Totty: a total order multicast

Jordi Garcia

Adapted with permission from Jordi Guitart & Johan Montelius (KTH)

November 19, 2018

Introduction

The task is to 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.

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 un-specified order but only deliver the messages to its worker in a **total order** i.e. all workers in the system will deliver the messages in the same order.

1.1 The worker

A worker is in our example a process that at random intervals wish to send a multicast message to the group. The worker will wait until it sees its own message before it sends another one to prevent an overflow of messages in the system.

The worker will be connected to a gui process that is simply a colored window. The window is initially black or in RGB talk {`0, 0, 0`}. This is also the initial state of the worker. Each message that is delivered to the worker is an integer N in some interval, say 1 to 20. A worker will change its state by adding N to the R value and rotate the values. If the state of the worker is {`R, G, B`} and the worker is delivered message N, the new state is {`G, B, (R+N) rem 256`}.

The color of a worker will thus change over time and the order of messages is of course important. The sequence `5, 12, 2` will of course not create the same color as the sequence `12, 5, 2`. If all workers start with a black color and we fail to deliver the messages in the same order the colors of the workers will start to diverge.

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 message is sent 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 initial state and 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'. You will also find in 'Appendix B' the code of a simple gui.

2 Basic multicast

Initially, we will use a process that implements basic multicast. We give a set of peer processes to this 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) ->
    {A1,A2,A3} = now(),
    random:seed(A1, A2, A3),
    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, _From, Msg} ->
            Master ! {deliver, Msg},
            server(Id, Master, Nodes, Jitter);
        stop ->
            ok
    end.

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

Experiments. Set up the basic multicast system, and use the following test program to experiment with different values for `Sleep` and `Jitter`. Does it keep workers synchronized? Justify why. Note that we are using the name of the module (i.e. `basic`) as a parameter to the start procedure. We will easily be able to test different multicast implementations. `Sleep` stands for the average number of milliseconds the workers should wait until the next message is sent. `Jitter` stands for the average number of milliseconds of network delay.

```
-module(toty).
-export([start/3, stop/0]).

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

stop() ->
    toty ! stop.

init(Module, Sleep, Jitter) ->
    Ctrl = self(),
    worker:start("P1", Ctrl, Module, 1, Sleep, Jitter),
    worker:start("P2", Ctrl, Module, 2, Sleep, Jitter),
    worker:start("P3", Ctrl, Module, 3, Sleep, Jitter),
    worker:start("P4", Ctrl, Module, 4, Sleep, Jitter),
    collect(4, [], []).

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

run(Workers) ->
    receive
        stop ->
            lists:foreach(
                fun(W) ->
                    W ! stop
                end,
                Workers)
    end.
```

3 Total order multicast

To avoid messages to be delivered out of order, we will implement a total order multicaster. 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) ->
    {A1,A2,A3} = now(),
    random:seed(A1, A2, A3),
    receive
        {peers, Nodes} ->
            server(Master, seq:new(Id), seq:new(Id), Nodes, [], [], Jitter)
    end.
```

The server procedure has the following state:

- **Master**: the 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 but no final sequence number have yet been assigned
- **Queue**: 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 C', you will find code to create, modify, and compare sequence numbers.

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

The `Queue` is an ordered list of entries representing messages that we have received but for which no agreed value exist. The list is ordered based in the proposed or agreed sequence number. The proposed entries are entries where we have proposed a 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.

3.1 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 should be sent to all nodes with a unique reference. This reference is also added to the casted set with information on how many nodes have to report back (using the function `cast/3`).

```
server(Master, MaxPrp, MaxAgr, Nodes, Cast, Queue, Jitter) ->
receive
    {send, Msg} ->
        Ref = make_ref(),
        request(... , ... , ... , ...),
        NewCast = cast(... , ... , ...),
        server(... , ... , ... , ... , ... , ... , ...);
```

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

3.2 Receiving a request

When the process receives a `request` message it should reply with a new proposed sequence number. The sequence number to be proposed is calculated by incrementing by one the maximum of `MaxAgr` and `MaxPrp`. What ever happens we must not propose a lower sequence number than the ones we have proposed already. It should also queue the message using the proposed sequence number as key. This is handled by the function `insert/4`.

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

3.3 Receiving a proposal

A `proposal` message is sent as a reply to a request that we have sent earlier. The proposal contains the message reference and the proposed sequence number.

If the proposal is the last proposal that we are waiting for, we have also found an agreed sequence number. We implement this by calling the function `proposal/3` that will update the set and either return {`agreed, MaxSeq, NewCast`} if an agreement was found or simply the updated `Cast` list.

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

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

3.4 Agree at last

An `agreed` message contains the agreed sequence number of a particular message. 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`.

