Paxy: the Paxos protocol

Jordi Guitart

Adapted with permission from Johan Montelius (KTH)

October 21, 2012

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 (read 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. ``...'' spaces must be filled with the missing pieces.

1 Paxos

The Paxos algorithm has three different processes: 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. The learner process will not be implemented since it is not needed to reach a consensus. In a real system, it is of course important to also know the outcome of the algorithm but we will do without learners.

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

We will need some basic support to handle sequence numbers. Since proposers need unique sequence numbers we need a way 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 small `order` module, that can be found in 'Appendix A', according to this description. It will be quite easy to see what we mean when we use the exported functions.

1.2 The *acceptor*

Let's start with the *acceptor*. It has a state consisting of:

- **Name**: the name of the *acceptor*
- **Promise**: the *acceptor* promised not to accept any ballot (i.e. sequence number) below this number
- **Voted**: the highest ballot number accepted
- **Accepted**: the value coming with the highest ballot that has been accepted
- **PanelId**: the process id of the gui panel that is connected to this acceptor

Note that an *acceptor* can accept many values during the execution but we must remember the value with the highest ballot number.

When we start an *acceptor* we have not promised anything nor accepted a value, so the `Promise` and `Voted` parameters are instantiated to *null* sequence numbers that are lower than any other sequence number. The `Accepted` parameter is initialized to `na` to indicate that it is *not applicable*.

```
-module(acceptor).
-export([start/3]).

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

init(Name, Seed, PanelId) ->
    random:seed(Seed, Seed, Seed),
    Promise = order:null(),
    Voted = order:null(),
    Accepted = na,
    acceptor(Name, Promise, Voted, Accepted, 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 last `Promise` already given. If the `Round` number is higher, the *acceptor* can return a promise, {`promise, Round, Voted, Accepted`}. It is important that this message returns the highest ballot number this *acceptor* has accepted (`Voted`), the accepted value in that ballot (`Accepted`), and which round we are promising (`Round`).

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 on the current promise and the round during which the acceptor has voted. The values voted 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. If we really want to make life hard for the *proposer* we could even send back a promise. If we have promised not to vote in round lower than round 17, we could of course promise not to vote in a round lower than 12. The *proposer* will of course take our promise as an indication that it is possible for us to vote for a value in round 12 but that will of course not happen. To help the *proposer* we should inform it that we have promised not to vote in the round requested by the *proposer* (we could even inform the proposer what we have promised but let's keep thing simple).

```
{prepare, Proposer, Round} ->
    case order:gr(..., ...) of
        true ->
            ... ! {promise, ..., ..., ...},
            % Update gui
            if
                Accepted == na ->
io:format("[Acceptor ~w] set gui: voted ~w promise ~w colour na~n",
                    [Name, ..., ...]),
                    PanelId ! {updateAcc, "Round voted: "
++ lists:flatten(io_lib:format("~p", [Voted])), "Cur. Promise: "
++ lists:flatten(io_lib:format("~p", [...])), {0,0,0}};
                true ->
io:format("[Acceptor ~w] set gui: voted ~w promise ~w colour ~w~n",
                    [Name, ..., ..., Accepted]),
                    PanelId ! {updateAcc, "Round voted: "
++ lists:flatten(io_lib:format("~p", [Voted])), "Cur. Promise: "
++ lists:flatten(io_lib:format("~p", [...])), Accepted}
            end,
            acceptor(Name, ..., Voted, Accepted, PanelId);
        false ->
            ... ! {sorry, ...},
            acceptor(Name, ..., Voted, Accepted, PanelId)
    end;
```

An `accept` request, sent by a `Proposer` when it has received promises from a majority, also has two outcomes; either we can accept the request and then save the value that comes in the ballot (if the ballot number is

higher than the current maximum one) or we have a promise that prevents us from accepting the request. 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 accept the request 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 ->
                  % Update gui
io:format("[Acceptor ~w] set gui: voted ~w promise ~w colour ~w~n",
                       [Name, ..., ..., ...]),
                       PanelId ! {updateAcc, "Round voted: "
++ lists:flatten(io_lib:format("~p", [...])), "Cur. Promise: "
++ lists:flatten(io_lib:format("~p", [Promise])), ...},
                       acceptor(Name, Promise, ..., ..., PanelId);
                 false ->
                       % Update gui
io:format("[Acceptor ~w] set gui: voted ~w promise ~w colour ~w~n",
                       [Name, ..., ..., ...]),
                       PanelId ! {updateAcc, "Round voted: "
++ lists:flatten(io_lib:format("~p", [...])), "Cur. Promise: "
++ lists:flatten(io_lib:format("~p", [Promise])), ...},
                       acceptor(Name, Promise, ..., ..., PanelId)
             end;
         false ->
             ... ! {sorry, ...},
             acceptor(Name, Promise, ..., ..., PanelId)
     end;
```

