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

start(Id, Master, Jitter) ->
    spawn(fun() -> init(Id, Master, Jitter) end).

init(Id, Master, Jitter) ->
    receive
        {peers, Nodes} ->
            server(Id, Master, lists:delete(self(), Nodes), Jitter, newVC(length(Nodes),[]) , [] )
    end.

newVC(0, List) ->
    list_to_tuple(List);
newVC(N, List) ->
    newVC(N-1, [0|List]).

server(Id, Master, Nodes, Jitter, VC, Queue) ->
    receive
        {send, Msg} ->
            NewVC = incrementVC(Id,VC),
            multicast(Msg, Nodes, Jitter, Id , NewVC ),
            Master ! {deliver, Msg},
            server(Id, Master, Nodes, Jitter, NewVC , Queue );
        {multicast, Msg, FromId, MsgVC} ->
            case checkMsg(FromId, MsgVC, VC, size(VC)) of
                ready ->
                    Master ! {deliver,Msg},
                    NewVC = incrementVC( FromId , VC ),
                    {NewerVC, NewQueue} = deliverReadyMsgs(Master, NewVC, Queue, Queue),
                    server(Id, Master, Nodes, Jitter, NewerVC, NewQueue);
                wait ->
                    server(Id, Master, Nodes, Jitter, VC, [{FromId, MsgVC, Msg}|Queue])
            end;
        stop ->
            ok;
        Error ->
            io:format("Process ~w: unsupported message: ~w~n", [Id, Error])
    end.

%% Increment position N of vector clock VC
incrementVC(N, VC) ->
    setelement(N,VC,element(N,VC)+1).

%% Check if a message can be delivered to the master
checkMsg(_, _, _, 0) -> ready;
checkMsg(FromId, MsgVC, VC, FromId) ->
    if ( element(FromId,MsgVC) == element(FromId,VC)+1 ) ->
           checkMsg(FromId, MsgVC, VC, FromId-1);
       true -> wait
    end;
checkMsg(FromId, MsgVC, VC, N) ->
    if ( element(N, MsgVC) =< element(N, VC) ) ->
           checkMsg(FromId, MsgVC, VC, N-1);
       true -> wait
    end.

%% Deliver to the master all the ready messages in the hold-back queue
deliverReadyMsgs(_, VC, [], Queue) ->
    {VC, Queue};
deliverReadyMsgs(Master, VC, [{FromId, MsgVC, Msg}|Rest], Queue) ->
    case checkMsg(FromId, MsgVC, VC, size(VC)) of
        ready ->
            Master ! {deliver, Msg},
```

```
            NewVC = incrementVC( FromId ✔ , VC ✔ ),
            NewQueue = lists:delete({FromId, MsgVC, Msg}, Queue),
            deliverReadyMsgs(Master, NewVC, NewQueue, NewQueue);
        wait ->
            deliverReadyMsgs(Master, VC, Rest, Queue)
    end.

multicast(Msg, Nodes, 0, Id, VC) ->
    lists:foreach(fun(Node) ->
                      Node ! {multicast, Msg, Id ✔ , VC ✔ }
                  end,
                  Nodes);
multicast(Msg, Nodes, Jitter, Id, VC) ->
    lists:foreach(fun(Node) ->
                      T = rand:uniform(Jitter),
                      timer:send_after(T, Node, {multicast, Msg, Id ✔ , VC ✔ })
                  end,
                  Nodes).
```