An exercise in communication

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Introduction

This exercise is partly about breaking up a problem in layers of abstractions and implement each layer as a process or a set of processes. We will model our solution as communicating finite state machines and each machine is of course implemented as an Elixir process.

The problem that we will look at is how to create a communication abstraction that provides addressing and FIFO delivery over a lossy channel. The situation will look similar to that of TCP/IP but our problem statement and solution will be much simpler.

1 A transport abstraction

Assume that we want to create a *transport service*. The service should provide the following functionality:

- identity: The service should provide addresses, we should be able to send and receive messages to other processes given an address.
- flow control: The service should take care of flow control so we should not be able to send more messages than the receiver can consume.
- ordered delivery: Packets should be delivered in order.
- reliable delivery: Packets should be delivered even if the underlying communication channel is lossy.

This is not that hard to achieve and the solution could easily be be described in a state diagram with 15 states and 123 transitions..... The problem we would have is that, if we want to solve all these properties in one go, we would have so many things to keep track of that the solution would look like a bowl of spaghetti.

Instead of solving all problems at once we divide the solution into layers, each layer providing a service that brings us closer to the final solution. The

layers that we will implement have similarities with the ISO communication layers or the layers in the IP stack. The layers that we will work with are:

- link: This layer is given, it will send a *frame* on a wire but does not know very much more. There are no guarantees that the frame will arrive nor that frames will arrive in order.
- network: This layer will introduce *network addresses* so we can have several nodes connected together. It uses the link layer straight off so we do not add ordered delivery nor guarantees of message delivery.
- order: This layer will add re-transmissions of missing messages and keep track of ordering of messages.
- flow: This layer will solve the flow-control problem.

The TCP protocol is similar to our order and flow layers but also implements congestion avoidance, process addressing, stream to datagram segmentation etc. In this implementation we only provide reliability and flow control to make things a bit easier. We will also implement them as separate layers to show how simple each layer can be implemented.

If we draw a diagram over the complete system it will look like Fig.1. Each layer will have one or more processes that implements the right abstractions. In its first incarnation each layer will be handled by a single process but when we extend the system we might want to have separate processes for different directions.

We now start form the ground up and implement layer by layer. We should be able to test that a layer works before implementing the next layer.

2 link layer

The link layer process is, as shown in Fig.2 quite simple. We will first start the process and give it the process identifier of the master process. We will then send it a message that connect the process with a destination link layer process.

Note the order here; we can not provide the link layer process with its connection point when we create the process. If we tried we would end up in a chicken or the egg question. We solve this by first creating both link processes and the send them a message with the process identifier of their peer.

The link layer should be able to handle the following messages:

- `{:send, msg}` : A message from the master (the network layer process), telling the link process to send the message, `msg`, to the destina-

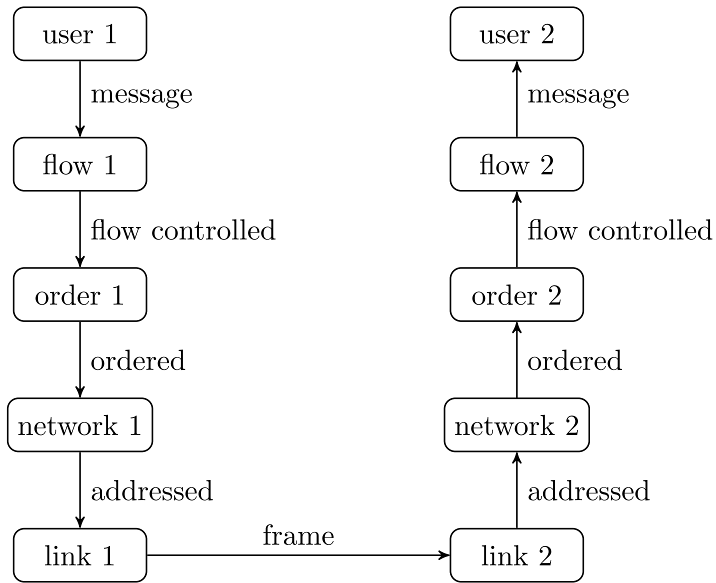


Figure 1: The architecture.

