Concurrent Programming (CPR) Assignment

Nicholas Drake
University of Oxford

25-29 January 2021

Total Number of Pages: 29

Part 1: An Auction Repository

Choice: How should the data be structured? And what data structures should be used?

Option 1: One massive data structure

Single data structure such as an ETS table where we have the tuple `{AuctionId, ItemId}` as the key

```
{{AuctionId, ItemId}, {Item, Desc, Bid}}
```

Advantages of using an ETS table as a single data structure

- Easy to keep a single table consistent.
- Data structures like ETS tables offer constant time access to the data e.g for `auction_data:get_item` (although $\mathcal{O}(\log n)$ for `ordered_set`) and with over 20 elements they are more efficient than lists.

Disadvantages of using an ETS table as a single data structure

- Single point of failure where if the process
- Auctions are independent so it does not make sense to have them in a single table.
- In Practical Part 2: The Auction Process we are told that we want to be able to lock our repository code which we cannot do if we have a single table and we want to lock on a set of keys.
- Match operations are implemented as BIFs and disrupt the real time properties of the system, leading to a slight bottleneck as BIFs prevent other processes from executing which if we wanted to do `auction_data:get_items` for example we would be blocking other unrelated auction activity
- May hit the max ETS table size of about 3.5GB.
- ETS tables are in-memory only and do not offered long-lived tables such as DETS (or Mnesia which is built on top).

Option 2: One data structure per auction

We could have one data structure per auction where if, for example, we use an ETS tables each process would have a table with

```
{ItemId, {Item, Desc, Bid}}
```

Advantages of using an ETS table for each auction

- The rule of thumb is that every concurrent activity in the system should be in its own process[1]. As auctions are independent it makes sense that they would have independent data structures both so they can be run concurrently (and potentially in parallel) and that they can fail independently as well.
- We want lexicographical order which ETS `ordered_sets` gives us for free with `first` and `next` operations.
- Easy to implement `auction_data:get_items` as a function that returns all the rows for a given `AuctionId`
- Can still have a separate process for the auction itself.

Disadvantages

- One ETS table per auction does not scale as well as say a Map inside each process. Although it is no longer true that the VM is limited to 1,400 ETS tables (although it was, even previously possible to change this with `erl -env ERL_MAX_ETS_TABLES Number`[1]) we should still avoid one table per process.

- We do not have a good way to get all the table names where `ets:all()` gets many other tables which we do not want to share with the user. Therefore we would potentially have to maintain a registry table of all the relevant ETS tables.
- ETS tables are in-memory only and do not offer long-lived tables such as DETS (or Mnesia which is built on top).
- Mnesia offers transactions, if we use ETS then only the owner of the process should be able to make changes. Transactions are important for operations where there is a mixed read and write[1] which we will need if we want to check whether a table exists first before deleting it (otherwise we would have to try-catch errors).

Option 3: One data structure per item

We could have one data structure per auction item where we might have

```
#{itemid => ItemId, item => Item, desc => Desc, bid => Bid}
```

Advantages of using a Map for each item

- Each item is independent so it makes sense that they be stored in separate processes, especially as they will be bid on separately as well.
- Using an ETS table for each item doesn't make sense as there is very little data to store, potentially just one single record, however, we would already be well set up to store the bid information on each item.

Disadvantages of using a Map for each item

- There is no easy way to aggregate items across an auction - it would involve messaging all the processes and waiting for them to message data back.

Solution: One Mnesia database

I decided to go for a single data structure using Mnesia. The problem with one ETS/Mnesia table per `AuctionId` is the number of tables constraint. The alternative of using a different data structure like a `map` makes it difficult to aggregate information as needed for `get_auctions`.

Mnesia offers the advantages of a single data structure but in addition to ETS in-memory performance offers transactions, sharding and distribution as well[1].

Choice: Should we use one or more tables?

Option 1: Table with `ItemId` key

We could have a single table with

```
-record(items_table, {item_id, auction_id, item, desc, bid})
```

The advantage of this is that all the information is available in one single table. The disadvantage is that `auction_data:get_auctions` would require a fold over all the items and then turning that into a set to remove duplicates, and similarly `auction_data:get_items` for a given `AuctionId` would require filtering and then returning all the keys. Furthermore,

Option 2: Table with a tuple of keys

We could have a single table with a tuple of keys

```
-record(items_table, {{auction_id, item_id}, item, desc, bid})
```

Unlike the previous implementation, this approach more naturally aligns with the `auction_data:get_item(AuctionId, ItemId)` API. However, `auction_data:get_items` and related functions that take just `AuctionId` as an argument there is a less natural implementation pattern matching on the key.

Option 3: Two tables

There are multiple options here including

```
-record(auction_ids, {auction_id, [item_id]}.
-record(auction_data, {item_id, item, desc, bid}).
```

The problem with this approach is we want to leverage the `ordered_set` to get `item_id` so there would not only be a duplication of `item_id` but also for `auction_data:get_items` we would not use the list of `item_id` as that would not be sorted. Of course, we could sort it and not use the `ordered_set` but this still feels like needless repetition.

Another option is to have a table with just the `auction_id`, and the rest being a tuple key of `{auction_id, item_id}`.

```
-record(auction_ids, {auction_id}).
-record(auction_data, {{auction_id, item_id}, item, desc, bid}).
```

The advantage of this is the `auction_ids` would effectively act as an index, negating the disadvantage of a tuple key and having to do a complicated filter and set to get all the auctions. However, it turns out that you cannot have a Mnesia table with a single column and you need both a key and a value[2].

Solution: Two tables with `lock` attribute

The solution I ended up going for was to have

```
-record(auction_ids, {auction_id, locked}).
-record(auction_data, {item_id, auction_id, item, desc, bid}).
```

The first table `auction_ids` makes it easy to implement `auction_data:create_auction` and `auction_data:get_auctions`, and the `lock` attribute provides a way to lock an auction efficiently and get around the single attribute constraint of `mnesia` tables. Although some of the methods (more detail on this later) will require updating multiple tables, or potentially checking multiple tables this is okay because `mnesia` offers transactions (something ETS does not offer) so we can ensure there will be no inconsistencies in the data from operations partially succeeding[3].

The second table uses `item_id` as a key and also includes `auction_id` to help perform a check that the auction is correct for methods like `auction_data:get_item(AuctionId, ItemId)`. We use a single key rather than a tuple key although both approaches could be made to work.

Choice: What data types to choose? Will it work in a distributed environment?

Variable: `Bid`

Our options include

- `non_neg_integer` = Do not allow negative numbers but do allow zero. The advantage of this is we can allow auctions to start at zero.
- `pos_integer` = Do not allow negative numbers or zero. Advantageous if we want to guarantee that an auction doesn't finish on zero but then requires starting bids to also start on non-zero.
- `float` = This allows for greater precision in bids but this could be potentially too much precision with amounts much smaller than pennies. The second disadvantage is it does not prevent negative numbers.

The use-case for our auction bidding backend is unclear, and for microtransactions such as with real time bidding for advertising space we might want to have floats, but for now we shall choose `non_neg_integer` for the non negative number check.

Variable: `AuctionId`

In the assignment page the hint says that we should use `erlang:make_ref/0` which we can then use as a unique table name.

We then have a choose as to whether to define in the spec the type as `ETSTableRef::term()`, or as `reference()`. We choose the latter in order to not tie ourselves in the spec to a specific implementation as although we have decided to use Mnesia this might potentially change in the future.

Variable: `ItemId`

Our options include

- `reference` = As hinted at in the assignment notes, we want to make sure `ItemId` works in a distributed environment as well, implying we should not use `erlang:make_ref()`. The problem with references is they may not be unique across restarts of nodes[4, 5, 6] and as Virding points out although 'A ref is unique over a distributed set of nodes. You should NEVER try and interpret the values in a pid/ref/port. They contain their node but how this is represented is NOT defined.'[7]. Furthermore, 'even though the access of a single object is always guaranteed to be `atomic` and `isolated` each traversal through a table to find the next key is not done with such guarantees'[8]
- `integer` = An alternative might be to generate some integer for example through `erlang:unique_integer`, or through a time `erlang:monotonic_time` although these are only unique for a given runtime system instance as well. We could potentially use Mnesia and its `dirty_update_counter`[9] as a form of auto-increment[10], but we would need cross-node locking with transactions to ensure that it increments appropriately[11].
- `tuple` = The best approach seems to be combine multiple keys together to ensure uniqueness [5]. For example, if we combine both `{erlang:monotonic_time(), make_ref()}` we can make sure that even if a node is restarted it will still generate a new unique reference[12], where we assume a restart will ensure that even if the same reference is generated we will get a new monotonic time. As noted in the Virding quote above this will also work in the distributed case as 'A ref is unique over a distributed set of nodes'[7]. In terms of spec, we define the type to be `itemid() :: {integer(), reference()}` where we can redefine it at a later stage if we see fit, if for example we do not want to expose our internal processing times etc. If we wanted we also have the flexibility to either add another element to the tuple such as `node()` or perhaps even hashing the elements to anonymous the information. Note that for parts 1 to 3 we use the definition we have here, but in part 4 for distributed erlang we change the definition of `ItemId` to include `node()` to make it work across different runtime systems or nodes[13]

Code Implementation

We will highlight some of the key code choices made in writing each function.

