```
-module(node1).
-export([start/1, start/2]).

-define(Stabilize, 1000).
-define(Timeout, 5000).

start(MyKey) ->
    start(MyKey, nil).

start(MyKey, PeerPid) ->
    timer:start(),
    spawn(fun() -> init(MyKey, PeerPid) end).

init(MyKey, PeerPid) ->
    Predecessor = nil,
    {ok, Successor} = connect(MyKey, PeerPid),
    schedule_stabilize(),
    node(MyKey, Predecessor, Successor).

connect(MyKey, nil) ->
    {ok, {MyKey ✔ , self() ✔ }};
connect(_, PeerPid) ->
    Qref = make_ref(),
    PeerPid ! {key, Qref, self()},
    receive
        {Qref, Skey} ->
            {ok, {Skey ✔ , PeerPid ✔ }}
    after ?Timeout ->
        io:format("Timeout: no response from ~w~n", [PeerPid])
    end.

schedule_stabilize() ->
    timer:send_interval(?Stabilize, self(), stabilize).

node(MyKey, Predecessor, Successor) ->
    receive
        {key, Qref, Peer} ->
            Peer ! {Qref, MyKey},
            node(MyKey, Predecessor, Successor);
        {notify, NewPeer} ->
            NewPredecessor = notify(NewPeer, MyKey, Predecessor),
            node(MyKey, NewPredecessor, Successor);
        {request, Peer} ->
            request(Peer, Predecessor),
            node(MyKey, Predecessor, Successor);
        {status, Pred} ->
            NewSuccessor = stabilize(Pred, MyKey, Successor),
            node(MyKey, Predecessor, NewSuccessor);
        stabilize ->
            stabilize(Successor),
            node(MyKey, Predecessor, Successor);
        stop ->
            ok;
        probe ->
            create_probe(MyKey, Successor),
            node(MyKey, Predecessor, Successor);
        {probe, MyKey, Nodes, T} ->
            remove_probe(MyKey, Nodes, T),
            node(MyKey, Predecessor, Successor);
        {probe, RefKey, Nodes, T} ->
            forward_probe(MyKey, RefKey, [MyKey|Nodes], T, Successor),
            node(MyKey, Predecessor, Successor);
        Error ->
            io:format("Reception of strange message ~w~n", [Error]),
            node(MyKey, Predecessor, Successor)
   end.

stabilize(Pred, MyKey, Successor) ->
  {Skey, Spid} = Successor,
  case Pred of
      nil ->
          Spid ! {notify, {MyKey, self()}}, ✔
```

```
            Successor;
        {MyKey, _} ->
            Successor;
        {Skey, _} ->
            Spid ! {notify, {MyKey, self()}},
            Successor;
        {Xkey, Xpid} ->
            case key:between(Xkey, MyKey, Skey) of
                true ->
                    self() ! stabilize,
                    Pred;
                false ->
                    Spid ! {notify, {MyKey, self()}},
                    Successor
            end
    end.

stabilize({_, Spid}) ->
    Spid ! {request, self()}.

request(Peer, Predecessor) ->
    case Predecessor of
        nil ->
            Peer ! {status, nil};
        {Pkey, Ppid} ->
            Peer ! {status, {Pkey, Ppid}}
    end.

notify({Nkey, Npid}, MyKey, Predecessor) ->
    case Predecessor of
        nil ->
            {Nkey, Npid};
        {Pkey, _} ->
            case key:between(Nkey, Pkey, MyKey) of
                true ->
                    {Nkey, Npid};
                false ->
                    Predecessor
            end
    end.

create_probe(MyKey, {_, Spid}) ->
    Spid ! {probe, MyKey, [MyKey], erlang:monotonic_time()},
    io:format("Node ~w created probe~n", [MyKey]).

remove_probe(MyKey, Nodes, T) ->
    T2 = erlang:monotonic_time(),
    Time = erlang:convert_time_unit(T2-T, native, microsecond),
    io:format("Node ~w received probe after ~w us -> Ring: ~w~n", [MyKey, Time, Nodes]).

forward_probe(MyKey, RefKey, Nodes, T, {_, Spid}) ->
    Spid ! {probe, RefKey, Nodes, T},
    io:format("Node ~w forwarded probe started by node ~w~n", [MyKey, RefKey]).
```