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

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

```
 stop ->
    ok
```

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 this does not work 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 proposal.

```
-module(proposer).
-export([start/5]).

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

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

init(Name, Proposal, Acceptors, Seed, PanelId) ->
    random:seed(Seed, Seed, Seed),
    Round = order:null(Name),
    round(Name, ?backoff, Round, Proposal, Acceptors, PanelId).
```

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 a 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) ->
    % Update gui
    io:format("[Proposer ~w] set gui: Round ~w Proposal ~w~n",
    [Name, Round, Proposal]),
    PanelId ! {updateProp, "Round: "
    ++ lists:flatten(io_lib:format("~p", [Round])), "Proposal: "
    ++ lists:flatten(io_lib:format("~p", [Proposal])), Proposal},
    case ballot(Name, ..., ..., ..., PanelId) of
        {ok, Decision} ->
            io:format("[Proposer ~w] ~w decided ~w in round ~w~n",
            [Name, Acceptors, Decision, Round]),
            {ok, Decision};
        abort ->
```

```
            timer:sleep(random:uniform(...)),
            Next = order:inc(...),
            round(Name, (2*...), ..., Proposal, Acceptors, PanelId)
    end.
```

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

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

The collect procedure will simply receive promises and, if no *acceptor* has any objection, learn in the variables `Max, Proposal` the so far accepted value with the highest ballot number. Note that we need a timeout since *acceptors* could take forever or simply refuse to reply. Also note that we have tagged the sent request with the sequence number and we only accept replies with the same sequence number. 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) ->
    {..., ...};
collect(N, Round, Max, Proposal) ->
    receive
        {promise, Round, _, na} ->
            collect(..., ..., ..., ...);
        {promise, Round, Voted, Value} ->
            case order:gr(..., ...) of
                true ->
                    collect(..., ..., ..., ...);
                false ->
                    collect(..., ..., ..., ...)
            end;
        {promise, _, _,  _} ->
            collect(N, Round, Max, Proposal);
        {sorry, Round} ->
            collect(N, Round, Max, Proposal);
        {sorry, _} ->
            collect(N, Round, Max, Proposal)
    after ?timeout ->
            abort
    end.
```

Collecting votes is almost the same procedure. We are only waiting for votes and need only count them until we have received them all. If we're unsuccessful we abort and hope for better luck next round. Here we also have catch-all clauses.

```
vote(0, _) ->
    ...;
vote(N, Round) ->
    receive
        {vote, Round} ->
            vote(..., ...);
        {vote, _} ->
            vote(N, Round);
        {sorry, Round} ->
            vote(N, Round);
        {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:map(Fun, Acceptors).

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

Let's set up a test and see if a set of *acceptors* can agree on something. We start five *acceptors* and we have 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 set up the experiments.

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

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

start(Seed) ->
    AcceptorNames = ["Acceptor 1", "Acceptor 2", "Acceptor 3",
    "Acceptor 4", "Acceptor 5"],
    AccRegister = [a, b, c, d, e],
    ProposerNames = ["Proposer 1", "Proposer 2", "Proposer 3"],
    PropInfo = [{kurtz, ?RED, 10}, {kilgore, ?GREEN, 2},
    {willard, ?BLUE, 3}],
    % computing panel heights
```

```
    AccPanelHeight = length(AcceptorNames)*50 + 20, %plus the spacer value
    PropPanelHeight = length(ProposerNames)*50 + 20,
    register(gui, spawn(fun() -> gui:start(AcceptorNames, ProposerNames,
    AccPanelHeight, PropPanelHeight) end)),
    gui ! {reqState, self()},
    receive
        {reqState, State} ->
            {AccIds, PropIds} = State,
            start_acceptors(AccIds, AccRegister, Seed),
            start_proposers(PropIds, PropInfo, AccRegister, Seed+1)
    end,
    true.

start_acceptors(AccIds, AccReg, Seed) ->
    case AccIds of
        [] ->
            ok;
        [AccId|Rest] ->
            [RegName|RegNameRest] = AccReg,
            register(RegName, acceptor:start(RegName, Seed, AccId)),
            start_acceptors(Rest, RegNameRest, Seed+1)
    end.

start_proposers(PropIds, PropInfo, Acceptors, Seed) ->
    case PropIds of
        [] ->
            ok;
        [PropId|Rest] ->
            [{RegName, Colour, Inc}|RestInfo] = PropInfo,
            proposer:start(RegName, Colour, Acceptors, Seed+Inc, PropId),
            start_proposers(Rest, RestInfo, Acceptors, Seed)
        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(gui),
    stop(a),
    stop(b),
    stop(c),
    stop(d),
```

```
    stop(e).

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

Add code to trace each state transition in the *acceptor* and the *proposer*. Try to follow the execution and the progress of the algorithm.

Open Questions. Do some experiments and see how the system works. Split the `paxy` module in two parts and make the needed adaptations to enable the acceptors (with the gui) and the proposers to run in different machines. Remember how names registered in remote nodes are referred and how Erlang runtime should be started to run distributed programs.

Try to introduce delays in the *acceptor* and the *proposer*. Insert larger delays and see if the algorithm still terminates. You can use the following lines to add delays.

