Seminar Report: Opty

Miguel Cruz; Javier Cano; Riccardo Cecco

December 26, 2021

1 Introduction

The seminar is about Opty, the implementation of a transaction server with Optimistic Concurrency Control. Each transaction has three phases: Working, Validation and Update. Multiple transactions can regularly complete without interfering with each other, according to Optimistic Concurrency Control. Transactions consume data resources while executing without collecting locks on those resources. Each transaction checks that no other transaction has affected the data it has read before committing (backwards validation). If the conflicting modifications are discovered, the committing transaction is rolled back and can be reopened.

2 Experiments

2.1 Performance Experiments

The main goal of these experiments is to see how well the algorithm performs with a different number of clients, entries, reads per transaction, writes per transaction and a different ratio of reads and writes .

We set a default value for these experiments, in order to compare and see what happened when we change the parameters.

The default values for most of the experiments are:
Clients: Number of concurrent clients in the system = 4
Entries: Number of entries in the store = 10
Reads: Number of read operations per transaction = 1
Writes: Number of write operations per transaction = 1
Time: Duration of the experiment (in secs) = 1

2.1.1 Clients Variation

In this section, we'll illustrate how client variations influence the performance. We use the list [4,8,12,16,20,24,28], to represent the different number of clients that we tested.
We averaged the success rate of all clients. And now we will present the plot.

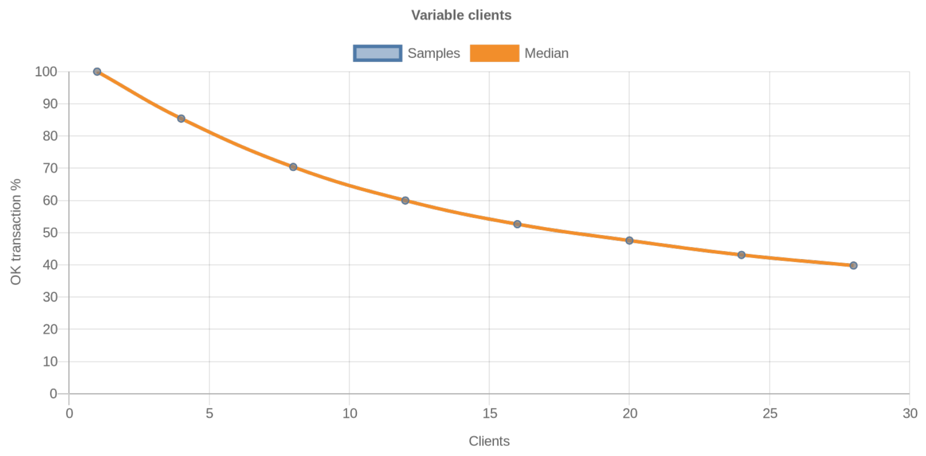


Figure 1: Client variation chart.

It's clear that an increasing number of concurrent clients affects the number of transactions that complete successfully negatively. This is quite intuitive as in most situations, the more actors that participate in a process, the more complicated it becomes to maintain some degree of organisation. In this case, the ratio of successful transactions drops as the actors increase.

2.1.2 Entries Variation

In this part, we'll show how different entries affect performance. We use the list [10,20,30,40,50,100,1000], to represent the different number of entries that we tested.
We averaged the success rate of all clients. And now we will present the plot.