Function: `auction_data:install/1`

Mnesia comes with some additional setup above and beyond say ETS. This is because in order to work across many nodes Mnesia needs to have a schema to tell it how to store tables on disk (using DETs), how to load them and which other nodes it should be synchronized with[1]. The problem is there is a catch-22 where Mnesia both depends on the schema and needs to create it itself, where the solution is to call `mnesia:create_schema` beforehand.

```
install(Nodes) ->
  ok = mnesia:create_schema(Nodes),
  rpc:multicall(Nodes, application, start, [mnesia]),
  mnesia:create_table(auction_ids,
                      [{type, set},
                       {attributes, record_info(fields, auction_ids)},
                       {disc_copies, Nodes}]),
  mnesia:create_table(auction_data,
                      [{attributes, record_info(fields, auction_data)},
```

```
                              {disc_copies, Nodes},
                              {type, ordered_set}]),
  rpc:multicall(Nodes, application, stop, [mnesia]).
```

We then call `mnesia:create_table` with the schemas we have already discussed. Crucially we set the `auction_data` table to be an `ordered_set` to take advantage of the lexicographic ordering of the keys. For `auction_ids` we keep it as the default `set` as we do not care about the ordering here and it has faster access than `ordered_set`($\mathcal{O}(\log n)$ compared to $\mathcal{O}(\log n^2)$). We then both tables select `disc_copies` which means that the data is stored both in ETS and on disk (where Mnesia is not restricted to DETS storage limit of 2GB[1]).

We surround the `mnesia:create_table` calls with `rpc:multicall`s to `start` the application. We need to use these remote procedure calls here to make Mnesia work on multiple nodes so that the tables are not only created on the main node but also replicated to other nodes[1].

Then we have our `start` function which includes a call to `mnesia:wait_for_tables`. This is necessary because there could be a significant delay between the time that Mnesia starts and the time it finishes loading all the tables from disk[1]. As noted in Cesarini and Vinoski's Designing for Scalability with Erlang/OTP `mnesia:wait_for_tables` also helps prevent getting exceptions when unexpected restarts and asynchronous loading of tables can mean that other parts of the application can outpace the Mnesia part[14]. It is true that when repairing tables this can lead to a substantial delay, but for our system it is a useful synchronization point[14], although of course if it occurred in the middle of an auction it could have significant consequences.

Function: `auction_data:create_auction/0`

Like all other methods we run the Mnesia function call in a transaction where

```
F = fun() ->
     ...
end,
mnesia:activity(transaction, F).
```

Where we choose to use `mnesia:activity(transaction, F)` over `mnesia:transaction` because the former gives us more flexibility over selecting the access mode (we could switch to `mnesia_frag` for example if we wanted to fragment our data our multiple nodes[3, 1] with relative ease). Secondly, `mnesia:transaction` can fail silently where `mnesia:activity` simply returns the result on success and an error otherwise[3].

The rest of the implementation we have discussed including using `make_ref` for the `AuctionId` as suggested in the assignment. We set `locked` to false and crucially only update the `auction_ids` table so we don't have to deal with nulls in the `auction_ids` table.

Function: `auction_data:add_items/2`

As before, we wrap our function in a Mnesia transaction which is important because we perform two Mnesia methods here: `mnesia:read` to check whether the auction exists, and then after that `mnesia:write` to add the data to the database.

Note that the function description we were given in the assignment document has

```
add_items(AuctionId, [{Item, Desc, Bid}]) ->
```

but we cannot actually write the function clause head this way, instead writing `add_items(AuctionId, ItemsList)` specifying the types of the components in the spec as

```
-type item_info() :: {nonempty_string(),
                      nonempty_string(),
                      non_neg_integer()}.
```

As discussed previously for each `Item` we generate an `ItemId` and then `mnesia:write` to the table `auction_data`. As the table is an `ordered_set` and `ItemId` is the key the rows will automatically be ordered as we need them to be from the specification.

Then to generate the list of items we wish to return we use a fold left accumulating the `{ItemId, Item}` pairs we need to return. This is list is not necessarily ordered as if two items are generated with the same `erlang:monotonic_time()` they might have `make_ref()` in an order different from insertion order. Therefore, we need to be careful in testing that we do not assume any order of the returned list.

Function: `auction_data:get_items/1`

The main point to note is that we have two functions `F1` and `F2`. The former takes the table `auction_data` and for each row if the `AuctionId` matches extracts the `ItemId`. The latter function first checks that the auction exists in `auction_ids` and then if it uses `mnesia:foldr` to fold right over `auction_data` using `F1`, exploiting the ordering of `ordered_set` to return the list in correct `ItemId` order (we do not use fold left here as that would give reverse ordering).

```
F1 = fun(AuctionData, Output) ->
     case AuctionData of
       {auction_data, ItemId, AuctionId, _, _, _} ->
         [ItemId | Output];
       _ ->
         Output
     end
  end,
F2 = fun() ->
     ...
     mnesia:foldr(F1, [], auction_data)
     ...
mnesia:activity(transaction, F2).
```

Function: `auction_data:get_item/2`

This has similar logic to `auction_data:get_items` where we pattern match on `AuctionId` reading out the data. It is worth noting that the API requires us to return `ItemId` so even though we insert `Item` with `auction_data:add_items` we actually never extract it.

Function: `auction_data:remove_auction/1`

The main point of note here is that if we remove an auction we do not delete all the item data in `auction_data`. This is because our first table `auction_ids` effectively represents the live and pending auctions, whereas `auction_data` also holds past data for winning bids etc. (see Part 2: The Auction Process).

Function: `auction_data:remove_item/2`

Again simple pattern match on `AuctionId` and a `mnesia:delete` wrapped in a transaction.

Testing

There are two main testing frameworks: EUnit and Common Test - the former is well-suited for unit tests and the latter for integration tests and distributed Erlang. Ideally we want to be able to use the two of them together but it can be difficult to include Common Tests in the EUnit framework, but the reverse is much easier therefore we shall primarily use Common Tests.

In order to run the tests move into the correct folder and from there run `rebar3 ct` where seven tests should pass.

```
cd practical/part_1
rebar3 ct
```

Tests: `auction_data_SUITE` setup and teardown

Common tests offer three levels of setup and teardown functions at a suite, group and testcase level. We use the `init_per_suite` method to load and install both our `auction_data` application and `mnesia`. Note that we have given `auction_data` the `-behaviour(application).` rather than having a separate supervisor and app in order to align with the required specification.

We then use `init_per_testcase` methods for each specific method. There is some repetition here but in general not a huge amount. Where appropriate I used underscore to capture common cases for example

```
end_per_testcase(_, Config) ->
  AuctionId = ?config(auction, Config),
  ok = auction_data:remove_auction(AuctionId).
```

to remove any auctions after a testcase has finished. It is worth noting that `init_per_testcase` and `end_per_testcase` methods are run in the same process as that test is run, this is unlike `init_per_group` which will require us to change our initialisation process later on[1].

Tests: `auction_data_SUITE` tests

The tests are fairly self-explanatory, where I am careful to test for both successful outputs like

```
{ok, {ItemId1, "blue cap", 1}} =
    auction_data:get_item(AuctionId, ItemId1),
```

where we pattern match on `ItemId1` which has already been defined and therefore acts as an assertion, and for errors such as

```
{error, unknown_auction} =
    auction_data:remove_item(InvalidAuctionId, ItemId2).
```

Part 2: The Auction Process

Choice: When should `auction_data` be updated with sold items?

Option 1: Update `auction_data` after each Item is sold

Advantages

- We can store and replicate the bid data - if you had an auction on expensive items you wouldn't want to have to re-run it.
- We could avoid the `lock` on `auction_data` rows by having a separate function to add the winning bidders which does not perform the `lock` check - unlike `auction_data:add_items`

Disadvantages

- It is unclear if the starting bid is an actual bid that users can submit in advance of the auction starting or if it is a minimum bid and perhaps the item will be unsold if that bid level isn't reached. We shall assume that it represents a minimum bid as most auctions require live bidding rather than in advance - I think!
- Once an auction is `lock`ed then it is unclear how we can still allow updates to the `auction_data` rows.

