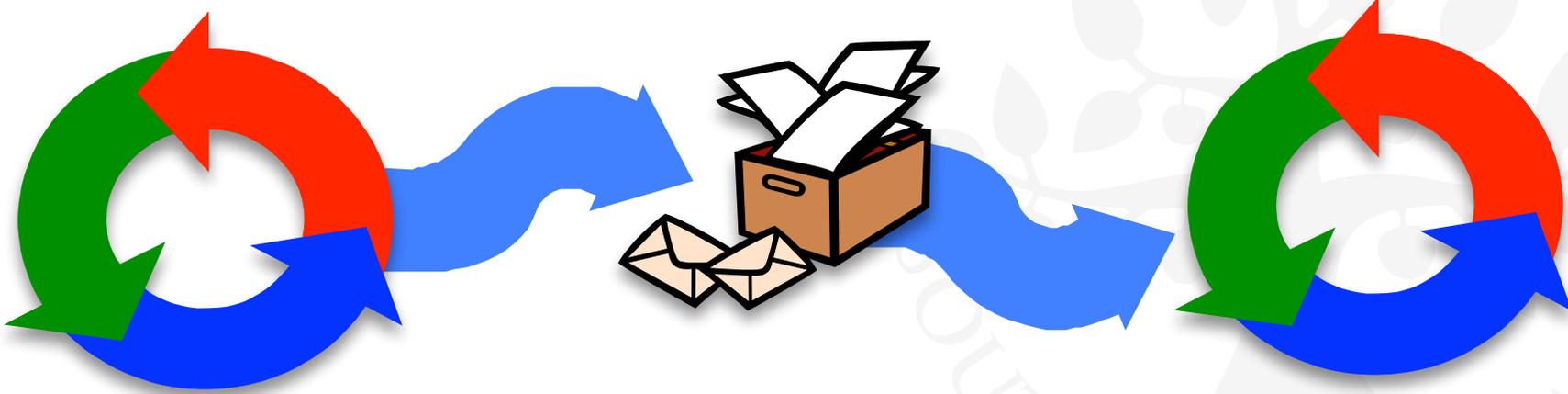


# Message Passing





# Absence Of Shared Memory

**In previous lectures interaction between threads has been via shared memory**

- In Java, we refer to shared objects.
- Usually encapsulate shared memory in Monitors.

**In a distributed setting there is no shared memory**

- Communication is achieved via passing messages between concurrent threads.
- Same message passing abstraction can also be used in non-distributed settings.



