Paxy: the Paxos protocol

Jordi Guitart

Adapted with permission from Johan Montelius (KTH)

September 24, 2025

Introduction

This exercise will give you the opportunity to learn the Paxos algorithm for gaining consensus in a distributed system. We assume a system where processes can propose values and the consensus algorithm ensures that a single one among the proposed values is chosen. You should know the basic operation of the algorithm (you could also read the paper 'Paxos Made Simple' by Lamport), but you do not have to know all the details, that is the purpose of this exercise.

The given code is not complete. Ellipsis (...) must be replaced with the missing pieces.

1 Paxos

The Paxos algorithm has three different roles: *proposers*, *acceptors*, and *learners*. The functionality of all three is often included in one process but it will be easier to implement the *proposer* and the *acceptor* as two separate processes. As the *learner* role is not needed to reach a consensus, we will not implement *learners* and we will notify the outcome of the algorithm only to the *proposers*.

We also include a gui module in order to illustrate better how the algorithm works. The gui contains two sets of panels; *proposers* on the left and *acceptors* on the right. Each *proposer* and *acceptor* is represented by a panel process that is updated every time the state of the *proposer* or the *acceptor* changes. The complete code for the gui is given in 'Appendix C'.

1.1 Sequence numbers

Since *proposers* need unique sequence numbers, we will need some basic support to handle them, in particular, functions to generate and compare sequence numbers. One way of guaranteeing uniqueness is to use a tuple and let the first element be an increasing integer (per *proposer*) and the second a unique identifier for the *proposer*. We have built a simple `order` module, which can be found in 'Appendix A', according to this description. We can operate easily on sequence numbers by using the exported functions.

1.2 The *acceptor*

The *acceptor* has a state consisting of:

- **Name**: the name of the *acceptor*. This is also used as its identifier.
- **Promised**: the sequence number of the highest promise given so far. The *acceptor* promised not to vote any ballot below this number.

- **Voted**: the sequence number of the highest ballot voted so far.
- **Value**: the value (i.e. color) of the highest ballot voted so far.
- **PanelId**: the process id of the gui panel that is connected to this *acceptor*.

Note that an *acceptor* can vote many ballots during the execution but it must remember only the one with the highest sequence number.

When we start an *acceptor* we have neither promised nor voted anything, so the `Promised` and `Voted` parameters are instantiated to *null* sequence numbers that are lower than any other sequence number. The `Value` parameter is initialized to `na` to indicate that it is *not applicable*.

```
-module(acceptor).
-export([start/2]).

start(Name, PanelId) ->
  spawn(fun() -> init(Name, PanelId) end).

init(Name, PanelId) ->
  Promised = order:null(),
  Voted = order:null(),
  Value = na,
  acceptor(Name, Promised, Voted, Value, PanelId).
```

The *acceptor* is waiting for two types of messages: `prepare` requests and `accept` requests. A `prepare` request from process `Proposer` ({`prepare, Proposer, Round`}) will result in a promise, if we have not made any promise that prevents us to make such a promise. In order to check this, the `Round` number of the `prepare` request must be compared with the highest promise already given (`Promised`). If the `Round` number is higher, the *acceptor* can return a promise, {`promise, Round, Voted, Value`}. It is important that this message returns which round we are promising (`Round`), and the sequence number (`Voted`) and the value (`Value`) of the highest ballot this *acceptor* has voted so far.

In this case, in addition to informing the *proposer*, the *acceptor* should also send an update message to the corresponding panel process of the gui. Acceptor panels contain information for each *acceptor* on its highest promise given and its highest ballot voted. The voted values are represented by different colors. Black color corresponds to no value voted yet.