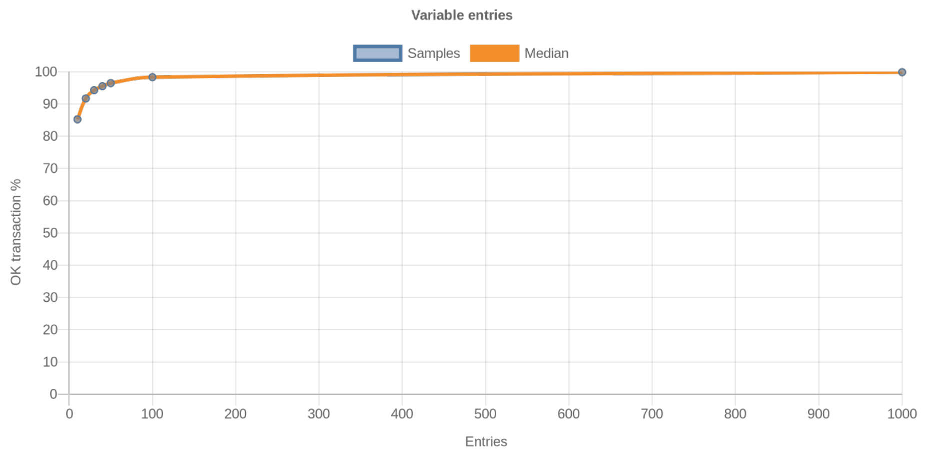


Figure 2: Entries variation chart.

We can observe that an increase in the number of entries has a direct impact on the number of transactions that are completed successfully. We believe this is connected to the number of reads/writes operations. This indicates that as the number of entries grows, the success rate tends to reach 100% (for read/write operations with a constant value).

2.1.3 RDxTR Variation

We'll look at how different numbers of reads per transaction effect performance in this section. We use this list [1,2,3,4,5,6,7], to represent the different number of reads per transaction that we tested.
We averaged the success rate of all clients. And now we will present the plot.

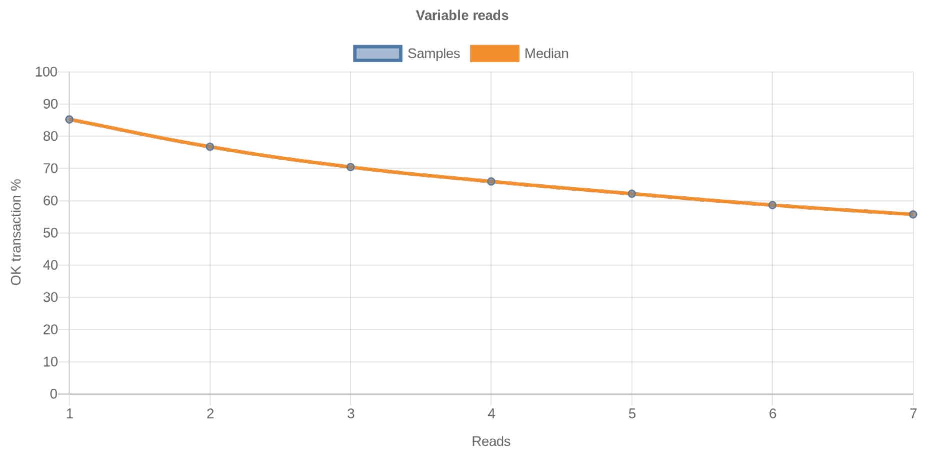


Figure 3: RDxTR variation chart.

The number of reads per transaction affects the ratio of successful transactions negatively, as the more reads a transaction does, the more chances for a read to conflict with another operation's writes. As a result, as the number of reads per transaction increases, the success rate tends to drop.

2.1.4 WRxTR Variation

We use this list [1,2,3,4,5,6,7], to represent the different number of writes per transaction that we tested.

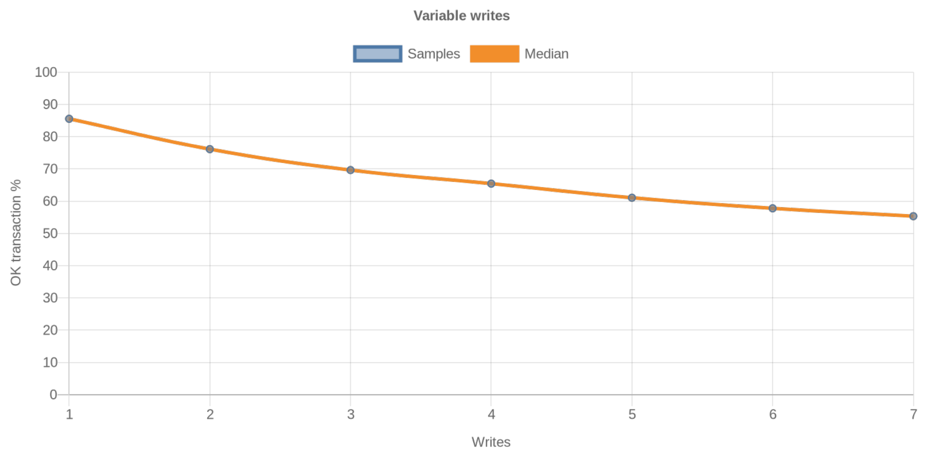


Figure 4: WRxTR variation chart.

