THE EVENT SERVER IN ERLANG

Let's deal with the event server. According to the protocol, the skeleton for that one should look a bit like this:

```
-module(evserv).
-compile(export_all).

loop(State) ->
receive
{Pid, MsgRef, {subscribe, Client}} ->
...
{Pid, MsgRef, {add, Name, Description, TimeOut}} ->
...
{Pid, MsgRef, {cancel, Name}} ->
...
{done, Name} ->
...
shutdown ->
...
{'DOWN', Ref, process, _Pid, _Reason} ->
...
code_change ->
...
Unknown ->
io:format("Unknown message: ~p~n",[Unknown]),
loop(State)
end.
```

You'll notice I have wrapped calls that require replies with the same `{Pid, Ref, Message}` format as earlier. Now, the server will need to keep two things in its state: a list of subscribing clients and a list of all the event processes it spawned. If you have noticed, the protocol says that when an event is done, the event server should receive `{done, Name}`, but send `{done, Name, Description}`. The idea here is to have as little traffic as necessary and only have the event processes care about what is strictly necessary. So yeah, list of clients and list of events:

```
-record(state, {events,    %% list of #event{} records
clients}). %% list of Pids

-record(event, {name="",
description="",
pid,
timeout={{1970,1,1},{0,0,0}}}).
```

And the loop now has the record definition in its head:

```
loop(S =#state{}) ->
receive
```

```
...
end.
```

It would be nice if both events and clients were orddicts. We're unlikely to have many hundreds of them at once. If you recall the chapter on data structures, orddicts fit that need very well. We'll write an `init` function to handle this:

```
init() ->
%% Loading events from a static file could be done here.
%% You would need to pass an argument to init telling where
the
%% resource to find the events is. Then load it from here.
%% Another option is to just pass the events straight to
the server
%% through this function.
loop(#state{events=orddict:new(),
clients=orddict:new()}).
```

With the skeleton and initialization done, I'll implement each message one by one. The first message is the one about subscriptions. We want to keep a list of all subscribers because when an event is done, we have to notify them. Also, the protocol above mentions we should monitor them. It makes sense because we don't want to hold onto crashed clients and send useless messages for no reason. Anyway, it should look like this:

```
{Pid, MsgRef, {subscribe, Client}} ->
Ref = erlang:monitor(process, Client),
NewClients = orddict:store(Ref, Client, S#state.clients),
Pid ! {MsgRef, ok},
loop(S#state{clients=NewClients});
```

So what this section of `loop/1` does is start a monitor, and store the client info in the orddict under the key *`Ref`*. The reason for this is simple: the only other time we'll need to fetch the client ID will be if we receive a monitor's `EXIT` message, which will contain the reference (which will let us get rid of the orddict's entry).

The next message to care about is the one where we add events. Now, it is possible to return an error status. The only validation we'll do is check the timestamps we accept. While it's easy to subscribe to the `{{Year,Month,Day}, {Hour,Minute,seconds}}` layout, we have to make sure we don't do things like accept events on February 29 when we're not in a leap year, or any other date that doesn't exist. Moreover, we don't want to accept impossible date values such as "5 hours, minus 1 minute and 75 seconds". A single function can take care of validating all of that.

The first building block we'll use is the function `calendar:valid_date/1`. This one, as the name says, checks if the date is valid or not. Sadly, the weirdness of the calendar module doesn't stop at funky names: there is actually no function to confirm that `{H,M,S}` has valid values. We'll have to implement that one too, following the funky naming scheme:

```
valid_datetime({Date,Time}) ->
try
```

```
calendar:valid_date(Date) andalso valid_time(Time)
catch
error:function_clause -> %% not in {{Y,M,D},{H,Min,S}} format
false
end;
valid_datetime(_) ->
false.
```

```
valid_time({H,M,S}) -> valid_time(H,M,S).
valid_time(H,M,S) when H >= 0, H < 24,
M >= 0, M < 60,
S >= 0, S < 60 -> true;
valid_time(_,_,_) -> false.
```

The `valid_datetime/1` function can now be used in the part where we try to add the message:

```
{Pid, MsgRef, {add, Name, Description, TimeOut}} ->
case valid_datetime(TimeOut) of
true ->
EventPid = event:start_link(Name, TimeOut),
NewEvents = orddict:store(Name,
#event{name=Name,
description=Description,
pid=EventPid,
timeout=TimeOut},
S#state.events),
Pid ! {MsgRef, ok},
loop(S#state{events=NewEvents});
false ->
Pid ! {MsgRef, {error, bad_timeout}},
loop(S)
end;
```

If the time is valid, we spawn a new event process, then store its data in the event server's state before sending a confirmation to the caller. If the timeout is wrong, we notify the client rather than having the error pass silently or crashing the server. Additional checks could be added for name clashes or other restrictions (just remember to update the protocol documentation!)

The next message defined in our protocol is the one where we cancel an event. Canceling an event never fails on the client side, so the code is simpler there. Just check whether the event is in the process' state record. If it is, use the `event:cancel/1` function we defined to kill it and send ok. If it's not found, just tell the user everything went right anyway -- the event is not running and that's what the user wanted.

```
{Pid, MsgRef, {cancel, Name}} ->
Events = case orddict:find(Name, S#state.events) of
{ok, E} ->
```

```
event:cancel(E#event.pid),
orddict:erase(Name, S#state.events);
error ->
S#state.events
end,
Pid ! {MsgRef, ok},
loop(S#state{events=Events});
```

Good, good. So now all voluntary interaction coming from the client to the event server is covered. Let's deal with the stuff that's going between the server and the events themselves. There are two messages to handle: canceling the events (which is done), and the events timing out. That message is simply `{done, Name}`:

```
{done, Name} ->
case orddict:find(Name, S#state.events) of
{ok, E} ->
send_to_clients({done, E#event.name, E#event.description},
S#state.clients),
NewEvents = orddict:erase(Name, S#state.events),
loop(S#state{events=NewEvents});
error ->
%% This may happen if we cancel an event and
%% it fires at the same time
loop(S)
end;
```

And the function `send_to_clients/2` does as its name says and is defined as follows:

```
send_to_clients(Msg, ClientDict) ->
orddict:map(fun(_Ref, Pid) -> Pid ! Msg end, ClientDict).
```

That should be it for most of the loop code. What's left is the set different status messages: clients going down, shutdown, code upgrades, etc. Here they come:

```
shutdown ->
exit(shutdown);
{'DOWN', Ref, process, _Pid, _Reason} ->
loop(S#state{clients=orddict:erase(Ref, S#state.clients)});
code_change ->
?MODULE:loop(S);
Unknown ->
io:format("Unknown message: ~p~n",[Unknown]),
loop(S)
```

The first case (`shutdown`) is pretty explicit. You get the kill message, let the process die. If you wanted to save state to disk, that could be a possible place to do it. If you wanted safer save/exit semantics, this could be done on every`add`, `cancel` or `done` message. Loading events from disk could then be done in the `init` function, spawning them as they come.

The `'DOWN'` message's actions are also simple enough. It means a client died, so we remove it from the client list in the state.

Unknown messages will just be shown with `io:format/2` for debugging purposes, although a real production application would likely use a dedicated logging module

And here comes the code change message. This one is interesting enough for me to give it its own section.

Source : http://learnyousomeerlang.com/designing-a-concurrent-application