Message Passing

CS 3SD3

Ryszard Janicki

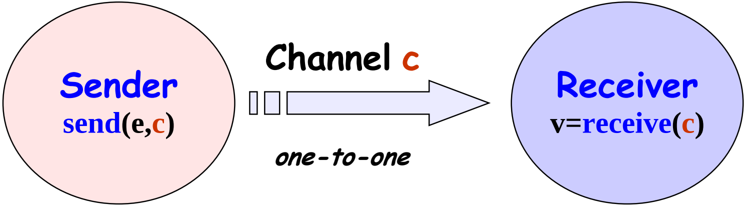
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Concepts: **synchronous** message passing - **channel**
asynchronous message passing - **port**
- send and receive / selective receive
rendezvous bidirectional comms - **entry**
- call and accept ... reply

Models:	**channel**	: relabelling, choice & guards
	port	: message queue, choice & guards
	entry	: port & channel

Practice: distributed computing (disjoint memory)
threads and monitors (shared memory)

Synchronous Message Passing - Channel



♦ **send(e,c)** - send the value of the expression *e* to channel *c*. The process calling the send operation is ***blocked*** until the message is received from the channel.

♦ **v = receive(c)** - receive a value into local variable *v* from channel *c*. The process calling the receive operation is ***blocked*** waiting until a message is sent to the channel.

cf. distributed assignment **v = e**

- Popular notation: $v = e$, $c!e \leftarrow send$, $c?v \leftarrow receive$

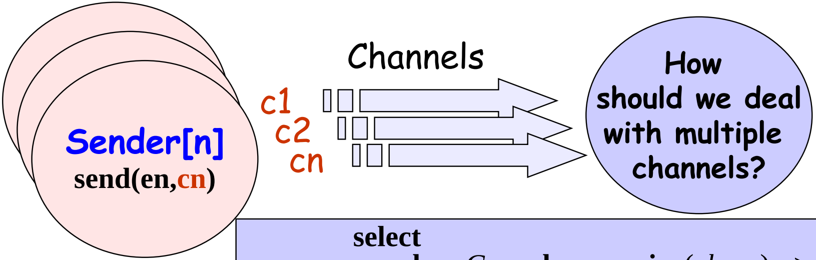
Model

```
range M = 0..9                    // messages with values up to 9

SENDER = SENDER[0],               // shared channel chan
SENDER[e:M] = (chan.send[e]-> SENDER[(e+1)%10]).

RECEIVER = (chan.receive[v:M]-> RECEIVER).

                                  // relabeling to model synchronization
||SyncMsg = (SENDER || RECEIVER)
            /{chan/chan.{send,receive}}.
```

How can this be modeled directly without the need for relabeling?

message operation	*FSP model*
send(e,chan)	`chan.[e]`
v = receive(*chan*)	`chan.[v:M]`

- Wrong question! Why should we avoid relabeling?



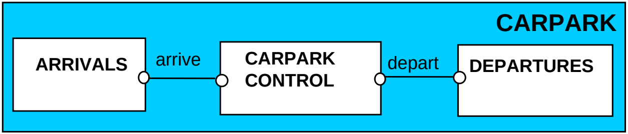
Select statement...

select
  when G_1 **and** v_1=**receive**(*chan*$_1$) => S_1;
or
  when G_2 **and** v_2=**receive**(*chan*$_2$) => S_2;
or
  ...
or
  when G_n **and** v_n=**receive**(*chan*$_n$) => S_n;
end

How would we model this in FSP?

- If more than one of G_i's is true, the choice is **non-deterministic**.

Car Park

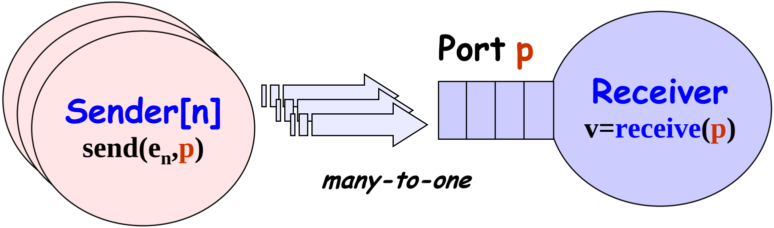



```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).

ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).
||CARPARK = (ARRIVALS||CARPARKCONTROL(4)
                     ||DEPARTURES).
```

Interpret as channels

Implementation using message passing?

- I think it is wrong example, *CARPARK* is in my opinion a **shared memory problem**, not *distributed computing* (i.e. message passing) problem.
- It show that we can model *shared memory problem* using *distributing computing* tools, but this should always be rather an exception, never the rule.



♦ **send(e,p)** - send the value of the expression *e* to port *p*. The process calling the send operation is **not** ***blocked***. The message is queued at the port if the receiver is not waiting.

♦ *v* = **receive(*p*)** - receive a value into local variable *v* from port *p*. The process calling the receive operation is ***blocked*** if there are no messages queued to the port.

FSP Model of Port

```
range M = 0..9              // messages with values up to 9
set   S = {[M],[M][M]}      // queue of up to three messages

PORT                        //empty state, only send permitted
  = (send[x:M]->PORT[x]),
PORT[h:M]                   //one message queued to port
  = (send[x:M]->PORT[x][h]
    |receive[h]->PORT
    ),
PORT[t:S][h:M]              //two or more  messages queued to port
  = (send[x:M]->PORT[x][t][h]
    |receive[h]->PORT[t]
    ).

// minimise to see result of abstracting from data values
||APORT = PORT/{send/send[M],receive/receive[M]}.
```

LTS?

What happens if send 4 values?

- For this model forth sent will go to *ERROR* state.
- For queue up to 3 messages, an LTS has 1111 states, for queue up to 4 messages an LTS has 11111 states.
- In general, for a range of x different values and queue up to n, we need $\frac{x^{n+1}-1}{x-1}$ states.
- Is such an approach proper?