Timey: time based concurrency control

Johan Montelius

October 2, 2016

Introduction

In this session you will implement a transaction server using time based 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 time based concurrency control.

1 The architecture

The architecture consist of a server having access to a store. The store consist of a set of entries, each holding a committed value and a list of suspended read and write operations. An entry is responsible for the concurrency control and will, as we will see, have quite complex state transitions.

A client starts a transaction by requesting a transaction handler from the server. The transaction handler is given a unique time stamp when created and will keep track of all write operations performed during the transaction. When the client request the transaction to be committed the handler makes sure that all write operation are also committed.

1.1 an entry

An entry is, as you will see, the most complex part. We will describe it gradually, adding requirements as we realize the important role it serves. To start, the entry has four values:

- **committed**: the value that has been committed.
- **written**: the time this value was committed.
- **read**: the last time (highest time stamp) this value was read.
- **susp**: an ordered list of suspended read and write operations.

We will have four messages that the entry must be able to handle. All messages are from a transaction handler. This is a short description of the protocol.

- **Read**: a read request containing a time stamp. The handler should be given a committed **value** when available. If the request is too late the reply is a **abort** message.
- **Write**: a write request containing a time stamp The handler is sent a `ok` or `abort` message.
- **Commit**: a commit request specifying a earlier write request. The write request, if not previously aborted, is promoted to a committed value. No reply is sent back to the handler.
- **Abort**: am abort request specifying a earlier write request. The write request, if not previously aborted, is aborted. No reply is sent back to the handler.

In the implementation we will of course see how to connect requests to replies and how to keep track of which commit and abort request belong to which write operations but we will go through the general idea first.

1.2 a read request

We will start by looking at the read request and solve the easy case. When a read request is received by an entry it will check if the time of the read request is later than the currently committed value. If not it means that the read operation came too late and an abort reply is sent to the handler.

If the request was not to late we must check the suspension list. It could be that there is a suspended write operation, not yet committed, that has a time stamp value that is earlier than the read operation. If this is the case the read operation must be inserted behind the write operation in the suspension lists. A special case occur if we find a write operation with the same time stamp. This would mean that a client first writes a value to an entry and then reads the same entry. The client should of course read its writes so we must return the written value even if this is only a tentative value.

If the suspension list is empty or only contains suspended write operations that have time stamps later than the read operation we can safely read the current value. We then send a reply to the handler and also update the read time of the committed value.

1.3 a write request

A write request is almost simpler. We must first examine the read time of the current value. If the read time is later than the time of the write operation there is not much to do, we have to abort. Assuming that this is not the case; examine the write time of the current value. If the write time is later than the time of the write operation we could simply forget that this operation was ever done. If a value has been written (and committed) at time 14 but no one has read the value since time 10, then what difference can a write operation at time 12 do? We can simply report to the handler that the operation has been handled.

If the write operation is later than the current write time we must do some bookkeeping. We can of course not replace the current value straight away, instead we must suspend the operation as a tentative write operation. The write operation is thus inserted into the list of suspended operations. The handler is of course informed that the operation has been correctly handled.

1.4 a commit request

Now for the tricky part; what should we do when a write operation later is committed? If a write operation to be committed is the first operation in the suspension list we should remove it from the list and promote the tentative value to current value of the entry.

If the write operation was followed by a sequence of suspended read operations these can now be scheduled and the committed value is sent back as a reply. This will of course also update the read time of the committed value.

Now assume that the first operation in the list is another write operation and the operation to be committed is somewhere down the list. We should promote the value to become a committed value but the operation is still in the list of suspended operation. The list would then contain tentative write, committed writes and suspended read operations.

Having committed values in the suspension list changes how a write operation is committed. If the write operation to be committed is the first element in the list and all following read operations have been removed, it could be that the next element is a committed value. This operation must now be promoted to the current value of the entry; a process that might trigger even more read operations.

1.5 the handler

The role of the handler becomes quite simple since most of the concurrency control is handled by the entries. The handler need only keep track of a time stamp and attache this to all read and write operations that it sends to the entries. Read request and read replies are thus very simple to handle. Write requests have to be stored so that the handler can schedule commit requests when the client wants to commit. The handler should also keep track of which write request that have been successfully handled. If any write request has to abort the whole transaction must abort.

It's important to understand that the commit request that the transaction is sending to all entries that have been involved in write requests does not return any message. The handler knows that the write operations have succeeded if they all return an ok message when issued. The ok message means that the request is added as an tentative write operation and nothing can make this operation fail. The commit request will promote the value to a committed value but nothing can prevent this from happening.

1.6 a client

The client is happily unaware of what is going on. It will contact the server and request a new transaction handler. It then communicates with the transaction handler and sends read and write request. When it is done it will send a commit request and wait for a result.

