# Opty: optimistic concurrency control

**Jordi Guitart**

Adapted with permission from Johan Montelius (KTH)

September 23, 2021

# Introduction

In this session, you will implement a transaction server using optimistic concurrency control with backward validation. You will also learn how to implement a updatable data structure in Erlang that can be accessed by concurrent, possibly distributed, processes. Before you start, you should know how optimistic concurrency control works (you could also read the paper 'On Optimistic Methods for Concurrency Control' by Kung and Robinson).

# 1 The architecture

The architecture consists of a *transaction server* process having access to a *store* and a *validator*. The *store* consists of a set of *entries*, each holding a value and a unique reference. The reference is updated in each write operation so we can validate that nothing has happened with the *entry* since we last read its value.

A *client* starts a transaction by creating a *transaction handler* that is given access to the *store* and the *validator*. The *transaction server* is thus not involved in the transaction; it is simply a process from which we can get access to the *store* and the *validator*.

## 1.1 The handler

The *transaction handler* will handle read and write requests from the *client* and will also close the transaction. In each read operation, the *handler* keeps track of the unique reference of the *entry* that is read. It also keeps the write operation in a local store so that the real *store* is not modified until we want to close the transaction.

When the transaction is closed, the *handler* sends the read and write sets to the *validator*.

## 1.2 The validator

The *validator* process will be able to tell if a value of an *entry* has been changed since the transaction read the entry. If none of the *entries* have changed, the transaction can commit and the associated write operations are performed.

Since there is only one *validator* in the system, it is the only process that actually writes anything to the *store*.

# 2 The implementation

Since the implementation has some complexity, it is probably a good idea that you draw a diagram containing the different involved processes and their interaction via messages in order to fully understand how the system works.

Since data structures in Erlang are not mutable (we cannot change their values), we need to do a trick. The *store* is represented by a tuple of process identifiers. Each process is an *entry* and we can change its value by sending it write messages.

A *transaction handler* is given a copy of the tuple (i.e. the *store*) when created, but the processes that represent the *entries* are of course not copied. The *store* can thus be shared by several *transaction handlers*, all of which can send read messages to the *entries*.

The *validator* will not need access to the whole *store* since it will be given the read and write sets from the *transaction handlers*. These sets will contain the process identifiers of the relevant *entries*.

## 2.1 An entry

A process that implements an *entry* should only have a value and a unique reference as its state. We will below call this reference "timestamp" but it has nothing to do with time, it's simply a unique reference. The reference will be given to a reader of the value so that the *validator* later can determine if the value has been changed. We could do without the reference but it has its advantages.

```
-module(entry).
-export([new/1]).

new(Value) ->
    spawn_link(fun() -> init(Value) end).

init(Value) ->
    entry(Value, make_ref()).
```

Note that we are using the primitive `spawn_link/1`. This is to ensure that if the creator of the *entry* dies, then the *entry* should also die.

Three messages should be handled by an *entry*:

- {`read, Ref, From`}: a read request tagged with a reference. We will return a message tagged with the same reference so that the requester can identify the correct message. The reply will contain also the process identifier of the *entry*, the value, and the current timestamp.

- {`write, Value`}: update the current value of the *entry*, no reply needed. The timestamp of the *entry* will be updated.

- {`check, Ref, Readtime, From`}: check if the timestamp of the *entry* has changed since we read the value (at timestamp `Readtime`). A reply, tagged with the reference, will tell the *validator* if the timestamp is still the same or if the client has to abort.

- `stop`: terminate.

```
entry(Value, Time) ->
    receive
        {read, Ref, From} ->
            %% TODO: ADD SOME CODE
            entry(Value, Time);
        {write, New} ->
            entry(... , make_ref());  %% TODO: COMPLETE
        {check, Ref, Readtime, From} ->
            if
                 ... == ... ->    %% TODO: COMPLETE
                    %% TODO: ADD SOME CODE
                true ->
                    From ! {Ref, abort}
            end,
            entry(Value, Time);
        stop ->
            ok
    end.
```

Why do we want the reply to the read message to include the process identifier of the *entry*? Not quite clear at the moment but you will see that it makes it easy to write an asynchronous *transaction handler*.

