Ordy: ordered multicasts

Jordi Guitart

Adapted with permission from Johan Montelius (KTH)

February 9, 2023

Introduction

The task is to implement a multicast service that supports different message orderings. Before you start you should have good theoretical knowledge of causally-ordered multicast implemented using vector clocks and totally-ordered multicast where the processes jointly agree on the sequence numbers of the messages.

This assignment includes a number of coding and experimental milestones. On accomplishing each of them, the team must show their outcome to the professor during the practical sessions, so that he can track the progress and evaluate the team accordingly.

This assignment also includes a number of open questions. The team must answer each of them in a report, which must be formatted according to the provided template and must be uploaded in PDF format through the *Racó* before the deadline.

1 The architecture

We will have a set of workers that communicate with each other using multicast messages. Each worker will have access to a multicast process that will hide the complexity of the system. The multicast processes are connected to each other and will have to agree on an order on how to **deliver** the messages. There is a clear distinction between receiving a message, which is done by the multicast process, and delivering a message, which is when the worker sees the message. A multicast process will receive messages in unspecified order but only deliver the messages to its worker in a given order. In particular, we will consider causal and total order.

1.1 The worker

In our example, a worker will represent a user in a distributed newsgroup. Each worker will publish posts at random intervals (or will respond to posts published by other workers) by sending multicast messages to the group.

Message ordering is desirable in a distributed newsgroup. In particular, if FIFO ordering is supported then every posting from a given user will be received in the same order, and users can talk consistently about that user's second posting. Causal ordering will ensure that response posts appear after the messages to which they refer. Finally, total ordering will allow the global

message numbering to be consistent between users, although maybe this is not compliant with FIFO and/or causal ordering.

We will experiment with different implementations and to prepare for the future we make the initialization a bit cumbersome. To start with, we provide some parameters to the worker to make experiments easier to manage. We can change the module of the multicast process and we can experiment with different values of sleep and jitter time. The sleep time is for up to how many milliseconds the workers should wait until the next new post is sent (responses to existing posts are sent immediately) and the jitter time is a parameter to the multicast process to simulate different network delays for the individual messages.

When started, the worker should register with a group manager and will be given the process identifier of the other members in the group.

Since the purpose of the exercise is not to debug the worker, the code is given in 'Appendix A'.

2 Basic multicast

Initially, we will use a system process that implements basic multicast. We give a set of peer processes to each multicast process and when it is told to multicast a message, the message is simply sent to the rest of peers, one by one.

To make the experiments more interesting we include a `Jitter` parameter when we start the process. The process will set an asynchronous timer up to these many milliseconds before sending each message. This will allow messages to interleave and possibly cause problems to the workers.

```
-module(basic).
-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)
    end.

server(Id, Master, Nodes, Jitter) ->
    receive
        {send, Msg} ->
            multicast(Msg, Nodes, Jitter),
            Master ! {deliver, Msg},
            server(Id, Master, Nodes, Jitter);
        {multicast, Msg} ->
            Master ! {deliver, Msg},
            server(Id, Master, Nodes, Jitter);
        stop ->
            ok
    end.
```