```
    {agreed, Ref, Seq} ->
        Updated = update(... , ... , ...),
        {Agreed, NewQueue} = agreed(...),
        deliver(... , ...),
        NewMaxAgr = ... ,
        server(... , ... , ... , ... , ... , ... , ...);
    stop ->
        ok
end.
```

If the first message in the queue now has an agreed sequence number it could be delivered. The function `agreed/2` will remove the messages that can be delivered and return them in a list. These messages can then be delivered using the `deliver/2` procedure. The largest agreed sequence number it has seen so far must be updated as the maximum of `MaxAgr` and `Seq`.

The remaining code that support the previous functionality is as follows:

```
%% Sending 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 = random:uniform(Jitter),
                      timer:send_after(T, Node, ... ) %% TODO: COMPLETE
                  end,
                  Nodes).

%% Sending an agreed message to all nodes
agree(Ref, Seq, Nodes)->
```

```
    lists:foreach(fun(Pid)->
                        %% TODO: ADD SOME CODE
                  end,
                  Nodes).

%% Delivering messages to the master
deliver(Master, Messages) ->
    lists:foreach(fun(Msg)->
                        Master ! {deliver, Msg}
                  end,
                  Messages).

%% Adding a new entry to the set of casted messages
cast(Ref, Nodes, Cast) ->
    L = length(Nodes),
    [{Ref, L, seq:null()}|Cast].

%% Update the set of casted messages
proposal(Ref, Proposal, [{Ref, 1, Sofar}|Rest])->
    {agreed, seq:max(Proposal, Sofar), Rest};
proposal(Ref, Proposal, [{Ref, N, Sofar}|Rest])->
    [{Ref, N-1, seq:max(Proposal, Sofar)}|Rest];
proposal(Ref, Proposal, [Entry|Rest])->
    case proposal(Ref, Proposal, Rest) of
        {agreed, Agreed, Rst} ->
            {agreed, Agreed, [Entry|Rst]};
        Updated ->
            [Entry|Updated]
    end.

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

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

%% Insert a new message into the queue
```

```
insert(Ref, Msg, Proposal, Queue) ->
    queue(Ref, Msg, propsd, Proposal, Queue).

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

Experiments. i) Set up the total order multicast system, and repeat the previous tests. Does it keep workers synchronized? ii) We have a lot of messages in the system. Derive a theoretical quantification of the number of messages needed to deliver a multicast message as a function of the number of workers and check experimentally that your formulation is correct. iii) Compare with the basic multicast implementation regarding the number of messages needed.

Appendix A: *worker.erl*

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

-define(change, 20).

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

init(Name, Grp, Module, Id, Sleep, Jitter) ->
    {A1,A2,A3} = now(),
    random:seed(A1, A2, A3),
    Gui = gui:start(Name),
    Cast = apply(Module, start, [Id, self(), Jitter]),
    Grp ! {join, self(), Cast},
    receive
        {state, Color, Peers} ->
            Cast ! {peers, Peers},
            Gui ! {color, Color},
            cast_change(Id, Cast, Sleep),
            worker(Id, Cast, Color, Gui, Sleep),
            Cast ! stop,
            Gui ! stop
    end.

worker(Id, Cast, Color, Gui, Sleep) ->
    receive
        {deliver, {From, N}} ->
            Color2 = change_color(N, Color),
            Gui ! {color, Color2},
            if
                From == Id ->
                    cast_change(Id, Cast, Sleep);
                true ->
                    ok
            end,
            worker(Id, Cast, Color2, Gui, Sleep);
        stop ->
            ok;
        Error ->
            io:format("strange message: ~w~n", [Error]),
            worker(Id, Cast, Color, Gui, Sleep)
    end.
```

```
change_color(N, {R,G,B}) ->
    {G, B, ((R+N) rem 256)}.

cast_change(Id, Cast, Sleep) ->
    Msg = {Id, random:uniform(?change)},
    timer:send_after(Sleep, Cast, {send, Msg}).
```

Appendix B: *gui.erl*

```
-module(gui).
-export([start/1]).
-define(width, 200).
-define(height, 200).
-include_lib("wx/include/wx.hrl").

start(Name) ->
    spawn_link(fun() -> init(Name) end).

init(Name) ->
    Frame = make_frame(Name),
    loop(Frame).

make_frame(Name) ->     %Name is the window title
    Server = wx:new(),  %Server will be the parent for the Frame
    Frame = wxFrame:new(Server, -1, Name, [{size,{?width, ?height}}]),
    wxFrame:setBackgroundColour(Frame, ?wxBLACK),
    wxFrame:show(Frame),
    %monitor closing window event
    wxFrame:connect(Frame, close_window),
    Frame.

loop(Frame)->
    receive
        %check if the window was closed by the user
        #wx{event=#wxClose{}} ->
            wxWindow:destroy(Frame),
            ok;
        {color, Color} ->
            color(Frame, Color),
            loop(Frame);
        stop ->
            ok;
        Error ->
            io:format("gui: strange message ~w ~n", [Error]),
            loop(Frame)
    end.

color(Frame, Color) ->
    wxFrame:setBackgroundColour(Frame, Color),
    wxFrame:refresh(Frame).
```

Appendix C: *seq.erl*

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

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

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

increment({Pn, Pi}) ->
    {Pn+1, Pi}.

max(Proposal, Sofar) ->
    case lessthan(Proposal, Sofar) of
        true ->
            Sofar;
        false ->
            Proposal
    end.

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