SINF2345 : A simulated bike race

Report

Crombé Henri (72580800)
Declercq Mallory (41661200)

April 2016

1 Introduction

In this project, we were asked to simulate a bike race using Erlang and Riak-Core. Erlang is a programming language used to build scalable real-time systems with requirements on high availability. The application is developed on top of the Riak-Core distributed key-value store, which can be thought of as a toolkit for building distributed, scalable, fault-tolerant applications. In the context of the project, a race consists of distinct bicker nodes that exchange messages to agree on the global state of the race. The race advances by rounds. During these rounds, each bicker node receives as input the strategy to follow and then act according to the broadcast technique chosen (beb or tob). When a round is over, the global state of the race is known by every bickers and a new round starts again. The race is over when a bicker reach the total distance to cover.

2 Problem 0 : Best-effort broadcast

Best-effort broadcast is a very simple broadcasting technique. Basically, with this technique, every node can update its view of the global system according to the inputs received and then broadcast its updated view to all other nodes. In the case of this project, beb was not well suited and we observed strange behaviour during the race. The strange behaviour came from the fact that nodes didn't agree on an well-defined order to perform their updates. Consequently, we observed in some cases that the global state of the system seen by a node was not the expected one. This problem can be frequently observed during a round. For example, if the node corresponding to *biker1* tries to recover the global state of the system, and unfortunately this state has already been updated by another node (say *biker4*) a few microseconds before, then the result for *biker1* is a wrong approximation of the current state of the system.

3 Problem 1 : Total-order broadcast

To fix the problem discussed in the previous section, it's crucial that all nodes agree on the same order to perform the updates. For this purpose, we built a total-order broadcast based on a hierarchical consensus [1]. The key idea of the hierarchical consensus is to exploit the ranking among the processes (from 1 to N) in order to elect a leader. The leader is the only node that can impose its value to all the other nodes. In our case, we modify a little bit the hierarchical consensus to elect a leader according to the round number of the race. Namely, the round numbers loop between 1, 2, 3, 4, 5 because we have 5 bikers and each biker becomes a leader in

1. cf. reference book, section 5.1.3 "Fail-stop algorithm : Hierarchical Consensus" (p.209)

its turn. For example, if the round number of the race is 4 then we know that the leader for this round is biker4. As rounds are synchronized and each biker knows the current round of the race, all biker knows who is the leader during the current round. The fact that all bikers are aware of the leader is very important. Indeed, this specification allows each biker to send its user's input to the leader (unicast). Once the leader has received the input from all bikers still in the race (i.e. with a remaining energy higher than zero). He updates the global state of the system and broadcast it to all other bikers (beb). As soon as, all bikers have updated their local state with the leader's global state, a new round can start (a new leader is elected and the overall process is repeated until a biker wins the race).

Without taking into account faulty nodes, this way of doing ensures that the system is always consistent because a single node performs updates and imposes its updated result to all other nodes. Consequently, all nodes observes exactly the same state of the system.

4 Problem 2 : Total-order broadcast with failing nodes

Unfortunately, we did not have enough time to finish the fault tolerant total-order broadcast. In the remainder of this section, we explain a way to build it inside the Riak-Core framework. First of all, let's assume that we can cause a failure on a node by doing a CTRL+C (then press the key *a* to abort) in the console attached to the node. Based on that assumption, we can implement a failure detector that sends a heartbeat message to all nodes present in the cluster. If the node is alive, it will answer to the heartbeat message. Otherwise, an exception will be raised indicating that the node is crashed. Figure 1 depicts the implementation of such a failure detector.

```
heartbeat()->
   Members = get_member_list(),

   VnodesB1 = [{Index, NodeId} || {Index, NodeId} <- Members, NodeId =:= 'biker1@127
       .0.0.1'],
   VnodesB2 = [{Index, NodeId} || {Index, NodeId} <- Members, NodeId =:= 'biker2@127
       .0.0.1'],
   VnodesB3 = [{Index, NodeId} || {Index, NodeId} <- Members, NodeId =:= 'biker3@127
       .0.0.1'],
   VnodesB4 = [{Index, NodeId} || {Index, NodeId} <- Members, NodeId =:= 'biker4@127
       .0.0.1'],
   VnodesB5 = [{Index, NodeId} || {Index, NodeId} <- Members, NodeId =:= 'biker5@127
       .0.0.1'],

   B1Crashed = heartbeatBiker1(VnodesB1),

   B2Crashed = heartbeatBiker2(VnodesB2),

   B3Crashed = heartbeatBiker3(VnodesB3),

   B4Crashed = heartbeatBiker4(VnodesB4),

   B5Crashed = heartbeatBiker5(VnodesB5),

   L = [{'biker1@127.0.0.1', B1Crashed},
       {'biker2@127.0.0.1', B2Crashed},
       {'biker3@127.0.0.1', B3Crashed},
       {'biker4@127.0.0.1', B4Crashed},
       {'biker5@127.0.0.1', B5Crashed}],

   CrashedList = [Id || {Id, C} <- L, C =:= true],
   CrashedList.


   % return true if biker1 is crashed, false otherwise
heartbeatBiker1(VnodesB1) ->
  {I, N} = hd(VnodesB1),
   try riak_core_vnode_master:sync_spawn_command({I, N}, heartbeat, biker_vnode_master)
        of
     {_, _} -> false
   catch exit:{{nodedown, _}, _} ->
      true
   end.

   % heartbeatBiker2, heartbeatBiker3, heartbeatBiker4 and heartbeatBiker5 have the
       same behavior than heartbeatBiker1.
```

Figure 1. Implementation of failure detector.

Thanks to this failure detector, we are able to know which nodes of the cluster are faulty at a given time *t*. Then, if we want to keep the system consistent, it's crucial to detect when the leader is down because in our implementation of the total-order broadcast, the leader is responsible of broadcasting the new global state of the system to all other nodes. Consequently, we suggest to use the heartbeat method at the beginning of the start_race() method. This way, we will be able to directly go to the next round if we detect a crash on the leader. Furthermore, we suggest that all nodes verify, via the heartbeat method, if the leader is alive before sending their input to it. If they detect a crash on the leader then they go to the next round. The key idea is to always know if the leader is working or not. Once a failure is detected on the leader then we must force each correct nodes to go to the next round. The next round will be the same as the previous one because only the leader can broadcast new state. Consequently, the system remains consistent. When other nodes than the leader crash during a round, we can handle them by flagging them in the global state of the system (e.g. if biker1 crashed and biker1 is not the leader then put the field *crashed* = true for this biker) This way, the leader knows that it doesn't need to wait an input from these nodes. Once the leader gets the input of all correct nodes, he updates the global state of the system and broadcast it to all correct nodes.

5 Conclusion

This project was interesting as it made us work with different abstractions and theoretical parts of the course. Being able to implement and observe the behavior of a distributed system based on different broadcasting mechanisms was helpful to catch the pros and cons of such techniques. Finally, we found this project quite hard because we didn't know Erlang and Riak at all. That's why we didn't finish the last part of the project on time.