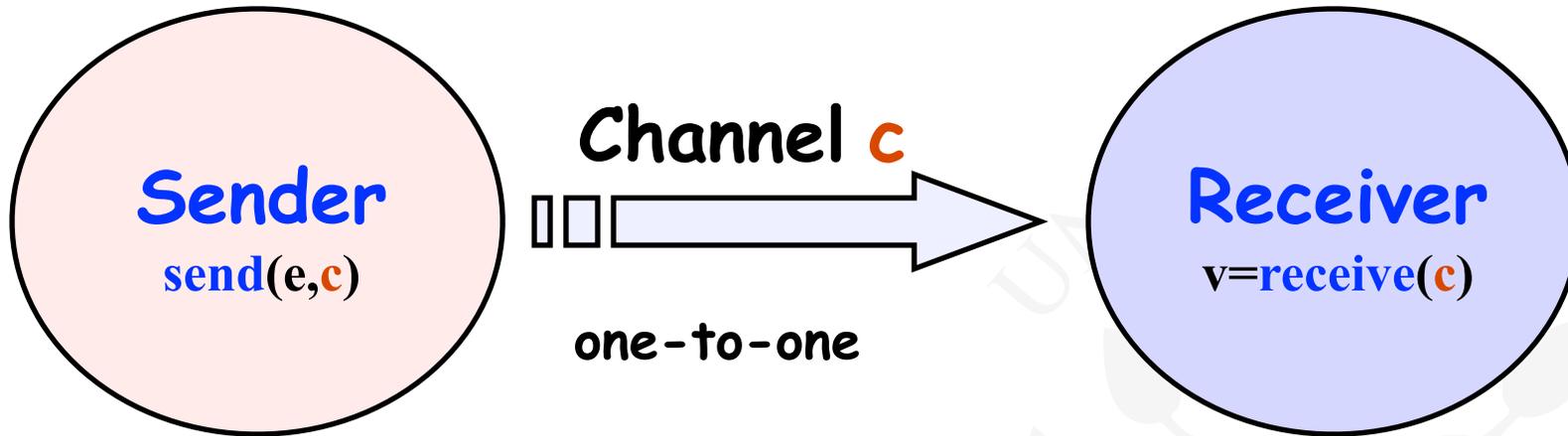
# Message Passing

**Concepts:** **synchronous** message passing - **channel**  
**asynchronous** message passing - **port**  
- **send** and **receive** / **selective receive**  
**rendezvous** bidirectional comm. - **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)

# 10.1 Synchronous Message Passing - Channel



◆ `send(e, c)` - send  $e$  to channel  $c$ . The sender is **blocked** until the message is received from the channel.

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

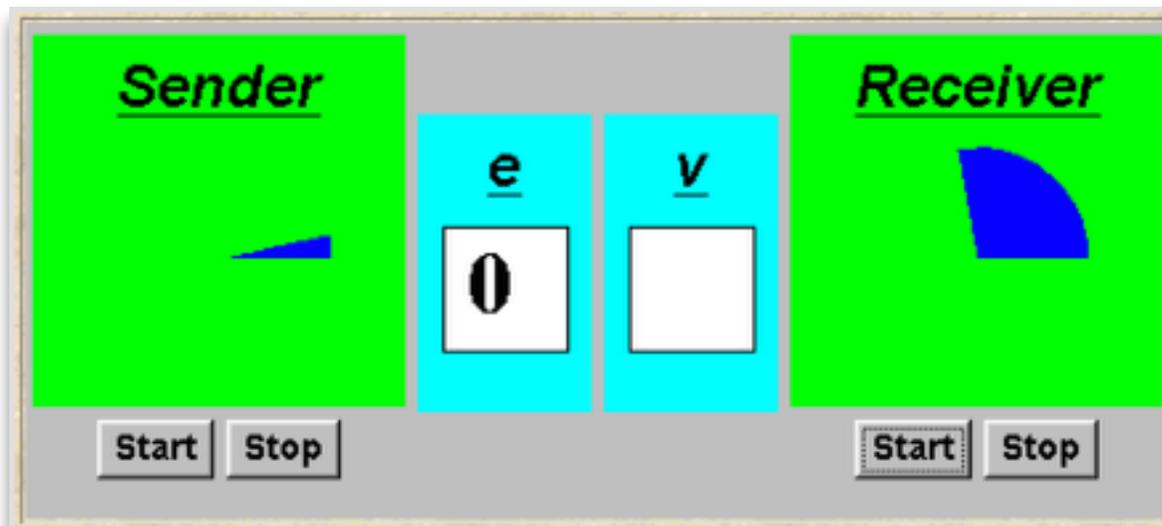
Channel has no buffering

Corresponds to " $v = e$ "

# Synchronous Message Passing - Applet

A sender communicates with a receiver using a single **channel**.

The sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.



```
Channel<Integer> chan = new Channel<Integer>();  
tx.start(new Sender(chan, senddisp));  
rx.start(new Receiver(chan, recvdisp));
```

Instances of ThreadPanel

Instances of SlotCanvas

# Synchronous Message Passing In Java

## Java has no built in message passing primitives

- Unlike Occam, Erlang, or Ada.

## Can still do message passing in Java, but it's clunky:

- Encapsulate message passing abstractions in monitor Channel:

```
class Channel<T> extends Selectable {  
    public synchronized void send(T v)  
        throws InterruptedException {...}  
  
    public synchronized T receive() {...}  
}
```



# Java Implementation - Channel

```
public class Channel<T> extends Selectable {
    T chan_ = null;

    public synchronized void send(T v)
        throws InterruptedException {
        chan_ = v;
        signal();
        while (chan_ != null) wait();
    }

    public synchronized T receive()
        throws InterruptedException {
        block(); clearReady(); // part of Selectable
        T tmp = chan_; chan_ = null;
        notifyAll(); // could be notify()
        return(tmp);
    }
}
```

Channel is a monitor that has synchronized access methods for `send` and `receive`.

Selectable is described later.