Similarly to the number of reads variation, the number of writes also affects the ratio of successful transactions negatively, as the more writes a transaction performs, the more chances for a write to conflict with a read. After that, we may state that as the number of writes per transaction increases, the success rate decreases.

2.1.5 Variable ratio of reads and writes

For this experiment, we will define a fixed number of operations (reads+writes) per transaction, and will vary the ratio of reads vs writes. In the following plot, we represent increments in steps of 10% for the reads, thus, when reads are at per example 20%, writes are at 100-20 = 80%.

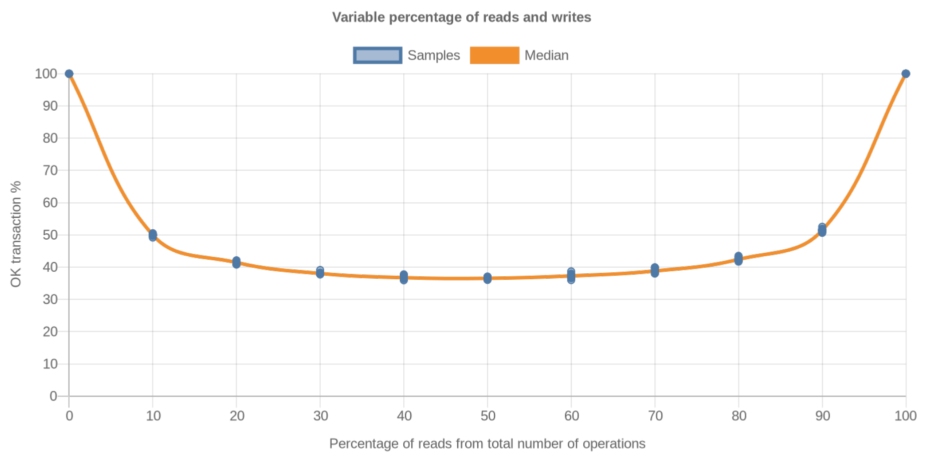


Figure 5: Varying ratio of operations chart.

This is a very interesting result. First, we see that whenever one of the operations is at 0%, every transaction is completed successfully. This was predictable, as writes are checked against reads in the validation process. When one of the operations isn't done at all, every transaction should succeed.

Then, one thing that catches our eye instantly is that the data seems to be almost symmetrical. The point where it gets the lowest transactions successfully processed is right at the middle, 50% reads and 50% writes.

Overall, it seems that the most difference there is in ratio of reads vs ratio of writes, the highest the ratio of successful transactions we get.

2.1.6 VI Experiment

The goal of this experiment is to set a different percentage of accessed entries with respect to the total number of entries and see how it works.

First of all, we modified the code of the Client and added a new entry parameter called Percentage. This variable represents the percentage of the accessed entries.

```
-export([start/6]).

start(ClientID, Entries, Reads, Writes, Server, Percentage) ->
```

Then, we calculated how many entries it must access and modified the spawn call with the new parameters.

```
    NumberRandom = round(Entries * Percentage),
    RandomEntries = lists:sublist([X || {_ , X} <- lists:sort([{
    rand:uniform(), E} || E <- lists:seq(1, Entries)])],
    NumberRandom),
    spawn(fun() -> open(ClientID, RandomEntries, Reads, Writes,
    Server, 0, 0) end).
```

Finally, we modified the code into the Opty file to be able to insert the new parameter.

```
start(Clients, Entries, Reads, Writes, Time, Percentage) ->
    register(s, server:start(Entries)),
    L = startClients(Clients, [], Entries, Reads, Writes,
    Percentage),
```

Now, the starting function has a new parameter : **opty:start(*,*,*,*,*,x).** where the parameter called x is the percentage set.

We did some tests with different parameters and the execution seems to work well.