Option 2: Store all item sold data in the process until the end of the auction

Advantages

- Allows bid information to be independent from the `auction_data`.
- Do not have to overly tax the Mnesia database with querie.s

Disadvantages

- An auction that restarts half-way will have to re-do all the previously sold items.

Solution: Update after every bid

On balance it is probably more important to update the `auction_data` after each item is sold to ensure we do not lose that data. We will only have the `lock` apply to `auction_data:add_items` and not `auction_data:add_winning_bidder`.

Choice: What OTP behaviour to use for `auction`: `gen_server` or `gen_statem`?

Option 1: `gen_server`

Advantages of `gen_server`

- Provides a robust abstract of the generic client-server interactions.

Disadvantages of `gen_server`

- For our `auction` use case we need the item to be sold if no one bids for 10s which will involve complicated management of timeouts, something which `gen_server` does not help us with.

Option 2: `gen_statem`

Advantages of `gen_statem`

- Provides the same functionality as `gen_server` but with a bit extra.
- `gen_statem` is well-suited for 'Easy-to-use time-outs (State Time-Outs, Event Time-Outs and Generic Time-Outs (named time-outs))'[15] and is preferred over the deprecated `gen_fsm`[16].

Disadvantages of `gen_statem`

- More complicated to implement

Solution: `gen_statem`

We want to leverage the built in `StateTimeout`.

Choice: one process per auction or one process for all live auctions

Option 1: One process per auction

Advantages of one process per auction

- We can build the supervision tree afterwards uses `start_link`.
- This allows us to have independence between different auctions.

Disadvantages of one process per auction

- The Part 4 Auction House API will need to handle directing each function call to the correct `auction` process.
- This approach creates a slight redundancy in the suggested `auction` API because technically we do not need need the `AuctionId` argument as part of the `auction:bid` call because each `Auction` should already know its own `AuctionId`. If the specification wanted us to implement one process per auction then it is the `auction` app that needs to know the `AuctionId` not the specific auction process.

Option 2: One process for all live auctions

Advantages of one process for all live auctions

- Will obviate the need to route the different `auction` calls to different auctions.
- Aligns with the assessment APIs were bid calls can go directly to the `auction` set of functions.

Disadvantages of one process for all live auctions

- Will needlessly create a single dependency and single bottleneck for all auctions
- Will lead to complicated state needing to be maintained to keep track of all the bidding for all the live auctions.

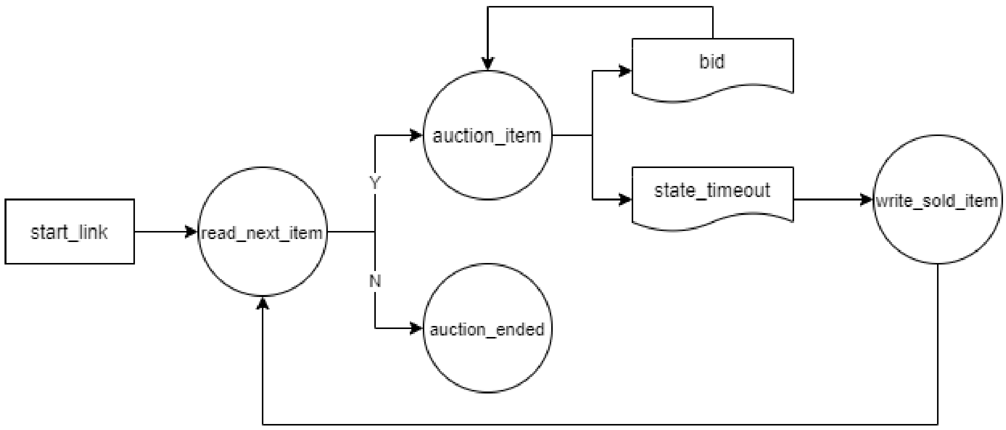
Solution: One process per auction

This should tie in better with a supervision hierarchy and allow greater independence.

Choice: What states for OTP `gen_statem` state machine?

Option 1: States for reading and writing to `auction_data`

We could have four different states `read_next_item`, `auction_item`, `write_sold_item` and `auction_ended` representing in the diagram as circles. The auction would start and would first check to see if there any items to auction with `read_next_item` by reading from `auction_data` and if not then would transition to `auction_ended`. If there is an item we transition to `auction_item` state awaiting bids, and if there are bids we return to that state, and if there are no more bids we receive the `state_timeout` message transitioning to `write_sold_item`. Here we can then write to the `auction_data` database and transition to see if there is another item in `read_next_item`.



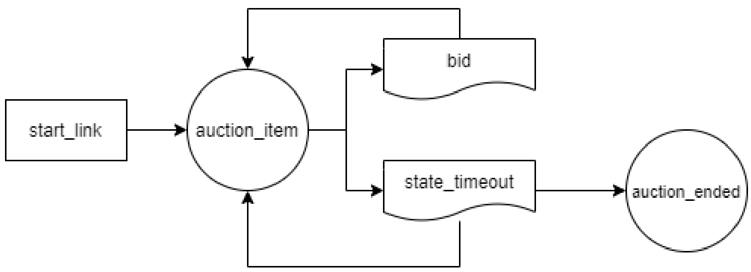
Advantages of separate states for reading and writing to `auction_data`

- Clean code where each state has a separate purpose

Disadvantages of separate states for reading and writing to `auction_data`

- For the time that we are in the reading and writing states messages that are not matched will be added to a save queue via Erlang's selective receive mechanism[17] and then will be retried again once a new message matches. However, the re-checking of the save queue only happens if a message in the `auction_item` state is matched which means that bids will be out of order.
- The alternative is ignoring messages that are slightly too early or slightly too late but this could lead to brief periods of no response or for which no bid is valid etc.
- There is no need for `write_sold_item` because `state_timeout` is actually a state itself.
- We needlessly have a `read_next_item` state when we only need to get the list of items once.
- In general we have multiple states when we do not want to react differently for them

Option 2: No separate states for reading and writing to `auction_data`



Advantages of no separate states for reading and writing to `auction_data`

- No issues with unreceived messages once an auction has started

Disadvantages of no separate states for reading and writing to `auction_data`

- Have to get the next item for both the initialized state and after each bid

Solution: No separate states

We want to avoid the complicated selective receive and `gen_statem` implementation and we go for a cleaner two state implementation with `auction_item` and `state_timeout`

Code Implementation

Function: `auction:start_link/1`

First, there was a choice of whether the `auction_data:get_items` call should happen in `start_link` or `init`. For separation we would like it to occur in the latter, but we need it to occur in the former so we can return `{error, unknown_auction}` if the `AuctionId` is not found.

When we get the items we want to be able to lock as well in a single transaction otherwise there could be a race condition where we `get_items` and then before we `lock` more items are added. Then within `start_link` the key function is we separate the returned list of items into a head and a tail and initialize the `gen_statem:start_link` function.

Function: `auction_data:get_items_and_lock_auction/1`

We want to be able to `get_items` without locking the table, therefore we need a separate function. We could write one completely from scratch but Mnesia nested transactions[18] allows us to use `auction_data:get_items` inside `auction_data:get_items_and_lock_auction`. I tried writing a general `lock` function, but it is difficult to make it work with varying numbers of arguments and there are no other use-cases for it thus far so we stuck with the more specific version.

Function: `auction:init/1`

For `gen_statem` we need to initialize `State`, `Data` and additional `Args`. We call the main state `auction_item` (rather than `auction_item` as in the diagrams). And we initialize the `Data` map with the `AuctionId`, `HeadItemId` and `TailItemIds` from the `start_link` function.

Then the crucial additional argument and the main motivation for using `gen_statem` is that it allows us to use `state_timeout`[15] where in our case we have set it to timeout after `10000` ms or in other words `10 s`

```
{ok, State, Data, [{state_timeout, 10000, next_item}]}.
```

Function: `auction:bid/4` **and** `auction_item`

The `bid` function works by call the synchronous `gen_statem:call` method which allows us to change state and also return a reply. This calls the method implemented as

```
auction_item({call,From},
             {bid, BidAuctionId, BidItemId, Bid, Bidder},
             ... = Data) ->
    ...
```

There are two main parts of the method. First we check whether the received bid is valid or not with the `check_for_invalid_bid` method. If the bid is invalid it will return the appropriate error messageand `keep_state` which significantly for our purposes does not reset the `state_timeout` timer (because of course we do not want invalid bids to keep the auction going for no reason!).