# Java Implementation - Sender

```
class Sender implements Runnable {
    private Channel<Integer> chan;
    private SlotCanvas display;
    Sender(Channel<Integer> c, SlotCanvas d)
        {chan=c; display=d;}

    public void run() {
        try { int ei = 0;
            while(true) {
                display.enter(String.valueOf(ei));
                ThreadPanel.rotate(12);
                chan.send(new Integer(ei));
                display.leave(String.valueOf(ei));
                ei=(ei+1)%10; ThreadPanel.rotate(348);
            }
        } catch (InterruptedException e) {}
    }
}
```



# Java Implementation - Receiver

```
class Receiver implements Runnable {
    private Channel<Integer> chan;
    private SlotCanvas display;
    Receiver(Channel<Integer> c, SlotCanvas d)
        {chan=c; display=d;}

    public void run() {
        try { Integer v=null;
            while(true) {
                ThreadPanel.rotate(180);
                if (v!=null) display.leave(v.toString());
                v = chan.receive();
                display.enter(v.toString());
                ThreadPanel.rotate(180);
            }
        } catch (InterruptedException e) {}
    }
}
```



# 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).

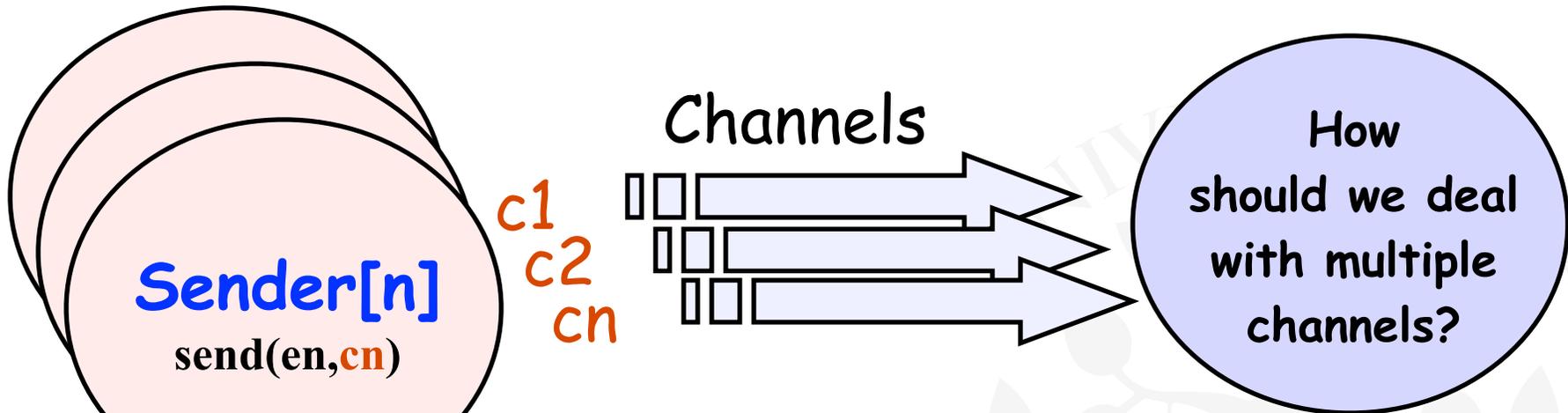
// relabeling to model synchronization
|| SyncMsg = (SENDER || RECEIVER)
             / {chan/chan. {send, receive}} .
```

LTS?

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

message operation	FSP model
<code>send(e,chan)</code>	<code>chan.[e]</code>
<code>v = receive(chan)</code>	<code>chan.[v:M]</code>

# Selective Receive



Select statement...

```
select
  when  $G_1$  and  $v_1 = \text{receive}(\text{chan}_1) \Rightarrow S_1;$ 
or
  when  $G_2$  and  $v_2 = \text{receive}(\text{chan}_2) \Rightarrow S_2;$ 
or
  ...
or
  when  $G_n$  and  $v_n = \text{receive}(\text{chan}_n) \Rightarrow S_n;$ 
end
```

How would we model this in FSP?

# Example: Selective Receive



```
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?