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

You can use the following program to test the basic multicast system with different values for `Sleep` and `Jitter`. `Sleep` stands for up to how many milliseconds the workers should wait until the next message is sent. `Jitter` stands for up to how many milliseconds the messages are delayed in the network. `Duration` stands for the duration of the experiment in milliseconds. Note that we are using the name of the multicast module (i.e. `basic`) as a parameter to the start procedure. We will easily be able to test different multicast implementations. Note also that workers are spawned in different nodes, hence you need to start an Erlang runtime for each one and name it accordingly.

```
-module(ordy).
-export([start/4, stop/0, pause/0, resume/0]).

start(Module, Sleep, Jitter, Duration) ->
    register(ordy, spawn(fun() -> init(Module, Sleep, Jitter, Duration) end)).

stop() ->
    ordy ! stop.

pause() ->
    ordy ! pause.

resume() ->
    ordy ! resume.

init(Module, Sleep, Jitter, Duration) ->
    Ctrl = self(),
    spawn('p1@127.0.0.1', fun() -> group_leader(whereis(user), self()),
                          worker:start("P1", Ctrl, Module, 1, Sleep, Jitter)
                          end),
    spawn('p2@127.0.0.1', fun() -> group_leader(whereis(user), self()),
                          worker:start("P2", Ctrl, Module, 2, Sleep, Jitter)
                          end),
    spawn('p3@127.0.0.1', fun() -> group_leader(whereis(user), self()),
                          worker:start("P3", Ctrl, Module, 3, Sleep, Jitter)
                          end),
    spawn('p4@127.0.0.1', fun() -> group_leader(whereis(user), self()),
```

```
                        worker:start("P4", Ctrl, Module, 4, Sleep, Jitter)
                        end),
    Workers = collect(4, [], []),
    run(Workers, Duration).

collect(N, Workers, Peers) ->
    if
        N == 0 ->
            lists:foreach(fun(W) -> W ! {peers, Peers} end, Workers),
            Workers;
        true ->
            receive
                {join, W, P} ->
                    collect(N-1, [W|Workers], [P|Peers])
            end
    end.

run(Workers, Duration) ->
    receive
        stop ->
            lists:foreach(fun(W) -> W ! stop end, Workers);
        pause ->
            lists:foreach(fun(W) -> W ! pause end, Workers),
            run(Workers, Duration);
        resume ->
            lists:foreach(fun(W) -> W ! resume end, Workers),
            run(Workers, Duration)
    after Duration ->
        io:format("Stopping...~n"),
        lists:foreach(fun(W) -> W ! stop end, Workers),
        io:format("Stopped~n")
    end.
```

EXPERIMENTAL MILESTONES (1 points)	
MS1 (1 points)	Set up the basic multicast system, make tests with different `Sleep` and `Jitter` parameters, and look for examples of the orderings that are not fulfilled.

OPEN QUESTIONS (0.25 points)
a) Are the posts displayed in FIFO, causal, and total order? Justify why.

3 Causal order multicast

We will implement a causal order multicast service using vector clocks. Take a copy of the module `basic` and call it `causal`.

The vector clock of each process is going to be represented with a tuple. Its size corresponds to the number of processes in the group, and can be created

with this invocation: `newVC(length(Nodes), [])`, being this function defined as follows:

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

You must modify the code to keep track of the vector clock of each process (i.e. `VC`), to include its vector clock and its identifier when it sends a multicast message, and to update the vector clock when necessary. In particular, **a process must increase its position before sending a multicast message and the position of the sender process after delivering a multicast message**. You can use the Erlang functions `element/2` and `setelement/3` to read and update a position of the vector clock, respectively. In addition, you must add a hold-back queue (i.e. `Queue`) so that a process can postpone the delivery of a given message to the worker until two conditions are met, namely that the message is the next one expected from the sender and the process has seen all the messages seen by the sender before sending the message.

```
{multicast, Msg, FromId, MsgVC} ->
    case checkMsg(FromId, MsgVC, VC, size(VC)) of
        ready ->
            %% TODO: ADD SOME CODE
            NewVC = incrementVC ( ... , ... ),  %% TODO: COMPLETE
            {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;
```

This new multicast functionality (which replaces the former basic multicast) is supported by the following three procedures:

```
%% Increment position N of vector clock VC
incrementVC(N, VC)
    %% TODO: ADD SOME CODE

%% Check if a message can be delivered to the master
checkMsg(_, _, _, 0) -> ready;
checkMsg(FromId, MsgVC, VC, FromId) ->
    if ( ... == ... ) ->  %% TODO: COMPLETE
           checkMsg(FromId, MsgVC, VC, FromId-1);
       true -> wait
    end;
checkMsg(FromId, MsgVC, VC, N) ->
    if ( ... =< ... ) ->  %% TODO: COMPLETE
           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 ->
            %% TODO: ADD SOME CODE
            NewVC = incrementVC ( ... , ... ),  %% TODO: COMPLETE
            NewQueue = lists:delete({FromId, MsgVC, Msg}, Queue),
            deliverReadyMsgs(Master, NewVC, NewQueue, NewQueue);
        wait ->
            deliverReadyMsgs(Master, VC, Rest, Queue)
    end.
```

CODE REVIEW (2.5 points)
Causally-ordered multicast: Erlang file `causal.erl`.

EXPERIMENTAL MILESTONES (1 points)	
MS2 (1 points)	Set up the causal order multicast system and repeat the tests on `MS1` (looking for examples of the orderings that are not fulfilled).