```
    -define(delay, 20).

    R = random:uniform(?delay),
    timer:sleep(R),
```

Could you even come to an agreement when you ignore messages? Try ignoring to send `sorry` messages or simply randomly drop a vote. If you drop too many messages a quorum will of course never be found, but we could probably lose quite many. Does the algorithm ever report conflicting answers?

What happens if we increase the number of *acceptors* to say 9 or 17? Will we reach a decision? What if we have also have 10 *proposers*?

3 Fault tolerance

In order to make the implementation fault tolerant we need to remember what we promise and what we vote for. If we use the module `pers`, that can be found in 'Appendix B', we can recover our state to the state we had when we crashed and store state changes as we make promises. Where in the *acceptor* should we add this?

We also have to be careful when we send 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 we knew that the *acceptor* was registered on

a remote node we could ignore this procedure 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) ->
    case whereis(Name) of
        undefined ->
            down;
        Pid ->
            Pid ! Message,
    end.
```

Open Questions. Simulate a crash and restart using the `crash` procedure below (to be located in the `paxy` module) and see if the protocol still comes to a consensus. You might have to increase the sleep period in the *acceptor* to make the execution run slower. You can specify a 'na' value for the `PanelId` of the *acceptor*, as it will get this value from the persistent storage.

```
crash(Name) ->
    case whereis(Name) of
        undefined ->
            ok;
        Pid ->
            unregister(Name),
            exit(Pid, "crash"),
            register(Name, acceptor:start(Name, Seed, PanelId))
    end.
```

4 Other improvements

There are some improvements that could be made in the implementation of the *proposer*. 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.

Open Questions. Change the code of the `collect/4` and `vote/2` procedures to implement the afore-mentioned improvement.

Appendix A: *order* module

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

null() ->
    {0,0}.

null(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([read/1, store/5, delete/1]).

%% dets module provides term storage on file

%% returns the object with the key 'perm' stored in the table 'Name'
read(Name) ->
    {ok, Name} = dets:open_file(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}).

delete(Name) ->
    dets:delete(Name, perm),
    dets:close(Name).
```

Appendix C: *gui* module

```
-module(gui).
-export([start/4]).
-include_lib("wx/include/wx.hrl").

-define(WindowSize, {550, 600}).
-define(PanelSize, {250, 400}).
-define(OuterSizerMinWidth, 250).
-define(OuterSizerMaxHeight, 600). % maximum sizer size
-define(InSizerMinWidth, 200).
-define(InSizerMinHeight, 50).
-define(PropTitle, "Proposers").
-define(PropText1, "Round:").
-define(PropText2, "Last:").
-define(AccTitle, "Acceptors").
-define(AccText1, "Round Voted: {}").
-define(AccText2, "Cur. Promise: {}").

start(Acceptors, Proposers, AccPanelHeight, PropPanelHeight) ->
    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),

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

    % add spacers
    wxSizer:addSpacer(MainSizer, 10),  %spacer
    wxSizer:addSpacer(ProposerSizer, 20),
    wxSizer:addSpacer(AcceptorSizer, 20),

    % 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, 20),

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

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

    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{}} ->
            io:format("~p Closing window ~n",[self()]), %optional, goes to shell
            % 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("loop default triggered: Got ~n ~p ~n", [Msg]),
            gui(State)
    end.