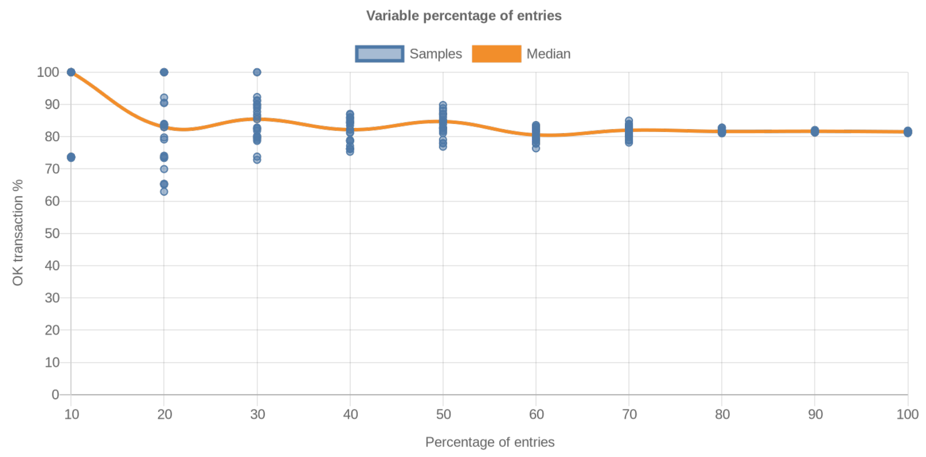


Figure 6: Different percentage set

We can see that the resulting samples, with the decrease of the percentage, deviates more and more from the result obtained without the percentage variable.

2.2 Distributed execution

The objective of this experiment is to run the same system as before (using the opty module) but with the server and client running in different erlang instances.

In this case, the server will run on a remote instance while the client will run in the same instance as opty.

First of all, we need to run two separate erlang instances with different names:

- **erl -name opty@127.0.0.1** (instance where opty and the clients run)
- **erl -name opty-srv@127.0.0.1** (instance for the server)

If we take a look at the opty module, we can see that the server start function is being wrapped in a register function. This means that the node that runs this register function will have a local name for the server process. This experiment requires for this name to be registered on the server node, so we will wrap this call in another function and spawn it remotely in the server node.

The wrapper function:

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

startServer(Entries) ->
    register(s, server:start(Entries)).
```

The spawn call:

```
start(Clients, Entries, Reads, Writes, Time) ->
    spawn(?server_node, opty, startServer, [Entries]),
    //rest of the code
```

Finally, we have to make sure that the created clients know where the server is. To do this, we replace the Server parameter in the client:start calls from a name to a tuple containing the name and the remote node.

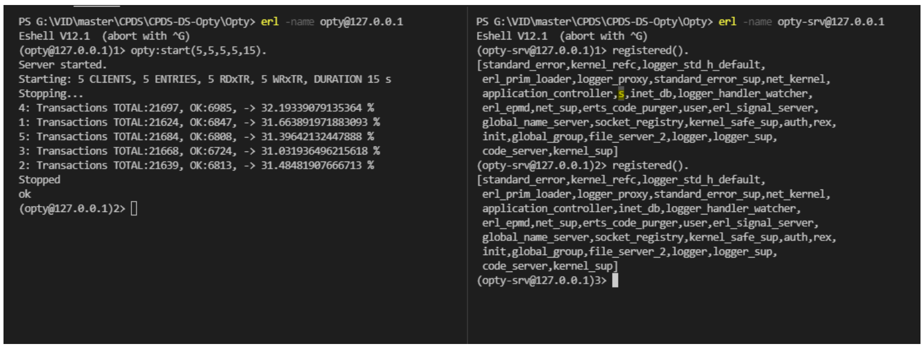
```
Pid = client:start(Clients, Entries, Reads, Writes, {s, ?
    server_node}),
```

At first, we thought that these changes to the code would work right away, but the program was behaving in an erratic way. After some investigation, we found out that sometimes the clients were being created earlier than the server, thereby trying to send messages to an unregistered process. This happened because after calling server:start in another node, the execution of that code would stop being synchronous to opty, resulting in opty continuing his normal procedure without knowing for sure if the server process had been actually created. To solve this, we thought that a quick solution would be to add a brief sleep time in opty before starting the clients. In our tests, 1 second was enough to ensure a correct execution.

In the end, the code of every changed function in opty ended up being the following:

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

-define(server_node, 'opty-srv@127.0.0.1').

start(Clients, Entries, Reads, Writes, Time) ->
    spawn(?server_node, opty, startServer, [Entries]),
    timer:sleep(1000), %wait for server to be created on remote node
    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, ?server_node} ! stop,
    io:format("Stopped~n").

startServer(Entries) ->
    register(s, server:start(Entries)).

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

To provide some proof of it working, here we have an image of two terminals with both erlang instances. The first one executes opty, and while it's running, we check the registered names on the second instance. We can see how ”s”, the server process name is registered. After opty stops, we also see how it is correctly unregistered. The execution seems to work well.

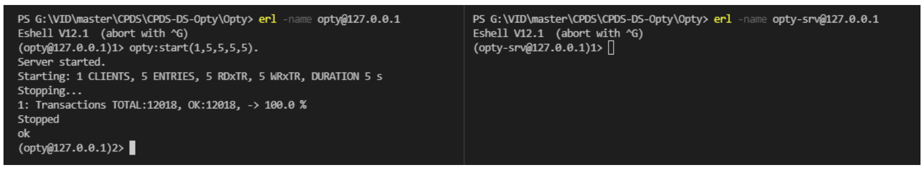