OPEN QUESTIONS (0.25 points)
b) Are the posts displayed in FIFO, causal, and total order? Justify why.

4 Total order multicast

To avoid messages to be delivered out of order, we will implement a total order multicast service using a distributed algorithm. The algorithm is the one used in the ISIS system and is based on requesting proposals from all nodes in a group.

We will here go through the code but you will have to do some programing yourself. We are leaving '...' at places in the code where you have to fill in the right values.

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

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

init(Id, Master, Jitter) ->
    receive
        {peers, Nodes} ->
            server(Master, seq:new(Id), seq:new(Id), Nodes, [], [], Jitter)
    end.
```

The server procedure has the following state:

- **Master**: the worker process to which messages are delivered
- **MaxPrp**: the largest sequence number proposed so far
- **MaxAgr**: the largest agreed sequence number seen so far
- **Nodes**: all peers in the network
- **Cast**: a set of references to messages that have been sent out asking for proposals but for which no agreed sequence number exist yet
- **Queue**: hold-back queue holding messages that have been received but not yet delivered
- **Jitter**: this parameter induces some network delay

The sequence numbers are represented by a tuple {`N, Id`}, where `N` is an integer that is incremented every time we make a proposal and `Id` is our process identifier. In 'Appendix B', you will find a module `seq` to create, modify, and compare sequence numbers.

The `Cast` set is represented as a list of tuples {`Ref, L, Max`}, where `L` is the number of proposals that we are still waiting for and `Max` the highest proposal received so far.

The `Queue` is an ordered list of entries representing messages that we have received but we have not yet delivered. The list is ordered according to the sequence numbers of the messages (proposed or agreed). Entries with status `propsd` are those where we have proposed a sequence number. Status `agrd` identifies those entries with an agreed sequence number. If we have entries with agreed sequence numbers at the front of the queue these can be removed and delivered to the worker.

4.1 Sending of a message

A `send` message is a directive to multicast a message. We first have to agree in which order to deliver the message and therefore send a request for proposals to all peers (using the function `request/4`).

The request is sent to all nodes with a unique reference (`Ref`). This reference is also added to the set of pending proposals (`Cast`) with information on how many nodes have to report back.

```
server(Master, MaxPrp, MaxAgr, Nodes, Cast, Queue, Jitter) ->
receive
    {send, Msg} ->
        Ref = make_ref(),
        request(... , ... , ... , ...),
        NewCast = [{... , ..., seq:null()}|Cast],
        server(... , ... , ... , ... , ... , ... , ...);
```

Note that we are also sending a request to ourselves. We will handle our own proposal in the same way as proposals from everyone else. This might look strange but it makes the code much easier.

4.2 Receiving a request

When a process receives a `request` message it should reply with a `proposal` message. The sequence number to be proposed is calculated by incrementing by one the maximum of `MaxPrp` and `MaxAgr`, but including always as a suffix the process identifier of the actual process (you can use the `maxIncrement/2` function of module `seq`). This ensures that we never propose a sequence number that is lower than the ones we have proposed already or than any agreed number. We should also queue the message using the proposed sequence number as key and with status `propsd`. This is handled by the function `queue/5`.

```
{request, From, Ref, Msg} ->
    NewMaxPrp = ... ,
     ... ! { ... , ... , ...},
    NewQueue = queue(... , ... , ... , ... , ...),
    server(... , ... , ... , ... , ... , ... , ...);
```

4.3 Receiving a proposal

A `proposal` message is received as a reply to a request that we have sent earlier. It contains the message reference (`Ref`) and the proposed sequence number (`Num`).

If the proposal is the last one that we are waiting for, we can calculate the agreed sequence number. We implement this by calling the function `proposal/3` that will update the set of pending proposals and either return {`agreed, MaxNum, NewCast`} if an agreement was found or simply the updated cast set (`NewCast`).

If we have an agreement this should be sent to all nodes in the network. This is handled by the `agree/3` procedure.