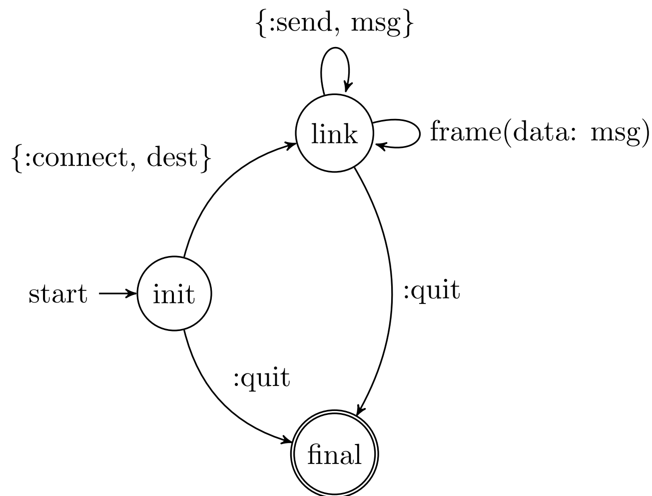


Figure 2: The link layer.

tion process. The message is wrapped in a *Frame* and sent using the regular Elixir send primitive.

- `frame(data: msg)`: A frame from the peer link process. This message should be forwarded to the master process using the regular Elixir send primitive. The message is sent as is, `msg`, without being wrapped in any special data structure.

The Elixir implementation is given in Appendix A. Before we continue we write a small test program and connect two nodes together.

```
def test_link() do
  sender = spawn(fn() -> sender() end)
  receiver = spawn(fn() -> receiver() end)
  {:ok, ls} = Link.start(sender)
  {:ok, lr} = Link.start(receiver)
  send(ls,{:connect, lr})
  send(lr, {:connect, ls})
  send(sender, {:connect, ls})
  send(receiver, {:connect, lr})
  :ok
end
```

This program will start two processes, the `sender` and the `receiver`; note the order in which we have to start the processes. The two master processes are started first in order to provide the link processes with their respective master. Once the link processes are created we can connect them together and then connect the master processes to the link processes.

The `link_sender` process could look like follows. Implement the corresponding receive process and see if your systems runs.

```
def sender() do
  receive do
    {:connect, lnk} ->
      :io.format("sender connected to link ~w ~n", [lnk]),
      :io.format("sending hello~n", [])
      send(lnk, {:send, :hello})
  end
end
```

3 a hub

If we only have two nodes in our network it is rather pointless, so we will implement a “hub” to which we can connect several nodes. The hub will

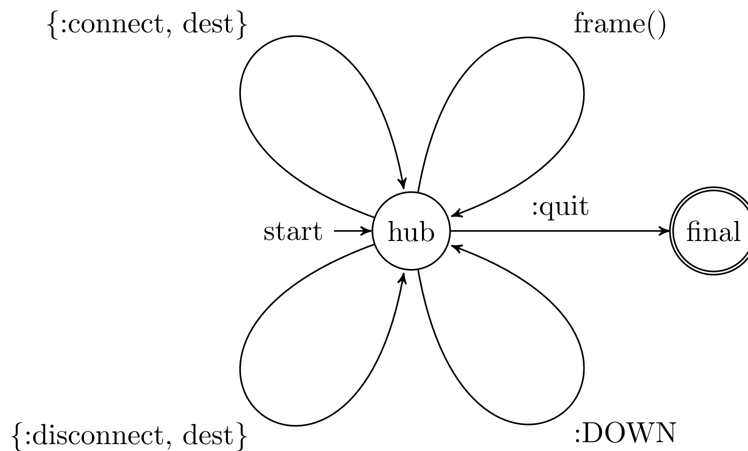


Figure 3: The hub.

accept new connection and forward incoming frames to all connected nodes. The state machine is shown in Fig. 3. In the implementation we only need to solve how connections should be stored and how to implement broadcasting of messages.