```
PS G:\VID\master\CPDS\CPDS-DS-Opty\Opty> erl -name opty@127.0.0.1
Eshell V12.1  (abort with ^G)
(opty@127.0.0.1)1> opty:start(5,5,5,5,15).
Server started.
Starting: 5 CLIENTS, 5 ENTRIES, 5 RDxTR, 5 WRxTR, DURATION 15 s
Stopping...
4: Transactions TOTAL:21697, OK:6985, -> 32.19339079135364 %
1: Transactions TOTAL:21624, OK:6847, -> 31.663891971883093 %
5: Transactions TOTAL:21684, OK:6808, -> 31.39642132447888 %
3: Transactions TOTAL:21668, OK:6724, -> 31.031936496215618 %
2: Transactions TOTAL:21639, OK:6813, -> 31.48481907666713 %
Stopped
ok
(opty@127.0.0.1)2>
```

```
PS G:\VID\master\CPDS\CPDS-DS-Opty\Opty> erl -name opty-srv@127.0.0.1
Eshell V12.1  (abort with ^G)
(opty-srv@127.0.0.1)1> registered().
[standard_error,kernel_refc,logger_std_h_default,
 erl_prim_loader,logger_proxy,standard_error_sup,net_kernel,
 application_controller,s,inet_db,logger_handler_watcher,
 erl_epmd,net_sup,erts_code_purger,user,erl_signal_server,
 global_name_server,socket_registry,kernel_safe_sup,auth,rex,
 init,global_group,file_server_2,logger,logger_sup,
 code_server,kernel_sup]
(opty-srv@127.0.0.1)2> registered().
[standard_error,kernel_refc,logger_std_h_default,
 erl_prim_loader,logger_proxy,standard_error_sup,net_kernel,
 application_controller,inet_db,logger_handler_watcher,
 erl_epmd,net_sup,erts_code_purger,user,erl_signal_server,
 global_name_server,socket_registry,kernel_safe_sup,auth,rex,
 init,global_group,file_server_2,logger,logger_sup,
 code_server,kernel_sup]
(opty-srv@127.0.0.1)3>
```

Figure 7: Server process registered in a remote node.

To double check, we execute opty with only one client, which should report a 100 % transaction success:



```
PS G:\VID\master\CPDS\CPDS-DS-Opty\Opty> erl -name opty@127.0.0.1
Eshell V12.1  (abort with ^G)
(opty@127.0.0.1)1> opty:start(1,5,5,5,5).
Server started.
Starting: 1 CLIENTS, 5 ENTRIES, 5 RDxTR, 5 WRxTR, DURATION 5 s
Stopping...
1: Transactions TOTAL:12018, OK:12018, -> 100.0 %
Stopped
ok
(opty@127.0.0.1)2>
```

