Groupy: a group membership service

Jordi Guitart

Adapted with permission from Johan Montelius (KTH)

February 6, 2013

Introduction

This is an assignment where you will implement a group membership service that provides atomic multicast. The aim is to have several application layer processes with **a coordinated state** i.e. they should all perform the same sequence of state changes. A node that wishes to perform a state change must first multicast the change to the group so that all nodes can execute it. Since the multicast layer provides total order, all nodes will be synchronized.

The problem in this assignment is that all nodes need to be synchronized even though nodes may come and go (crash). As you will see it is not as trivial as one might first think.

1 The architecture

We will implement a group membership service that provides atomic multicast in view synchrony. The architecture of this service consists of a set of nodes where one is the elected leader. All nodes that wish to multicast a message will send the message to the leader and the leader will do a basic multicast to all members of the group. If the leader dies a new leader is elected.

A new node that wishes to enter the group will contact any node in the group and request to join the group. The leader will determine when the node is to be included and will deliver **a new group view** to the group.

The application layer processes will use this group membership service to synchronize their states. As commented, this service is implemented by means of a set of group processes. Each application layer process will have its own group process that it communicates with. The application layer process will send the messages to be multicasted to the group process and will receive all multicasted messages from it. The group process will tell the application level process to deliver a message only when the rest of processes have also delivered it. The application layer process must also be prepared to decide if a new node should be allowed to enter the group and also decide the initial state of this node.

Note that we will not deliver any group views to the application layer process. We could adapt the system so that it reports any view changes but for the application that we are targeting this is not needed. We will keep it as simple as possible and then discuss extensions and how much they would cost.

1.1 View synchrony

Each node in the group should be able to multicast messages to the members of the group. The communication is divided into views and messages will be said to be delivered in a view. For all messages in a view we will guarantee the following:

- in FIFO order: in the order that they were sent by a node
- in total order: all nodes see the same sequence
- reliably: if a correct node delivers a message, all correct nodes deliver the message

The last statement seems to be a bit weak, what do we mean by a correct node? A node will fail only by crashing and will then never be heard from again. A correct node is a node that does not fail during a view (i.e. it survives to install the next view).

It will not be guaranteed that sent messages are delivered, we will use asynchronous sending without acknowledgment and if we have a failing leader a sent message might disappear.

1.2 The leader

A node will either play the role of a leader (let's hope there is only one) or a slave. The slaves will forward messages to the leader and the leader will tag each message with a sequence number and multicast it to all nodes. The leader can also accept a message directly from its own master (i.e. the application layer process). The application layer process is unaware of whether its group process is acting as a leader or a slave.

1.3 A slave

A slave will receive messages from its application layer process and forward them to the leader. It will also receive messages from the leader and forward them to the application layer process. If nodes would not fail this would be the easiest job in the world but since we must be able to act if the leader dies we need to do some book keeping.

In our first version of the implementation we will not deal with failures but only with adding new nodes to the system. This is complicated enough to start with.

1.4 The election

The election procedure is very simple. All slaves have the same list of peers and they all elect the first node in the list as the leader. A slave that detects that it is the first node will of course adopt the role as leader.

1.5 The application layer process

An application layer process will create a group process and contact any other application layer process it knows of. It will request to join the group providing the process identifier of its group process. It will then wait for a view to be delivered from its group process containing the state that it should adopt.

There is no guarantee that the request is delivered to the leader, or rather the leader could be dead and we have not detected this yet. The requesting application layer process is however not told about this so we cannot do anything but wait and hope for the best. We will use a timeout and if we have not been invited in a second we might as well abort the attempt.

2 The first implementation

Our first version, called `gms1`, will only handle starting of a single node and adding more nodes. Failures will not be handled so some of the states that we need to keep track of are not described. We will then extend this implementation to handle failures later on.

The group process will when started be a slave but might in the future become a leader. The first process that is started will however become a leader directly.