As an extra precaution we add monitoring of connected link layer processes. If a process dies we remove it from the set of connected processes. This will save the state of the hub from growing in an environment where nodes fail before having been disconnected.

An implementation can be found in Appendix A. Write a small test program, similar to the previous one, where two link layer processes are connected using a hub.

4 network layer

So we now have a link layer that works but the thing is that the hub will happily broadcast any message to all connected nodes. If we connect three nodes to the hub all three nodes will receive all messages. We need a network layer that introduces an addressing scheme so we can send a message to a particular node in the network.

In Fig.4 we have the state diagram of the network layer and as you see it is not much more complicated than then link layer. You might wonder why

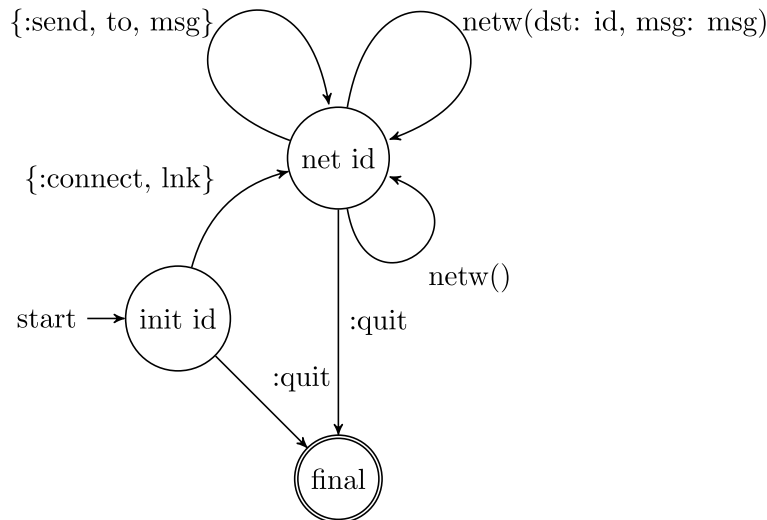


Figure 4: The network layer.

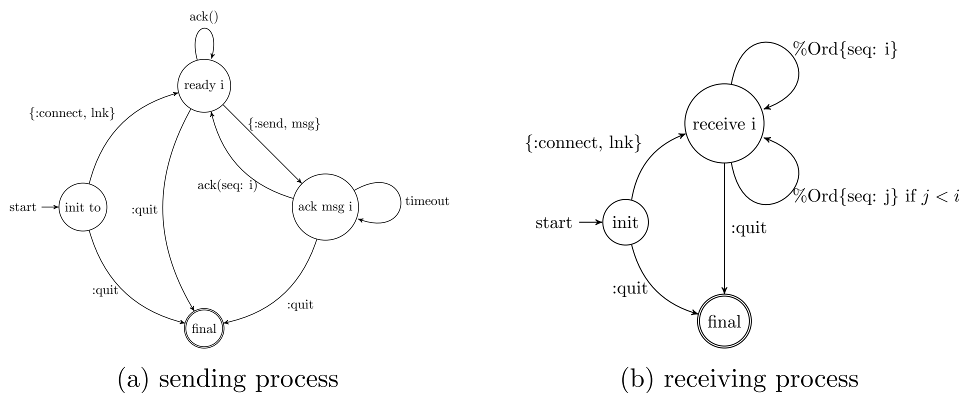
we did not implement the network layer directly since the link layer does not give us any thing besides message delivery but we might want to add more features, such as error detection, in the link layer and want to be able to do this without modifying the network layer.

The network layer is given a unique identifier when it is started and it will use this address to filter incoming messages. When we start the process we also provide its master process so that it knows to whom it should redirect incoming messages. Before we start sending messages we connect the processes to a link layer process; following the same pattern as for the test procedures for the link layer.

In Appendix **??** you will find an implementation of a network layer process. Implement a simple test function that as before creates a sender and receiver process that are connected using two network layer processes. The tricky part is how to start the processes in the right order: application layer, network layer, link layer, connect link layers, connect network layer to link layer, connect application layer to network layer.