Then the second part of the method `get_starting_bid` if there is one and then with that updated information `check_leading_bid`. Getting the starting bid here is awkward, because in particular the implementation checks with we have a starting bid for every bid that comes in even if it isn't relevant to do so because we already have a leading bid. However, this is only a local check and will avoid pinging the `auction_data` database unless necessary. The other possibility would be to integrate it into the `init` and `state_timeout` functions as we have done for other aspects of state but this would involve calling the method in two places, whereas here at least it is only called in one.

Regardless having gotten the starting bid (if there is one) we can then check whether we have a leading bid with the `check_leading_bid` function which tests the four scenarios giving the appropriate reply, using the `From` argument to reply to the client, and return state. Of note, is that if a bid is valid then we transition to the `auction_item` state again but with updated `leading_bid` and `leading_bidder` and we refresh the `state_timeout` to 10s again.

```
{next_state,
 auction_item,
```

```
 Data#{leading_bid := Bid, leading_bidder := Bidder},
[{reply, From, {ok, leading}}, {state_timeout, 10000, next_item}]}
```

Function: `auction_item(state_timeout,...)`

After 10s with no bids we transition to the `state_timeout` state. First we `add_winning_bidder` assuming there is one to the `auction_data` table. Then we `get_next_itemid` from the `RemainingItemIds`. This will either return a `NewCurrentItemId` which is `undefined` and then we transition to the `auction_ended` state (where we simply reply with `auction_ended` no matter what the message) or it will return another item to auction and we reset to `auction_item` state with a new timeout.

Here the API is a little restrictive where we could have an item that isn't sold but we still need to transition to the next state so we maintain a list of `auction_itemids` which we can use to return the `item_sold` error, but if an item is not sold it won't be in that list so we instead respond with `invalid_item`.

Testing

Ideally I would like to use a mocking library like `meck` for the `auction_data` side effects calls in `auction` but we have been told to only use librarires from the Erlang/OTP distribution so instead I simply run the tests with `auction_data`. To keep the parts separate this is copied into `part_2` and can be run independently there, this works well as you can also more easily see the enhancements and changes that have been made to the code, for example `auction_data` now has an additional `get_items_and_lock` method and corresponding tests.

The tests are similar to `part_1` and although include common test groups it is only in `part_3` that we start to leverage parallel testing with difference processes calling each other.

In order to run the tests cd into the correct folder and from there run `rebar3 ct` where twenty-one tests should pass

```
cd practical/part_2
rebar3 ct
```

It is worth noting that they will take about a minute to run because they use `timer:sleep` to test that the `gen_statem` timeouts are being triggered correctly.

Part 3: Pub Sub Engine

Choice: What OTP framework to use?

Solution: `gen_event`

Advantages of `gen_event`

- Good if the server has many subscribers
- Don't need to spawn processes for short-lived tasks
- Process is simple where we just spawn an event manager and attach it and that handles init handler loop and exit handlers[1]

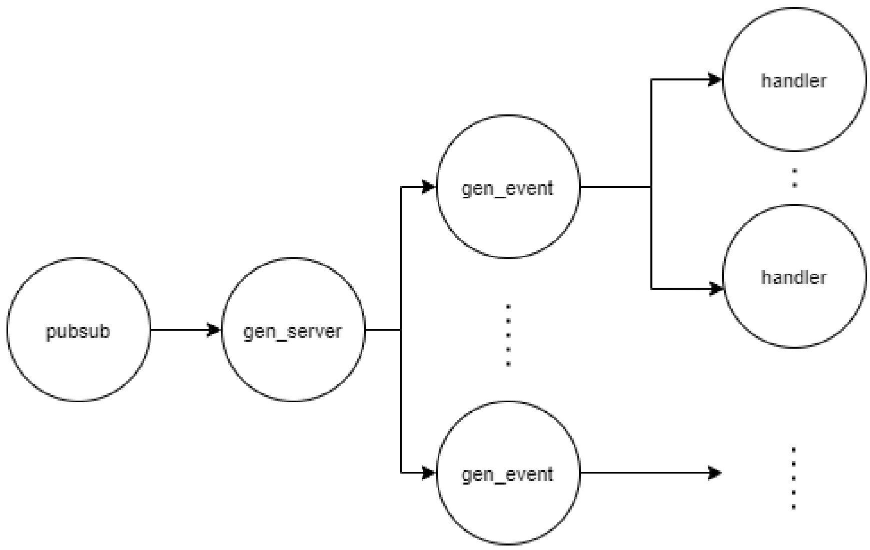
Disadvantages of `gen_event`

- Functions that run for a long time will block each other
- A function that loops indefinitely can prevent any new event from being handled.

We could have also used a simple `gen_server` but in the interest of brevity we only discuss `gen_event` as this was the obvious choice for managing multiple subscribers.

Choice: How do we model channels, and subscribers?

Option 1: One event manager per `AuctionId`



Advantages of one event manager per `AuctionId`

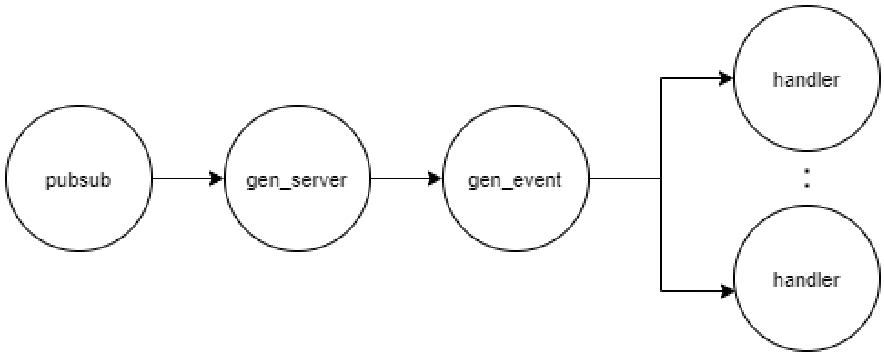
- We can then have the application choose which event manager `Pid` to update which will in turn update any subscribers.
- We have a separation where if one event manager fails it will only affect that auction and not all auctions.
- We filter the messages on the publishing side rather than the subscribing side only sending messages to the handlers that want them.

Disadvantages of one event manager per `AuctionId`

- Potentially inefficient to have so many `gen_event` event managers.

- We will then need some state to manager the list of event managers - perhaps a map or an ets database etc. Would like to use `gproc` but we are not allowed to use non-OTP libraries.
- We could have multiple event handlers for a given subscriber if they are subscribed to more than one `AuctionId`.

Option 2: One event manager with a way to create topics



Advantages of one event manager with a way to create topics

- This seems like a cleaner way to manage all the subscribers especially as some clients might want to subscribe to more than one `AuctionId`

Disadvantages of one event manager with a way to create topics

- There does not seem to be a good way to create topics on the handler side dynamically. We could set the `gen_event` handler to only handle a subset of events with `handle_event` but I wasn't clear on how to do this dynamically for different `AuctionIds`.
- Furthermore, having a single event manager would mean sending all messages to all handlers and having the handlers filter which is potentially quite wasteful.

Solution: One event manager per AuctionId

Decided to go for option 1 with one event manager for each auction keeping them independent and avoiding sending messages about all auctions to every handler.

Code Implementation

Function: `pubsub:handle_call/3`

The heart of the `pubsub gen_server` is the `handle_call` implementations. We handle each message where the `Channel` is the `AuctionId` reference. We then maintain a `gen_server` state map `Channels` where for each key `Channel` we have a value which is the `gen_event:start_link ChannelPid` for that auction's event manager.

We can then add and remove subscribers to that event manager with the `gen_event:add_sup_handler` and `gen_event:delete_handler` methods where we keep track of each subscriber with unique `HandlerIds`

```
{ClientPid, _} = From,
HandlerId = {channel_feed, ClientPid},
gen_event:add_sup_handler(ChannelPid, HandlerId, [ClientPid]),
```

which we create from the `From` variable. Note we could not use the client-tag here as that is unique to each query.

Function: `channel_feed:handle_event/3`

We also need to implement a `gen_event` behaviour to tell the handler how to act. In our case we just simply send the `Event` message to the `ClientPid` we specified in the `gen_event:add_sup_handler` call.

Function: `auction:subscribe/1`

In order to implement the subscribe function we need to return a `MonitorRef`. It is not clear from the specification what specific reference we should be monitoring but for our implementation it makes sense that when we subscribe to an `AuctionId` we want to monitor the corresponding `ChannelPid` event manager.

