Report for Distributed systems project

Rémi Coudert

30 avril 2017

Topology

The topology I chose to use is a ring in which nodes are connected to their direct neighbours and to their second neighbours, it permits to be more robust to sudden node shut down. Indeed its direct neighbours still have a link between themselves so they need to

1. know that their neghbours died
2. retrieve their new second degree neighours through messages
3. go back to idle

It can make the ring really more robust than a standard one, to disturb the topology, we need to have two nodes dying without prior notice at the exact same time being direct neighbours. Obviously the chances of this to happen, assuming every node is independant from its neighbours, is $\frac{1}{n^2}$, n being the size of the ring, for each time slot, which scales really well.

On the other hand, message passing in this topology does not scale so well, the number of messages is indeed linear in n (it's $n/2$ since we can communicate in both ways). I did not implement message passing in both ways but it can be implemented easily since each node keeps track of its neighbours.

To represent the neighbours of each node I used Erlang records. Records are more or less the equivalent of structs in C. I created 4 types of records, each one described in the file **records.hrl** (.hrl files are erlang header files, it permits me to carry them accross my files without redefining them).

The topology has 3 special case, when its size n is 1, 2 or 3 so I created a record for each of this case and for the general one. Each record contains the list of UIDs for the current files held by the node.

The four records are :

— **singletonState** : For $n = 1$.

— **pairState** : For $n = 2$, it holds the value **peer** which is the Pid of its only neighbour.

— **tripletState** : For $n = 3$, it holds the values **next, prev** which is the Pids of its two neighbours.

— **genState** : For $n > 3$, it holds the values **next, prev, nnext, pprev** which are the Pid of its 4 connections.

Since records can be pattern-matched they are very useful for simplicity and modularity of the code.

Files structure

There are 3 folders worth to be mentioned : **src** that holds all source files, **ebin** that holds .beam files from the compilation as well as application resources files (used for applications, as explained below) and **test**, the tests folder. The test folder is not longer usable with the current version but I did not erase it because it contained useful tests I could have reused (such as for broadcast and node creation).

In **src**, there is the folder **basic** that contains the old version of the code in case it's interesting. It contains the following files :

— **distrData.erl** : This is the main application

— **distrDataLocal.erl** : This is the local version of the main application.

— **handle_X.erl** where $X \in \{deletion, event, idle, insertion, message, work\}$: These files contain the different handlers for the messages depending on the state of the node or the type of the message, they permit to keep some modularity in the code.

— **local_supervisor** : The top supervisor for the local version (see below)

— **node_statem** : This is the main file for the nodes, it represents their state machine even though the files does not hold much code thanks to the handler file.

— **node_supervisor** : The supervisor for the basic nodes as discussed below
— **record.hrl** : This header file contains the definition of the records I used.

There are also the **Makefile** and the **EMakefile** (erlang makefile, for compiling).

Erlang OTP

To implement the topology I used Erlang OTP behaviours. They are a set of implementation of programming designs for Erlang. The basic idea is that an OTP Application is composed of an application behaviour that start the root of the supervisors tree. Below are the behaviours I used and what for :
— State machine : I used a state machine to represent the nodes, see below for a full description of nodes behaviour
— Supervisors : Supervisors are processes used to spawn some given node, and supervise them, that is restart them if they die (among other things) depending on the strategy chosen. Every node of the ring is supervised. See below for a full description of my supervisors' behaviour.
— Applications : Applications are a convenient way to abstract a set of functions, it is often used to spawn the supervisor at the top of the supervisor tree. It is useful for easy and maintainable deployment.

gen_statem behaviour, states and nodes

Official documentation
I used this behaviour to implement a finite state machine. Each state has an 'internal state' as described in the topology and some internal data, which hold the UUIDs for the data the node received. The possible state for the FSM are :
— idle : This is the basic state in which the node is when started when in this state the node is waiting either to start as a singleton or to be asked to connect to an existing topology
— work : This is the main state, in which the node can receive messages and events, both are handled in separate files.
— addingNode : When in this state a node is waiting either for a signal indicating that the node connecting to it is indeed connected or a cancel signal
— removingNode : Same as addingNode but for removal

message passing

Message passing in OTP is mostly done in three ways :
— asynchronous messages : called 'cast'
— synchronous messages : called 'call', that need a reply
— info messages : basically normal erlang messages (sent with '!')

I only used cast and call.

supervisor behaviour

Official documentation
This behaviour is used to supervise processes, that is start them and restart them when they crash depending on the rules we choose.
What I did is creating a supervisor for every node, so that a node that dies is immediatly revived, (we need to reconnect it though).