```
PS G:\VID\master\CPDS\CPDS-DS-Opty\Opty> erl -name opty-srv@127.0.0.1
Eshell V12.1  (abort with ^G)
(opty-srv@127.0.0.1)1>
```

Figure 8: Distributed execution working properly with 1 client.

In conclusion, in this version of the program, we get the following system:

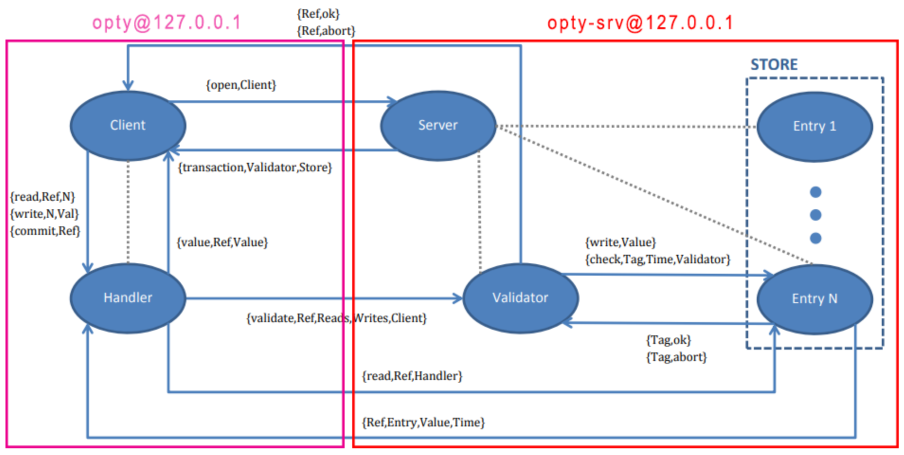


Figure 9: Diagram of the distributed system.

2.3 Forward validation

Our choice for another concurrency control technique has been forward validation. With this technique, we will keep track of which active transactions have read each entry, and in the validation process, we will check the write sets against the reads of the other active transactions. A write will create a conflict if another transaction has read the value of the same entry.

As we know, in a forward validation scheme, we have many options of what to do in the case of a validation failing. In our case, we chose to abort the current transaction, as it's the easiest option to implement. As there are a fair amount of changes to the code, we will comment the changes to each module separately.

First of all, the entry module:

```
-module(entry).
```

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

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

init(Value) ->
    entry(Value, []).

entry(Value, Active) ->
    receive
        {read, Ref, From, Client} ->
            From ! {Ref, self() ,Value},
            case lists:member(Client, Active) of
                false -> entry(Value,[Client|Active]);
                true -> entry(Value, Active)
            end;
        {write, New} ->
            entry(New, Active);
        {check, Ref, From, Client} ->
            EmptyList = length(lists:delete(Client,Active)) == 0,
            if
                 EmptyList ->
                        From ! {Ref, ok};
                 true ->
                        From ! {Ref, abort}
            end,
            entry(Value, Active);
        {deactivate, From, Client} ->
            From ! ok,
            entry(Value, lists:delete(Client, Active));
        stop ->
            ok
    end.
```

In this version, we keep a list of active transactions instead of a timestamp. When a read is performed, the transaction is added to the list, making sure we don't introduce a duplicate. Some of the messages now require the pid of the client to be sent, so it can be added to the list.

When a check message is received, it only returns "ok" if the list of active transactions doesn't contain any transaction other than the current one. A deactivate message is also introduced so that a transaction can delete itself from the active list when it finishes.

