GAME TRADING BETWEEN TWO PLAYERS

The first thing that needs to be done to implement our protocol with OTP's `gen_fsm` is to create the interface. There will be 3 callers for our module: the player, the `gen_fsm` behaviour and the other player's FSM. We will only need to export the player function and `gen_fsm` functions, though. This is because the other FSM will also run within the trade_fsm module and can access them from the inside:

```
-module(trade_fsm).
-behaviour(gen_fsm).

%% public API
-export([start/1, start_link/1, trade/2, accept_trade/1,
make_offer/2, retract_offer/2, ready/1, cancel/1]).
%% gen_fsm callbacks
-
export([init/1, handle_event/3, handle_sync_event/4, handle
_info/3,
terminate/3, code_change/4,
% custom state names
idle/2, idle/3, idle_wait/2, idle_wait/3, negotiate/2,
negotiate/3, wait/2, ready/2, ready/3]).
```

So that's our API. You can see I'm planning on having some functions being both synchronous and asynchronous. This is mostly because we want our client to call us synchronously in some cases, but the other FSM can do it asynchronously. Having the client synchronous simplifies our logic a whole lot by limiting the number of contradicting messages that can be sent one after the other. We'll get there. Let's first implement the actual public API according to the protocol defined above:

```
%%% PUBLIC API
start(Name) ->
gen_fsm:start(?MODULE, [Name], []).

start_link(Name) ->
gen_fsm:start_link(?MODULE, [Name], []).

%% ask for a begin session. Returns when/if the other
accepts
trade(OwnPid, OtherPid) ->
gen_fsm:sync_send_event(OwnPid, {negotiate,
OtherPid}, 30000).

%% Accept someone's trade offer.
accept_trade(OwnPid) ->
gen_fsm:sync_send_event(OwnPid, accept_negotiate).
```

```
%% Send an item on the table to be traded
make_offer(OwnPid, Item) ->
gen_fsm:send_event(OwnPid, {make_offer, Item}).

%% Cancel trade offer
retract_offer(OwnPid, Item) ->
gen_fsm:send_event(OwnPid, {retract_offer, Item}).

%% Mention that you're ready for a trade. When the other
%% player also declares being ready, the trade is done
ready(OwnPid) ->
gen_fsm:sync_send_event(OwnPid, ready, infinity).

%% Cancel the transaction.
cancel(OwnPid) ->
gen_fsm:sync_send_all_state_event(OwnPid, cancel).
```

This is rather standard; all these 'gen_fsm' functions have been covered before (except start/3-4 and start_link/3-4which I believe you can figure out) in this chapter.

Next we'll implement the FSM to FSM functions. The first ones have to do with trade setups, when we first want to ask the other user to join us in a trade:

```
%% Ask the other FSM's Pid for a trade session
ask_negotiate(OtherPid, OwnPid) ->
gen_fsm:send_event(OtherPid, {ask_negotiate, OwnPid}).

%% Forward the client message accepting the transaction
accept_negotiate(OtherPid, OwnPid) ->
gen_fsm:send_event(OtherPid, {accept_negotiate, OwnPid}).
```

The first function asks the other pid if they want to trade, and the second one is used to reply to it (asynchronously, of course).

We can then write the functions to offer and cancel offers. According to our protocol above, this is what they should be like:

```
%% forward a client's offer
do_offer(OtherPid, Item) ->
gen_fsm:send_event(OtherPid, {do_offer, Item}).

%% forward a client's offer cancellation
undo_offer(OtherPid, Item) ->
gen_fsm:send_event(OtherPid, {undo_offer, Item}).
```

So, now that we've got these calls done, we need to focus on the rest. The remaining calls relate to being ready or not and handling the final commit. Again, given our protocol above, we have three calls: `are_you_ready`, which can have the replies `not_yet` or `ready!`:

```
%% Ask the other side if he's ready to trade.
are_you_ready(OtherPid) ->
gen_fsm:send_event(OtherPid, are_you_ready).

%% Reply that the side is not ready to trade
%% i.e. is not in 'wait' state.
not_yet(OtherPid) ->
gen_fsm:send_event(OtherPid, not_yet).

%% Tells the other fsm that the user is currently waiting
%% for the ready state. State should transition to 'ready'
am_ready(OtherPid) ->
gen_fsm:send_event(OtherPid, 'ready!').
```

