ID2201 Distributed Systems, basic course

Groupy: A Group Membership Service

Johan Montelius, Vladimir Vlassov and Klas Segeljakt
Email: {johanmon,vladv,klasseg}@kth.se
July 7, 2023

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 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 group members. 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 deliver a new view to the group.

Each application layer process will have its group process that it communicates with. The application layer will send multicast messages to the group process and receive all multicasted messages from it. The application layer 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 views to the application layer. 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 a node sent them;
- 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. All 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 master, i.e., the application layer. The application layer is unaware whether its group process acts 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. If nodes did not fail, it 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 bookkeeping.

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 simple. All slaves have the same list of peers and 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 of leader. The leader must resend the last message received, and the slaves must monitor the new leader.

1.5 The application layer

An application process will create a group process and contact any other application process it knows of. It will request to join the group by providing the process identifier of its group process. It will then wait for a view delivery containing the peer processes in the group.

There is no guarantee that the request will be delivered to the leader, or the leader could be dead, and we have not detected this yet. However, the requesting application process is not told about this, so we can not do anything but wait and hope for the best. We will use a timeout; we abort the attempt if we have not been invited.

Once added to the group, the application process has the problem of obtaining the correct state (the color). It does this by using the atomic multicast layer in a clever way. It sends a request to obtain the state and waits for this message to be delivered to itself. It now knows that the other processes see this message and respond by sending the state using the multicast layer.

However, The state message might not be the first message delivered. We might have other state changes in the pipeline. Once the state is received, these state changes must be applied to the state before the process is up and running. The implementation uses the implicit deferral of Erlang. It simply lets any state change messages remain in the message queue and chooses to handle the state message first before the state change messages.

2 The first implementation

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

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

2.1 The leader process

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;
- Slaves: an ordered list of the process identifiers of all slaves in the group;
- Group: a list of all application layer processes in the group.

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

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 `Msg` is sent to the application layer.
- {`join, Wrk, Peer`}: a message from a peer or the master that is a request from a node to join the group. The message contains the process identifier of the application layer, `Wrk`, and the process identifier of its group process.

The following procedure implements the state of a leader. We use a function `bcast/3` to send a message to each process in a list.

```
leader(Id, Master, Slaves, Group) ->
    receive
        {mcast, Msg} ->
            bcast(Id, {msg, Msg}, Slaves),
            Master ! Msg,
            leader(Id, Master, Slaves, Group);
        {join, Wrk, Peer} ->
            Slaves2 = lists:append(Slaves, [Peer]),
            Group2 = lists:append(Group, [Wrk]),
            bcast(Id, {view, [self()|Slaves2], Group2}, Slaves2),
            Master ! {view, Group2},
            leader(Id, Master, Slaves2, Group2);
        stop ->
            ok
    end.
```

Notice that we add the new node at the end of the list of peers. This is important, and 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, and 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 exactly the same as that of the leader, with the only exception that the slaves keep explicit track of the leader.

The messages from the master are the following:

- {`mcast, Msg`}: a request from its master to multicast a message, the message is forwarded to the leader.
- {`join, Wrk, Peer`}: a request from the 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 `Msg` is sent to the master.
- {`view, Peers, Group`}: a multicasted view from the leader. A view is delivered to the master process.

This is the implementation of the slave:

```
slave(Id, Master, Leader, Slaves, Group) ->
    receive
        {mcast, Msg} ->
            Leader ! {mcast, Msg},
            slave(Id, Master, Leader, Slaves, Group);
        {join, Wrk, Peer} ->
            Leader ! {join, Wrk, Peer},
            slave(Id, Master, Leader, Slaves, Group);
        {msg, Msg} ->
            Master ! Msg,
            slave(Id, Master, Leader, Slaves, Group);
        {view, [Leader|Slaves2], Group2} ->
            Master ! {view, Group2}
            slave(Id, Master, Leader, Slaves2, Group2);
        stop ->
            ok
    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 and let it know that its master is the only node in the group. Since it is the only node in the group, it will, of course, be the leader of the group.

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

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

Starting a node that should join an existing group is only slightly more problematic. We need to send a `{join, Master, 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(),
    {ok, spawn_link(fun()-> init(Id, Grp, Self) end)}.

init(Id, Grp, Master) ->
    Self = self(),
    Grp ! {join, Master, Self},
    receive
        {view, [Leader|Slaves], Group} ->
            Master ! {view, Group},
            slave(Id, Master, Leader, Slaves, Group)
    end.
```

2.4 The application process

To do some experiments, we create a worker that uses a GUI to describe its state. Make sure that you can create a group and add some peers.