# Java Implementation - Selective Receive

```
class MsgCarPark implements Runnable {
    private Channel<Signal> arrive, depart;
    private int spaces, N;
    private StringCanvas disp;

    public MsgCarPark(Channel<Signal> a,
                      Channel<Signal> l,
                      StringCanvas d,int capacity) {
        depart=l; arrive=a; N=spaces=capacity; disp=d;
    }
    ...
    public void run() {...}
}
```

**Implement**  
CARPARKCONTROL as a  
thread MsgCarPark  
which receives signals  
from channels arrive  
and depart.

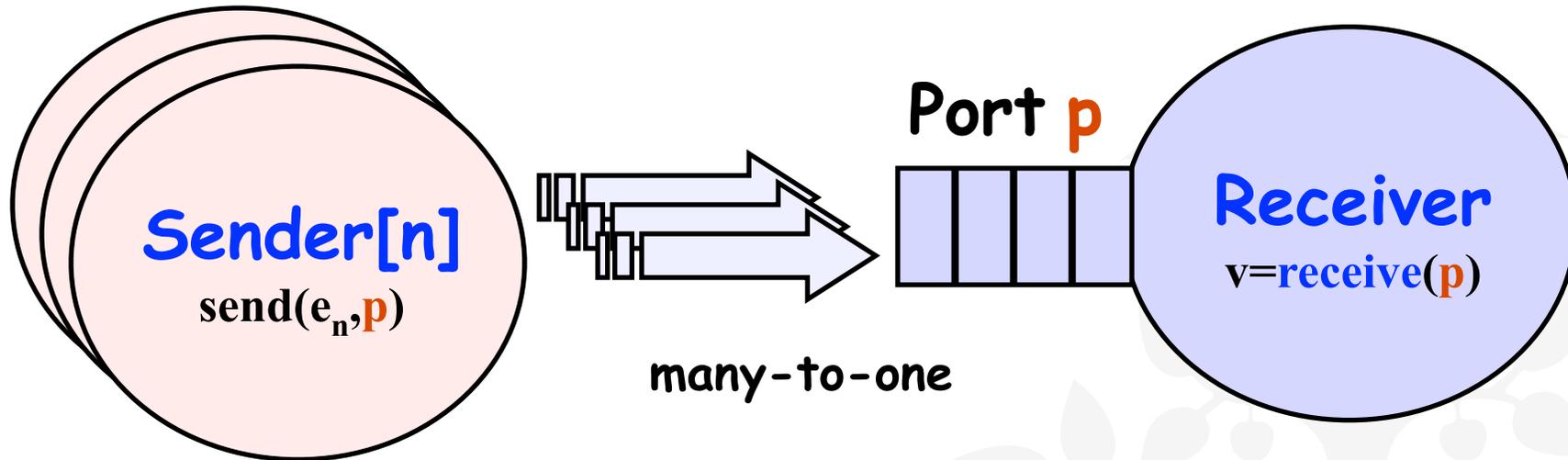


# Java Implementation - Selective Receive

```
public void run() {
    try {
        Select sel = new Select();
        sel.add(depart);
        sel.add(arrive);
        while(true) {
            ThreadPanel.rotate(12);
            arrive.guard(spaces>0);
            depart.guard(spaces<N);
            switch (sel.choose()) {
                case 1:depart.receive();display(++spaces);
                        break;
                case 2:arrive.receive();display(--spaces);
                        break;
            }
        }
    } catch InterruptedException{}
}
```