In addition, I used supervisors for the local version of the application. I created a **local_supervisor** that supervised the node supervisors, it is easier to start and maintain the application this way.

some implementation details

To test my implementation, I created a new supervisor (local_supervisor), that permits me to create local nodes (i.e. all worker in the topology are in a single node / erlang shell).
I did not have time to implement the real distributed system (that is, it only work really on the local version), all the functions I implemented can be easily ported though, it suffices to replaces the Pids by global Pids or {Pid, Node}.

I used the *md*5 from erlang to compute the UUIDs for the data, the data being written to a file on the node's computer.

Concerning node addition and deletion I made the choice of completely pause 4 nodes each time (in the general case), so that no message is processed during this time by these nodes beside the wanted ones. This means that there is a very low chance to have problems during node addition/deletion such as double insertion for example. The downsides of this are

1. Other messages are not processed during that time
2. In case of an internal problem (e.g. one of the paused node crashes), it may cripple the whole ring

To ease a it the constrains I added a (not used) way to cancel an addition/deletion via a message.
Handling addition and deletion was not so easy because I needed first to pause the nodes correctly and then to add/remove the given node. I had to take care of each case whether in term of size in the topology or in term of node position (who is next whom ? who is before whom ? etc..).

For the easy cases I computed the new local topology on a single node and then sent the result to the concerned nodes. For the more general cases I had to compute for each of the concerned nodes their new neighbours, then compute the neighbours for the new node (in the case of node addition), and finally send the result to the new node.

Deletion was a bit easier since I only needed to break links and not to add new ones but it wasn't trivial either.

some implemented functionalities

This is the list of what I've implemented :

— Broadcast : I have a simple test broadcast message I used at first, but a broadcast can easily be done using the function $sendMsgNext$, this function remembers which nodes were visited, and has a message to broadcast as well as an action to do when the broadcast is done (e.g. send a message), under the form {M, F, A}.

— I did not implement scatter as such but it is easy to use $sendMsgNext$ to create a scatter, but if there are n messages, only the first n nodes will receive a message.

— adding and removing nodes

— reacting to node death is node implemented as such but since the application uses supervisors, a dead node is immediately respawned, but since it's a new one it didn't keep track of its neighbours. One way to handle this would be to keep a copy of the neighbours in the supervisor.

— data can be added and retrieved from the topology. I wanted to implement data saving by automatically copying uniformly the data to random nodes in the topology thus resolving both robustness and speed of data retrieval but I wasn't able to finish it on time so the current implementation is a standard one, that is keeping the data to one node only.

— size query : I implemented a function to retrieve the size of the ring.

unimplemented functionalities and possible improvements discussion

I'll discuss here of the functionalities I did not implement and how I'd have done with more time :

Reacting to the crash of a node can be done relatively easily thanks to the use of supervisors as discussed above. The main issues are to add the respawned node to the ring, and find a way to tell its old neighbours that it's dead.

I did not make use of the 4 links usable by each node, but this can be easily used for future usage since they're kept in the internal state anyway.

The use of cast and call is inconsistent in my code. I used call (i.e. synchronous messages) it critical cases such as node addition/deletion for example because it was clearly required. However if we don't make the assumption of a reliable network all the cast (asynchronous messages) have to be replaced (either by calls or something else).

Leader election can be very straightforward given that message passing works correctly, it depends however on how the election is done.

Remotely spawning agents can be done using the **rpc :call()** function which permits to execute functions on a node, it would suffice to simply launch the application on another node (**application :start(distrData)**).

usage

You can use the makefile first to compile and run the application :
— **make compile** to compile
— **make run** to compile and run the main application (not implemented, only work for one node as discussed above).
— **make local** to compile and run the local version of the application (implemented). It will automatically create the first node.

Then you can use the following commands (local) :
— **distrDataLocal :addNode().** to add a node to the topology, some messages will output the status.
— **distrDataLocal :removeNode().** to remove a node from the topology, some messages will output the status.
— **distrDataLocal :size().** to query the size of the ring.
— **distrDataLocal :addData(Data).** to add some data, the function return the UUID for the data.
— **distrDataLocal :retrieveData(UUID).** to retrieve some data added to the system previously.

Conclusion

As you can see, there lack some required functions as well as many optional ones. This is due to the fact that I spent a **lot** of time thinking about how to handle my topology (it was quite harder than I thought), and to understand Erlang and OTP more in depth.

My choice of using OTP behaviours in my code yielded a big amount of time spent rewriting entirely my code (3 times), each time more modular and more like a real Erlang application.

This implies that my code, for the most part, is easily changeable and improvable, you basically just need to handle messages in the correct file. The obvious downside is that it took me too much time to do it with regards to the time we had.

With the time (well) spent on this project doing it the best way possible in addition with my interest in it it is quite possible that I'll dig deeper on it (that's why the code is on github).