```
{proposal, Ref, Num} ->
    case proposal(... , ... , ...) of
        {agreed, MaxNum, NewCast} ->
            agree(... , ... , ...),
            server(... , ... , ... , ... , ... , ... , ...);
        NewCast ->
            server(... , ... , ... , ... , ... , ... , ...)
    end;
```

4.4 Agree at last

An `agreed` message contains the agreed sequence number (`Num`) of a particular message with reference `Ref`. The message that is in the queue must be updated and possibly moved back in the queue (the agreed number could be higher than the proposed number). This is handled by the function `update/3`.

If the first message in the queue now has an agreed sequence number it could be delivered. The function `agreed/1` will remove the messages that can be delivered from the queue and return them in a list (`AgrMsg`). These messages can then be delivered using the `deliver/2` procedure. The largest agreed sequence number seen so far must be updated as the maximum of `MaxAgr` and `Num` (you can use the `max/2` function of module `seq`).

```
    {agreed, Ref, Num} ->
        NewQueue = update(... , ... , ...),
        {AgrMsg, NewerQueue} = agreed(...),
        deliver(... , ...),
        NewMaxAgr = ... ,
        server(... , ... , ... , ... , ... , ... , ...);
    stop ->
        ok
end.
```

The remaining procedures that support the previous functionality are as follows:

```
%% Send a request message to all nodes
request(Ref, Msg, Nodes, 0) ->
    Self = self(),
    lists:foreach(fun(Node) ->
                      %% TODO: ADD SOME CODE
                  end,
                  Nodes);
request(Ref, Msg, Nodes, Jitter) ->
    Self = self(),
    lists:foreach(fun(Node) ->
                      T = rand:uniform(Jitter),
                      timer:send_after(T, Node, ... ) %% TODO: COMPLETE
                  end,
                  Nodes).

%% Send an agreed message to all nodes
agree(Ref, Num, Nodes)->
    lists:foreach(fun(Node)->
                      %% TODO: ADD SOME CODE
                  end,
                  Nodes).

%% Deliver messages to the master
deliver(Master, Messages) ->
    lists:foreach(fun(Msg)->
                      %% TODO: ADD SOME CODE
                  end,
                  Messages).

%% Update the set of pending proposals
proposal(Ref, PrpNum, [{Ref, 1, Max}|Rest])->
    {agreed, seq:max(PrpNum, Max), Rest};
proposal(Ref, PrpNum, [{Ref, N, Max}|Rest])->
    [{Ref, N-1, seq:max(PrpNum, Max)}|Rest];
proposal(Ref, PrpNum, [Entry|Rest])->
    case proposal(Ref, PrpNum, Rest) of
        {agreed, AgrNum, NewRest} ->
```

```
            {agreed, AgrNum, [Entry|NewRest]};
        Updated ->
            [Entry|Updated]
    end.

%% Remove all messages in the front of the queue that have been agreed
agreed([{_Ref, Msg, agrd, _Agr}|Queue]) ->
    {AgrMsg, NewQueue} = agreed(Queue),
    {[Msg|AgrMsg], NewQueue};
agreed(Queue) ->
    {[], Queue}.

%% Update the queue with an agreed sequence number
update(Ref, AgrNum, [{Ref, Msg, propsd, _}|Rest])->
    queue(Ref, Msg, agrd, AgrNum, Rest);
update(Ref, AgrNum, [Entry|Rest])->
    [Entry|update(Ref, AgrNum, Rest)].