We use a newly created `pubsub:monitor` method to expose the `ChannelPid` and then create a monitor reference by using `erlang:monitor`

```
Reference = erlang:monitor(process, ChannelPid),
```

Functions: adding `pubsub` calls

In order to implement the rest of the code we add `pubsub` calls as appropriate although on occassion we have to ping the `auction_data` layer for more data

```
  {ok, {HeadItemId, Description, StartingBid}} =
    auction_data:get_item(AuctionId, HeadItemId),
  pubsub:publish(
    AuctionId,
    {auction_event, {new_item, HeadItemId, Description, StartingBid}}),
```

This leads to more than one call to the `auction_data` layer per `ItemId` which perhaps might be worth reducing by carrying more data in `State`.

Testing

Tests: `pubsub_SUITE`

In addition to the standard unit tests we also have an integration test which combine multiple function calls together. In order to implement this we use parallel tests that interact with each other with two publishers `pub1` and `pub2` and two subscribers `sub1` and `sub2`.

```
groups() ->
  [{pubsub_session,
    [],
    [{group, pubs_and_subs}]},
   {pubs_and_subs,
    [parallel],
    [pub1, pub2, sub1, sub2]}].
```

However, `init_per_group` and `end_per_group` methods, unlike their test case counterparts, are run in separate processes which means that when we call `pubsub:start_link()` we tie the `pubsub` process to the `init_per_group` process which ends before the testing starts. Therefore a crucial step that is needed is to `unlink(Pid)`[1]. Furthermore, as we wish to share the `pubsub` we need to do so by nesting the `pubs_and_subs` group inside the `pubsub_session`.

Having done this we are able to demonstrate the two publishers and two subscribers publishing, subscribing and unsubscribing. To run the `pubsub_SUITE` specifically move into the `part_3` directory and run `rebar3 ct` adding the specific suite and `--verbose` to see the printed messages

```
cd practical/part_3
rebar3 ct --suite apps/pubsub/test/pubsub_SUITE --verbose
```

Tests: `auction_SUITE`

We extend the tests of the `auction` module to take account of the new functionality and in particular use parallel tests similar to those in the `pubsub_SUITE` to demonstrate a real auction in action.

As with the `pubsub_SUITE` we use nested groups to setup the integration test properly

```
{bidder_integ, [], [{group, bidder_integ_components}]},
{bidder_integ_components, [parallel], [pub1,
                                        sub1,
                                        sub2]}].
```

We can then run the tests in verbose mode to see the messages that are sent between our two bidders.

```
cd practical/part_3
rebar3 ct --suite apps/auction/test/auction_SUITE --verbose
```

To see test output of the form where `sub1` and `sub2` indicate which subscriber is seeing the message. Here we have both subscribers seeing that a new item, a blue cap, is put up for auction, that it is bid on for 5 pounds, and that the item is sold for 5 pounds.

```
----------------------------------------------------
2021-03-08 18:30:56.666
sub2 {auction_event,
  {new_item,{66944000,#Ref<0.2669465380.2526543873.73387>},
  "blue cap",3}}
----------------------------------------------------
2021-03-08 18:30:56.666
sub1 {auction_event,
  {new_item,{66944000,#Ref<0.2669465380.2526543873.73387>},
  "blue cap",3}}
----------------------------------------------------
2021-03-08 18:31:01.671
sub2 {auction_event,
  {new_bid, {66944000,#Ref<0.2669465380.2526543873.73387>}, 5}}
----------------------------------------------------
2021-03-08 18:31:01.671
sub1 {auction_event,
  {new_bid, {66944000,#Ref<0.2669465380.2526543873.73387>}, 5}}
----------------------------------------------------
2021-03-08 18:31:11.673
sub2 {auction_event,
  {item_sold, {66944000,#Ref<0.2669465380.2526543873.73387>}, 5}}
----------------------------------------------------
2021-03-08 18:31:11.673
sub1 {auction_event,
  {item_sold, {66944000,#Ref<0.2669465380.2526543873.73387>}, 5}}
```

And of course, we could also just run all the tests with the usual `rebar3 ct` command where 35 tests should pass.

```
cd practical/part_3
rebar3 ct
```

Part 4: Implementing the Client

Choice: Supervising auctions

Option 1: One supervisor for one auction

Advantages of one supervisor for one auction

- Easier to implement supervisor structure.

Disadvantages of one supervisor for one auction

- Will have to refactor `auction` `gen_statem` to manage multiple auctions at once which will be uglier
- Will also lose independence of different auctions so if one fails they all do.

Option 2: One supervisor with a dynamic number of children auctions

Advantages of one supervisor with a dynamic number of children auctions

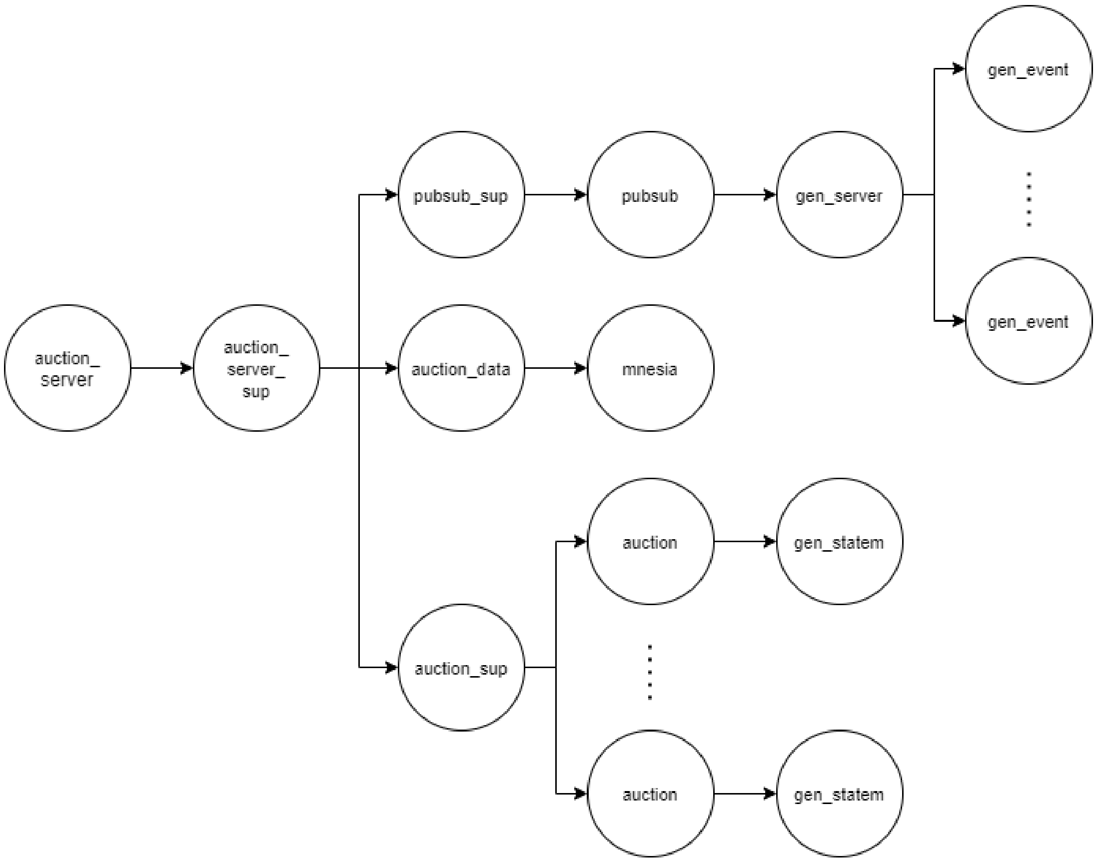
- Can have a separate process per auction

Disadvantages of one supervisor with a dynamic number of children auctions

- Will have to refactor the `auction` api to include the `pid` of the child process

Solution: One supervisor with a dynamic number of children auctions

We refactor the system to allow for dynamic children (more discussion on the code implementation follows in Function: `auction_sup:start_auction/1` section).



This is represented in the diagram as a single `auction_sup` which manages a dynamic number of `auction`s which under the hood each use `gen_statem`. In the diagram, we can also see the `auction_data` and `pubsub` implementations we have already discussed in Parts 1 to 3.

auction_data is an application and does not have its own specific supervisor, although pubsub as a gen_server does. This is a more standard supervisor. These are then all managed by a single auction_server_sup supervisor which includes the loading and installation of auction_data and mnesia and managing the pubsub_sup and auction_sup children which are paired in a ChildSpecList for synchronous starts[14]

```
SupFlags = #{strategy => rest_for_one,
             intensity => 3, % max number of restarts / period
             period => 3600}, % period is 60 mins
ChildSpecList = [child(pubsub_sup), child(auction_sup)],
{ok, {SupFlags, ChildSpecList}}.
```

Finally, these sit under an auction_server application. We want to avoid bottlenecks so when running auction_server apart from starting and stopping auctions most of the messages are sent directly to the component parts.