2.1 The leader

The leader process can be in two states, the regular state where it is multicasting messages or a state where it has asked the application layer process if a new node can enter the group. The leader keeps the following state:

- **Id**: a unique name of the node, only used for debugging
- **Master**: the process identifier of the application layer process
- **Peers**: an ordered list of the process identifiers of all slaves in the group

The list of peers is ordered based on when they were admitted to the group. We will use this order in the election procedure.

In its regular state the leader should be able to handle the following messages:

- {`mcast, Msg`}: a message either from its own master or from a peer node. A message {`msg, Msg`} is multicasted to all peers and a message {`deliver, Msg`} is sent to the application layer process (i.e. `Master`). We use a function `bcast/3` that will send a message to each of the processes in a list.
- {`join, Peer`}: a message, from a peer or the master, that is a request from a node to join the group. A `request` message is sent to the application layer process and the leader moves into the joining state.

```
leader(Id, Master, Peers) ->
    receive
        {mcast, Msg} ->
            bcast(Id, ..., ...),  %% TODO: COMPLETE
            %% TODO: ADD SOME CODE
            leader(Id, Master, Peers);
        {join, Peer} ->
            %% TODO: ADD SOME CODE
            joining(Id, ..., ..., ...);  %% TODO: COMPLETE
        stop ->
            ok;
        Error ->
            io:format("leader ~w: strange message ~w~n", [Id, Error])
    end.

bcast(_, Msg, Nodes) ->
    lists:foreach(fun(Node) -> Node ! Msg end, Nodes).
```

In the joining state the leader can only accept one message:

- {`ok, State`}: a message from the master deciding that the peer node should join the group and that it should start with a specified state. The leader returns to its regular state.

We ignore the possibility that a node is refused to enter the group but this is of course a simple extension.

The joining of a new node **is not an asynchronous process**. The leader has to stop delivering messages until it has got the reply from the application layer process. If this was not the case we could have a situation where the application layer process accepts the new peer but the state that it is given is already old.

```
joining(Id, Master, Peer, Peers) ->
    receive
        {ok, State} ->
```

```
            Peers2 = lists:append(Peers, [Peer]),
            bcast(Id, {view, State, self(), Peers2}, Peers2),
            leader(Id, Master, Peers2);
        stop ->
            ok
    end.
```

Notice that we add the new node at the end of the list of peers. This is important, we want the new node to be the last one to see the view message that we send out. More on this later when we look at failing nodes.

2.2 A slave

A slave has an even simpler job, it will not make any complicated decisions. It is simply forwarding messages from its master to the leader and vice versa. The state of a slave is as follows:

- **Id**: a unique name of the node, only used for debugging
- **Master**: the process identifier of the application layer process
- **Leader**: the process identifier of the leader
- **Peers**: an ordered list of the process identifiers of all slaves in the group

The messages from the master and the leader are the following:

- `{mcast, Msg}`: a request from its master to multicast a message, the message is forwarded to the leader.
- `{join, Peer}`: a request from its master to allow a new node to join the group, the message is forwarded to the leader.
- `{msg, Msg}`: a multicasted message from the leader. A message `{deliver, Msg}` is sent to the master.
- `{view, State, Leader, View}`: a multicasted view from the leader. For the slave only the new set of peers (`View`) is of interest.

```
slave(Id, Master, Leader, Peers) ->
    receive
        {mcast, Msg} ->
            %% TODO: ADD SOME CODE
            slave(Id, Master, Leader, Peers);
        {join, Peer} ->
            %% TODO: ADD SOME CODE
```

```
            slave(Id, Master, Leader, Peers);
        {msg, Msg} ->
            %% TODO: ADD SOME CODE
            slave(Id, Master, Leader, Peers);
        {view, _, _, View} ->
            slave(Id, Master, Leader, View);
        stop ->
            ok;
        Error ->
            io:format("slave ~w: strange message ~w~n", [Id, Error])
    end.
```

Since we will not yet deal with failure there is no transition between being a slave and becoming a leader. We will add this later but first let us have this thing up and running.