5 order and reliability

In the next layer we will create a process for one particular flow of messages. We will then make sure that these messages are delivered properly by adding a sequence number to the messages and re-transmit messages that could be lost.



(a) sending process

(b) receiving process

Figure 5: The ordering layer.

If messages can come out of order we need a way to impose some order. The ordering is solved if the sending process numbers the messages as they are sent and the receiver will buffer received messages and deliver them in the right order.

A problem occurs if a message is lost; if we do not do anything to solve this the receiver will wait forever for a message that will never arrive. The solution is of course to let the sender keep a copy of each message in case it is lost and to let the receiver request a re-transmission. This can be solved either by explicitly request transmissions or by acknowledge messages that do arrive correctly, we will choose the latter alternative.

To understand how the layer works we describe the process of a sender process and the receiver process as two different state machines. In the implementation we will however combine these machines into one Elixir process.

In Fig.5 we see the state machines of the sender and receiver processes. This description in simplified in that it will only allow one outstanding message. The sender will be directed to send a message to the receiver and will move in to a state where it is waiting for an *acknowledgement* for that particular message *i*. If everything works it will receive the *Ack* message and go back to the ready state. If no acknowledge message is received it will receive a timeout and resend the message (*msg*).

The receiver knows what sequence number it is waiting. When the correct message arrives it will send an acknowledge message in return and forward it to its master. It could be that the sender has already sent a copy of the message so it must be prepared to discard messages that it has already seen. The sender must likewise be prepared to discard *Ack* messages that it has seen.

Both processes are started and given a specified network address that it is communication with. It is then given access to a network process to

use when sending messages. The network processes have addresses and we here assume that the sender and receiver are the only processes that are communicating using these addresses. The network processes are thus dedicated to this communication channel.

In this simple version we only allow one message to be outstanding at any given time. Only when a message has been acknowledged will we send the next message. This is of course a severe limitation; our communication channel will be depending on the round trip thus being very slow.

To solve this we extend the sender, as shown in Fig. 6, to allow several outstanding messages at the same time. We extend the receive *ack* state so that we can send new messages while we are waiting for an acknowledgment. We have too keep track of all sent messages and make sure that we resend the right one so we keep all sent messages, with their corresponding sequence number, in a buffer (an list of all messages with the last message sent last).

The sender can now receive the following messages:

- {`:send, msg`}: The process will send the message in a numbered datagram to the receiving process. It will store the message with its corresponding sequence number as the last element in the buffer. The sequence number is then updated.
- `ack(seq: a)`: if a is equal to the sequence number of the first element in the buffer then this element is removed from the buffer.If we receive an acknowledgment and remove the last element from the buffer we move to the *ready* state.
- `ack(seq: a)`: if a is less than the sequence number of the first element in the buffer, this is a duplicate acknowledgment and can be ignored.
- `timeout` : the process will resend the first element in the buffer.

Note that we have excluded one alternative from the description. A sending process can send messages 14, 15 and 16, and then receive an acknowledgment for 15 - what should we do? We could of course remove the element from the buffer immediately but in our description we defer the message until 15 is the first element in the buffer. Implicit deferral is a technique that allows us to simplify the description but it also introduce complications that we will discuss later.

In the description we have chosen to describe the sender and the receiver as two state machines, which they are. In the implementation however, we will implement them both as one Elixir process in the same way as we have implemented the link layer or network layer. We do this in order to provide a duplex communication channel without having to create two pairs or processes for each direction. The implementation becomes trickier since