Note that the code has been refactored where because, technically pubsub is not a separate application it has been moved into the auction_server folder.

Code Implementation

Function: auction_sup:start_auction/1

In order to manage the dynamic children we use a simple_one_for_one strategy, which is appropriate because we only have one child specification shared by all the processes under a single supervisor[14].

```
init([]) ->
  SupFlags = #{strategy => simple_one_for_one,
               intensity => 3, % max number of restarts / period
               period => 60}, % period is 60
  ChildSpecs = [{auction_child,
                 {auction, start_link, []},
                 transient, 1000, worker, [auction]}],
  {ok, {SupFlags, ChildSpecs}}.
```

We then add and remove auctions with separate functions start_auction/1 and stop_auction/1 methods

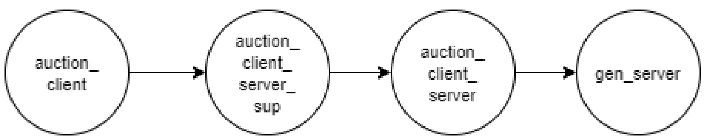
```
start_auction(AuctionId) ->
  supervisor:start_child(?MODULE, [AuctionId]).

stop_auction(AuctionPid) ->
  supervisor:terminate_child(?MODULE, AuctionPid).
```

Then finally to make this work and be able to message the correct auction process we have to change the definition of bid to add the AuctionPid arguments

```
bid(AuctionPid, AuctionId, ItemId, Bid, Bidder) ->
```

Function: auction_client:start_link/1

Most of the auction_client implementation is fairly standard gen_server where requests to subscribe, bid etc. are handled in handle_call. Each client will have their own auction_client with a unique string BidderName.



Function: `auction_client:handle_info/3`

In addition to this, the `auction_client` can also receive information from the `pubsub` system by `handle_info` which handles messages that are not explicitly sent via the `gen_server` API. These are then printed out to the user as requested in the specification with `io:format`. For example we have

```
handle_info({{AuctionId, auction_event},
  {item_sold, _ItemId, WinningBid}}, State) ->
  io:format("AuctionId␣~p:␣Item␣sold.␣Winning␣bid␣~p~n",
    [AuctionId, WinningBid]),
  {noreply, State};
```

Crucially, because we have each `auction` as a separate child process, we only know the `AuctionPid` once that process is started. So then the request is how do we distribute the `AuctionPid` to all the subscribers so they can bid on items etc. The answer is using the `pubsub` system where when an `auction` is started we adjust the message to include both the `AuctionId` and the `AuctionPid`

```
handle_info(
  {{AuctionId, auction_event}, {auction_started, AuctionPid}}, State) ->
  io:format("AuctionId␣~p:␣Started~n", [AuctionId]),
  AuctionIdPidMap = maps:get(auction_id_to_pid_map, State),
  UpdatedAuctionIdPidMap = maps:put(AuctionId, AuctionPid,
    AuctionIdPidMap),
  {noreply, State#{auction_id_to_pid_map := UpdatedAuctionIdPidMap}};
```

We then inside each client maintain a map of `AuctionIds` to `AuctionPids` which the user can use to bid to correct `AuctionPid` by

```
AuctionIdPidMap = maps:get(auction_id_to_pid_map, State),
case maps:get(AuctionId, AuctionIdPidMap, undefined) of
  ...
  AuctionPid ->
    ...
    Result = auction:bid(AuctionPid, AuctionId, ItemId, Bid, Bidder)
```

This then abstracts the client from having to every manage `AuctionPids` and potentially could be extended such that if an `auction` went down then the client could be updated with a new `AuctionPid`.

For two of these messages: `new_item` and `new_bid` we may need to respond by `bid_automatically` which is discusssed below.

Function: `auction_client:bid_automatically/4`

Users can submit `add_automated_bid_to_maxs` where they set a starting bid and a max bid which they are willing to go up to. Then anytime that `ItemId` is bidded on (or is a new item) it will be automatically bid on. This works in `bid_automatically` where we maintain a map of `automated_bidding` strategies with the `ItemId` as key and the value of the `CurrentBid` and `MaxBid`. If there is a strategy, then we check to see if we can outbid the incoming `Bid`.

```
if
    Bid + 1 =< MaxBid ->
      {BidderName, _} = maps:get(bidder, State),
      automated_bid(BidderName, AuctionId, ItemId, Bid + 1, From),
      UpdatedAutomatedBiddingMap =
        AutomatedBiddingMap#{ItemId := {From, Bid + 1, MaxBid}},
      {noreply, State#{automated_bidding :=
        UpdatedAutomatedBiddingMap}};
    true ->
      {noreply, State} % do nothing as too high
  end
```

However, there is a problem where we can end up in deadlock. This is because if we simply `bid/4` within the `gen_server` then we will be waiting for the response to the call but will be unable to process that response because we are still waiting for it! Therefore to resolve this we need to use automated_bid where we spawn a new process to wait for the response, and then to separately handle replying to the client[19]

```
automated_bid(BidderName, AuctionId, ItemId, Bid, From) ->
  spawn(fun() ->
    BidResponse =
      gen_server:call({global, BidderName}, {bid, AuctionId, ItemId, Bid}),
    gen_server:reply(From, BidResponse)
  end).
```

Test output proving this automated bidding is discussed further in Tests: auction_client_server_SUITE.

Function: auction_client:start/2

In order to implement a distributed OTP application we need to interact with the application_controller which sits on top of the applications in our system.

There are two important concepts for distributed applications. The first is failover which is the idea of restarting an application somewhere other than where it stopped running[1]. The second is takeover which is the act of a dead node coming back from the dead[1].

For our application we can achieve this in the client through changing the start/2 application definition to

```
start(normal, [BidderName]) ->
    auction_client_server_sup:start_link(BidderName);
start({takeover, _OtherNode}, [BidderName]) ->
    auction_client_server_sup:start_link(BidderName).
```

Here, we have BidderName listed a StartArgs but in order to use it this way we would have to change how application:start(auction_client) works and use environmental variables (perhaps through a config to set it - so in the code we instead hard-code the BidderName instead.

Note that the client names are all global so they can be accessed via the global registry with for example in auction_client_server_sup

```
supervisor:start_link({global, BidderNameSup}, ?MODULE, [BidderName]).
```

Then to make the application distributed we need to add configuration files main.config and backup.config

```
[{kernel,
  [{distributed, [{auction_client,
                   5000,
                  [main@localhost, {backup@localhost}]}]},
   {sync_nodes_optional, [backup@localhost]},
   {sync_nodes_timeout, 30000}]}].
```

Then to run the client in distributed mode we can type from one terminal

```
cd part_4/apps/auction_client
erl -sname main -config config/main -pa ebin/
```

And another terminal

```
cd part_4/apps/auction_client
erl -sname backup -config config/backup -pa ebin/
```

I have set sync_nodes_optional rather than sync_nodes_mandatory. Also note that you might need to change localhost to your own localhost. For some reason, I could use just localhost here but when I tried to run the distributed tests in auction_simulation I had to explicitly set it as DESKTOP-N3ROL9Q.lan. Also note that it is important that the auction_client.app file exists in ebin which means you may need to both create the folder and run rebar3 compile to get it there. On windows I had trouble with this so just copied the .src file in the folder instead.

To see the backup capabilities in action, we can then run application:which_applications() which will show auction_client on the main but not the backup. If I then kill the main it will then show up on backup.