The client does not receive a message for each write request. The handler will check that all write requests are handled properly and only report on the transaction as a whole.

2 The Implementation

Let's start with the most complicated part, the implementation of the entry process.

2.1 the entry

The process will have a state consisting of: the current value, its write time, last time it was read and the list of suspended operations. Let's call the module `entry` and prepare to export the procedures we need and also some for testing that our implementation works.

```
-module(entry).

-export([new/1]).
-export([suspend_read/4, suspend_write/4, suspend_commit/2, abort_write/2]).

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

init(Value) ->
    entry(Value, 0, 0, []).

entry(Value, Written, Read, Susp) ->
    receive
      {read, Ref, Time, Handler}  ->
```

```
           :

      {write, Ref, Time, Tentative, Handler} ->
           :

      {commit, Ref} ->
           :

      {abort, Ref} ->
           :

      stop ->
           ok
    end.
```

The entry procedure will go into a loop and wait for request from any transaction handler. Request from the transaction handler will be tagged with unique references so that the handler can properly identify the reply from the entry. The handler will operate asynchronously and might have several outstanding requests.

2.1.1 a read request

The first message to handle is a read request. The request contains a unique reference, the time stamp of the request and the process identifier of the handler so that we can reply. We will make use of a supporting function called `suspend_read/4`. This function will, if necessary, insert a read operation in the list of suspended operation and then return the updated list. If however, there is no tentative write operation in the list with lower time stamp then the current value of the entry is read.

```
{read, Ref, Time, Handler}  ->
    if
        (Time >= Written) ->
            case suspend_read(Time, Ref, Handler, Susp) of
                no ->
                    Handler ! {reply, Ref, {ok, Value}},
                    entry(Value, Written, Time, Susp);
                Susp2 ->
                    entry(Value, Written, Read, Susp2)
            end;
        true ->
            Handler ! {reply, Ref, abort},
            entry(Value, Written, Read, Susp)
    end;
```

2.1.2 a write request

The write request is equally simple; the request consist of a unique reference, a time stamp, the tentative value and a process identifier of the handler. We first check that the request is not too late i.e. that a read operation has read the value at a "later" time.

```
{write, Ref, Time, Tentative, Handler} ->
    if
        Time >= Read ->
            if
                (Time >= Written)  ->
                    Susp2 = suspend_write(Time, Ref, Tentative, Susp),
                    Handler ! {ok, Ref},
                    entry(Value, Written, Read, Susp2);
                true ->
                    Handler ! {ok, Ref},
                    entry(Value, Written, Read, Susp)
            end;
        true ->
            Handler ! {abort, Ref},
            entry(Value, Time, Read, Susp)
    end;
```

We make use of a supporting function called `suspend_write/4` that will insert the write operation at the right position in the suspension list. As opposed to the insertion of a read operation, the write operation will always be inserted the list. The suspended write request contains the unique reference so that we can identify it later when the handler wants to commit.

When we report back to the handler we of course tag the reply with the unique reference so that the handler can match the request to the reply.

Note that the write request will always result in a reply directly. We know if the operation was too late or arrived in time. For the read request we only replied directly if it was possible to read the committed value. If not, the reply comes when a commit request triggers the suspended read operation to be scheduled.

2.1.3 a commit request

The commit request contains a reference that identifies a previous issued write request. As we discussed there are two cases that we must take care of; either it is the first write operation in the list or it is somewhere down the list (or not in the list at all).

A suspended write operation is represented with a tuple containing the reference so it is easy to identify it. If it is the first write operation that should be promoted we should of course also schedule all suspended read operations (and possibly also suspended commit operations). We could have made use of a function also in this case but it turns out to be more practical to call a procedure `commit/4` that when it has done its job will call the `entry/4` procedure with the right state.

```
{commit, Ref} ->
    case Susp of
        [{Time, {write, Ref, Tentative}}|Susp2] ->
            commit(Tentative, Time, Read, Susp2);
        _ ->
            Susp2 = suspend_commit(Ref, Susp),
            entry(Value, Written, Read, Susp2)
    end;
```

If it is not the most recent write operation that should be committed we promote a suspended write operation somewhere down the list. For this we use the function `suspend_commit/2` that will simply transform a write operation to a commit operation.

2.1.4 a abort request

The abort request is trivial. The request contains a reference that identifies a write request that should be removed.

```
{abort, Ref} ->
    Susp2 = abort_write(Ref, Susp),
    entry(Value, Written, Read, Susp2);
```

2.1.5 committing

The procedure `commit/4` is called when we have received a commit request and the first suspended write operation in the list was identified as the operation to commit. We call the procedure with the committed value, the time this was written, the last time the entry was read and the list of suspended operations.

