Université catholique de louvain-la-neuve

LSINF2345 - Languages and Algorithms for Distributed Applications

Final Project : Support for Atomic Transactions in Lasp

Scott Ivinza Mbe
Jos Zigabe

5 mai 2017

Introduction

For this course project, we have to adapting an existing system in Erlang to add new functionality. More precisely, this project aims to add support for *atomic transactions* in the *Lasp programming system.* Lasp programming system is a programming model for building *distributed applications.* The goal of Lasp is to allow developers to write a program once and know that it will behave the same whether it is executed on one computer or distributed across many. In addition, *Lasp*, outside of being a programming system, it is just a *distributed database.* It offers *two* main actions : *query*, to get the current value of an object, and *update*, to update the value of an object based on a strict API that ensures updates won't conflict with each other.

More concretely, we're going to extend the model to have a new command that let's us specify a *list of updates* to be applied *at the same time.* Then, we have to ensure we only send the list of updates together (by disabling the behavior where we try to send around individual changes) and ensure updates are delivered together in all of the nodes in the distributed systems.

1 Project Tasks

1.1 Create a new API call named lasp :transaction()

In order to add a new API call named *lasp :transaction()* in Lasp system, we were inspired by the way that *lasp :update()* call was implemented inside Lasp system and then we modified all *.erl* files which are concerned when a change is propagated through the Lasp system programming (Anatomy of a Request). These *.erl* files are the following : *lasp.erl, lasp_distribution_backend.erl*, *lasp_core.erl*, *lasp_storage_backend.erl*, *lasp_ets_storage_backend.erl*, *lasp_synchronization_backend.erl* and *lasp_state_based_synchronization_backend.erl.*

Unlike to *lasp :update()* call, *lasp :transaction()* takes a list of updates to be performed. Thus, the format of parameters of transaction call is the following : ([ListOfObjectIdsAndOps], Actor) where *ListOfObjectIdsAndOps* is a list of that contains pairs of *ObjectId* and *Operation* in the form of tuples that is [{*ObjectId1,Operation*}, {*ObjectId2,Operation*},....]. *Actor* is an Actor identifier that identify this actor uniquely in the system : this is used to detect concurrent operations from different actors in the system, and it's assumed that each actor acts sequentially.

Here, we've added the value 5 and value 20 to the set on this local replica in the form of a transaction. *Eventually*, all of the nodes in the system will see these values (5 and 20) in the set, but it might take some time before that happens.

```
lasp:transaction([{{<<"set">>,state_orset}, {add,5}},{{<<"set">>,state_orset}, {add,20}}], self()).
```

And we get the following result in Erlang shell :

```
{ok,{{<<"set">>,state_orset},
    state_orset,
    [{clock,[{<<170,227,142,126,63,64,19,227,195,66,39,125,58,
                195,75,134,148,109,168,...>>,
              3}]}],
    {state_orset,[{5,
                   [{<<231,91,83,72,94,78,144,98,37,135,83,174,237,235,230,
                       105,...>>,
                     true}]},
                  {20,
                   [{<<37,210,132,184,49,72,155,245,12,198,80,159,172,100,43,
                       ...>>,
                     true}]}]]}}}
```

We can now see that the set reflects the added elements.

```
> {ok, Value1} = lasp:query({<<"set">>, state_orset}), sets:to_list(Value1).
[5,20]
```

1.1.1 Handling of a transaction rollback

By definition, a transaction is *atomic* when either it *succeeds entirely* or it *fails entirely*. Thus, we note that when a transaction is performed it is important to handle the case where the transaction can fail, for instance if among the list of updates there is at least one operation which is not correct then the entire transaction fails. In our transaction function, we handle the *transaction rollback* as the following : First of all, we put all the updates of the transaction into a buffer and at the sending of each update of this list (or buffer) of updates we make sure to receive an *Ack* before moving to the next update of the list. So, once the *Ack* of the last update is received the transaction will be entirely executed. Otherwise, it fails entirely.

1.2 Disable the normal synchronization mechanism

In order to allow the sending of several updates belonging to the same transaction at the same time. First of all, we had to put them together in a list. Then, in the *init_state_sync(.....)* function, a case has been added to process this list in order to be able to create a *blocking function* for each update of the list before sending them at the same time. In this way, we ensure that transaction will be performed once the last function is created. Finally, since the transaction will require an *Ack* of each of its update, we also had to make a small change in *handle_info (state_sync,...)* function by setting the *blocking_sync* to true. Otherwise, the transaction remains blocked indefinitely while awaiting the *Ack*.

```
init_state_sync(Peer, ObjectFilterFun, Blocking, Store, BufferId, Key) ->
    case ObjectFilterFun of
        %Check if we have a list of ObjectFilterFun
        [Head_ObjectFilterFun|Tail] ->
            ...
        _ ->
            ...
    end.

...
%% Ship buffered updates for the fanout value.
SyncFun = fun(Peer) ->
                    case lasp_config:get(reverse_topological_sync,
                                         ?REVERSE_TOPOLOGICAL_SYNC) of
                        true ->
                            init_reverse_topological_sync(Peer, ObjectFilterFun, Store);
                        false ->
                            BufferId = ets:new(buffer, [ordered_set, protected]),
                            init_state_sync(Peer, ObjectFilterFun, true, Store, BufferId, 0),
                            ets:delete(BufferId)
                      end
              end,
%% Send the object.
            SyncFun = fun(Peer) ->
                            {ok, Os} = init_state_sync(Peer, ObjectFilterFun, true, Store),
                            [{Peer, O} || O <- Os]
                      end,
            Objects = lists:flatmap(SyncFun, Peers),
```