## 2.2 The store

We will hide the representation of the *store* and provide only an API to create a new *store* consisting of N *entries* and to lookup an *entry* in the *store*. Creating a tuple is done simply by first creating a list of all the process identifiers and then turning it into a tuple.

```
-module(store).
-export([new/1, stop/1, lookup/2]).

new(N) ->
    list_to_tuple(entries(N, [])).

stop(Store) ->
    lists:foreach(fun(E) ->
                    E ! stop
                  end,
                  tuple_to_list(Store)).

lookup(I, Store) ->
    element(I, Store). % this is a built-in function

entries(0, ListSoFar) ->
    ListSoFar;
entries(N, ListSoFar) ->
    Entry = entry:new(0),
    entries(N-1, [Entry|ListSoFar]).
```

### 2.3 The transaction handler

A *client* should never access the *store* directly. It will perform all operations through a *transaction handler*. A *transaction handler* is created for a specific *client* and holds the *store* and the process identifier of the *validator*.

We will implement the *handler* so that a *client* can make asynchronous reads to the *store*. If latencies are high, there is no point in waiting for one read operation to complete before initiating a second operation.

The task of the *transaction handler* is to record all read operations (and at what time these took place) and make write operations only visible in a local store. In order to achieve this, the *handler* will keep two sets: the **read set** (`Reads`) and the **write set** (`Writes`). The read set is a list of tuples {`Entry, Time`} and the write set is a list of tuples {`N, Entry, Value`}. When it is time to commit, the *handler* sends the read and write sets to the *validator*.

When the *handler* is created it is also linked to its creator. This means that if one dies both die. This sounds hard but it will be explained once we implement the *server*.

```
-module(handler).
-export([start/3]).

start(Client, Validator, Store) ->
    spawn_link(fun() -> init(Client, Validator, Store) end).

init(Client, Validator, Store) ->
    handler(Client, Validator, Store, [], []).
```

The message interface to the *handler* is as follows:

- {`read, Ref, N`}: a read request from the *client* containing a reference that we should use in the reply message. The integer `N` is the index of the *entry* in the *store*. The *handler* should first look through the write set to see if *entry* `N` has been written. In this case, the written value is returned to the *client*. If no matching operation is found, a message is sent to the Nth *entry* process in the *store*. This *entry* will reply to the *handler* since we need to record the read timestamp.

- {`Ref, Entry, Value, Time`}: a reply from an *entry* that should be forwarded to the *client*. The {`Entry, Time`} tuple is saved in the read set of the *handler*. The reply to the *client* is {`value, Ref, Value`}.

- {`write, N, Value`}: a write request from the *client*. The integer `N` is the index of the *entry* in the *store* and `Value`, the new value. The {`N, Entry, Value`} tuple is saved in the write set of the *handler*.

- {`commit, Ref`}: a commit request from the *client*. We must contact the *validator* and see if there are any conflicts in our read set. If not, the *validator* will perform the write operations in the write set and reply directly to the *client*.

Here is some skeleton code for the *handler*.

```
handler(Client, Validator, Store, Reads, Writes) ->
    receive
        {read, Ref, N} ->
            case lists:keyfind(..., ..., ...) of  %% TODO: COMPLETE
                {N, _, Value} ->
                    %% TODO: ADD SOME CODE
                    handler(Client, Validator, Store, Reads, Writes);
                false ->
                    %% TODO: ADD SOME CODE
                    %% TODO: ADD SOME CODE
                    handler(Client, Validator, Store, Reads, Writes)
            end;
        {Ref, Entry, Value, Time} ->
            %% TODO: ADD SOME CODE HERE AND COMPLETE NEXT LINE
            handler(Client, Validator, Store, [...|Reads], Writes);
        {write, N, Value} ->
            %% TODO: ADD SOME CODE HERE AND COMPLETE NEXT LINE
            Added = lists:keystore(N, 1, ..., {N, ..., ...}),
            handler(Client, Validator, Store, Reads, Added);
        {commit, Ref} ->
            %% TODO: ADD SOME CODE
        abort ->
            ok
    end.
```

## 2.4 The validator

The *validator* is responsible of doing the final validation of transactions. The task is made quite easy since only one transaction is validated at a time. There are no concurrent operations that could possibly conflict with the validation process.

When we start the *validator*, we also link it to the process that creates it. This is to ensure that we don't have any zombie process.

```
-module(validator).
-export([start/0]).

start() ->
    spawn_link(fun() -> init() end).

init()->
    validator().
```

The *validator* receives a request from a *handler* containing everything that is needed both to validate that the transaction is allowed and to perform the write operations that will be a result of the transaction. The request contains:

- `Ref`: a unique reference to tag the reply message.

- `Reads`: the list of read operations that have been performed. The *validator* must ensure that the *entries* of the read operations have not been changed.
- `Writes`: the pending write operations that, if the transaction is valid, should be applied to the *store*.
- `Client`: the process identifier of the *client* to whom we should return the reply.