The only functions left are those which are to be used by both FSMs when doing the commit in the `ready` state. Their precise usage will be described more in detail later, but for now, the names and the sequence/state diagram from earlier should be enough. Nonetheless, you can still transcribe them to your own version of trade_fsm:

```
%% Acknowledge that the fsm is in a ready state.
ack_trans(OtherPid) ->
gen_fsm:send_event(OtherPid, ack).

%% ask if ready to commit
ask_commit(OtherPid) ->
gen_fsm:sync_send_event(OtherPid, ask_commit).

%% begin the synchronous commit
do_commit(OtherPid) ->
gen_fsm:sync_send_event(OtherPid, do_commit).
```

Oh and there's also the courtesy function allowing us to warn the other FSM we cancelled the trade:

```
notify_cancel(OtherPid) ->
gen_fsm:send_all_state_event(OtherPid, cancel).
```

We can now move to the really interesting part: the `gen_fsm` callbacks. The first callback is `init/1`. In our case, we'll want each FSM to hold a name for the user it represents (that way, our output will be nicer) in the data it keeps passing on to itself. What else do we want to hold in memory? In our case, we want the other's pid, the items we offer and the items the other offers. We're also going to add the reference of a monitor (so we know to abort if the other dies) and a `from` field, used to do delayed replies:

```
-record(state, {name="",
other,
```

```
ownitems=[],
otheritems=[],
monitor,
from}).
```

In the case of `init/1`, we'll only care about our name for now. Note that we'll begin in the `idle` state:

```
init(Name) ->
{ok, idle, #state{name=Name}}.
```

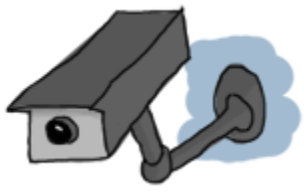
The next callbacks to consider would be the states themselves. So far I've described the state transitions and calls that can be made, but We'll need a way to make sure everything goes alright. We'll write a few utility functions first:

```
%% Send players a notice. This could be messages to their
clients
%% but for our purposes, outputting to the shell is enough.
notice(#state{name=N}, Str, Args) ->
io:format("~s: "++Str++"~n", [N|Args]).

%% Unexpected allows to log unexpected messages
unexpected(Msg, State) ->
io:format("~p received unknown event ~p while in state
~p~n",
[self(), Msg, State]).
```

And we can start with the idle state. For the sake of convention, I'll cover the asynchronous version first. This one shouldn't need to care for anything but the other player asking for a trade given our own player, if you look at the API functions, will use a synchronous call:

```
idle({ask_negotiate, OtherPid}, S=#state{}) ->
Ref = monitor(process, OtherPid),
notice(S, "~p asked for a trade negotiation", [OtherPid]),
{next_state, idle_wait,
S#state{other=OtherPid, monitor=Ref}};
idle(Event, Data) ->
unexpected(Event, idle),
{next_state, idle, Data}.
```



A monitor is set up to allow us to handle the other dying, and its ref is stored in the FSM's data along with the other's pid, before moving to the `idle_wait` state. Note that we will report all unexpected events and ignore

them by staying in the state we were already in. We can have a few out of band messages here and there that could be the result of race conditions. It's usually safe to ignore them, but we can't easily get rid of them. It's just better not to crash the whole FSM on these unknown, but somewhat expected messages.

When our own client asks the FSM to contact another player for a trade, it will send a synchronous event. The `idle/3` callback will be needed:

```
idle({negotiate, OtherPid}, From, S=#state{}) ->
ask_negotiate(OtherPid, self()),
notice(S, "asking user ~p for a trade", [OtherPid]),
Ref = monitor(process, OtherPid),
{next_state, idle_wait,
S#state{other=OtherPid, monitor=Ref, from=From}};
idle(Event, _From, Data) ->
unexpected(Event, idle),
{next_state, idle, Data}.
```

We proceed in a way similar to the asynchronous version, except we need to actually ask the other side whether they want to negotiate with us or not. You'll notice that we do *not* reply to the client yet. This is because we have nothing interesting to say, and we want the client locked and waiting for the trade to be accepted before doing anything. The reply will only be sent if the other side accepts once we're in `idle_wait`.

When we're there, we have to deal with the other accepting to negotiate and the other asking to negotiate (the result of a race condition, as described in the protocol):

```
idle_wait({ask_negotiate, OtherPid},
S=#state{other=OtherPid}) ->
gen_fsm:reply(S#state.from, ok),
notice(S, "starting negotiation", []),
{next_state, negotiate, S};
%% The other side has accepted our offer. Move to negotiate
state
idle_wait({accept_negotiate, OtherPid},
S=#state{other=OtherPid}) ->
gen_fsm:reply(S#state.from, ok),
notice(S, "starting negotiation", []),
{next_state, negotiate, S};
idle_wait(Event, Data) ->
unexpected(Event, idle_wait),
{next_state, idle_wait, Data}.
```