2.3 Initialization

Initializing a process that is the first node in a group is simple. The only thing we need to do is to give it an empty list of peers. Since it is the only node in the group it will of course be the leader of the group.

```
-module(gms1).
-export([start/1, start/2]).

start(Id) ->
    Self = self(),
    spawn_link(fun()-> init(Id, Self) end).

init(Id, Master) ->
    leader(Id, Master, []).
```

Starting a node that should join an existing group is only slightly more problematic. We need to send a {`join, self()`} message to a node in the group and wait for an invitation. The invitation is delivered as a view message containing everything we need to know. The initial state is of course as a slave.

```
start(Id, Grp) ->
    Self = self(),
    spawn_link(fun()-> init(Id, Grp, Self) end).

init(Id, Grp, Master) ->
    Self = self(),
    Grp ! {join, Self},
```

```
    receive
        {view, State, Leader, Peers} ->
            Master ! {ok, State},
            slave(Id, Master, Leader, Peers)
    end.
```

Note that the view that is delivered contains the state that should be passed on to the application layer process (Master).

2.4 The application layer process

To do some experiments we create an application layer process (so-called worker) that uses a gui to describe its state. A worker and the gui are given at the appendices.

Experiments. i) Do some experiments to see that you can create a group, add some peers and keep their state coordinated. You can use the following code to start and stop the whole system. Note that we are using the name of the module (i.e. gms1) as a parameter to the start procedure. ii) Split the groupy module and make the needed adaptations to enable each worker to run in different machines. Remember how names registered in remote nodes are referred and how Erlang runtime should be started to run distributed programs.

```
-module(groupy).
-export([start/2, stop/0]).

start(Module, Sleep) ->
Leader = worker:start("1", Module, 1, Sleep),
register(a, Leader),
register(b, worker:start("2", Module, 2, Leader, Sleep)),
register(c, worker:start("3", Module, 3, Leader, Sleep)),
register(d, worker:start("4", Module, 4, Leader, Sleep)),
register(e, worker:start("5", Module, 5, Leader, Sleep)).

stop() ->
a ! stop,
b ! stop,
c ! stop,
d ! stop,
e ! stop.
```

3 Handling failures

We will build up our fault tolerance gradually. First we will make sure that we detect crashes, then ensure that a new leader is elected an then make sure that the multicast service preserves the properties of the atomic multicast. Keep `gms1` as a reference and call the adapted module `gms2`.

3.1 Failure detectors

We will use the Erlang built-in support to detect and report that processes have crashed. A process can monitor another process and if that process dies a message will be received. For now, we will assume that the monitors are perfect i.e. they will eventually report the crash of a process and they will never report the death of a process that has not died.

We will also assume that the message that informs a process about a death of another process is the last message that it will see from it. The message will thus be received in FIFO order as any regular message.

The question we first need to answer is, who should monitor who? In our architecture we do not need to report new views when a slave dies and there is nothing to prevent a dead slave to be part of a view so we will keep things simple; the only node that will be monitored is the leader. A slave that detects that the leader has died will move to an election state.

This is implemented by first adding a call to `erlang:monitor/2` in the initialization of the slave:

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

and a new clause in the state of the slave:

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

In the election state the process will select the first process in its list of peers and elect this as the leader. It could of course be that the process finds itself being the first node and it will thus become the leader of the group.

```
election(Id, Master, [Leader|Rest]) ->
    if
        Leader == self() ->
            leader(Id, Master, Rest);
        true ->
            erlang:monitor(process, Leader),
            slave(Id, Master, Leader, Rest)
    end.
```

Since the leader can crash it could be that a node that wants to join the group will never receive a reply. The message could be forwarded to a dead leader and the joining node is never informed of the fact that its request was lost. We simply add a timeout when waiting for an invitation to join the group in the initialization of the slave.

```
    after 1000 ->
        Master ! {error, "no reply from leader"}
```

Experiments. Do some experiments to see if the peers can keep their state coordinated even if nodes crash.

