A transport layer

Johan Montelius
KTH
VT19

Communication service

Assume we have a communication channel that allows us to send *frames* between two connected nodes. The channel is not reliable so messages can be lost or delivered out of order.

We want to build a communication service that is .... better.

Our task is to build a communication service that provides:
- reliable delivery: despite frames being lost
- ordered delivery: FIFO - first-in-first-out
- identity: an addressing scheme
- flow control: prevented from overflowing a receiver

layered architecture

Build a solution using a layered architecture.

Each layer provides an abstraction that the layer above can make use of.

layers

```
user 1
  message
  flow 1
    flow controlled
    order 1
      ordered
      network 1
        addressed
        link 1
          frame

user 2
  message
  flow 2
    flow controlled
    order 2
      ordered
      network 2
        addressed
        link 2
```
the link layer

the link process - a state diagram

the link process

```
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, dest} ->
      link(master, dest)
    :quit ->
      :ok
  end
end
```

def link(master, dest) do
  receive do
    {:send, msg} ->
      send(dest, frame(data: msg))
      link(master, dest)
    frame(data: msg) ->
      send(master, msg)
      link(master, dest)
    :quit ->
      :ok
  end
end
def test() do
    sender = spawn(fn() -> sender() end)
    receiver = spawn(fn() -> receiver() end)
    link1 = Link.start(sender)
    link2 = Link.start(receiver)
    send(link1, {:connect, link2})
    send(link2, {:connect, link1})
    send(sender, {:connect, link1})
    send(receiver, {:connect, link2})
    :ok
end

def hub(connected) do
    receive do
        {:connect, pid} ->
            hub([pid|connected])
        {:disconnect, pid} ->
            hub(List.delete(connected, pid))
        frame() = frm ->
            Enum.each(connected, fn(pid) -> send(pid, frm) end)
            hub(connected)
        :quit ->
            :ok
    end
end
The setup - a sequence diagram

```
sysop
start()

user
start()

link
start(user)
{:connect, hub}
{:connect, link}
{:connect, link}
```

The network layer

```
master
{:send, to, msg}
msg

network id
{:send, netw(dst: to, data: msg)}
netw(dst: id, data: msg)

link
```

The network layer will only forward messages with the right destination.

The network process

```
def network(master, id, link) do
  receive do
    {:send, to, msg} ->
      send(link, {:send, netw(src: id, dst: to, data: msg)})
    network(master, id, link)
    netw(dst: ^id, data: msg) ->
      send(master, msg)
    network(master, id, link)
    netw() ->
      network(master, id, link)
    :quit ->
      :ok
  end
end
```
A communication channel is a duplex flow of an ordered sequence of messages.

- add a sequence number to each message
- order messages as they arrive and
- resend lost messages

The layer will need to buffer messages and use a timeout to detect missing datagrams.
the order process

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

the order process

def order(master, to, n, i, [{a,res}|rest]=buffer, netw) do
  receive do
    : ack(seq: ^a) ->
      order(master, to, n, i, rest, netw)
    : after 10 ->
      dgr = ord(seq: a, data: res)
      send(netw, {:send, to, dgr})
      order(master, to, n, i, buffer, netw)
  end
end

flow control

- do not overflow the receiver
- keep track of the receiver buffer size
- wait for the user to actively read messages

We are introducing a synchronous interface - only send if receiver prepared.
the flow control

```
{:send, msg, pid} {:read, n, pid} {:ok, i, [msg]}
{:send, msg(data: msg)}
```

the flow sending process

```
{:connect, net} :quit
start init send s syn() :quit
ready {send, msg, pid}
sent s syn() syn()
final :quit {send, msg, pid}
```

the flow receiving process

```
{:connect, net} {read, n, pid}
start init size :quit
receive s msg(data: msg)
```

extensions

- What if the link layer could only send sequences of bytes?
- Can we add error detection in the link layer?
- Could we build a switch or router?
- divide a service into processes
- layers of abstraction
- finite State Machine (FSM) description of a process
- sequence diagrams to show protocol
- asynchronous and synchronous interfaces

.. and hopefully, you have learned about communication stacks