This gives us two transitions to the `negotiate` state, but remember that we must use `gen_fsm:reply/2` reply to our client to tell it it's okay to start offering items. There's also the case of our FSM's client accepting the trade suggested by the other party:

```
idle_wait(accept_negotiate, _From,
S=#state{other=OtherPid}) ->
accept_negotiate(OtherPid, self()),
notice(S, "accepting negotiation", []),
{reply, ok, negotiate, S};
idle_wait(Event, _From, Data) ->
unexpected(Event, idle_wait),
{next_state, idle_wait, Data}.
```

Again, this one moves on to the `negotiate` state. Here, we must handle asynchronous queries to add and remove items coming both from the client and the other FSM. However, we have not yet decided how to store items. Because I'm somewhat lazy and I assume users won't trade that many items, simple lists will do it for now. However, we might change our mind at a later point, so it would be a good idea to wrap item operations in their own functions. Add the following functions at the bottom of the file with `notice/3` and `unexpected/2`:

```
%% adds an item to an item list
add(Item, Items) ->
[Item | Items].

%% remove an item from an item list
remove(Item, Items) ->
Items -- [Item].
```

Simple, but they have the role of isolating the actions (adding and removing items) from their implementation (using lists). We could easily move to proplists, arrays or whatever data structure without disrupting the rest of the code.

Using both of these functions, we can implement offering and removing items:

```
negotiate({make_offer, Item},
S=#state{ownitems=OwnItems}) ->
do_offer(S#state.other, Item),
notice(S, "offering ~p", [Item]),
{next_state, negotiate, S#state{ownitems=add(Item,
OwnItems)}};
%% Own side retracting an item offer
negotiate({retract_offer, Item},
S=#state{ownitems=OwnItems}) ->
undo_offer(S#state.other, Item),
notice(S, "cancelling offer on ~p", [Item]),
{next_state, negotiate, S#state{ownitems=remove(Item,
OwnItems)}};
%% other side offering an item
```

```
negotiate({do_offer, Item},
S=#state{otheritems=OtherItems}) ->
notice(S, "other player offering ~p", [Item]),
{next_state, negotiate, S#state{otheritems=add(Item,
OtherItems)}};
%% other side retracting an item offer
negotiate({undo_offer, Item},
S=#state{otheritems=OtherItems}) ->
notice(S, "Other player cancelling offer on ~p", [Item]),
{next_state, negotiate, S#state{otheritems=remove(Item,
OtherItems)}};
```

This is an ugly aspect of using asynchronous messages on both sides. One set of message has the form 'make' and 'retract', while the other has 'do' and 'undo'. This is entirely arbitrary and only used to differentiate between player-to-FSM communications and FSM-to-FSM communications. Note that on those coming from our own player, we have to tell the other side about the changes we're making.

Another responsibility is to handle the `are_you_ready` message we mentioned in the protocol. This one is the last asynchronous event to handle in the `negotiate` state:

```
negotiate(are_you_ready, S=#state{other=OtherPid}) ->
io:format("Other user ready to trade.~n"),
notice(S,
"Other user ready to transfer goods:~n"
"You get ~p, The other side gets ~p",
[S#state.otheritems, S#state.ownitems]),
not_yet(OtherPid),
{next_state, negotiate, S};
negotiate(Event, Data) ->
unexpected(Event, negotiate),
{next_state, negotiate, Data}.
```

As described in the protocol, whenever we're not in the `wait` state and receive this message, we must reply with`not_yet`. Were also outputting trade details to the user so a decision can be made.

When such a decision is made and the user is ready, the `ready` event will be sent. This one should be synchronous because we don't want the user to keep modifying his offer by adding items while claiming he's ready:

```
negotiate(ready, From, S = #state{other=OtherPid}) ->
are_you_ready(OtherPid),
notice(S, "asking if ready, waiting", []),
{next_state, wait, S#state{from=From}};
negotiate(Event, _From, S) ->
unexpected(Event, negotiate),
{next_state, negotiate, S}.
```

At this point a transition to the `wait` state should be made. Note that just waiting for the other is not interesting. We save the *`From`* variable so we can use it with `gen_fsm:reply/2` when we have something to tell to the client.

