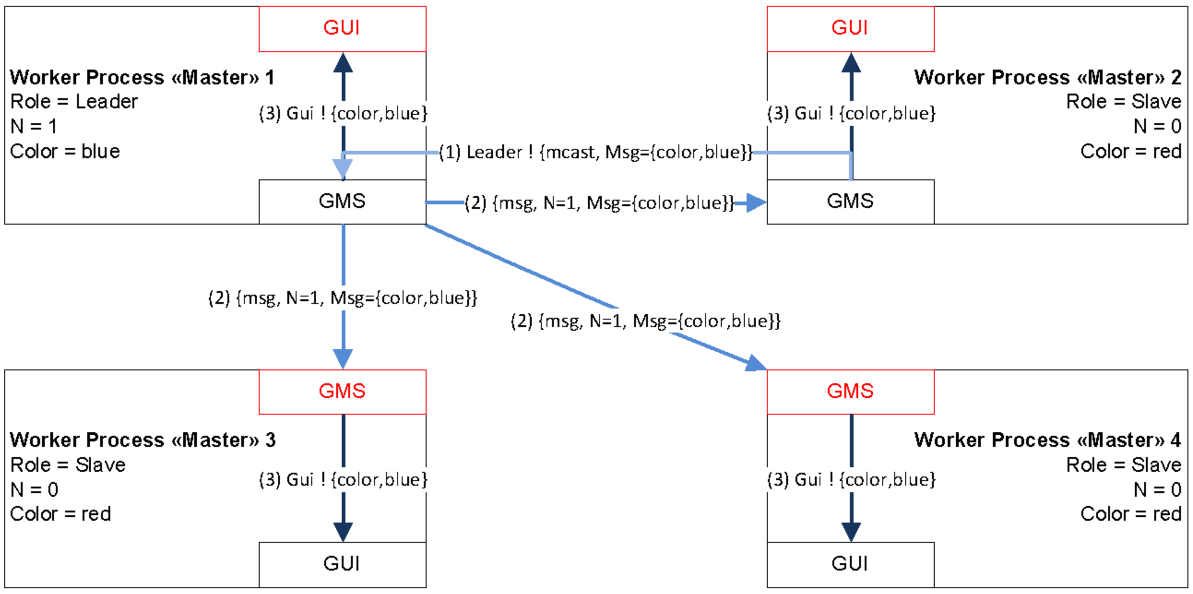
Report 4: Groupy

Thomas Galliker

September 28, 2011

1 Introduction

In this exercise we have implemented a group membership service that provides atomic multicast. The target is to build distributed applications that share a coordinated state. Nodes may come and go on-the-fly. Some might also crash unexpectedly. State changes on nodes are forwarded to all nodes of the same multicast group. This forwarding process is done by a so called "leader", all the other nodes are called "slaves" as illustrated in Figure 1.



GUI
Worker Process «Master» 1
Role = Leader
N = 1
Color = blue
(3) Gui ! {color,blue}
GMS
(1) Leader ! {mcast, Msg={color,blue}}
(2) {msg, N=1, Msg={color,blue}}
GUI
Worker Process «Master» 2
Role = Slave
N = 0
Color = red
(3) Gui ! {color,blue}
GMS
(2) {msg, N=1, Msg={color,blue}}
(2) {msg, N=1, Msg={color,blue}}
GMS
Worker Process «Master» 3
Role = Slave
N = 0
Color = red
(3) Gui ! {color,blue}
GUI
GMS
Worker Process «Master» 4
Role = Slave
N = 0
Color = red
(3) Gui ! {color,blue}
GUI

Figure 1: Sample configuration with 4 nodes. Slave #2 is sending out a new color (=blue) to its leader on node #1. The leader in turn multicasts this message to all known slaves.

2 Main problems and solutions

In a first version we simply had to implement the membership service based upon the given code. This was not a huge task, but it took some time to read through the script and to understand the given code. The first version didn't concern about failures. A group leader was defined first and group members (=slaves) were added afterwards.
In the second part of this exercise we were asked to extend our application with failsafe procedures. To detect connection failures between slaves and their leader, we simply added Erlang monitors:

```
erlang:monitor(process, Leader)
```

As soon as slave processes detect that their leader crashed, they start electing a new one:

```
{'DOWN', _Ref, process, Leader, _Reason} ->
        election(Id, Master, N, Last, Peers);
```

It is very important that all nodes apply the same rules to elect the leader. Otherwise we will end up in having several independent groups after a while. In this application we simply take the first node of the group membership list to become the new leader.

To get proof of our safety actions, we have implemented a crash simulator that automatically tears down the leader node while sending the multicast messages to its slave nodes. The reelection of the leader was successful, but we faced another problem: No atomicity was guaranteed during the multicast transmission, means, we don't know how much messages have actually been delivered to our slaves (Figure 2).