Before we continue we must schedule and remove and suspended read operations. If there are no operations in the list we can of course call the `entry/4` procedure directly with the parameters we have at hand. If the first operation is a read operation we send a reply with the current value to the handler that issued the request. When we continue down the list we update our read time stamp.

If the first operation is a committed value it means that this value is now the current value so we should continue to remove read operations but now with the committed value as the current value. We also update the write time stamp.

```
commit(Value, Written, Read, Susp) ->
    case Susp of
        [] ->
            entry(Value, Written, Read, []);
        [{R, {read, Rf, Handler}}|Susp2] ->
            Handler ! {reply, Rf, {ok, Value}},
            commit(Value, Written, R, Susp2);
        [{W, {committed, Committed}}|Susp2] ->
            commit(Committed, W, Read, Susp2);
        _ ->
            entry(Value, Written, Read, Susp)
    end.
```

If the first operation is not a read nor a commit we have done our job and continue by calling the `entry/4` procedure.

2.1.6 supporting functions

The only thing that remains is writing the supporting functions. The list of suspended operations is represented using the following tuples:

- {`Time`, {`read`, `Reference`, `Handler`}}: a read operation at `Time` with a unique `Reference` issued by a `Handler`.
- {`Time`, {`write`, `Reference`, `Tentative`}}: a write operation at `Time` with a unique `Reference` and a `Tentative` value.

  {`Time`, {`committed`, `Committed`}}: a committed write operation at `Time` with a `Committed` value.

The description of the supporting functions is as follows:

- `suspend_read(Time, Ref, Handler, Susp)`: if there is a write operation in `Susp` that is earlier than `Time` then insert the read operation {`read`, `Time`, `Handler`} in the list. If there is a write operation at `Time` then reply to the `Handler` with the value of the write operation and return `Susp`. If neither holds true, return `no`.
- `suspend_write(Time, Ref, Tentative, Susp)`: insert the write operation {`Time`, {`write`, `Ref`, `Tentative`}} in `Susp`.
- `suspend_commit(Ref, Susp)`: if there exist a write operation {`Time`, {`write`, `Ref`, `_`}} in `Susp` then replace it with {`Time`, {`committed` `Tentative`}}.
- `abort_write(Ref, Susp)`: if there exist a write operation {`Time`, {`write`, `Ref`, `_`}} in `Susp` then remove it.

The implementation is left as an exercise.

2.2 the store

The store is simple a tuple of process identifiers of entries to which the handler can send read and write requests. We create it by first constructing a list of all the process identifiers and then turning this list into a tuple. The handler, that will access the store, need only use the interface `lookup/2` to get the right process identifier.

```
-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 built-in

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

2.3 the handler

The transaction handler will be started by the client and is given a unique identifier, a unique time stamp and a store. The identifier is used to report the result of the transaction back to the client. The identifier allows a client to have more than one transaction open.

The job of the handler is mainly to accept request from the client and to forward them to the right entry. It will also keep track of all write operations and thus keep two extra parameters: the set of unconfirmed write operations and the set of confirmed write operations.

```
-module(handler).

-export([start/4]).
```

```
start(Client, Id, Time, Store) ->
    spawn(fun() -> init(Client, Id, Time, Store) end).

init(Client,  Id, Time, Store) ->
    handler(Client, Id, Time, Store, [], []).

handler(Client, Id, Time, Store, Unconf, Conf) ->
    receive
        :
        :

        Error ->
            io:format("strange message ~w~n", [Error])
    end.
```

Read operations are forwarded directly to the right entry with the handler as the sender. The replies are returned to the client and one could ask if it is really necessary to send these first to the handler and then to the client; why not send them to the client directly? We will discuss these questions later but for now it's fine to do the simple things.

Note that the handler does not care if a read operation fails. It simply passes the result back to the client.

```
{read, Ref, N} ->
    Entry = store:lookup(N, Store),
    Entry ! {read, Ref, Time, self()},
    handler(Client, Id, Time, Store,  Unconf, Conf);

{reply, Ref, {ok, Value}} ->
    Client ! {Ref, {ok, Value}},
    handler(Client, Id, Time, Store,  Unconf, Conf);

{reply, Ref, abort} ->
    Client ! {Ref, abort},
    handler(Client, Id, Time, Store,  Unconf, Conf);
```

The write operations must be handler with a bit more care. When the client wish to commit we must know if all write operations could be committed. We will thus save the reference of a write operation in the list of unconfirmed operations. When a `ok` message arrives the operation is removed from the set of unconfirmed to the set of confirmed operations. Since we later have to send a message to the same entry we

If an `abort` message is returned then the whole transaction must abort and the client is informed by an abort message tagged with the identifier of the transaction.