Next, is the handler module:

```
-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, [], []).

handler(Client, Validator, Store, Reads, Writes) ->
    receive
        {read, Ref, N} ->
```

```
            case lists:keyfind(N, 1, Writes) of
                {N, _, Value} ->
                    Client ! {value, Ref, Value},
                    handler(Client, Validator, Store, Reads, Writes);
                false ->
                    store:lookup(N,Store) ! {read, Ref, self(), Client},
                    handler(Client, Validator, Store, Reads, Writes)
            end;
        {write, N, Value} ->
            Added = lists:keystore(N, 1, Writes, {N, store:lookup(N,Store), Value}),
            handler(Client, Validator, Store, Reads, Added);
        {Ref, Entry, Value} ->
            Client ! {value, Ref, Value},
            handler(Client, Validator, Store, [{Entry}|Reads], Writes);
        {commit, Ref} ->
            Validator ! {validate, Ref, Reads, Writes, Client};
        abort ->
            ok
    end.
```

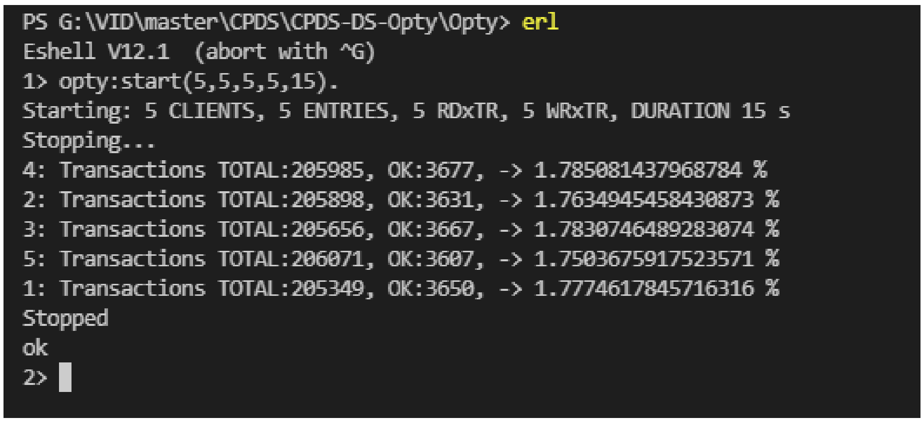
In this module, the changes made are only to conform to the new parameters of the entry module. It's interesting to mention that now we listen to a message that has 3 undefined variables. Because of this, this message has to be written after the other 3 parameter messages. If it was introduced earlier than these in the code, it would always pattern match and no read or write messages would be processed correctly.

Finally, the validator module:

```
-module(validator).
-export([start/0, update/1]).

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

init()->
    validator().

validator() ->
    receive
        {validate, Ref, Reads, Writes, Client} ->
            Tag = make_ref(),
            send_write_checks(Writes, Tag, Client),
            case check_writes(length(Writes), Tag) of
                ok ->
                    update(Writes),
                    deactivate_before_ending(Reads,Client),
                    Client ! {Ref, ok};
                abort ->
                    deactivate_before_ending(Reads,Client),
                    Client ! {Ref, abort}
            end,

            validator();
        stop ->
            ok;
        _Old ->
            validator()
    end.

update(Writes) ->
    lists:foreach(fun({_, Entry, Value}) ->
                    Entry ! {write,Value}
                  end,
                  Writes).

send_write_checks(Writes, Tag, Client) ->
    Self = self(),
    lists:foreach(fun({_, Entry, _}) ->
                  Entry ! {check, Tag, Self, Client}
                  end,
                  Writes).

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

deactivate_before_ending(Reads, Client) ->
    send_reads_deactivations(Reads, Client),
    wait_for_deactivations_confirmation(length(Reads)).

send_reads_deactivations(Reads, Client) ->
    Self = self(),
    lists:foreach(fun({Entry}) ->
                  %% TODO: ADD SOME CODE
                  Entry ! {deactivate, Self, Client}
                  end,
                  Reads).

wait_for_deactivations_confirmation(0) ->
    ok;
wait_for_deactivations_confirmation(N) ->
    receive
        ok -> wait_for_deactivations_confirmation(N-1)
    end.
```

The read checks have been replaced with write checks. These messages conform with the new check interface in the entry module. Also, some helper functions have been introduced to deregister the transaction from the entries' active lists. Of course, this process of deactivation has to be made either if the transaction aborts or commits.