1.3 When API is called, buffer the updates

When the new API call *lasp :transaction()* is called, we have to put the updates that are done in the same transaction in a buffer. In order to implement our buffer, we chose to use a data structure named *ETS Table* as I already mentionned it in the previous section. An *ETS table* is just a collection of Erlang tuples. In an *ETS table*, one of the elements in the tuple (by default, the first) is called the key of the table. Thus, an *ETS table* is data structure for associating keys with values. Furthermore, we can insert tuples into the table and extract tuples from the table based on the key.

One of our difficulties was first of all to know where to define the buffers. After thinking, we told ourself that it was better to have a table that would be defined in the *init()* function of the *state_synchronization_backend.erl* file and that would be placed in the record state. Then once in the *handle_call (blocking_sync,..)* method, we defined another subtable that we placed in the table contained in the state record with key as the name of the node peer. In this way, we ensured that each peer will have its own buffer to store the update of the transaction.

The following code fragment illustrates the implementation of our buffer using a ETS table data structure :

```
%% @private
-spec init([term()]) -> {ok, #state{}}.
init([Store, Actor]) ->

    ...

%tab that will contain the buffers for all nodes
Buffer_list = ets:new(buffer_list, [named_table, ordered_set, public]),

    {ok, #state{gossip_peers=[],
                blocking_syncs=BlockingSyncs,
                store=Store,
                actor=Actor,
                buffer=Buffer_list}}.


handle_call({blocking_sync, ObjectFilterFun}, From,
            #state{gossip_peers=GossipPeers,
                   store=Store,
                   blocking_syncs=BlockingSyncs0,
                   buffer=Buffer_list}=State) ->

    ...


    case length(Peers) > 0 of
```

```
         true ->
            %% Send the object.
            SyncFun = fun(Peer) ->
BufferId = ets:new(Peer, [ordered_set, protected]),
%add the peer's buffer in a list of buffer
ets:insert(Buffer_list, {Peer, BufferId}),
{ok, Os} = init_state_sync(Peer, ObjectFilterFun, true, Store, BufferId, 0),
%delete the peer's buffer from the list
ets:delete(Buffer_list, Peer),
[{Peer, O} || O <- Os]
                        end,
            ...
    end;
```

1.4 Periodically, synchronize with peers

We noted that the periodical synchronization go through the function *init_state_sync()*. To ensure that the synchronization runs correctly with the other nodes of the system, we first set *blocking_sync* to true in order that we receive an *Ack* of each peer. We also ensure that it is only when we receive the *Ack* that the buffer is deleted. Finally, we ensure that buffered events arrive in the correct order using an ETS table with a key of type integer.

1.5 On receive, acknowledge them and apply them locally

For each update sent, we apply the updates event locally (that is, in each node in the system that receives this update) using the *?CORE :update API* and then each node acknowledges to the node sender that he receives this update.

The following code fragment illustrates that for each update sent we apply them locally in the node receiver :

```
handle_cast({state_send, From, {Id, Type, _Metadata, Value}, AckRequired},
            #state{store=Store, actor=Actor}=State) ->
    lasp_marathon_simulations:log_message_queue_size("state_send"),

%apply the updates locally
    {ok, Object} = ?CORE:receive_value(Store, {state_send,
                                       From,
                                       {Id, Type, _Metadata, Value},
                                       ?CLOCK_INCR(Actor),
                                       ?CLOCK_INIT(Actor)}),
%acknowledge the update
    case AckRequired of
        true ->
            ?SYNC_BACKEND:send(?MODULE, {state_ack, node(), Id, Object}, From);
        false ->
            ok
    end,

    %% Send back just the updated state for the object received.
    case ?SYNC_BACKEND:client_server_mode() andalso
         ?SYNC_BACKEND:i_am_server() andalso
         ?SYNC_BACKEND:reactive_server() of
        true ->
BufferId = ets:new(buffer, [ordered_set, protected]),
            ObjectFilterFun = fun(Id1) ->
                                      Id =:= Id1
                              end,
            init_state_sync(From, ObjectFilterFun, false, Store, BufferId, 0),
ets:delete(BufferId),
            ok;
        false ->
            ok
    end,

    {noreply, State};
```

1.6 Ensure atomicity and FIFO

To ensure atomicity and FIFO, when we perform a transaction this can be achieved in the following way : if the argument *blocking_sync* in the method named *handle_call({blocking_sync, ObjectFilterFun}, From, #state{gossip_peers=GossipPeers, store=Store, blocking_syncs=BlockingSyncs0}=State)* which is located in *lasp_state_based_synchronization_backend.erl* file is set to *true* then all the updates will wait for an acknowledgement of all the nodes and our implementation ensures that updates are applied all at the same time before resuming. Moreover, these updates will be delivered in the correct order and more particularly in the FIFO order thanks to the ETS table data structure that we defined as an *ordered set.* In this way, for each update sent to a node peer, this node peer must acknowledge us that it receives the update before that the next update be sent to this node.

