Opty: optimistic concurrency control

Johan Montelius

October 2, 2016

Introduction

In this session you will implement a transaction server using optimistic concurrency control. 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 with backwards validation works.

1 The architecture

The architecture consist of a server having access to a store and a validator. The store consist 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 process. 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 process.

1.1 the handler

The transaction handler will handle read and write requests from the client and will also to 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 to be closed the handler send the read and write sets to the validator.

1.2 the validator

The validation process will be able to tell if a value of an entry has been changes since the transaction read the entry. If non of the entries have changed the transacting can commit and the associated write operations are performed.

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

2 The Implementation

Since data structures in Erlang are not mutable (we can not 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 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 who 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 "time stamp" 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, Handler`}: a read request from a handler tagged with a reference. We will return a message tagged with the reference so that the handler can identify the correct message. The reply will contain the process identifier of the entry, the value and the current time stamp.
- {`write, Value`}: changes the current value of the entry, no reply needed. The time stamp of the entry will be updated.

- {`check, Ref, Read, Handler`}: check if the time stamp of the entry has changed since we read the value (at time `Read`). A reply, tagged with the reference, will tell the handler if the time stamp is still the same or if it has to abort.

- `stop`: terminate.

We here talk about *the handler* and not *the client*, this will be clear later. Why do we want the read message to include the process identifier? Not quite clear at the moment but you will see that it makes it easy to write an asynchronous transaction handler.

```
entry(Value, Time) ->
    receive
        {read, Ref, Handler} ->
            Handler ! {Ref, self(), Value, Time},
            entry(Value, Time);
        {check, Ref, Read, Handler} ->
            if
                Read == Time ->
                    Handler ! {Ref, ok};
                true ->
                    Handler ! {Ref, abort}
            end,
            entry(Value, Time);
        {write, New} ->
            entry(New, make_ref());
        stop ->
            ok
    end.
```

2.2 the store

We will hide the representation of the store and provide only an API to create a new store and to look-up an entry in the store. Creating a tuple is done simply by first creating a list of all 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:map(fun(E) -> E ! stop end, tuple_to_list(Store)).

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

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

2.3 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 process.

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. 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 time.

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

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

- {`commit, Ref`}: a commit message from the client. This is the time to 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:keysearch(N, 1, Writes) of
                {value, {N, _, Value}} ->
                        :
                    handler(Client, Validator, Store, Reads, Writes);
                false ->
                        :
                        :
                    handler(Client, Validator, Store, Reads, Writes)
            end;

        {Ref, Entry, Value, Time} ->
               :
            handler(Client, Validator, Store, [...|Reads], Writes);

        {write, N, Value} ->
            Added = [{N, ..., ...}|...],
            handler(Client, Validator, Store, Reads, Added);

        {commit, Ref} ->
            Validator ! {validate, Ref, Reads, Writes, Client};
```

```
        abort ->
            ok
    end.
```

2.4 validation

The validation handler 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 processes that creates it. This is to ensure that we don't have any zombie processes.

```
-module(validator).

-export([start/0]).

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

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

The validator receives a request from a client 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: a 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.

```
validator() ->
    receive
```

```
        {validate, Ref, Reads, Writes, Client} ->
            case  validate(Reads) of
                ok ->
                    update(Writes),
                    Client ! {Ref, ok};
                abort ->
                    Client ! {Ref, abort}
            end,
            validator();
         _Old ->
            validator()
    end.
```

Since a read operation is represented with a tuple {`Entry, Time`} the validator need only send a `check` message to the entry and make sure that the current time-stamp of the entry is the same.

```
validate(Reads) ->
    {N, Tag} = send_checks(Reads),
    check_reads(N, tag).
```

For better performance the validator can first send check messages to all entries 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 them with a unique reference so that we know that we're counting the right replies.

```
send_checks(Reads) ->
    Tag = make_ref(),
    Self = self(),
    N = length(Reads),
    lists:map(fun({Entry, Time}) ->
                      Entry ! {check, Tag, Time, Self}
              end,
              Reads),
    {N, Tag}.
```

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(N, Tag) ->
    if
        N == 0 ->
            ok;
```

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

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

```
-module(server).

-export([start/1, open/1, stop/1]).

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

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

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). 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.

```
open(Server) ->
    Server ! {open, self()},
    receive
        {transaction, Validator, Store} ->
            handler:start(self(), Validator, Store)
    end.
```

This will also have implications on where the transaction handler is running. Something that we might discuss further when we are done with the server.

The server now becomes almost trivial.

```
server(Validator, Store) ->
    receive
        {open, Client} ->
             :
            server(Validator, Store);
        stop ->
             store:stop(Store)
    end.
```

You're done, you have implemented a transaction server using optimistic concurrency control.

3 Performance

Does it perform? How many transaction can we do per second? What are the limitations on the number of concurrent transactions and the success rate? This does of course depend on the size of the store, how many write operations each transaction does and how much delay we have in between the read instructions of the transaction and the final commit instruction.

Some Erlang related questions can be fair to raise. Is it realistic with the store implementation that we have? Regardless of the transaction handler, how fast can we operate on the store? What happens if we run this in a distributed Erlang network, what is actually copied when a transaction handler is started? Where is the handler running? Pros and cons of this implementation strategy?