If we cannot give a promise, we do not have to do anything, but it would be polite to send a `sorry` message informing the *proposer* that we cannot give a promise for the round it has requested (we could additionally inform the proposer what round we have already promised but let's keep things simple) [1].

```
acceptor(Name, Promised, Voted, Value, PanelId) ->
  receive
    {prepare, Proposer, Round} ->
```

[1] Alternatively, if we want to make life hard for the *proposer* we could even send back a promise. If we have promised not to vote in a round lower than 17, we could of course promise not to vote in a round lower than 12. The *proposer* will take our promise as an indication that it is possible for us to vote for a ballot in round 12 but that will of course not happen.

```
case order:gr(..., ...) of
  true ->
    ... ! {promise, ..., ..., ...},
io:format("[Acceptor ~w] Phase 1: promised ~w voted ~w colour ~w~n",
           [Name, ..., ..., Value]),
    % Update gui
    Colour = case Value of na -> {0,0,0}; _ -> Value end,
    PanelId ! {updateAcc, "Voted: " ++ io_lib:format("~p", [Voted]),
                "Promised: " ++ io_lib:format("~p", [...]), Colour},
    acceptor(Name, ..., Voted, Value, PanelId);
  false ->
    ... ! {sorry, {prepare, ...}},
    acceptor(Name, ..., Voted, Value, PanelId)
end;
```

An `accept` request, sent by a `Proposer` when it has received promises from a majority of *acceptors*, also has two outcomes; either we can vote the ballot and then, if the ballot number is higher than the current maximum one, we must save the number and the value that come in the ballot, or we made a promise that prevents us from voting that ballot. Note that we do not change our promise just because we vote for a new value. Here, again, we need to update the corresponding gui process.

Again, if we cannot vote the ballot we could simply ignore the message but it is polite to inform the `Proposer`.

```
{accept, Proposer, Round, Proposal} ->
  case order:goe(..., ...) of
    true ->
      ... ! {vote, ...},
      case order:goe(..., ...) of
        true ->
  io:format("[Acceptor ~w] Phase 2: promised ~w voted ~w colour ~w~n",
             [Name, Promised, ..., ...]),
          % Update gui
          PanelId ! {updateAcc, "Voted: " ++ io_lib:format("~p", [...]),
                      "Promised: " ++ io_lib:format("~p", [Promised]), ...},
          acceptor(Name, Promised, ..., ..., PanelId);
        false ->
          acceptor(Name, Promised, ..., ..., PanelId)
      end;
    false ->
      ... ! {sorry, {accept, ...}},
      acceptor(Name, Promised, Voted, Value, PanelId)
  end;
```

Nothing prevents an *acceptor* to vote a value in round 17 and then vote another value if asked to do so in round 12 (provided of course that it has not any promise preventing that). This is a very strange situation but it is allowed. If we vote a value in a lower round we should of course still remember the value of the ballot with the highest sequence number.

We also include a message to terminate the *acceptor*. You can also add messages for status information, a catch-all clause, and print-out statements so that you can track what the *acceptor* has done.

```
    stop ->
      PanelId ! stop,
      ok
  end.
```

1.3 The *proposer*

The *proposer* works in rounds. In each round it will try to get acceptance of a proposed value (`Proposal`) or at least make the *acceptors* agree on any value. If it fails, it will try again and again, but each time with a higher round number. The proposer panel in the gui contains information on the current round and the current proposed value.

```
-module(proposer).
-export([start/6]).

-define(timeout, 2000).
-define(backoff, 10).

start(Name, Proposal, Acceptors, Sleep, PanelId, Main) ->
  spawn(fun() -> init(Name, Proposal, Acceptors, Sleep, PanelId, Main) end).

init(Name, Proposal, Acceptors, Sleep, PanelId, Main) ->
  timer:sleep(Sleep),
  Begin = erlang:monotonic_time(),
  Round = order:first(Name),
  {Decision, LastRound} = round(Name, ?backoff, Round, Proposal, Acceptors, PanelId),
  End = erlang:monotonic_time(),
  Elapsed = erlang:convert_time_unit(End-Begin, native, millisecond),
  io:format("[Proposer ~w] DECIDED ~w in round ~w after ~w ms~n",
            [Name, Decision, LastRound, Elapsed]),
  Main ! done,
  PanelId ! stop.
```

In a round the *proposer* will wait for `promise` and `vote` messages for up to `timeout` milliseconds. If it has not received the necessary number of replies it will abort the round. It will then sleep for an increasing number of milliseconds (calculated from `backoff`) before starting the next round. It will try its best to get the *acceptors* to vote for its proposal but, as you will see, it will be happy if they can agree on anything.