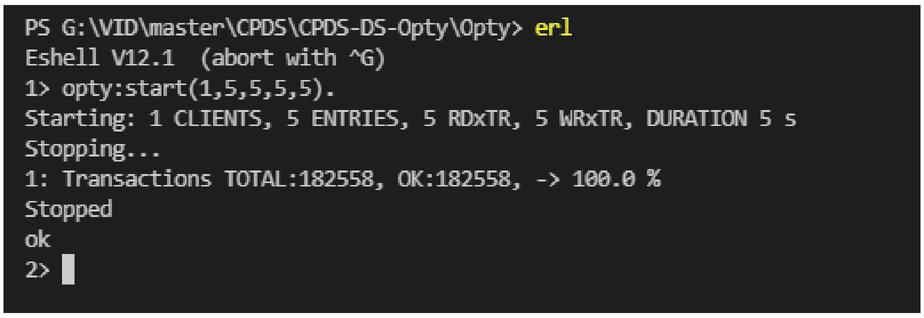
Here is an execution of this version. It reports in general more aborted transactions that the backwards validation version.



```
PS G:\VID\master\CPDS\CPDS-DS-Opty\Opty> erl
Eshell V12.1  (abort with ^G)
1> opty:start(5,5,5,5,15).
Starting: 5 CLIENTS, 5 ENTRIES, 5 RDxTR, 5 WRxTR, DURATION 15 s
Stopping...
4: Transactions TOTAL:205985, OK:3677, -> 1.785081437968784 %
2: Transactions TOTAL:205898, OK:3631, -> 1.7634945458430873 %
3: Transactions TOTAL:205656, OK:3667, -> 1.7830746489283074 %
5: Transactions TOTAL:206071, OK:3607, -> 1.7503675917523571 %
1: Transactions TOTAL:205349, OK:3650, -> 1.7774617845716316 %
Stopped
ok
2>
```

Figure 10: Example execution with forward validation.

And another execution, checking that when there's only one client, all transactions must commit.



```
PS G:\VID\master\CPDS\CPDS-DS-Opty\Opty> erl
Eshell V12.1  (abort with ^G)
1> opty:start(1,5,5,5,5).
Starting: 1 CLIENTS, 5 ENTRIES, 5 RDxTR, 5 WRxTR, DURATION 5 s
Stopping...
1: Transactions TOTAL:182558, OK:182558, -> 100.0 %
Stopped
ok
2>
```

Figure 11: Only 1 client with forward validation.

3 Open questions

Q1) What is the impact of each of these parameters on the success rate?

We explained each of these separately on section 2.1 of this document. To sum it up, we found out that increasing the number of clients, writes and reads, seems to make the number of transactions that complete successfully drop almost linearly. Contrary to this, an increasing number of entries seems to affect the ratio of successful transactions positively.

The varying ratio of reads vs writes gives an almost symmetrical plot, which indicates that the closer the ratio, the worst ratio of successful transactions we get.

For the sixth experiment, making smaller subsets of entries seems to affect the ratio of successful transactions somewhat positively, but samples seem to deviate more. This is due to the randomness that comes from generating these random subsets per client. In the most extreme case, a client can get a subset of entries that are not shared by any other client, making his transactions to always complete successfully. As the subset of entries gets smaller, this scenario is more likely to happen.

Q2) Is the success rate the same for the different clients?

No, the success rate varies from client to client. They are, nevertheless, values that are quite near to one other. The only instance where we found a noticeable deviation has been, as stated in the former answer, in experiment 6, by assigning subsets of entries, thus creating a somewhat more random execution.

Q3) If we run this in a distributed Erlang network, where is the handler running?

As we showed in figure 9, the handler runs alongside the client. This is because in this system, the handler is created by the client after the server has passed references to the store and validator to it. The reason for the client creating the handler is, as written in the assignment statement, to make it so that if the client dies, the handler does too.

4 Personal opinion

We found that it was very useful to implement a transaction server with an Optimistic Concurrency Control algorithm by ourselves, because by developing the project we were able to better understand the process and its functioning.

As this is a structure with quite a few number of different elements, at first it looked a bit overwhelming, but as we followed the statement, it wasn't that hard to understand the role that every module plays.

It was a pleasure, and to work better together we created a common git repository where each of us could develop the code and try new experiments. The resulting branches will all be included in separate directories in the final product.

It was a good experience and in our opinion the seminar should be proposed again next year.