1.7 Adapt the test suite

Our test that verifies that each node receives all of the updates in the correct order is shown in the following code fragment :

```
 %% @doc Transaction.
transaction_test(_Config) ->
%% Create new set-based CRDT.
    {ok, {SetId, _, _, _}} = lasp:declare(?SET),

    %% Determine my pid.
    Me = self(),

    ?assertMatch({ok, _},
                 lasp:transaction([{SetId, {add, 1}}, {SetId, {add, 2}}], actor)),

    {ok, {_, _, _, V0}} = lasp:transaction([{SetId, {add, 3}}], actor),

    ?assertMatch({ok, _},
                 lasp:transaction([{SetId, {add, 4}}], actor)),
    %% Spawn fun which should block until lattice is strict inflation of
    %% V0.
    I1 = first_read,
    spawn(fun() -> Me ! {I1, lasp:read(SetId, {strict, V0})} end),

    %% Ensure we receive [1, 2, 3, 4].
    Set1 = receive
        {I1, {ok, {_, _, _, X}}} ->
            lasp_type:query(?SET, X)
    end,

    %% Perform more inflations.
    {ok, {_, _, _, V1}} = lasp:transaction([{SetId, {add, 5}}], actor),

    ?assertMatch({ok, _}, lasp:transaction([{SetId, {add, 6}}], actor)),

    %% Spawn fun which should block until lattice is a strict inflation
    %% of V1.
    I2 = second_read,
    spawn(fun() -> Me ! {I2, lasp:read(SetId, {strict, V1})} end),

    %% Ensure we receive [1, 2, 3, 4, 5].
    Set2 = receive
        {I2, {ok, {_, _, _, Y}}} ->
            lasp_type:query(?SET, Y)
    end,

    ?assertEqual({sets:from_list([1,2,3,4]), sets:from_list([1,2,3,4,5,6])},
                 {Set1, Set2}),

    ok.
```

2 Project Choices

2.1 At-Least Once Message Delivery

Our choice for messages delivery is *at least once message delivery*. That is, each peer in the system can receive a message at least once. More precisely, we can execute the updates on the node where the updates originated immediately, then buffer the effect of those updates and send the values in the buffer : these changes will be idempotent, which means we won't have to ensure they are delivered at-most once and can keep sending them until you receive an acknowledgement.

3 Bonus Points

3.1 Trade-offs

We did the choice of *at least once message delivery* because in a *point-to-point* message communication where a message sent is delivered *at least* once that is example of *liveness* property that we saw during the lectures. By definition, a liveness property is property that state that something good *eventually* happens that is liveness can always be made true in finite time. And often liveness is just *termination*. For instance, as I said in the section 1.1, when we've added the value 5 and value 20 to the set on this local replica in the form of a transaction. *Eventually*, all of the nodes in the system will see these values (5 and 20) in the set, but it might take some time before that happens. This is guaranteed by the liveness property. The trade-off with liveness is that in one hand liveness can only be satisfied in *finite time*(that is when good happens) and in the other hand liveness is violated in *infinite time*(there's alaways hope).

3.2 Configurability

In order to make the system *configurable* based on our choice for messages delivery that is either *at least once message delivery* or *at most once message delivery* using a configuration parameter. First of all, given that our implementation choice for messages delivery is *at-least once message delivery*, we just need to modify *lasp_config.erl* file, such that we can configure the parameter to wait for node acknowledgments or not wait for them at all. In the first case, we will have the first approach which is *at-least once message delivery* and in the second will have the second approach which is *at most once message delivery.*

Conclusion

This project was interesting in fact by working on this project, we understood better the algorithms for distributed application seen during the lectures and therefore we improved our hard skills on distributed programming. Moreover, we learned Erlang programming language and Lasp programming system things that we did not know before. We hope that our implementation will be integrated in the final version of Lasp and thus will be used by a few compagnies in the Silicon Valley.

4 Appendices

User Manual

Our application works with *Erlang 19*, the standard distribution that is used for Lasp.

Installing

If you want to install and build Lasp :

1. Open a terminal at the root of the project(for instance *cd/lasp*)
2. Install and build Lasp using : *mak*e

Running a shell

If you want to run a Erlang shell where you can interact with a Lasp node :

1. Run a Erlang shell using : *make shell*

Running the test suite

In order to run the test suite, which will execute all of the Lasp scenarios :

1. Run the test suite using : *make check*