Each round consists of one ballot attempt. The ballot either succeeds or aborts, in which case a new round is initiated. The gui is updated in the beginning of each round.

```
round(Name, Backoff, Round, Proposal, Acceptors, PanelId) ->
  io:format("[Proposer ~w] Phase 1: round ~w proposal ~w~n",
            [Name, Round, Proposal]),
```

```
  % Update gui
  PanelId ! {updateProp, "Round: " ++ io_lib:format("~p", [Round]), Proposal},
  case ballot(Name, ..., ..., ..., PanelId) of
    {ok, Value} ->
      {Value, Round};
    abort ->
      timer:sleep(rand:uniform(Backoff)),
      Next = order:inc(...),
      round(Name, (2*Backoff), ..., Proposal, Acceptors, PanelId)
  end.
```

A ballot is initialized by multicasting a `prepare` message to all *acceptors* (`prepare/2` function). The *proposer* then collects the promises and also the voted value with the highest sequence number so far (`collect/4` function). If we receive promises from a majority of *acceptors* (`Quorum`) we start the voting process by multicasting an `accept` message to all *acceptors* (`accept/3` function). In the `accept` message we include the value with the highest sequence number voted by a member in the quorum. Then it is time to collect the votes (`vote/2` function) and determine whether consensus has been reached or not.

```
ballot(Name, Round, Proposal, Acceptors, PanelId) ->
  prepare(..., ...),
  Quorum = (length(...) div 2) + 1,
  MaxVoted = order:null(),
  case collect(..., ..., ..., ...) of
    {accepted, Value} ->
      io:format("[Proposer ~w] Phase 2: round ~w proposal ~w (was ~w)~n",
                 [Name, Round, Value, Proposal]),
      % update gui
      PanelId ! {updateProp, "Round: " ++ io_lib:format("~p", [Round]), Value},
      accept(..., ..., ...),
      case vote(..., ...) of
        ok ->
          {ok, ...};
        abort ->
          abort
      end;
    abort ->
      abort
  end.
```

The collect procedure simply waits to receive `N` promises and learns in the variables `(MaxVoted, Proposal)` the number and the value of the highest ballot voted so far. We need a timeout since *acceptors* could take forever or simply refuse to reply. Note that we have tagged the requests sent with the round number and we only consider replies that correspond with that round. Note also that we need catch-all alternatives for `promise` and `sorry` messages, since there might be delayed messages out there that otherwise would just stack up.

```
collect(0, _, _, Proposal) ->
  {accepted, Proposal};
```

```
collect(N, Round, MaxVoted, Proposal) ->
  receive
    {promise, Round, _, na} ->
      collect(..., ..., ..., ...);
    {promise, Round, Voted, Value} ->
      case order:gr(..., ...) of
        true ->
          collect(..., ..., ..., ...);
        false ->
          collect(..., ..., ..., ...)
      end;
    {promise, _, _,  _} ->
      collect(N, Round, MaxVoted, Proposal);
    {sorry, {prepare, Round}} ->
      collect(..., ..., ..., ...);
    {sorry, _} ->
      collect(N, Round, MaxVoted, Proposal)
  after ?timeout ->
    abort
  end.
```