```
{write, N, Value} ->
    Ref = make_ref(),
    Entry = store:lookup(N, Store),
    Entry ! {write, Ref, Time, Value, self()},
    handler(Client, Id, Time, Store, [{Ref, Entry}|Unconf], Conf);

{ok, Ref} ->
    {value, Op} = lists:keysearch(Ref, 1, Unconf),
    Unconf2 = lists:keydelete(Ref, 1, Unconf),
    handler(Client, Id, Time, Store, Unconf2, [Op|Conf]);

{abort, _Ref} ->
    Client ! {abort, Id};
```

When the handler is asked to commit the transaction it will send commit messages to all write operations issued during the transaction. It will then make sure that there are no unconfirmed transactions left. It does not have to check if the commit operations succeeded since this is guaranteed by protocol.

```
commit ->
    lists:map(fun({Ref,Pid}) -> Pid ! {commit, Ref} end, Unconf),
    lists:map(fun({Ref,Pid}) -> Pid ! {commit, Ref} end, Conf),
    case commit(Unconf) of
        commit ->
            Client ! {committed, Id};
        abort ->
            Client ! {abort, Id}
    end;
```

An abort message from the client will simply result in abort messages to all outstanding write operations.

```
abort ->
    lists:map(fun({Ref,Pid}) -> Pid ! {abort, Ref} end, Unconf),
    lists:map(fun({Ref,Pid}) -> Pid ! {abort, Ref} end, Conf);
```

To complete the handler we need one procedure that is waiting for the last `ok` messages. If all unconfirmed write operations have confirmed that they will be able to commit the transaction s a whole can commit .

```
commit([]) ->
    commit;
commit(Unconf) ->
    receive
```

```
        {abort, _Ref} ->
            abort;
        {ok, Ref} ->
            commit(lists:keydelete(Ref, 1, Unconf))
    end.
```

2.4 the server

The server simply keeps track of the store and the current time stamp. A client will call `open/1` to get access to newly created transaction handler with the correct time stamp, access to the store and and a unique transaction identifier.

```
-module(server).

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

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

init(N) ->
    Store = store:new(N),
    server(1, Store).

open(Server) ->
    Server ! {open, self()},
    receive
        {transaction, Time, Store} ->
            Id = make_ref(),
            Handler = handler:start(self(), Id, Time, Store),
            {Id, Handler}
    end.

stop(Server) ->
    Server ! stop.
```

We have a option here; should the transaction handler be started by the server or by the client. In the code listed, the handler is started by the client and thus will reside at the node of the client. This means that the tuple that represents the store has to be copied, possibly to another machine. If the handler was created by the server the store would also be copied but then only between processes on the same Erlang node. The approach chosen have some advantages in that once we have started the handler it will not place any burden on the node of the server.

```
server(Time, Store) ->
    receive
        {open, Client} ->
            Client ! {transaction, Time, Store},
            server(Time+1, Store);
        stop ->
            store:stop(Store)
    end.
```

3 Evaluation and discussion

3.1 performance

How well does the transaction handler perform? We can run some simple test with a couple of read and write operations to see how long time a singe transaction takes. We can then extend the tests to have several clients running on individual machines and run transaction concurrently to see how many transaction we could handle per second.

The number of failed transactions does of course depend on the number of read and write operations of each transaction and how large the store is compared to the number of concurrent transactions. The smaller the store the more likely it is that two transaction collide and one of them turns up to late.

One can also do experiment to see how long the lists of suspended operations are. Is it worth the trouble to optimize the handling of large lists or it is something else that we can do to increase performance?

3.2 read short-cut

One question that we skipped over was how come the handler ignores if a read operation had to abort. The reasoning is as follows. Assume that a client issues two read operation and is informed that one of them could not be handled but had to abort. The client has not seen an old value nor a value that is not yet committed so the client can not violate the consistency constraints by continue the transaction and later commit. It is thus up to the handler to decide what to do.

This of course also means that the reply from an entry could be sent directly to the client form the entry. Change the code and see if there is any advantages in performance.

3.3 advanced bookkeeping

Assume that transactions are long lived; a client might open a transaction and keep it open for minutes. Now if one transaction commits a value at time 17 but this value is still not the current value of the entry (there is a tentative value at time 14). Now what will happen if we have a read request to the entry at time 18? In our current implementation this request will be suspended but we could of course also send a reply back since we know what the value will be.

Design a scheme where read request can be replied to even if a committed value is yet not the current value of entry. You have to be careful; what would happen if we reply to a read request with time stamp 22 and then later receives a write request at time stamp 19.