The `wait` state is a funny beast. New items might be offered and retracted because the other user might not be ready. It makes sense, then, to automatically rollback to the negotiating state. It would suck to have great items offered to us, only for the other to remove them and declare himself ready, stealing our loot. Going back to negotiation is a good decision:

```
wait({do_offer, Item}, S=#state{otheritems=OtherItems}) ->
gen_fsm:reply(S#state.from, offer_changed),
notice(S, "other side offering ~p", [Item]),
{next_state, negotiate, S#state{otheritems=add(Item,
OtherItems)}};
wait({undo_offer, Item}, S=#state{otheritems=OtherItems}) -
>
gen_fsm:reply(S#state.from, offer_changed),
notice(S, "Other side cancelling offer of ~p", [Item]),
{next_state, negotiate, S#state{otheritems=remove(Item,
OtherItems)}};
```

Now that's something meaningful and we reply to the player with the coordinates we stored

in *`S#state.from`*. The next set of messages we need to worry about are those related to with synchronising both FSMs so they can move to the `ready` state and confirm the trade. For this one we should really focus on the protocol defined earlier.

The three messages we could have are `are_you_ready` (because the other user just declared himself ready), `not_yet` (because we asked the other if he was ready and he was not) and `ready!` (because we asked the other if he was ready and he was).

We'll start with `are_you_ready`. Remember that in the protocol we said that there could be a race condition hidden there. The only thing we can do is send the`ready!` message with `am_ready/1` and deal with the rest later:

```
wait(are_you_ready, S=#state{}) ->
am_ready(S#state.other),
notice(S, "asked if ready, and I am. Waiting for same
reply", []),
```

```
{next_state, wait, S};
```

We'll be stuck waiting again, so it's not worth replying to our client yet. Similarly, we won't reply to the client when the other side sends a `not_yet` to our invitation:

```
wait(not_yet, S = #state{}) ->
notice(S, "Other not ready yet", []),
{next_state, wait, S};
```

On the other hand, if the other is ready, we send an extra `ready!` message to the other FSM, reply to our own user and then move to the `ready` state:

```
wait('ready!', S=#state{}) ->
am_ready(S#state.other),
ack_trans(S#state.other),
gen_fsm:reply(S#state.from, ok),
notice(S, "other side is ready. Moving to ready state",
[]),
{next_state, ready, S};
%% DOn't care about these!
wait(Event, Data) ->
unexpected(Event, wait),
{next_state, wait, Data}.
```

You might have noticed that I've used `ack_trans/1`. In fact, both FSMs should use it. Why is this? To understand this we have to start looking at what goes on in the `ready!` state.

When in the ready state, both players' actions become useless (except cancelling). We won't care about new item offers. This gives us some liberty. Basically, both FSMs can freely talk to each other without worrying about the rest of the world. This lets us implement our bastardization of a two-phase commit. To begin this commit without either player acting, we'll need an event to trigger an action from the FSMs. The `ack` event from`ack_trans/1` is used for that. As soon as we're in the ready state, the message is treated and acted upon; the transaction can begin.

Two-phase commits require synchronous communications, though. This means we can't have both FSMs starting the transaction at once, because they'll end up deadlocked. The secret is to find a way to decide that one finite state machine should initiate the commit, while the other will sit and wait for orders from the first one.

It turns out that the engineers and computer scientists who designed Erlang were pretty smart (well, we knew that already). The pids of any process can be compared to each other and sorted. This can be done no matter when the process was spawned, whether it's still alive or not, or if it comes from another VM (we'll see more about this when we get into distributed Erlang).

Knowing that two pids can be compared and one will be greater than the other, we can write a function `priority/2`that will take two pids and tell a process whether it's been elected or not:

```
priority(OwnPid, OtherPid) when OwnPid > OtherPid -> true;
priority(OwnPid, OtherPid) when OwnPid < OtherPid -> false.
```

And by calling that function, we can have one process starting the commit and the other following the orders.

Here's what this gives us when included in the `ready` state, after receiving the `ack` message:

```
ready(ack, S=#state{}) ->
case priority(self(), S#state.other) of
true ->
try
notice(S, "asking for commit", []),
ready_commit = ask_commit(S#state.other),
notice(S, "ordering commit", []),
ok = do_commit(S#state.other),
notice(S, "committing...", []),
commit(S),
{stop, normal, S}
catch Class:Reason ->
%% abort! Either ready_commit or do_commit failed
notice(S, "commit failed", []),
{stop, {Class, Reason}, S}
end;
false ->
{next_state, ready, S}
end;
ready(Event, Data) ->
unexpected(Event, ready),
{next_state, ready, Data}.
```

This big `try ... catch` expression is the leading FSM deciding how the commit works.