Collecting votes follows almost the same procedure. We must wait until we have received `N` votes for the corresponding round. If we're unsuccessful we abort and hope for better luck next round. Here we also have catch-all clauses.

```
vote(0, _) ->
  ok;
vote(N, Round) ->
  receive
    {vote, Round} ->
      vote(..., ...);
    {vote, _} ->
      vote(N, Round);
    {sorry, {accept, Round}} ->
      vote(..., ...);
    {sorry, _} ->
      vote(N, Round)
  after ?timeout ->
    abort
  end.
```

The only remaining thing is to implement the sending of `prepare` and `accept` requests.

```
prepare(Round, Acceptors) ->
  Fun = fun(Acceptor) ->
    send(Acceptor, {prepare, self(), Round})
  end,
  lists:foreach(Fun, Acceptors).

accept(Round, Proposal, Acceptors) ->
```

```
  Fun = fun(Acceptor) ->
    send(Acceptor, {accept, self(), Round, Proposal})
  end,
  lists:foreach(Fun, Acceptors).
```

Sending a message is of course trivial but we will, for reasons described later, implement it in a separate procedure.

```
send(Name, Message) ->
  Name ! Message.
```

2 Experiments

Once you have completed the code (you can validate it through the review questionnaires at ATENEA), we can set up a test and see if the processes can agree on something. We have five *acceptors* and three *proposers*. The *proposers* try to make the *acceptors* vote for their suggestion. The *proposers* will hopefully find a quorum and then learn the agreed value. A test module (`paxy`) will help us to set up the experiments. You can use the `Sleep` parameter to vary the initial sleep time of each proposer (in milliseconds), which will allow different consensus values to be agreed.

```
-module(paxy).
-export([start/1, stop/0, stop/1]).

-define(RED, {255,0,0}).
-define(BLUE, {0,0,255}).
-define(GREEN, {0,255,0}).

% Sleep is a list with the initial sleep time for each proposer
start(Sleep) ->
  AcceptorNames = ["Homer", "Marge", "Bart", "Lisa", "Maggie"],
  AccRegister = [homer, marge, bart, lisa, maggie],
  ProposerNames = [{"Fry", ?RED}, {"Bender", ?GREEN}, {"Leela", ?BLUE}],
  PropInfo = [{fry, ?RED}, {bender, ?GREEN}, {leela, ?BLUE}],
  register(gui, spawn(fun() -> gui:start(AcceptorNames, ProposerNames) end)),
  gui ! {reqState, self()},
  receive
    {reqState, State} ->
      {AccIds, PropIds} = State,
      start_acceptors(AccIds, AccRegister),
      spawn(fun() ->
        Begin = erlang:monotonic_time(),
        start_proposers(PropIds, PropInfo, AccRegister, Sleep, self()),
        wait_proposers(length(PropIds)),
        End = erlang:monotonic_time(),
        Elapsed = erlang:convert_time_unit(End-Begin, native, millisecond),
        io:format("[Paxy] Total elapsed time: ~w ms~n", [Elapsed])
      end)
  end.
```

```
start_acceptors(AccIds, AccReg) ->
  case AccIds of
    [] ->
      ok;
    [AccId|Rest] ->
      [RegName|RegNameRest] = AccReg,
      register(RegName, acceptor:start(RegName, AccId)),
      start_acceptors(Rest, RegNameRest)
  end.

start_proposers(PropIds, PropInfo, Acceptors, Sleep, Main) ->
  case PropIds of
    [] ->
      ok;
    [PropId|Rest] ->
      [{RegName, Colour}|RestInfo] = PropInfo,
      [FirstSleep|RestSleep] = Sleep,
      proposer:start(RegName, Colour, Acceptors, FirstSleep, PropId, Main),
      start_proposers(Rest, RestInfo, Acceptors, RestSleep, Main)
  end.

wait_proposers(0) ->
  ok;
wait_proposers(N) ->
  receive
    done ->
      wait_proposers(N-1)
  end.
```

Since the *acceptors* stay alive even if a decision has been made, we need to terminate them explicitly. The code below becomes useful during debugging since a crashed acceptor will be de-registered (and sending a message to an unregistered name will cause an exception).

```
stop() ->
  stop(homer),
  stop(marge),
  stop(bart),
  stop(lisa),
  stop(maggie),
  stop(gui).

stop(Name) ->
  case whereis(Name) of
    undefined ->
      ok;
    Pid ->
      Pid ! stop
  end.
```

Experiments and Questions. i) Try introducing **different** delays in the `promise` messages, `vote` messages, or both sent by the *acceptor*. Q) Does the algorithm still terminate? Does it require more rounds? How does the impact of message delays depend on the value of the timeout at the proposer?