3.2 Missing messages

It seems to be too easy and unfortunately it is not. To show that current implementation is not working we can change the `bcast/3` procedure and introduce a random crash. We define a constant `arghh` that defines the risk of crashing. A value of `100` means that a process will crash in average once in a hundred attempts. The definition of `bcast/3` now looks like this:

```
bcast(Id, Msg, Nodes) ->
    lists:foreach(fun(Node) ->
                      Node ! Msg,
                      crash(Id)
                  end,
                  Nodes).

crash(Id) ->
    case random:uniform(?arghh) of
        ?arghh ->
            io:format("leader ~w: crash~n", [Id]),
            exit(no_luck);
        _ ->
            ok
    end.
```

We also add seeding of the random number generator when starting a process so that we will not have all processes crashing at the same time. The leader initialization is for example done as follows (the slave will be initialized in a similar manner).

```
start(Id) ->
    Rnd = random:uniform(1000),
    Self = self(),
    spawn_link(fun()-> init(Id, Rnd, Self) end).

init(Id, Rnd, Master) ->
    random:seed(Rnd, Rnd, Rnd),
    leader(Id, Master, []).
```

Experiments. Repeat the experiments and see if you can have the state of the workers become out of synch.

Open Questions. Why is this happening?

3.3 Reliable multicast

To remedy the problem we could replace the basic multicaster with a reliable multicaster. A process that would forward all messages before delivering them to the higher layer.

Assume that we keep a copy of the last message that we have seen from the leader. If we detect the death of the leader it could be that it died during the basic multicast procedure and that some nodes have not seen the message. We will now make an assumption that we will discuss later:

- Messages are reliably delivered and thus,
- if the leader sends a message to A and then B, and B receives the message, then also A will receive the message.

The leader is sending messages to the peers in the order that they occur in the list of peers. If anyone receives a message then the first peer in the list receives the message. This means that only the next leader needs to resend the message.

This will of course introduce the possibilities of doublets of messages being received. In order to detect this we will number all messages and only deliver new messages to the application layer process.

Let's go through the changes that we need to make and create a new module `gms3` that implements these changes.

- `slave(Id, Master, Leader, N, Last, Peers)`: the slave procedure is extended with two arguments: `N` and `Last`. `N` is the expected sequence number of the next message and `Last` is a copy of the last message (a regular message or a view) received from the leader.
- `election(Id, Master, N, Last, Peers)`: the election procedure is extended with the same two arguments.
- `leader(Id, Master, N, Peers)`: the leader procedure is extended with the argument `N`, the sequence number of the next message (regular message or view) to be sent.

The messages are also changed and will now contain the sequence number.

- `{msg, N, Msg}`: a regular message with a sequence number.
- `{view, N, State, Leader, View}`: a view message with a sequence number.

We must also add clauses to the slave to accept and ignore duplicate messages. If we do not remove these from the message queue they will add up and after a year generate a very hard to handle trouble report.

When discarding messages we only want to discard messages that we have seen i.e. messages with a sequence number less than N. We can do this by using the `when` construction. For example:

```
{msg, I, _} when I < N -> ...
```

The crucial part is then in the election procedure where **the elected leader will forward the last received message to all peers** in the group. Hopefully this will be enough to keep slaves synchronized.

Experiments. i) Repeat the experiments to see if now the peers can keep their state coordinated even if nodes crash. ii) Try to keep a group rolling by adding more nodes as existing nodes die.

Assuming all tests went well we're ready to ship the product. There is however one thing we need to mention and that is that our implementation does not work!!! Well, it sort of work depending on what the Erlang environment guarantees and how strong our requirements are.

3.4 What could possibly go wrong

The first thing we have to realize is what guarantees the Erlang system actually gives on message sending. The specifications only guarantee that messages are delivered in FIFO order, not that they actually do arrive. We have built our system relying on reliable delivery of messages, something that is not guaranteed.