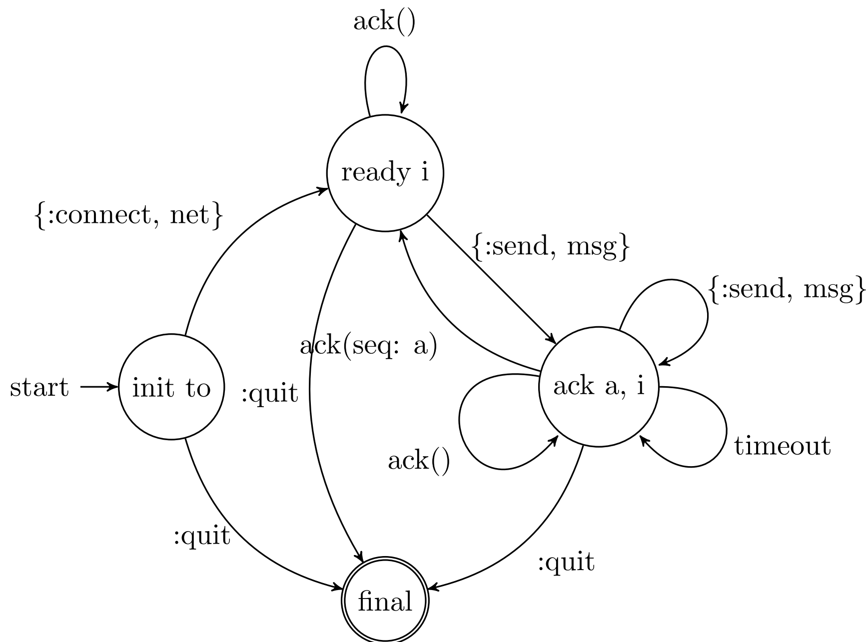


Figure 6: The ordering process: multiple outstanding messages.

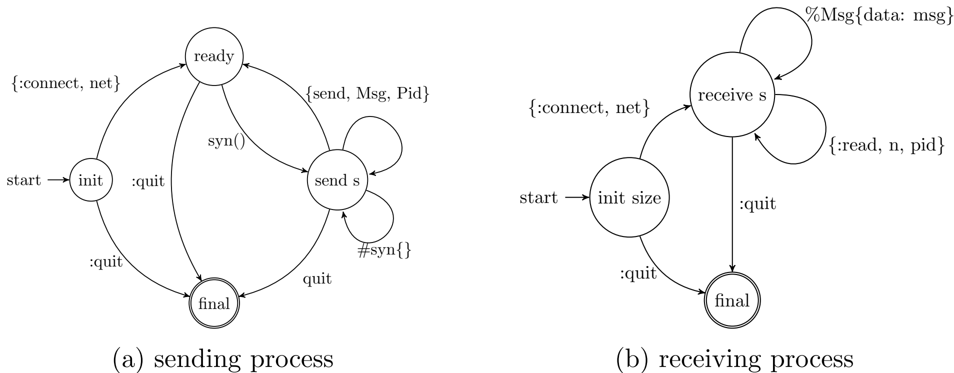
we now have to keep track of both outstanding buffered messages as well as what messages to receive and acknowledge.

6 flow control layer

The next layer will be slightly different since we now will do synchronous reads of messages. The previous layers have simply passed an incoming message to the master process but now we require the master process to actively do a read operations. Similarly the send operation is acknowledged so the user of the sending process knows that it is safe to send the next message. The application layer thus has a message interface that looks like follows:

- `{:send, msg, pid}` : sent to the sending process, a `:ok` message is delivered to the process `pid`, when its safe to send the next message.
- `{:read, n, pid}` : sent to the reading process, at most `n` messages are returned in a message `{:ok, l, mgs}` where `l` is the number of messages.

This layer would be easy to implement if we only allowed one outstanding message but we would of course like to implement a buffered system to increase the throughput. The read process thus keeps a buffer of incoming



(a) sending process

(b) receiving process

Figure 7: The flow control layer.

messages that are waiting to be read. The buffer does of course have a maximum size and the sender must be careful not to overflow the reader with messages. Before the sender can start to send messages it must therefore receive a *synchronize* message that informs the sender of how many more messages it can send. In the first message the reader informs the sender of the maximum size of the buffer and it is then the responsibility of the sender to keep track of how many more messages it can send.