%% Queue a new entry using Number as key
queue(Ref, Msg, State, Number, []) ->
    [{Ref, Msg, State, Number}];
queue(Ref, Msg, State, Number, Queue) ->
    [Entry|Rest] = Queue,
    {_, _, _, Next} = Entry,
    case seq:lessthan(Number, Next) of
        true ->
            [{Ref, Msg, State, Number}|Queue];
        false ->
            [Entry|queue(Ref, Msg, State, Number, Rest)]
    end.
```

CODE REVIEW (3 points)
Totally-ordered multicast: Erlang file `total.erl`.

EXPERIMENTAL MILESTONES (1 points)	
MS3 (1 points)	Set up the total order multicast system and repeat the tests on `MS1` (looking for examples of the orderings that are not fulfilled).

OPEN QUESTIONS (1 points)

c) Are the posts displayed in FIFO, causal, and total order? Justify why.

d) Given a multicast system that is both totally-ordered (using an implementation like ours, which guarantees that the sequence number of any message sent is greater than that of any seen by the sender) and FIFO-ordered, justify whether it is also causally-ordered as a consequence.

e) We have a lot of messages in the system. Derive a theoretical quantification of the number of messages needed to deliver a multicast message (from the sending by a worker to the delivery by all the workers) as a function of the number of workers.

f) Compare with the basic multicast implementation regarding the number of messages needed.

Appendix A: *worker.erl*

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

start(Name, Main, Module, Id, Sleep, Jitter) ->
    spawn(fun() -> init(Name, Main, Module, Id, Sleep, Jitter) end).

init(Name, Main, Module, Id, Sleep, Jitter) ->
    Cast = apply(Module, start, [Id, self(), Jitter]),
    Main ! {join, self(), Cast},
    receive
        {peers, Peers} ->
            Cast ! {peers, Peers},
            Wait = if Sleep == 0 -> 0; true -> rand:uniform(Sleep) end,
            {ok, Tref} = timer:send_after(Wait, new_topic),
            worker(Name, Id, Cast, 1, 1, Sleep, Tref),
            Cast ! stop
    end.

worker(Name, Id, Cast, NxtSnd, NxtRcv, Sleep, Tref) ->
    receive
        {deliver, {FromName, From, Msg, SenderN, Nre}} ->
            io:format("~s RECV(~4w) ; From: ~s(~4w) ; Subject: ~s~n",
                      [Name, NxtRcv, FromName, SenderN, Msg]),
            if From == Id ->
                  worker(Name, Id, Cast, NxtSnd, NxtRcv+1, Sleep, Tref);
               true ->
                  Op = rand:uniform(),
                  if (Op >= 0.8) and (Tref /= null) and (Nre < 3) ->
                        cast_response(Name, Id, Cast, Msg, NxtSnd, Nre+1),
                        worker(Name, Id, Cast, NxtSnd+1, NxtRcv+1, Sleep, Tref);
                     true ->
                        worker(Name, Id, Cast, NxtSnd, NxtRcv+1, Sleep, Tref)
                  end
            end;
        new_topic ->
            cast_new_topic(Name, Id, Cast, NxtSnd),
            Wait = if Sleep == 0 -> 0; true -> rand:uniform(Sleep) end,
            {ok, NewTref} = timer:send_after(Wait, new_topic),
            worker(Name, Id, Cast, NxtSnd+1, NxtRcv, Sleep, NewTref);
        pause ->
            timer:cancel(Tref),
            worker(Name, Id, Cast, NxtSnd, NxtRcv, Sleep, null);
        resume ->
            Wait = if Sleep == 0 -> 0; true -> rand:uniform(Sleep) end,
            {ok, NewTref} = timer:send_after(Wait, new_topic),
            worker(Name, Id, Cast, NxtSnd, NxtRcv, Sleep, NewTref);
        stop ->
            ok;
```

```
        Error ->
            io:format("strange message: ~w~n", [Error]),
            worker(Name, Id, Cast, NxtSnd, NxtRcv, Sleep, Tref)
    end.

cast_new_topic(Name, Id, Cast, N) ->
    Sbj = lists:sublist(shuffle_list("abcdefghijklmnopqrstuvwxyz"), 8),
    Msg = {Name, Id, Sbj, N, 0},
    io:format("~s SEND(~4w) ; Subject: ~s~n", [Name, N, Sbj]),
    Cast ! {send, Msg}.

cast_response(Name, Id, Cast, Text, N, Nre) ->
    Sbj = "Re:" ++ Text,
    Msg = {Name, Id, Sbj, N, Nre},
    io:format("~s SEND(~4w) ; Subject: ~s~n", [Name, N, Sbj]),
    Cast ! {send, Msg}.

shuffle_list(L) ->
    [X||{_,X} <- lists:sort([ {rand:uniform(), N} || N <- L])].
```

Appendix B: *seq.erl*

```
-module(seq).
-export([null/0, new/1, maxIncrement/2, max/2, lessthan/2]).

%%% Functions to handle the sequence numbers.
null() ->
    {0,0}.

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

maxIncrement({Pn, Pi}, {Nn, _}) ->
    {erlang:max(Pn, Nn) + 1, Pi}.

max(P, N) ->
    case lessthan(P, N) of
        true ->
            N;
        false ->
            P
    end.

lessthan({Pn, Pi}, {Nn, Ni}) ->
    (Pn < Nn) or ((Pn == Nn) and (Pi < Ni)).
```