Testing

Tests: `auction_client_server_SUITE`

In addition to some standard unit tests we, as we did with the `pubsub_SUITE`, again use nested groups to test our functionality.

```
groups() ->
  [{automated_integ,
    [],
    [{group, automated_integ_components}]},
   {automated_integ_components,
    [parallel],
    [test_add_automated_bid_to_max, auction_house, other_bidder]}].
```

These tests can be run by

```
cd practical/part_4
rebar3 ct --suite apps/auction_client/test/auction_client_server_SUITE
--verbose
```

These tests use `ct:capture_get/0` to get the `io:format/2` messages so we can assert that they are correct, for example we have this test to make sure that we print out the correct message when we subscribe.

```
{ok, _} = auction_client_server:subscribe(BidderName1, AuctionId1),
ExpectedString = lists:flatten(
    io_lib:format("AuctionId␣~p:␣Subscribed\n", [AuctionId1])),
[ExpectedString] = ct:capture_get(),
```

If you run the tests with the `verbose` flag as above you will get a print out using `ct:print/2` of the messages that are printed by `io:format/2` for the `test_automated_bid_to_max`. This test demonstrates the bid automatically increasing with each rival bid unless that would require bidding above the user's limit. First we add an automatic bid starting at 3 up to a max bid of 10. We then submit this bid of 3 which is accepted, but then a rival bidder bids 5. We then counter with 6 using the automatic bidding before the rival bids 10, which would require a counter bid of 11 and the automatic bidder does not bid any more.

```
2021-03-15 12:39:14.848
AuctionId #Ref<0.1268505216.3229351937.55215>: Added automatic bid.
----------------------------------------------------
2021-03-15 12:39:14.848
              Start bid: 3, Max bid: 10
----------------------------------------------------
2021-03-15 12:39:14.848
AuctionId #Ref<0.1268505216.3229351937.55215>: Submitted bid 3
----------------------------------------------------
2021-03-15 12:39:14.848
AuctionId #Ref<0.1268505216.3229351937.55215>: Bid 3
----------------------------------------------------
2021-03-15 12:39:16.856
AuctionId #Ref<0.1268505216.3229351937.55215>: Bid 5
----------------------------------------------------
2021-03-15 12:39:16.856
AuctionId #Ref<0.1268505216.3229351937.55215>: Submitted bid 6
----------------------------------------------------
2021-03-15 12:39:16.856
AuctionId #Ref<0.1268505216.3229351937.55215>: Bid 6
----------------------------------------------------
2021-03-15 12:39:18.849
AuctionId #Ref<0.1268505216.3229351937.55215>: Bid 10
```

Tests: `auction_server_sup_SUITE`

Need to add `timer:sleep(10000)` to `end_per_suite` of `auction_SUITE` to make sure there is enough time to `application:stop(mnesia)`

Tests: `auction_simulation_SUITE`

We also want to be able to demonstrate the system working in a distributed fashion. For this we have a separate folder `auction_simulation`.

To run the system you will need to replace all the references to `DESKTOP-N3ROL9Q.lan` with your localhost in both the `configs` files `auction_server.config` and `client.config` and in the `auction_simulation.spec` file.

Then run

```
cd practical/part_4/apps/auction_simulation
erl -name ct
ct_master:run("auction_simulation.spec").
```

You should then get output like where the tests are shown to be passing on the `client` and `auction_server` with `{1,0,{0,0}}`.

```
=== Master Logdir ===
c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/part_4/apps/
auction_simulation/simulation_logs
=== Master Logger process started ===
<0.88.0>
Node 'auction_server@DESKTOP-N3ROL9Q.lan' started successfully with callback
ct_slave
Node 'client@DESKTOP-N3ROL9Q.lan' started successfully with callback ct_slave
=== Cookie ===
'OOZLJKRRLPZGKSKOOTBA'
=== Starting Tests ===
Tests starting on: ['ct@DESKTOP-N3ROL9Q.lan','client@DESKTOP-N3ROL9Q.lan',
                    'auction_server@DESKTOP-N3ROL9Q.lan']
=== Test Info ===
Starting test(s) on 'ct@DESKTOP-N3ROL9Q.lan'...
=== Test Info ===
Starting test(s) on 'client@DESKTOP-N3ROL9Q.lan'...

********** node_ctrl process <0.108.0> started on 'ct@DESKTOP-N3ROL9Q.lan'
**********
=== Test Info ===
Starting test(s) on 'auction_server@DESKTOP-N3ROL9Q.lan'...

Common Test starting (cwd is c:/Users/lao8n/OneDrive/Documents/oxford_cpr/
assignment/practical/part_4/apps/auction_simulation)


Common Test: Running make in test directories...
Including the following directories:
"c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/part_4/
apps/auction_simulation"

CWD set to: "c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/
part_4/apps/auction_simulation/simulation_logs/ct_run.ct@DESKTOP-N3ROL9Q.lan.
2021-03-15_13.52.56"

TEST INFO: 0 test(s), 0 case(s) in 0 suite(s)

Updating c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/
part_4/apps/auction_simulation/simulation_logs/index.html ... done
Updating c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/
part_4/apps/auction_simulation/simulation_logs/all_runs.html ... done
```

```
=== Test Info ===
Test(s) on node 'ct@DESKTOP-N3ROL9Q.lan' finished.
=== Test Info ===
Test(s) on node 'client@DESKTOP-N3ROL9Q.lan' finished.
=== Test Info ===
Test(s) on node 'auction_server@DESKTOP-N3ROL9Q.lan' finished.
=== TEST RESULTS ===
ct@DESKTOP-N3ROL9Q.lan_________________{0,0,{0,0}}
client@DESKTOP-N3ROL9Q.lan_____________{1,0,{0,0}}
auction_server@DESKTOP-N3ROL9Q.lan______{1,0,{0,0}}

=== Info ===
Updating log files
Updating c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/
part_4/apps/auction_simulation/simulation_logs/index.html ... done
Updating c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/
part_4/apps/auction_simulation/simulation_logs/all_runs.html ... done
Logs in c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/
part_4/apps/auction_simulation/simulation_logs refreshed!
=== Info ===
Refreshing logs in "c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/
practical/part_4/apps/auction_simulation/simulation_logs"... ok
[{["c:/Users/lao8n/OneDrive/Documents/oxford_cpr/assignment/practical/
part_4/apps/auction_simulation/auction_simulation.spec"],
  ok}]
```

I was not able to get a fully distributed system with multiple clients working in the time because this would have involved having `application:start` accept multiple different arguments for the different bidder names but this working example shows that the system can work in a distributed manner and then you can run the Part 4 tests with

```
cd practical/part_4
rebar3 ct
```

where the output is

```
===> Verifying dependencies...
===> Analyzing applications...
===> Compiling auction_data
===> Compiling auction_client
===> Compiling auction_server
===> Running Common Test suites...
%%% auction_client_server_SUITE: ..........
%%% auction_client_server_sup_SUITE: ..
%%% auction_data_SUITE: ..........
%%% auction_SUITE: .........
%%% auction_sup_SUITE: ...
%%% pubsub_SUITE: ...........
%%% pubsub_sup_SUITE: .
All 46 tests passed.
```

Theoretical

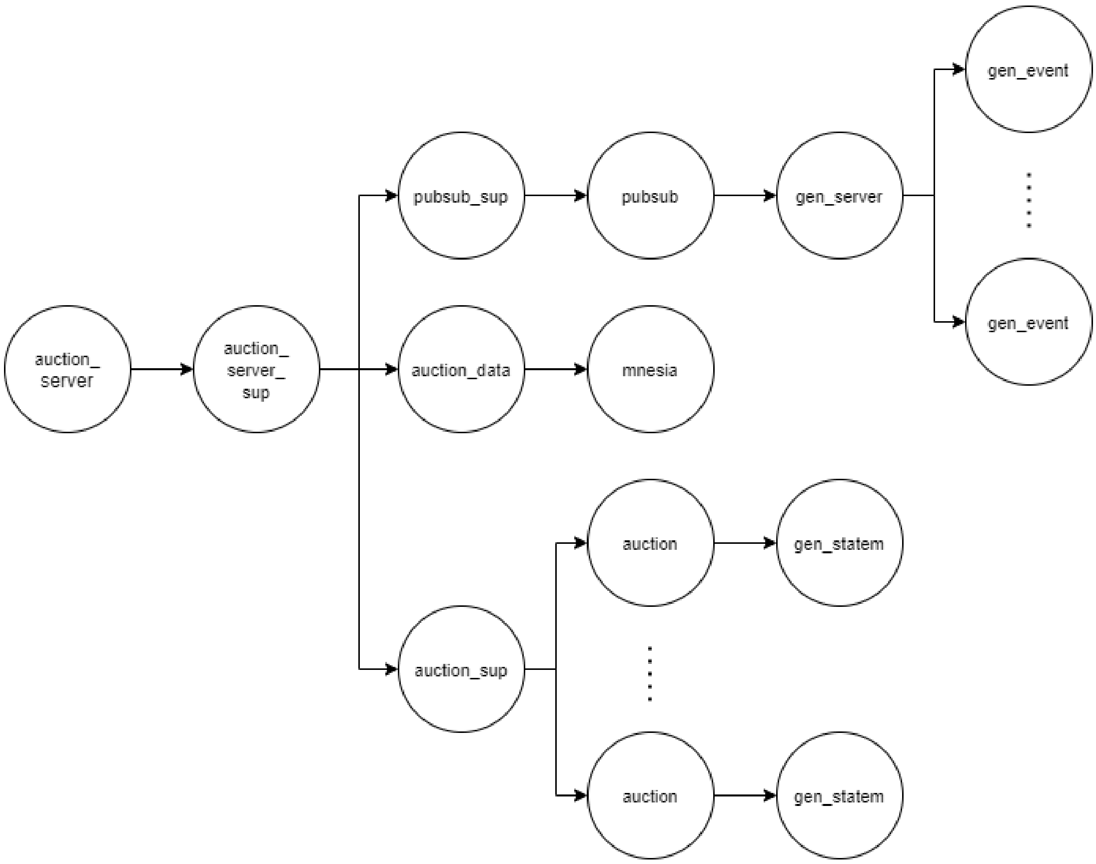
We have not implemented any fault tolerance in your system. How would you model dependencies in a supervision tree and make sure there is no single point of failure in your system?