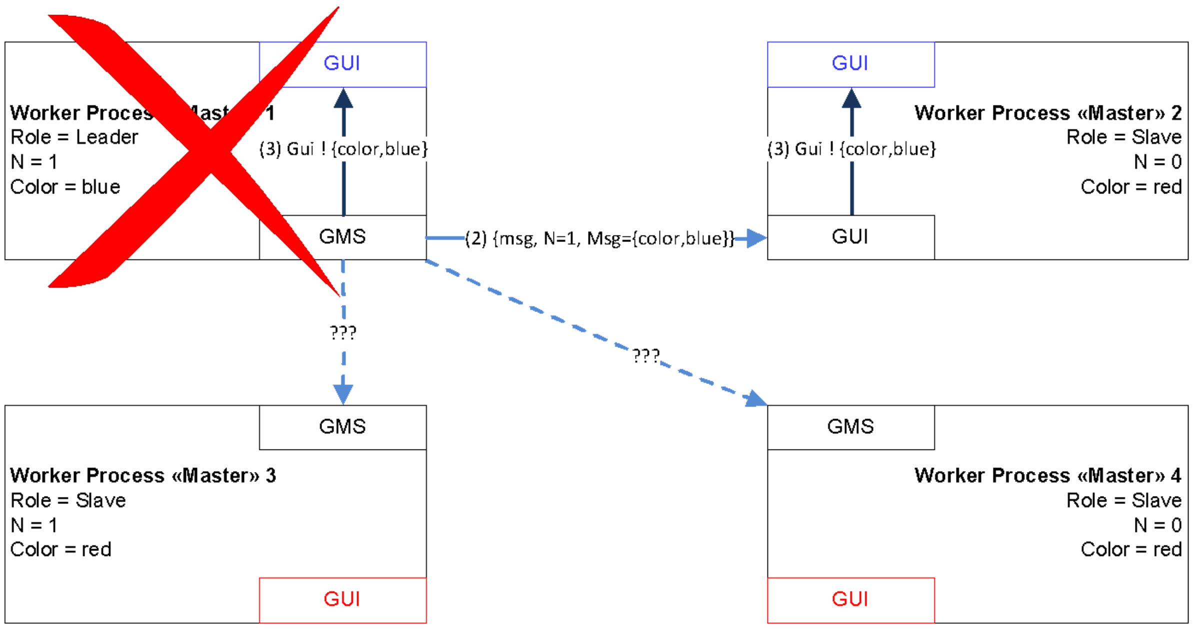


Figure 2: Atomic multicast guarantees that subscriber receives seamlessly every message,

Thus, the new leader must somehow be able to resend the last message. This is basically not a complicated task. But what happens if some of the slaves already got the messages? They would receive duplicated copies of the same message if we just resent the last message again. For this reason we have introduced a sequence number N that enables nodes to discard duplicated/old messages.

```
{msg, I, _} when I < N ->
            io:format("~p: Old sequence number detected (~p<~p)~n", [Id, I, N]),
```

New messages are indirectly forwarded to the GUI process. We must not forget to increase N as soon as a newly arrive message has been processed. (We need N+1 to compare with further incoming messages). The last message is always being kept in the slave loop since (as already mentioned) the slave at the top most position of the peer list is predestined to become the new leader and in this case it has to resend this last message to all its peers again.

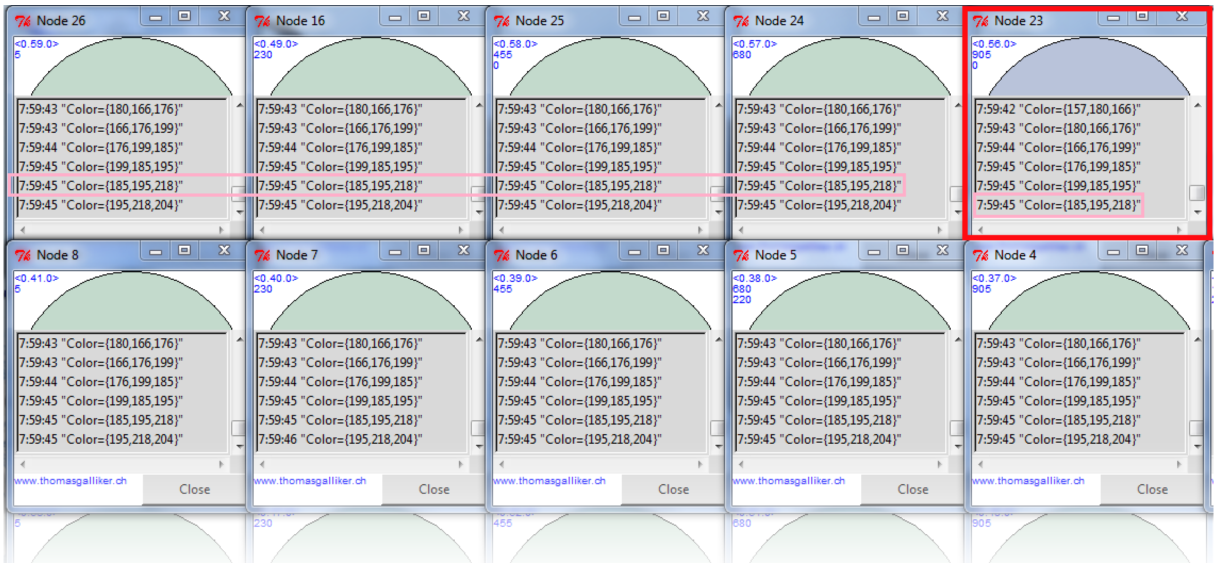
```
{msg, N, Msg} ->
            Master ! {deliver, Msg},
            slave(Id, Master, Leader, N+1, {msg, N, Msg}, Peers);
```