Validation is thus simply checking if read operations are still valid and if so update the *store* with the pending write operations.

Since a read operation is represented with a tuple {`Entry, Time`}, the *validator* needs only to send a `check` message to the *entry* to make sure that the current timestamp of the *entry* is the same.

```
validator() ->
    receive
        {validate, Ref, Reads, Writes, Client} ->
            Tag = make_ref(),
            send_read_checks(..., Tag),  %% TODO: COMPLETE
            case check_reads(..., Tag) of  %% TODO: COMPLETE
                ok ->
                    update(...),  %% TODO: COMPLETE
                    Client ! {Ref, ok};
                abort ->
                    %% TODO: ADD SOME CODE
            end,
            validator();
        stop ->
            ok;
        _Old ->
            validator()
    end.

update(Writes) ->
    lists:foreach(fun({_, Entry, Value}) ->
                  %% TODO: ADD SOME CODE
                  end,
                  Writes).
```

For better performance the *validator* can first send `check` messages to all the *entries* in the read set and then collect the replies. As soon as one *entry* replies with an `abort` message, we're done. Note however, that we must be careful so that we are not seeing replies that pertain to a previous validation. When we send our check request we therefore tag replies with a unique reference so that we know that we're counting the right replies.

```
send_read_checks(Reads, Tag) ->
    Self = self(),
    lists:foreach(fun({Entry, Time}) ->
                  %% TODO: ADD SOME CODE
```

```
                    end,
                    Reads).
```

Collecting the replies is a simple task and we only have to be careful so that we don't collect an old reply.

```
check_reads(0, _) ->
    ok;
check_reads(N, Tag) ->
    receive
        {Tag, ok} ->
            check_reads(N-1, Tag);
        {Tag, abort} ->
            abort
    end.
```

Old messages that are still in the queue must be removed somehow, this is why, and this is very important, the main loop of the *validator* includes a catch all clause.

## 2.5 The server and the client

We now have all the pieces to build the *transaction server*. The *server* will create the *store* and the *validator* process. *Clients* can then open a transaction and each transaction will be given a new *transaction handler* that is dedicated to the task.

```
-module(server).
-export([start/1]).

start(N) ->
    spawn(fun() -> init(N) end).

init(N) ->
    Store = store:new(N),
    Validator = validator:start(),
    server(Validator, Store).

server(Validator, Store) ->
    receive
        {open, Client} ->
            %% TODO: ADD SOME CODE
            server(Validator, Store);
        stop ->
            Validator ! stop,
            store:stop(Store)
    end.
```

The *server* could simply wait for requests from *clients* and spawn a new *transaction handler*. There is however a trap here that we don't want to fall into. If the *server* creates the *transaction handler*, we must let the *handler* be independent from the *server* (not linked), because if the *handler* dies we don't want the *server* to die. On the other hand, if the *client* dies we do want the *handler* to die. The solution is to let the process of the *client* create the *handler* when it receives the information about the *validator* and the *store* from the *server*. Then, it moves to a loop where it executes transactions until a *stop* message is received.

```
-module(client).
-export([start/5]).

start(ClientID, Entries, Reads, Writes, Server) ->
    spawn(fun() -> open(ClientID, Entries, Reads, Writes, Server, 0, 0) end).

open(ClientID, Entries, Reads, Writes, Server, Total, Ok) ->
    Server ! {open, self()},
    receive
        {stop, From} ->
            io:format("~w: Transactions TOTAL:~w, OK:~w, -> ~w % ~n",
            [ClientID, Total, Ok, 100*Ok/Total]),
            From ! {done, self()},
            ok;
        {transaction, Validator, Store} ->
            Handler = handler:start(self(), Validator, Store),
            case do_transaction(ClientID, Entries, Reads, Writes, Handler) of
                ok ->
                    open(ClientID, Entries, Reads, Writes, Server, Total+1, Ok+1);
                abort ->
                    open(ClientID, Entries, Reads, Writes, Server, Total+1, Ok)
            end
    end.

do_transaction(_, _, 0, 0, Handler) ->
    do_commit(Handler);
do_transaction(ClientID, Entries, 0, Writes, Handler) ->
    do_write(Entries, Handler, ClientID),
    do_transaction(ClientID, Entries, 0, Writes-1, Handler);
do_transaction(ClientID, Entries, Reads, 0, Handler) ->
    do_read(Entries, Handler),
    do_transaction(ClientID, Entries, Reads-1, 0, Handler);
do_transaction(ClientID, Entries, Reads, Writes, Handler) ->
    Op = rand:uniform(),
    if Op >= 0.5 ->
         do_read(Entries, Handler),
         do_transaction(ClientID, Entries, Reads-1, Writes, Handler);
       true ->
         do_write(Entries, Handler, ClientID),
         do_transaction(ClientID, Entries, Reads, Writes-1, Handler)
    end.
```

```
do_read(Entries, Handler) ->
    Ref = make_ref(),
    Num = rand:uniform(Entries),
    Handler ! {read, Ref, Num},
    receive
        {value, Ref, Value} -> Value
    end.

do_write(Entries, Handler, Value) ->
    Num = rand:uniform(Entries),
    Handler ! {write, Num, Value}.

do_commit(Handler) ->
    Ref = make_ref(),
    Handler ! {commit, Ref},
    receive
        {Ref, Value} -> Value
    end.
```