See applet

## 10.2 Asynchronous Message Passing - Port



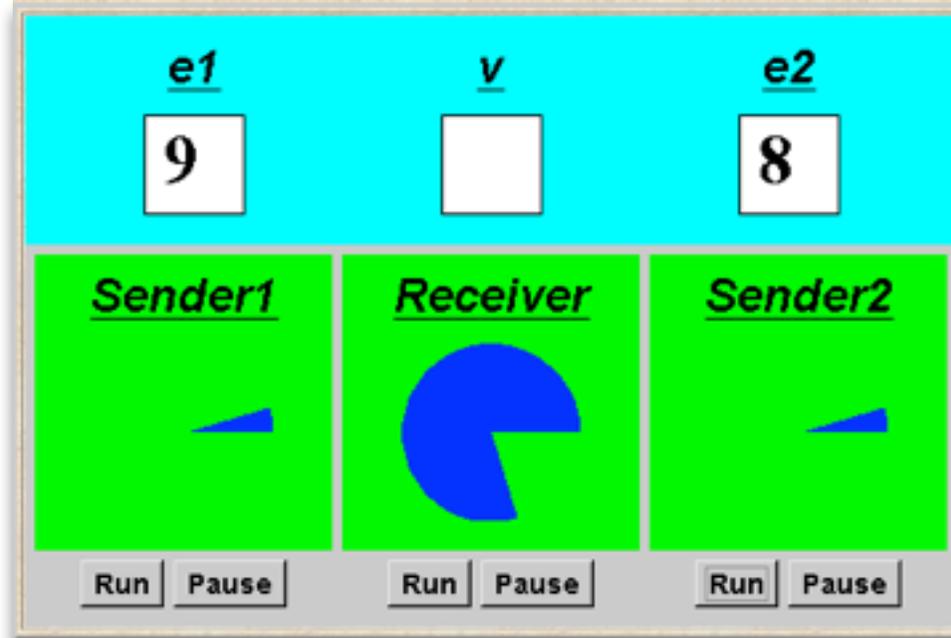
◆ `send(e, p)` - send  $e$  to port  $p$ .  
The calling process is **not blocked**.  
The message is queued at the port  
if the receiver is not waiting.

◆ `v = receive(p)` - receive a  
value into local variable  $v$  from  
port  $p$ . The calling process is  
**blocked** if no messages queued to  
the port.

# Asynchronous Message Passing - Applet

Two senders communicate with a receiver via an "unbounded" port.

Each sender sends a sequence of integer values from 0 to 9 and then restarts at 0 again.



```

Port<Integer> port = new Port<Integer> ();
tx1.start(new Asender(port, send1disp));
tx2.start(new Asender(port, send2disp));
rx.start(new Areceiver(port, recvdisp));
  
```

Instances of ThreadPanel

Instances of SlotCanvas



# Java Implementation - Port

```
class Port<T> extends Selectable {  
  
    Queue<T> queue = new LinkedList<T>();  
  
    public synchronized void send(T v) {  
        queue.add(v);  
        signal();  
    }  
  
    public synchronized T receive()  
        throws InterruptedException {  
        block(); clearReady();  
        return queue.remove();  
    }  
}
```

The implementation of Port is a monitor that has synchronized access methods for `send` and `receive`.



# Port Fsp Model

```
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]
    ) .

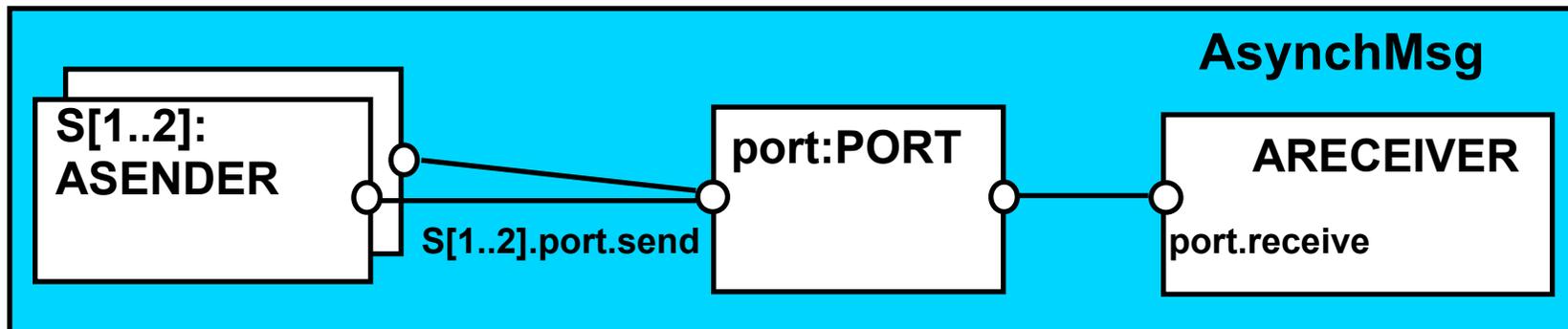
// minimise to see result of abstracting from data values
|| APORT = PORT / { send / send [M], receive / receive [M] } .
```

LTS?

What happens if  
you send 4 values?