% create acceptors
create_acceptors(AcceptorList, AcceptorIds, Panel, AcceptorSizer, Env) ->
    case AcceptorList of
        [] ->
            AcceptorIds;
        [AccName|Rest] ->
            Id = spawn(fun() -> initAcceptor(AccName, Panel, ?wxBLACK,
            AcceptorSizer, Env) end),
            create_acceptors(Rest, [Id|AcceptorIds], Panel, AcceptorSizer, Env)
        end.

% create proposers
create_proposers(ProposerList, ProposerIds, Panel, ProposerSizer, Env) ->
    case ProposerList of
        [] ->
            ProposerIds;
        [PropName|Rest] ->
            Id = spawn(fun() -> initProposer(PropName, Panel, ?wxBLACK,
            ProposerSizer, Env) end),
            create_proposers(Rest, [Id|ProposerIds], Panel, ProposerSizer, Env)
    end.

% initialize an acceptor
initAcceptor(AccTitle, InPanel, BgColour, AccSizer, AccEnv) ->
    wx:set_env(AccEnv),
```

```
    AcceptorSizerIn = wxStaticBoxSizer:new(?wxVERTICAL, InPanel,
    [{label, AccTitle}]),
    %set Sizer's min width/height
    wxSizer:setMinSize(AcceptorSizerIn, ?InSizerMinWidth, ?InSizerMinHeight),
    AcceptorPanel = wxPanel:new(InPanel, [{size, ?PanelSize}]),
    {L1, L2} = setPanel(AcceptorPanel, BgColour, ?AccText1, ?AccText2),
    wxSizer:add(AcceptorSizerIn, AcceptorPanel, []),
    wxSizer:add(AccSizer, AcceptorSizerIn),
    wxWindow:fit(InPanel),
    wxWindow:fit(AcceptorPanel),
    acceptor(AcceptorPanel, AcceptorSizerIn, BgColour, L1, L2).

% acceptor loop waiting updates
acceptor(AccPanel, AccSizerIn, BgColour, L1, L2) ->
    receive
        % update panel
        {updateAcc, Round, Promise, Colour} ->
            updatePanel(AccPanel, L1, L2, Round, Promise, Colour),
            wxWindow:fit(AccPanel),
            acceptor(AccPanel, AccSizerIn, BgColour, L1, L2)
    end.

% initialize a proposer
initProposer(PropTitle, InPanel, BgColour, PropSizer, PropEnv) ->
    wx:set_env(PropEnv),
    ProposerSizerIn = wxStaticBoxSizer:new(?wxVERTICAL, InPanel,
    [{label, PropTitle}]),
    % set Sizer's min width/height
    wxSizer:setMinSize(ProposerSizerIn, ?InSizerMinWidth, ?InSizerMinHeight),
    ProposerPanel = wxPanel:new(InPanel, [{size, ?PanelSize}]),
    {L1, L2} = setPanel(ProposerPanel, BgColour, ?PropText1, ?PropText2),
    wxSizer:add(ProposerSizerIn, ProposerPanel, []),
    wxSizer:add(PropSizer, ProposerSizerIn,[]),
    wxWindow:fit(InPanel),
    proposer(ProposerPanel, ProposerSizerIn, BgColour, L1, L2).

% proposer loop waiting for updates
proposer(PropPanel, PropSizerIn, BgColour, L1, L2) ->
    receive
        % update panel
        {updateProp, Round, Proposal, Colour} ->
            updatePanel(PropPanel, L1, L2, Round, Proposal, Colour),
            wxWindow:fit(PropPanel),
            proposer(PropPanel, PropSizerIn, BgColour, L1, L2)
```

```
    end.

% set a Panel
setPanel(InPanel, BgColour, Label1, Label2) ->
    wxPanel:setBackgroundColour(InPanel, BgColour),
    Round = wxStaticText:new(InPanel, 1, Label1,[{pos, {5, 5}}]),
    wxStaticText:setForegroundColour(Round, ?wxWHITE),
    Proposal = wxStaticText:new(InPanel, 1, Label2, [{pos, {5, 20}}]),
    wxStaticText:setForegroundColour(Proposal, ?wxWHITE),
    {Round, Proposal}.

updatePanel(PropPanel, Label1, Label2, Round, Proposal, Colour) ->
    wxPanel:setBackgroundColour(PropPanel, Colour),
    wxStaticText:setLabel(Label1, Round),
    wxStaticText:setLabel(Label2, Proposal),
    wxPanel:refresh(PropPanel).
```