Supervisors are processes whose only task is to monitor and manage children[14]. Fault-tolerance is achieved by creating supervision trees, where the supervisors are the nodes and the workers are the leaves[14].

In our system, both the `pubsub` and `auction` processes are managed by supervisors with `one_to_one`[1] and `simple_one_to_one`[2] strategies and are in turn managed by `auction_server_sup` with a `rest_for_one`[3] strategy. The restart strategy is determined by the `intensity`, which is the maximum number of restarts in a set `period` before the supervisor terminates the child.

We also have fault-tolerance in `auction_data` as use Mnesia's `disc_copies` to store the data both in ETS and on disk.

This 'let it crash' philosophy helps ensure fault-tolerance by killing processes as fast as possible and not letting the error propagate through the system[1]. Although the top supervisor `auction_server_sup` is not itself supervised its only job is to supervise and has no implementation code in it therefore is less likely to fail. Therefore we have removed any single point of failure in all parts of the system (except the supervision of the system itself).



[1]Only the crashed process is restarted
[2]Only the crashed process is restarted but where the children are dynamic and of the same type
[3]All processes started after the crashed one are terminated and restarted

There are race conditions in your system, such as (but not limited to) identical bids arriving at the same time. How are they handled? Can they be exploited?

A race condition is when the conditions of a system are dependent upon the sequence or timing of other uncontrollable events[20]. Race conditions from simultaneous bids are demonstrated in the test

```
cd practical/part_4/
[{group, race_integ}]. % uncomment on line 27 as we expect this to fail
rebar3 ct --suite apps/auction_client/test/race_conditions_SUITE
```

where we run two bidders in parallel ten times. `race_bidder_1` and `race_bidder_2` in `[parallel, {repeat, 10}]`

What we find is that about half the time `race_bidder_1` gets the bid returning `{ok, leading}` and the other half the time `race_bidder_2` does and hence in a trial run we found

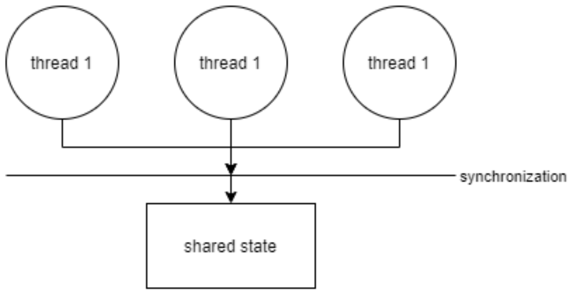
```
Failed 16 tests. Passed 14 tests.
```

This kind of race condition is acceptable because in erlang message receiving is an atomic operation with locks on the mailbox[21]. This means there is no risk that one `bid` might be part-processed and then another `bid` is processed. We handle the most significant race condition of updating the database by using `mnesia` transactions which is important when we mix `read` and `write` operations as we have done in `auction_data:add_items/2`.

One way to potentially try and exploit the system is to send a valid leading bid - and then send lots(!) of invalid bids which will need to be processed but will not reset the `gen_statem` `state_timeout`. This could then mean that a competing bid might not be processed in time.

How does a shared memory concurrency model (such as threads) compare and contrast to a no shared memory model (such as the actor model or Erlang style concurrency)?

There is a trade-off between reliability[4] and availability[5]. Shared memory concurrency models can make a system more reliable as any node can take over a failed request, whereas share nothing architectures are more fault tolerant as errors do not propagate from one part of the system to another[1].



In shared systems synchronization primitives are required like locks, mutexes, semaphores and monitors[22] which greatly complicates the programming model and can lead to further issues with lock contention, thread starvation and deadlocks.

On the other hand, with share-nothing systems each actor maintains its own separate memory asynchronously processing one message at a time removing the need for synchronization or locks and avoiding the risk of blocking other processes. The only way that data is shared is via messages which maps to how modern memory hierarchies work both at the cache line level and for remote communication[23] helping actor systems scale and distribute naturally.

[4] Where under particular, predefined conditions, errors included, your system continues to function. If a node is unresponsive because it has terminated, is slow, or got separated from the rest of the system in a network partition, your business logic should be capable of redirecting the request to a responsive node[14]

[5] The uptime of a system over a certain period of time where high availability refers to systems with very low downtime, software maintenance and upgrades included[14]

How do you handle bottlenecks when working with no shared memory concurrency models?

Bottlenecks can be found by monitoring process memory usage and mailbox queues using the `erlang:memory()` BIF[14].
There are a number of ways to deal with them.

- The number of reductions it costs to send a message is increased to slow a producer through a peak and allow the consumers to catch up[14].
- Another is to use synchronous calls even if no response is needed, again blocking the producer as a form of one-to-one back pressure[14].
- Use backpressure with load regulation where requests are rejected.
- Another strategy is to reduce the workload of the consumers, perhaps passing the work to the client[14].
- If the process can be parallelized spawn more children processes as we did with `auction` or as is done with pools of connections found in libraries like `ranch`.
- If data access is the bottleneck then use database like ETS which allow for parallel and concurrent access of the data[1].

References

[1] Fred Hebert. *Learn You Some Erlang for Great Good! A Beginner's Guide.* No Starch Press, USA, 2013.

[2] Stackoverflow, mnesia tables must have at least one attribute. https://stackoverflow.com/questions/41853702/erlang-mnesia-not-working-with-size-one-record .

[3] Stackoverflow, mnesia:transaction vs mnesia:activity(transaction, f). https://stackoverflow.com/questions/19611961/erlang-mnesia-mnesiatransactionf-vs-mnesiaactivitytransaction-f: :text=Second.

[4] Erlang documentation, make_ref known issue. https://erlang.org/doc/man/erlang.html#make$_ref-$0.

[5] Stackoverflow, references are not unique. https://stackoverflow.com/questions/10029154/are-erlang-refs-unique-between-nodes-vm-restarts.

[6] Erlang documentation: Unique references on a runtime system instance. https://erlang.org/doc/efficiency_guide/advanced.html#unique_references.

[7] Stackoverflow, do not interpret references. https://stackoverflow.com/questions/26337647/what-is-ref-in-erlang/26337986.

[8] Erlang documentation, ets traversal. https://erlang.org/doc/man/ets.html.

[9] Erlang documentation, mnesia dirty update. http://erlang.org/doc/man/mnesia.html# dirty_update_counter-3.

[10] Mnesia article ondirty update. https://www.programmersought.com/article/4597164850/.

[11] Erlang questions, mnesia dirty update. https://erlang-questions.erlang.narkive.com/GHaRicLJ/mnesia-dirty-update-counter-and-replicated-tables .

[12] Stackoverflow unique integer is the same. https://stackoverflow.com/questions/38882828/erlang-unique-integer-returns-the-same-integer-after-restart.

[13] Distributed erlang. https://erlang.org/doc/reference_manual/distributed.html.

[14] Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems.* O'Reilly Media, Inc., 1st edition, 2016.

[15] Erlang gen_statem documentation. https://erlang.org/doc/design$_principles/statem.html$.

[16] Erlang gen_fsm deprecated. https://erlang.org/doc/man/gen_fsm.html.

[17] Erlang selective receive. https://www.tutorialspoint.com/erlang/erlang_concurrency.htm: :text=Selectivel.

[18] Erlang nested transactions documentation. http://erlang.org/documentation/doc-6.0/lib/mnesia-4.12/doc/html/Mnesia$_chap4.htmlid$74395.

[19] Gen server calls to itself. https://stackoverflow.com/questions/41125837/proper-way-to-structure-genserver-calls-to-self/41128268.

[20] Wikipedia race condition definition. https://en.wikipedia.org/wiki/Race_condition .

[21] Stackoverflow erlang mailbox. https://stackoverflow.com/questions/65902202/how-do-erlang-akka-etc-send-messages-under-the-hood-why-doesnt-it-lead-to-dea/6590314365903143 .

[22] Shared vs non-shared. https://www.javacodegeeks.com/2016/10/developing-modern-applications-scala-concurrency-parallelism-akka.html .

[23] State is not shared in modern memory systems. https://doc.akka.io/docs/akka/current/typed/guide/ actors-intro.html .