You can use the following code to add delays when sending messages.

```
-define(delay, 500).

T = rand:uniform(?delay),
timer:send_after(T, Pid, Message),
```

ii) Avoid sending `sorry` messages by commenting the corresponding sentences in the acceptor. Q) Could you come to an agreement when `sorry` messages are not sent?

iii) Try randomly dropping `promise` messages, `vote` messages, or both in the *acceptor*. If you drop too many messages a quorum will of course never be found, but we could probably lose quite many. Q) How does the drop ratio affect the number of rounds required to reach consensus? What percentage of messages can we drop until consensus is no longer possible?

You can drop messages using the following code, which will drop in average one in 10 messages. Try different drop ratios.

```
-define(drop, 1).

P = rand:uniform(10),
if P =< ?drop ->
      io:format("message dropped~n");
   true ->
      %send message
end.
```

iv) Try increasing the number of *acceptors* and *proposers*. Q) What is the impact (in the number of rounds needed to get consensus) of having more acceptors while keeping the same number of proposers? What if we have more proposers while keeping the same number of acceptors?

v) Adapt the `paxy` module to create the *proposers* in a remote Erlang instance (named *paxy-pro*) and to ensure that they can connect correctly to the *acceptors*, which must be created in a different remote Erlang instance (named *paxy-acc*). Note that *acceptors* and *proposers* should not be modified, the *acceptors* have to use locally registered names, there should be a single 'start' (and 'stop') function that creates (and stops) all the processes, and the *proposers* must be created once the *acceptors* are up and running. Check the slides about Erlang to refresh how processes are created remotely, how names registered in remote nodes are referred, and how Erlang runtime should be started to run distributed programs.

3 Fault tolerance

In order to make the implementation fault tolerant, the *acceptor* needs to remember what it promises and what it votes for. If we use the module `pers`, which can be found in 'Appendix B', we can store state changes as we make promises and vote, and we can recover our state to the state we had before we crashed. Think about how to do these changes in the *acceptor* to make it fault tolerant while fulfilling the following requirements: i) the *acceptor* must save the most up-to-date version of its state in the presence of unexpected failures, ii) the *acceptor*'s state must be only saved when it has been modified, iii) the *acceptor*'s state must be removed from the disk when it finishes its execution successfully, iv) the GUI panel of the *acceptor* must always show its current state.

The *proposer* also has to be careful when sending a message to an *acceptor*. We should first check that the *acceptor* is actually registered, if not it means that the *acceptor* is down. If the *acceptor* was registered on a remote node we can send the message anyway since sending a message to a remote process always succeeds. If the *acceptor* is a locally registered process the send operation could throw an exception, something that we want to avoid.

```
send(Name, Message) ->
  if is_tuple(Name) -> %remote
    Name ! Message;
  true -> %local
    case whereis(Name) of
      undefined ->
        down;
      Pid ->
        Pid ! Message
    end
  end.
```

Experiments. Simulate a crash and restart of an *acceptor* using the `crash` procedure below and check whether it recovers successfully (with exactly the same state it had before failing). We specify a 'na' value for the `PanelId` of the *acceptor*, as it will get this value from the persistent storage. You can place the procedure in the `paxy` module, but it must be invoked from the Erlang instance where the acceptors are running.

```
crash(Name) ->
  case whereis(Name) of
    undefined ->
      ok;
    Pid ->
      unregister(Name),
      exit(Pid, "crash"),
      pers:open(Name),
      {_, _, _, Pn} = pers:read(Name),
      Pn ! {updateAcc, "Voted: CRASHED", "Promised: CRASHED", {0,0,0}},
      pers:close(Name),
      timer:sleep(3000),
      register(Name, acceptor:start(Name, na))
  end.
```

4 Improvement based on sorry messages

Some improvements in the implementation of the *proposer* are possible. In a given round, if we need three promises for a quorum and we have received three `sorry` messages from the in total five *acceptors* then we can abort the ballot.

