Toty: a total order multicast

Johan Montelius

October 2, 2016

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 unspecified 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 a 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.

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. This might look like over-kill but we will use it in the coming seminars.

1.2 basic multicast

As a first experiment we should use a process that implements basic multicast. The multicaster is simply give a set of peer processes and when it is told to multicast a message the message is simply sent to all the peers, one by one.

To make the experiments more interesting we include a jitter parameter when we start the process. The process will wait up to this many milliseconds before sending the next message. This will allow messages to interleave and possibly cause problems for the workers.

1.3 experiment

Experiment with different values for sleep and jitter and see when messages are not delivered in total order.

Ponder what would happen if we had a system that relied on total order delivery but that this was not clearly stated. If congestion was low and we did not have any delays in the network, how long time does it take before messages are delivered out of order? How hard would it be to debug this system and figure out what went wrong?

2 total order multicast

The multicast process is of course the tricky part. We will here go through the code but you will have to do some programming yourself.

The main procedure has the following state:

- Master: the process to which messages are delivered
- Next: the next value to propose
- 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: the jitter parameter that 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.

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

The `Queue` is an ordered list of entries represented messages that we have been 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.

2.1 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.

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 that have to report back. We're leaving `?` at places in the code where you have to fill in the right values.

```
{send, Msg} ->
    Ref = make_ref(),
    request(?, ?, ?, ?),
    Cast2 = cast(?, ?, ?),
    server(Master, Next, Nodes, Cast2, Queue, Jitter);
```

Note that we're 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.

2.2 receiving a request

When the process receives a request it should reply with a new sequence number. It should also queue the message using the proposed sequence number as key.

```
{request, From, Ref, Msg} ->
    From ! {proposal, ?, ?},
```

```
    Queue2 = insert(?, ?, ?, ?),
    Next2 = increment(?),
    server(Master, Next2, Nodes, Cast, Queue2, Jitter);
```

We increment the value for the next sequece number, what ever happens we must not propose the same or lower sequence number than the ones we have proposed already.

2.3 receiving a proposal

A proposal 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, Seq, Cast2`} if an agreement was found or simply the updated list.

```
{proposal, Ref, Proposal} ->
    case proposal(?, ?, ?) of
        {agreed, Seq, Cast2} ->
            agree(?, ?, ?),
            server(Master, Next, Nodes, Cast2, Queue, Jitter);
        Cast2 ->
            server(Master, Next, Nodes, Cast2, Queue, Jitter)
    end;
```

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

2.4 agree at last

An agree 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`.

We also need to increment our next

```
{agreed, Ref, Seq} ->
    Updated = update(?, ?, ?),
    {Agreed, Queue2} = agreed(?),
    deliver(?, ?),
    Next2 = increment(?, ?),
    server(Master, Next, Nodes, Cast, Queue2, Jitter);
```

If the agreed sequence number is greater than the one that we currently have we need to increment our value; we must make sure that we never propose a value that could be lower than an agreed value.

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.

3 experiments

Try running the tests using the total order multicaster. Does it keep workers synchronized? We have a lot of messages in the system, how many messages can we multicast per second and how does this depend on the number of workers?

Build a larger distributed network, how large can it be before we are down on our knees?