LSINF2345
Final project: Support for atomic transaction in LASP
Report

Martin Olivier - olivier.martin@student.uclouvain.be - 22341300
Naitali Brandon - brandon.naitali@student.uclouvain.be - 10261300

May 5, 2017

Introduction

This project consists in extending the Lasp API by adding atomic transaction.

Code

For each 'project tasks' of the project, we incated in the code with a big comment flag indicating which part of this project it is related to. We modified files:

- `lasp_state_based_synchronization_backend.erl`
- `lasp_synchronization_backend.erl`
- `lasp.erl`

We also create a new file `lasp_synchronisation_transaction.erl` for our buffer: we create a new process that keep in memory the buffer and allows us to update it by sending messages to this process. This allows us to easily split our program into different parts and avoid making the code heavier.

Decisions

In order to implement transactions, we made some choices. A first decision was the way we modeled the buffer. We chose a map containing mapping between the node, the transaction id and the list of updates to apply. It seems to be a good choice for the lookup in constant time (when we get an ack for example) and a intuitive way to store key-value pair. This map allows us to easily add, remove and send messages to all peers because, the peers are the keys of the map. You can found in our code more comments justifying our choices, decisions and how we implemented the transaction mechanism.

We also add a transaction id in every transaction. We needed to add this because when we get an ack from a peer, we need to remove the transaction from the buffer but, to identify which transaction we have to remove we need a way to identify the correct transactions.

Since we implement the `lasp:transaction()` and we disabled the normal synchronization mechanism, the `lasp:update()` function should not be used anymore because changes made by the update function will only be local to the node.

Trade-offs: Message delivery

We decided to do "At-Least Once Message Delivery". This approach is more "safe", thanks to acknowledgment. We send the message until an acknowledgment is received, it will not be lost; we guarantee that the message will be eventually delivered. But there are several drawbacks: the use of the link is intensive, duplication can occur and it needs an acknowledgment mechanism. We consider that updates are messages that need to be acknowledged because a node without the last version of the system can lead to a misunderstanding between the real version and previous one.

Troubles

We had a lot of troubles to do this project because we were often lost in the code of lasp and we lose lots of time to try to understand the code. We think some short description of what functions does and what are the input of the functions (specifications) would have been very helpful as well as some comments in the code. Moreover our lack of knowledge about Erlang did not help us.

Conclusion

We implemented a transaction mechanism in the lasp API: when the transaction function is called, the list of updates is saved in a buffer. Then we send every 4 seconds all transactions in the buffer to the corresponding peers. When the peer reply with an acknowledgment, we remove the transaction from the buffer. We chose the "At-Least Once Message Delivery" to ensure all peers get eventually the transactions. However, messages can be duplicated and receives twice by a peer.

How to run the project

```
Run the project: in "lasp" folder:
> make
> make shell

Run the tests: in "lasp" folder:
> make test

Tiny manual test: in "lasp" folder:
1) Build a cluster of two nodes by following:
   https://lasp-lang.readme.io/docs/starting-and-clustering-two-nodes

2) On node 1, run:
   > lasp:declare({<<"set">>, state_orset}, state_orset).
   > lasp:declare({<<"set2">>, state_orset}, state_orset).
   > lasp:transaction(
   [{{<<"set">>, state_orset}, {add, 1}}, {{<<"set2">>, state_orset}, {add, 2}}], self()).
   > lasp:transaction(
   [{{<<"set">>, state_orset}, {add, 3}}, {{<<"set2">>, state_orset}, {add, 4}}], self()).

3) On node 2, this query should return [3,1]:
   > lasp:query({<<"set">>, state_orset}), sets:to_list(ValueR1).
```