Experiments. Change the code of both the `collect/4` and `vote/2` procedures to implement the aforementioned improvement and check how it works, especially whether it allows getting consensus faster.

Appendix A: *order* module

```
-module(order).
-export([null/0, first/1, gr/2, goe/2, inc/1]).

null() ->
    {0,0}.

first(Id) ->
    {0, Id}.

%% compare sequence numbers: greater?
gr({N1,I1}, {N2,I2}) ->
    if
        N1 > N2 ->
            true;
        ((N1 == N2) and (I1 > I2)) ->
            true;
        true ->
            false
    end.

%% compare sequence numbers: greater or equal?
goe({N1,I1}, {N2,I2}) ->
    if
        N1 > N2 ->
            true;
        ((N1 == N2) and  (I1 >= I2)) ->
            true;
        true ->
            false
    end.

%% increase sequence number
inc({N, Id}) ->
    {N+1, Id}.
```

Appendix B: *pers* module

```
-module(pers).
-export([open/1, read/1, store/5, close/1, delete/1]).

%% dets module provides term storage on file

open(Name) ->
    dets:open_file(Name, []).

%% returns the object with the key 'perm' stored in the table 'Name'
read(Name) ->
    case dets:lookup(Name, perm) of
        [{perm, Pr, Vt, Ac, Pn}] ->
            {Pr, Vt, Ac, Pn};
        [] ->
            {order:null(), order:null(), na, na}
    end.

%% inserts one object {Pr, Vt, Ac, Pn} into the table 'Name'
store(Name, Pr, Vt, Ac, Pn)->
    dets:insert(Name, {perm, Pr, Vt, Ac, Pn}).

close(Name) ->
    dets:close(Name).

delete(Name) ->
    file:delete(Name).
```