In Fig 7 we sea the state diagrams of the sender and receiver. The sender keeps track of s, the free space in the receiver buffer, a value that is decremented with each send operation. If this value goes down to 0 it will enter the *ready* state were it must first see a synch message before it can accept more send operations.

As with the ordering layer we will implement the sender and receiver in one Elixir process. We could however have implemented them in separate processes but we would the of course need two processes on either side if we want to provide a duplex channel.

Do as before and implement a test function that creates a sender and a receiver and connects them over a flow controlled connection.

7 extensions

If you think you have everything working you can continue with some extensions.

7.1 testing

Hopefully you have managed so far but let's see if things actually work. If we tweak our hub a bit we can make it drop a packet every now and then. As

we receive a incoming frame we toss a coin and if we're unlucky we simply throw the frame away. If things works the ordering layer should resend the message and also order them in the right order.

While you at it you can give the hub a *loss* rate that determines how often it should loose a frame. The following code will get you going:

```
frame() = frm ->
  if :rand.uniform(100) <= loss do
     :io.format("nub: throwing away ~w\n", [frm])
     :ok
  else
     Enum.each(connected, fn({_,pid}) -> send(pid, frm) end)
  end
  hub(loss, connected)
```

7.2 a switch

Our hub is of course nothing but a hub, it will distribute every frame in all directions. What would a switch look like?

7.3 zombies

By now you probably have a hundred link processes running on you machine. Every time you test the system you create several processes that are then not properly terminated. You could of course continue for a while but if this was a real system we would eventually run out of resources.

All our processes can take a message `:quit` and should then of course terminate. The problem is of course to find them, or rather making sure that who ever started them also terminates them. Since we want to make sure that processes terminate even if their parent processes crash we could use linking when processes are started. Linking is however bi-directional so if a underlying network process dies it might take it's master process with it.

To solve these problems we could, as in the hub, use monitoring to let processes detect when their masters die. This soon starts to get complicated and it should be no surprise that Elixir has a whole framework, OTP, to handle larger systems.

7.4 marshaling

Our link layer can today send anything across the network but this is not a realistic assumption. Assume that we can only send a byte sequence. If the interface to the link layer was the message {`:send, string`}, the layers above would have to encode their sequence numbers, addresses, packet types

etc in a string. This is one challenge, to encode all data structures as strings before sending them.

The link layer can now turn into something more interesting and apply error detection etc to be able to send messages over connections where bit can be flipped. If you want to learn how a CRC code or a forward error correction code can be used this is the task to take on.

A linkl.ex

```
defmodule Link do

  require Record

  Record.defrecord(:frame, data: nil)

  def start(master) do
    {:ok, spawn(fn() -> init(master) end)}
  end

  defp init(master) do
    receive do
      {:connect, lnk} ->
        :io.format("link ~w: connected to ~w~n", [self(), lnk])
        link(master, lnk)
      :quit ->
        :ok
    end
  end

  def link(master, lnk) do
    receive  do
      {:send, msg} ->
        send(lnk, frame(data: msg))
        link(master, lnk)

      frame(data: msg) ->
        ##:io.format("link receiving ~w\n", [frm])
        send(master, msg)
        link(master, lnk)

      {:master, new} ->
        link(new, lnk)

      :status ->
        :io.format("link ~w: master: ~w,  link: ~w~n", [self(), master, l
        link(master, lnk)

      :quit ->
        :ok
    end
```

```
    end
end
```

B hub.ex

```
defmodule Hub do

  require Link

  def start() do
    {:ok, spawn(fn() -> init() end)}
  end

  def init() do
    :io.format("hub ~w: started~n", [self()])
    hub([])
  end

  def hub(connected) do
    receive do
      {:connect, pid} ->
        :io.format("hub ~w: connecting to ~w~n", [self(), pid])
        ref = :erlang.monitor(:process, pid)
        hub([{ref, pid}|connected])

      {:disconnect, pid} ->
        :io.format("hub ~w: disconnect ~w~n", [self(), pid])
        :erlang.demonitor(:process, pid)
        hub(List.keydelete(connected, pid, 1))

      {:DOWN, ref, :process, _, _} ->
        :io.format("hub ~w: died ~w~n", [self(), ref])
        hub(List.keydelete(connected, ref, 0))

      Link.frame() = frm ->
        Enum.each(connected, fn({_,pid}) -> send(pid, frm) end)
        hub(connected)

      :status ->
        :io.format("hub ~w: connected to ~w~n", [self(), connected])
        hub(connected)

      :quit ->
        :ok

    end
```

```
    end
end
```

