messages, and discard messages from the new, wrong, leader. The leader process could then also be made to listen for incoming messages from leaders (`msg, view`), and if the new, wrong, leader receives a message from his old leader (which he thinks is dead) he would demote himself back to slave.

### 6.2 Incorrect node delivers a message

What if one incorrect node delivers a message that will not be delivered by any correct node (this could happen even if we had reliable send operations and perfect failure detectors). How could this happen and how likely is it that it does?