Finally, it is worthwhile to mention that the leader node is responsible for providing increased sequence numbers on new messages.

```
leader(Id, Master, N+1, Peers);
```

3 Evaluation

To check the correctness of our atomic multicast algorithm I have built an extended GUI and added several console outputs. The results of the tests were mostly positive: Crashed leaders were immediately replaced by a successor. Only one test, a load test with 30 participating nodes has exposed strange display problems. Some of the nodes began to react very slow as it can be seen on Figure 3.



```
Node 26
<0.59.0>
5
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:45 "Color={195,218,204}"
```

```
Node 16
<0.49.0>
230
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:45 "Color={195,218,204}"
```

```
Node 25
<0.58.0>
455
0
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:45 "Color={195,218,204}"
```

```
Node 24
<0.57.0>
680
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:45 "Color={195,218,204}"
```

```
Node 23
<0.56.0>
905
0
7:59:42 "Color={157,180,166}"
7:59:43 "Color={180,166,176}"
7:59:44 "Color={166,176,199}"
7:59:45 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
```

```
Node 8
<0.41.0>
5
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:45 "Color={195,218,204}"
```

```
Node 7
<0.40.0>
230
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:46 "Color={195,218,204}"
```

```
Node 6
<0.39.0>
455
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:45 "Color={195,218,204}"
```

```
Node 5
<0.38.0>
680
220
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:45 "Color={195,218,204}"
```

```
Node 4
<0.37.0>
905
7:59:43 "Color={180,166,176}"
7:59:43 "Color={166,176,199}"
7:59:44 "Color={176,199,185}"
7:59:45 "Color={199,185,195}"
7:59:45 "Color={185,195,218}"
7:59:45 "Color={195,218,204}"
```

Figure 3: Display delays on some consoles after a test with 30 concurrent nodes.

The graphic system of Erlang could be the cause for delays captured in Figure 3. Another reason for this behavior could be the increasing delay time for retransmissions in case of a leader change. (Remember that the new leader always sends out messages to all nodes regardless of whether they already got the message. This can cause huge delays in large multicast groups. Real IP multicast could solve such issues).

4 Open Issues

There are still some cases where things can go wrong in our application. It could for instance happen that we run into a split-group situation if the leader shortly loses its network connection.

Furthermore we are not doing real IP multicasting in this exercise. We could extend this example by UDP sockets and send out messages via IP multicast, but then it's no longer (or at least not easily!) possible to simulate a crash during a multicast transmission.

Theoretically we could adapt the system to keep track of lost/unsent messages but to realise such a solution, all involved nodes should have a common storage (comparable with the quorum disk of a database cluster). This storage could then be used to run something like a journal in which the leader logs all performed actions. When the leader dies, this journal is consulated by its successor. Further measures are taken by the new leader to restore a consistent state.

5 Conclusions

This was an exiting exercise. Not only that we solve a problem that is of practical relevance but also that we got in contact with graphical Erlang applications. The most important point in such coordination tasks is that all participants follow the same rules. It took some time to understand how message tracking and especially the recovery from a failure state could be implemented. Most important to solve such exercises: Notepad and pen!

Since my group membership application was finished very soon, I tried to implement an IP multicast module. This was tricky since there were no usable documentations nor working code constructs on the Internet.

6 Links

http://www.erlang.org/doc/apps/gs/gs.pdf
http://www.erlang.org/doc/apps/gs/gs_chapter8.html#id67499
http://www.erlang.org/doc/apps/gs/gs_chapter3.html
http://stackoverflow.com/questions/588003/convert-an-integer-to-a-string-in-erlang
http://www.erlang.org/documentation/doc-5.2/doc/extensions/misc.html
http://www.trapexit.org/Dates_and_Time
http://www.thomasgalliker.ch