Both `ask_commit/1` and`do_commit/1` are synchronous. This lets the leading FSM call them freely. You can see that the other FSM just goes and wait. It will then receive the orders from the leading process. The first message should be `ask_commit`. This is just to make sure both FSMs are still there; nothing wrong happened, they're both dedicated to completing the task:

```
ready(ask_commit, _From, S) ->
notice(S, "replying to ask_commit", []),
```

```
{reply, ready_commit, ready, S};
```

Once this is received, the leading process will ask to confirm the transaction with `do_commit`. That's when we must commit our data:

```
ready(do_commit, _From, S) ->
notice(S, "committing...", []),
commit(S),
{stop, normal, ok, S};
ready(Event, _From, Data) ->
unexpected(Event, ready),
{next_state, ready, Data}.
```

And once it's done, we leave. The leading FSM will receive `ok` as a reply and will know to commit on its own end afterwards. This explains why we need the big `try ... catch`: if the replying FSM dies or its player cancels the transaction, the synchronous calls will crash after a timeout. The commit should be aborted in this case.

Just so you know, I defined the commit function as follows:

```
commit(S = #state{}) ->
io:format("Transaction completed for ~s. "
"Items sent are:~n~p,~n received are:~n~p.~n"
"This operation should have some atomic save "
"in a database.~n",
[S#state.name, S#state.ownitems, S#state.otheritems]).
```

Pretty underwhelming, eh? It's generally not possible to do a true safe commit with only two participants—a third party is usually required to judge if both players did everything right. If you were to write a true commit function, it should contact that third party on behalf of both players, and then do the safe write to a database for them or rollback the whole exchange. We won't go into such details and the current `commit/1` function will be enough for the needs of this book.

We're not done yet. We have not yet covered two types of events: a player cancelling the trade and the other player's finite state machine crashing. The former can be dealt with by using the callbacks `handle_event/3` and`handle_sync_event/4`. Whenever the other user cancels, we'll receive an asynchronous notification:

```
%% The other player has sent this cancel event
%% stop whatever we're doing and shut down!
handle_event(cancel, _StateName, S=#state{}) ->
notice(S, "received cancel event", []),
{stop, other_cancelled, S};
handle_event(Event, StateName, Data) ->
unexpected(Event, StateName),
{next_state, StateName, Data}.
```

When we do it we must not forget to tell the other before quitting ourselves:

```
%% This cancel event comes from the client. We must warn
the other
%% player that we have a quitter!
handle_sync_event(cancel, _From, _StateName, S = #state{}) -
>
notify_cancel(S#state.other),
notice(S, "cancelling trade, sending cancel event", []),
{stop, cancelled, ok, S};
%% Note: DO NOT reply to unexpected calls. Let the call-
maker crash!
handle_sync_event(Event, _From, StateName, Data) ->
unexpected(Event, StateName),
{next_state, StateName, Data}.
```

And voilà! The last event to take care of is when the other FSM goes down. Fortunately, we had set a monitor back in the `idle` state. We can match on this and react accordingly:

```
handle_info({'DOWN', Ref, process, Pid, Reason}, _,
S=#state{other=Pid, monitor=Ref})->
notice(S, "Other side dead", []),
{stop, {other_down, Reason}, S};
handle_info(Info, StateName, Data) ->
unexpected(Info, StateName),
{next_state, StateName, Data}.
```

Note that even if the `cancel` or `DOWN` events happen while we're in the commit, everything should be safe and nobody should get its items stolen.

Note: we used `io:format/2` for most of our messages to let the FSMs communicate with their own clients. In a real world application, we might want something more flexible than that. One way to do it is to let the client send in a Pid, which will receive the notices sent to it. That process could be linked to a GUI or any other system to make the player aware of the events. The `io:format/2` solution was chosen for its simplicity: we want to focus on the FSM and the asynchronous protocols, not the rest.

Only two callbacks left to cover! They're `code_change/4` and `terminate/3`. For now, we don't have anything to do with `code_change/4` and only export it so the next version of the FSM can call it when it'll be reloaded. Our terminate function is also really short because we didn't handle real resources in this example:

```
code_change(_OldVsn, StateName, Data, _Extra) ->
{ok, StateName, Data}.

%% Transaction completed.
terminate(normal, ready, S=#state{}) ->
notice(S, "FSM leaving.", []);
terminate(_Reason, _StateName, _StateData) ->
ok.
```

Whew.

We can now try it. Well, trying it is a bit annoying because we need two processes to communicate to each other. To solve this, I've written the tests in the file trade_calls.erl, which can run 3 different scenarios. The first one is`main_ab/0`. It will run a standard trade and output everything. The second one is `main_cd/0` and will cancel the transaction halfway through. The last one is `main_ef/0` and is very similar to `main_ab/0`, except it contains a different race condition. The first and third tests should succeed, while the second one should fail (with a crapload of error messages, but that's how it goes). You can try it if you feel like it.

Source : http://learnyousomeerlang.com/finite-state-machines