3 Handling failure

We will build up our fault tolerance gradually. First, we will make sure that we detect crashes, then make sure that a new leader is elected, and then make sure that the layer 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 node, and if that node 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 node, and they will never report the death of a node that has not died.

We will also assume that the message that informs a process about a death of a process is the last message that it will see from the node. 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 need not report new views when a slave dies, and there is nothing to prevent a dead slave from being 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 a 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, Slaves, Group);
```

In the election state, the process will select the first node in its lists 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, Slaves, [_|Group]) ->
    Self = self(),
    case Slaves of
        [Self|Rest] ->
            bcast(Id, {view, Slaves, Group}, Rest),
            Master ! {view, Group},
            leader(Id, Master, Rest, Group);
        [Leader|Rest] ->
            erlang:monitor(process, Leader),
            slave(Id, Master, Leader, Rest, Group)
    end.
```

We must pay attention to what we should do if, as a slave, we receive the *view message* from the new leader before we notice that the old leader is dead. Should we refuse to handle view messages unless we have seen the

Down message from the leader, or should we happily receive and accept the new view and then ignore trailing *Down messages*?

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 that its request was lost. We add a timeout when waiting for an invitation to join the group.

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

That is it; we can now add new nodes to the system and survive even if nodes crash. That was not that hard, was it? Do some experiments to see that it works, and then ship the product.

3.2 Missing messages

It seems to be too easy, and unfortunately, it is. To show that it 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 on 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 simultaneously. The initialization is, for example, done as follows; the slave will be initialized similarly.

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

init(Id, Rnd, Master) ->
```

The leader crashes before it sends messages to all slaves.

leader 2 send msg:{change,13}
slave 3 recv msg:{change,13}
slave 4 recv msg:{change,13}
leader 2 send msg:{change,18}
leader 2: crash
slave 3 recv msg:{change,18}
leader 3 send msg:{change,12}
slave 4 recv msg:{change,12}

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

Run some experiments and see if you can make the workers' state out of sync. What is 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. However, using a vanilla reliable multicaster would be very costly; we could try a smarter solution.

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 assume 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 sends messages to the peers in the order that they occur in the list of peers. If anyone receives a message, the first peer in the list receives the message. This means that only the next leader needs to resend the message.

This will introduce the possibility of doublets of messages being received. To detect this, we will number all messages and only deliver new messages to the application layer.

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

- `slave(Id, Master, Leader, N, Last, Slaves, Group)`: 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, Slaves, Group)`: the election procedure is extended with the same two arguments.
- `leader(Id, Master, N, Slaves)`: 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, Peers, Group`}: 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 generate a hard-to-handle trouble report after a year.

When discarding messages, we discard 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 ->
    slave(Id, Master, Leader, N, Last, Slaves, Group);
```

You might wonder how a message could possibly arrive early, but there is a small window where this could actually happen.

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.

```
      bcast(Id, Last, Rest),
```

This completes the transition, and `gms3` should be ready for release.

3.4 Some experiments

Run some experiments and create a group spanning several computers, if available. Can we 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 works depending on what the Erlang environment guarantees and how strong our requirements are.

4 Optional task for extra bonus: 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 the reliable delivery of messages, something that is not guaranteed.

How would we change the implementation to handle the possibly lost messages? How would this impact performance?

```
11> test:more(3, gms4, 1000).
<0.168.0>
leader 1 dropped message: {msg,17,{change,14}}
Message {msg,17,{change,14}} recast
leader 1 dropped message: {msg,28,{change,17}}
Message {msg,28,{change,17}} recast
```

***For the extra bonus*, change your implementation to handle the possibly lost messages. In the report, shortly explain your changes and discuss how this would impact performance.**

The second reason things will not work is that we rely on the Erlang failure detector to be perfect, i.e., it will never suspect any correct node for having crashed. Is this really the case? Can we adapt the system to behave correctly if it does make progress, even though it might not always make progress?

The third reason things do not work is that we could have a situation where one incorrect node delivers a message that any correct node will not deliver. 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? What would a solution look like?