Appendix C: *gui* module

```
-module(gui).
-export([start/2]).
-include_lib("wx/include/wx.hrl").

-define(WindowSize, {450, 420}).
-define(PanelSize, {175, 40}).
-define(OuterSizerMinWidth, 190).
-define(OuterSizerMaxHeight, 420). % maximum sizer size
-define(InSizerMinWidth, 175).
-define(InSizerMinHeight, 40).
-define(PropTitle, "Proposers").
-define(PropText1, "Round:").
-define(AccTitle, "Acceptors").
-define(AccText1, "Voted: {}").
-define(AccText2, "Promised: {}").

start(Acceptors, Proposers) ->
  % computing panel heights (plus the spacer value)
  AccPanelHeight = length(Acceptors)*?InSizerMinHeight + 10,
  PropPanelHeight = length(Proposers)*?InSizerMinHeight + 10,
  State = make_window(Acceptors, Proposers, AccPanelHeight, PropPanelHeight),
  gui(State).

make_window(Acceptors, Proposers, AccPanelHeight, PropPanelHeight) ->
  Server = wx:new(),
  Env = wx:get_env(),
  Frame = wxFrame:new(Server, -1, "Paxos Algorithm", [{size,?WindowSize}]),
  wxFrame:connect(Frame, close_window),
  Panel  = wxPanel:new(Frame),

  % create Sizers
  OuterSizer = wxBoxSizer:new(?wxVERTICAL),
  MainSizer = wxBoxSizer:new(?wxHORIZONTAL),
  ProposerSizer = wxStaticBoxSizer:new(?wxVERTICAL, Panel,
                                       [{label, "Proposers"}]),
  AcceptorSizer = wxStaticBoxSizer:new(?wxVERTICAL, Panel,
                                       [{label, "Acceptors"}]),

  % set Sizer's min width/height
  case AccPanelHeight > ?OuterSizerMaxHeight of
    true ->
      OuterAccSizerHeight = ?OuterSizerMaxHeight;
    false ->
      OuterAccSizerHeight = AccPanelHeight
  end,

  case PropPanelHeight > ?OuterSizerMaxHeight of
    true ->
```

```
      OuterPropSizerHeight = ?OuterSizerMaxHeight;
    false ->
      OuterPropSizerHeight = PropPanelHeight
  end,

  wxSizer:setMinSize(AcceptorSizer, ?OuterSizerMinWidth, OuterAccSizerHeight),
  wxSizer:setMinSize(ProposerSizer, ?OuterSizerMinWidth, OuterPropSizerHeight),
  % add spacers
  wxSizer:addSpacer(MainSizer, 10),  %spacer
  wxSizer:addSpacer(ProposerSizer, 5),
  wxSizer:addSpacer(AcceptorSizer, 5),

  % add ProposerSizer into MainSizer
  wxSizer:add(MainSizer, ProposerSizer,[]),
  wxSizer:addSpacer(MainSizer, 20),

  % add AcceptorSizer into MainSizer
  wxSizer:add(MainSizer, AcceptorSizer,[]),
  wxSizer:addSpacer(MainSizer, 20),
  wxSizer:addSpacer(OuterSizer, 10),

  % add MainSizer into OuterSizer
  wxSizer:add(OuterSizer, MainSizer, []),

  %% Now 'set' OuterSizer into the Panel
  wxPanel:setSizer(Panel, OuterSizer),

  % create Acceptors and Proposers Panels
  AccIds = create_acceptors(Acceptors, Panel, AcceptorSizer, Env),
  PropIds = create_proposers(Proposers, Panel, ProposerSizer, Env),

  wxFrame:show(Frame),
  {Frame, AccIds, PropIds}.

gui(State) ->
  {Frame, AccIds, PropIds} = State,
  receive
    % request State
    {reqState, From} ->
      io:format("[Gui] state requested ~n"),
      From ! {reqState, {AccIds, PropIds}},
      gui(State);
    % a connection gets the close_window signal
    % and sends this message to the server
    #wx{event=#wxClose{}} ->
      %optional, goes to shell
      io:format("[Gui] ~p closing window ~n", [self()]),
      % now we use the reference to Frame
      wxWindow:destroy(Frame),
      ok;  % we exit the loop
```

```
    stop ->
      wxWindow:destroy(Frame),
      ok;  % we exit the loop
    Msg ->
      %Everything else ends up here
      io:format("[Gui] unknown message: ~p ~n", [Msg]),