The second reason why things will not work is that we rely on that the Erlang failure detector is perfect.

Open Questions. i) How would we have to change the implementation to handle the possibly lost messages? ii) How would this impact performance? iii) Is really the case that we will never suspect any correct node for having crashed?

Appendix A: *worker.erl*

```
-module(worker).
-export([start/4, start/5]).

-define(change, 20).
-define(color, {0,0,0}).

start(Id, Module, Rnd, Sleep) ->
    spawn(fun() -> init(Id, Module, Rnd, Sleep) end).

init(Id, Module, Rnd, Sleep) ->
    Cast = apply(Module, start, [Id]),
    Color = ?color,
    init_cont(Id, Rnd, Cast, Color, Sleep).

start(Id, Module, Rnd, Peer, Sleep) ->
    spawn(fun() -> init(Id, Module, Rnd, Peer, Sleep) end).

init(Id, Module, Rnd, Peer, Sleep) ->
    Cast = apply(Module, start, [Id, Peer]),
    receive
        {ok, Color} ->
            init_cont(Id, Rnd, Cast, Color, Sleep);
        {error, Error} ->
            io:format("error: ~s~n", [Error])
    end.

init_cont(Id, Rnd, Cast, Color, Sleep) ->
    random:seed(Rnd, Rnd, Rnd),
    Gui = gui:start(Id, self()),
    Gui ! {color, Color},
    worker(Id, Cast, Color, Gui, Sleep),
    Cast ! stop,
    Gui ! stop.

worker(Id, Cast, Color, Gui, Sleep) ->
    Wait = if Sleep == 0 -> 0; true -> random:uniform(Sleep) end,
    receive
        {deliver, {_From, N}} ->
            Color2 = change_color(N, Color),
            Gui ! {color, Color2},
            worker(Id, Cast, Color2, Gui, Sleep);
        {join, Peer} ->
```

```
            Cast ! {join, Peer},
            worker(Id, Cast, Color, Gui, Sleep);
        request ->
            Cast ! {ok, Color},
            worker(Id, Cast, Color, Gui, Sleep);
        stop ->
            ok;
        Error ->
            io:format("strange message: ~w~n", [Error]),
            worker(Id, Cast, Color, Gui, Sleep)
    after Wait ->
            Cast !  {mcast, {Id, random:uniform(?change)}},
            worker(Id, Cast, Color, Gui, Sleep)
    end.

change_color(N, {R,G,B}) ->
    {G, B, ((R+N) rem 256)}.
```

Appendix B: *gui.erl*

```
-module(gui).
-export([start/2]).
-define(width, 200).
-define(height, 200).
-include_lib("wx/include/wx.hrl").

start(Id, Master) ->
    spawn_link(fun() -> init(Id, Master) end).

init(Id, Master) ->
    Frame = make_frame(Id),
    loop(Frame, Master).

make_frame(Id) ->       %Id is the window title
    Server = wx:new(),  %Server will be the parent for the Frame
    Frame = wxFrame:new(Server, -1, Id, [{size,{?width, ?height}}]),
    wxFrame:setBackgroundColour(Frame, ?wxBLACK),
    wxFrame:show(Frame),
    %monitor closing window event
    wxFrame:connect(Frame, close_window),
    Frame.

loop(Frame, Master)->
    receive
        %check if the window was closed by the user
        #wx{event=#wxClose{}} ->
            wxWindow:destroy(Frame),
            Master ! stop,
            ok;
        {color, Color} ->
            color(Frame, Color),
            loop(Frame, Master);
        stop ->
            ok;
        Error ->
            io:format("gui: strange message ~w ~n", [Error]),
            loop(Frame, Master)
    end.

color(Frame, Color) ->
    wxFrame:setBackgroundColour(Frame, Color),
    wxFrame:refresh(Frame).
```