A test module (opty) will help us to set up the experiments.

```
-module(opty).
-export([start/5, stop/1]).

%% Clients: Number of concurrent clients in the system
%% Entries: Number of entries in the store
%% Reads: Number of read operations per transaction
%% Writes: Number of write operations per transaction
%% Time: Duration of the experiment (in secs)

start(Clients, Entries, Reads, Writes, Time) ->
    register(s, server:start(Entries)),
    L = startClients(Clients, [], Entries, Reads, Writes),
io:format("Starting: ~w CLIENTS, ~w ENTRIES, ~w RDxTR, ~w WRxTR, DURATION ~w s~n",
           [Clients, Entries, Reads, Writes, Time]),
    timer:sleep(Time*1000),
    stop(L).

stop(L) ->
    io:format("Stopping...~n"),
    stopClients(L),
    waitClients(L),
    s ! stop,
    io:format("Stopped~n").

startClients(0, L, _, _, _) -> L;
startClients(Clients, L, Entries, Reads, Writes) ->
    Pid = client:start(Clients, Entries, Reads, Writes, s),
    startClients(Clients-1, [Pid|L], Entries, Reads, Writes).
```

```
stopClients([]) ->
    ok;
stopClients([Pid|L]) ->
    Pid ! {stop, self()},
    stopClients(L).

waitClients([]) ->
    ok;
waitClients(L) ->
    receive
        {done, Pid} ->
            waitClients(lists:delete(Pid, L))
    end.
```

# 3 Performance

**Experiments**. To assess the performance of your transaction server, you might use the provided client (updating it when needed), which creates several clients executing several concurrent transactions, to test: (i) different number of concurrent clients in the system, (ii) different number of entries in the store, (iii) different number of read operations per transaction, (iv) different number of write operations per transaction, (v) different ratio of read and write operations for a fixed amount of operations per transaction (including special cases having only read or write operations), (vi) different percentage of accessed entries with respect to the total number of entries (i.e. each client accesses a randomly generated subset of the store: the number of entries in each subset should be the same for all the clients, but the subsets should be different per client and should not contain necessarily contiguous entries[1]). Thus, you will be able to play with these parameters and see when conflicts arise (hint: for each experiment, set the system with a baseline configuration where the other parameters incur few conflicts and modify the studied parameter from that point).

**Open Questions**. (i) What is the impact of each of these parameters on the success rate (i.e. percentage of committed transactions with respect to the total)? (hint: the impact can be better appreciated if represented graphically) (ii) Is the success rate the same for the different clients?

# 4 Distributed execution

**Experiments**. Adapt the `opty` module to ensure that the clients can connect correctly to the server, which must be created in a different remote Erlang instance (named *opty-srv*). Note that the server has to use a locally registered name and there should be a single 'start' function that creates all the processes. Check the slides about Erlang to refresh how processes are created remotely, how names registered in remote nodes are referred, and how Erlang runtime should be started to run distributed programs.

[1] Next code may help to generate random subsets of the store of size S: `lists:sublist([X || {_, X} <- lists:sort([{rand:uniform(), E} || E <- lists:seq(1, Entries)])], S).`

**Open Questions**. If we run this in a distributed Erlang network, where is the handler running?

# 5 Other concurrency control techniques

**Experiments**. Choose ONE of the following activities:

a) Modify the source code to implement **forward validation** instead of backward validation and test how it performs regarding the number of concurrent clients (hint: each entry should keep a list of the ACTIVE transactions that have read it; the validator should consistently check the write sets of the validating transactions against these lists).

b) Compare ALL the previous performance results of the transaction server when using concurrency control with backward validation with respect to **timestamp ordering** (use the *timey* implementation available in the wiki page).