C network.ex

```
defmodule Network do

  require Record

  Record.defrecord(:netw, src: 0, dst: 0, data: nil)


 def start(master, id) do
   con = spawn(fn() -> init(master, id) end)
   {:ok, con}
 end

 def init(master, id) do
   :io.format("network ~w: process ~w started ~n", [id, self()])
   receive do
     {:connect, link} ->
       :io.format("network ~w: connected to ~w~n", [id, link])
       network(master, id, link)
     :quit ->
       :ok
   end
 end

 def network(master, id, link) do
   receive do
     {:send, to, msg} ->
       ##:io.format("network ~w sending ~w to ~w\n", [id, msg, to])
       send(link, {:send, netw(src: id, dst: to,  data: msg)})
       network(master, id, link)

     netw(dst: ^id, data: msg) ->
       ##:io.puts("network ~w receiving ~w\n", [id, msg])
       send(master, msg)
       network(master, id, link)

     netw() ->
       network(master, id, link)

     {:master, new} ->
       network(new, id, link)
```

```
      :status ->
        :io.format("network ~w: id ~w, master: ~w, link: ~w~n", [self(), i
        network(master, id, link)

      :quit ->
        :ok
    end
  end

end
```

D order.ex

```
defmodule Order do

  require Record

  Record.defrecord(:ord, seq: 0, data: nil)

  Record.defrecord(:ack, seq: 0)
  Record.defrecord(:dgr, seq: 0, data: [])

  def start(master, to) do
    {:ok, spawn(fn() -> init(master, to) end)}
  end

  def init(master, to) do
   :io.format("order to ~w: process ~w started~n", [to, self()])
    receive do
      {:connect, netw} ->
        :io.format("order to ~w: connected to ~w~n", [to, netw])
        order(master, to, 0, 0, [], netw)
    end
  end

  def order(master, to, n, i, [], netw) do
    receive do

      ord(seq: ^i, data: msg) ->
        send(netw, {:send, to, ack(seq: i)})
        send(master, msg)
        order(master, to, n, i+1, [], netw)

     ord(seq: j) when j < i ->
        send(netw, {:send, to, ack(seq: j)})
        order(master, to, n, i, [], netw)

      ack() ->
        order(master, to, n, i, [], netw)

      {:send, msg} ->
        send(netw, {:send, to, ord(seq: n, data: msg)})
        order(master, to, n+1, i, [{n,msg}], netw);
```

```
    {:master, new} ->
      order(new, to, n, i, [], netw)
  end
end
def order(master, to, n, i, [{a,res}|rest]=buffer, netw) do
  receive do

    ord(seq: ^i, data: msg) ->
      send(netw, {:send, to, ack(seq: i)})
      send(master, msg)
      order(master, to, n, i+1, buffer, netw)

    ord(seq: j) when j < i ->
      send(netw, {:send, to, ack(seq: j)})
      order(master, to, n, i, buffer, netw)

    ack(seq: ^a) ->
      order(master, to, n, i, rest, netw)

    ack(seq: b) when b < a ->
      order(master, to, n, i, buffer, netw);

    {:send, msg} ->
      send(netw,{:send, to, ord(seq: n, data: msg)})
      order(master, to, n+1, i, buffer++[{n,msg}], netw)

    {:master, new} ->
      order(new, to, n, i, buffer, netw)

  after 10 ->
      dgr = ord(seq: a, data: res)
      ##:io.format("order to ~w resending ~w\", [to, dgr])
      send(netw, {:send, to, dgr})
      order(master, to, n, i, buffer, netw)
  end
end
end
```