      gui(State)
  end.

% create acceptors
create_acceptors(AcceptorList, Panel, AcceptorSizer, Env) ->
  AcceptorData = lists:map(fun(AccTitle) ->
    AcceptorSizerIn = wxStaticBoxSizer:new(?wxVERTICAL, Panel,
    [{label, AccTitle}]),
    %set Sizer's min width/height
    wxSizer:setMinSize(AcceptorSizerIn, ?InSizerMinWidth, ?InSizerMinHeight),
    AcceptorPanel = wxPanel:new(Panel, [{size, ?PanelSize}]),
    {Lb1, Lb2} = setPanel2(AcceptorPanel, ?wxBLACK, ?AccText1, ?AccText2),
    wxSizer:add(AcceptorSizerIn, AcceptorPanel),
    wxSizer:add(AcceptorSizer, AcceptorSizerIn),
    {AcceptorPanel, AcceptorSizerIn, Lb1, Lb2}
  end,
  AcceptorList),

  lists:map(fun({AcceptorPanel, AcceptorSizerIn, Lb1, Lb2}) ->
    spawn(fun() ->
      wx:set_env(Env),
      acceptor(AcceptorPanel, AcceptorSizerIn, Lb1, Lb2)
    end)
  end,
  AcceptorData).

% create proposers
create_proposers(ProposerList, Panel, ProposerSizer, Env) ->
  ProposerData = lists:map(fun({PropTitle, TextColour}) ->
    ProposerSizerIn = wxStaticBoxSizer:new(?wxVERTICAL, Panel,
    [{label, PropTitle}]),
    % set Sizer's min width/height
    wxSizer:setMinSize(ProposerSizerIn, ?InSizerMinWidth, ?InSizerMinHeight),
    ProposerPanel = wxPanel:new(Panel, [{size, ?PanelSize}]),
    Lb1 = setPanel(ProposerPanel, ?wxBLACK, ?PropText1),
    wxSizer:add(ProposerSizerIn, ProposerPanel),
    wxSizer:add(ProposerSizer, ProposerSizerIn),
    StaticBox = wxStaticBoxSizer:getStaticBox(ProposerSizerIn),
    wxStaticText:setForegroundColour(StaticBox, TextColour),
    {ProposerPanel, ProposerSizerIn, Lb1}
  end,
  ProposerList),

  lists:map(fun({ProposerPanel, ProposerSizerIn, Lb1}) ->
```

```
    spawn(fun() ->
      wx:set_env(Env),
      proposer(ProposerPanel, ProposerSizerIn, Lb1)
    end)
  end,
  ProposerData).

% acceptor loop waiting updates
acceptor(AccPanel, AccSizerIn, L1Obj, L2Obj) ->
  receive
    % update panel
    {updateAcc, NewL1, NewL2, Colour} ->
      updatePanel2(AccPanel, L1Obj, L2Obj, NewL1, NewL2, Colour),
      wxWindow:fit(AccPanel),
      acceptor(AccPanel, AccSizerIn, L1Obj, L2Obj);
    stop ->
      ok
  end.

% proposer loop waiting for updates
proposer(PropPanel, PropSizerIn, L1Obj) ->
  receive
    % update panel
    {updateProp, NewL1, Colour} ->
      updatePanel(PropPanel, L1Obj, NewL1, Colour),
      wxWindow:fit(PropPanel),
      proposer(PropPanel, PropSizerIn, L1Obj);
    stop ->
      ok
  end.

% set a Panel
setPanel2(InPanel, BgColour, L1Text, L2Text) ->
  wxPanel:setBackgroundColour(InPanel, BgColour),
  Label1Obj = wxStaticText:new(InPanel, 1, L1Text, [{pos, {5, 5}}]),
  wxStaticText:setForegroundColour(Label1Obj, ?wxWHITE),
  Label2Obj = wxStaticText:new(InPanel, 1, L2Text, [{pos, {5, 20}}]),
  wxStaticText:setForegroundColour(Label2Obj, ?wxWHITE),
  {Label1Obj, Label2Obj}.

setPanel(InPanel, BgColour, L1Text) ->
  wxPanel:setBackgroundColour(InPanel, BgColour),
  Label1Obj = wxStaticText:new(InPanel, 1, L1Text, [{pos, {5, 12}}]),
  wxStaticText:setForegroundColour(Label1Obj, ?wxWHITE),
  Label1Obj.

updatePanel2(Panel, Label1Obj, Label2Obj, NewL1, NewL2, Colour) ->
  wxPanel:setBackgroundColour(Panel, Colour),
  wxStaticText:setLabel(Label1Obj, NewL1),
  wxStaticText:setLabel(Label2Obj, NewL2),
```

```
  wxPanel:refresh(Panel).

updatePanel(Panel, Label1Obj, NewL1, Colour) ->
  wxPanel:setBackgroundColour(Panel, Colour),
  wxStaticText:setLabel(Label1Obj, NewL1),
  wxPanel:refresh(Panel).
```