# Model Of Applet



```
ASENDER = ASENDER[0],
```

```
ASENDER[e:M] = (port.send[e] -> ASENDER[(e+1)%10]).
```

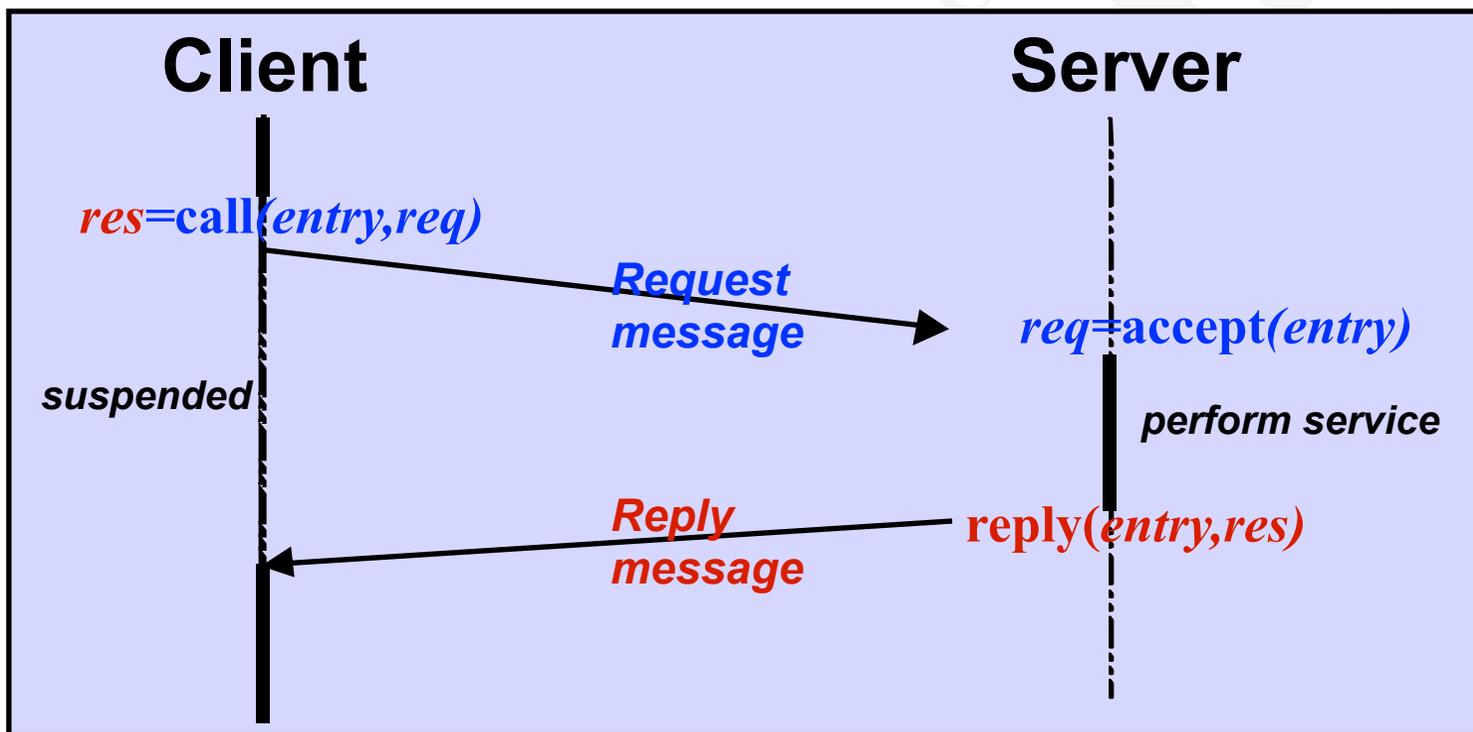
```
ARECEIVER = (port.receive[v:M] -> ARECEIVER).
```

```
|| AsyncMsg = (s[1..2]:ASENDER || ARECEIVER || port:PORT)  
/ {s[1..2].port.send/port.send}.
```

## Safety?

## 10.3 Rendezvous - Entry

Rendezvous is a form of **request-reply** to support **client server** communication. Many clients may request service, but only one is serviced at a time.



# Rendezvous

◆  **$res = call(e, req)$**  - send the value  **$req$**  as a request message which is queued to the entry  **$e$** .

◆ The calling process is **blocked** until a reply message is received into the local variable  **$req$** .

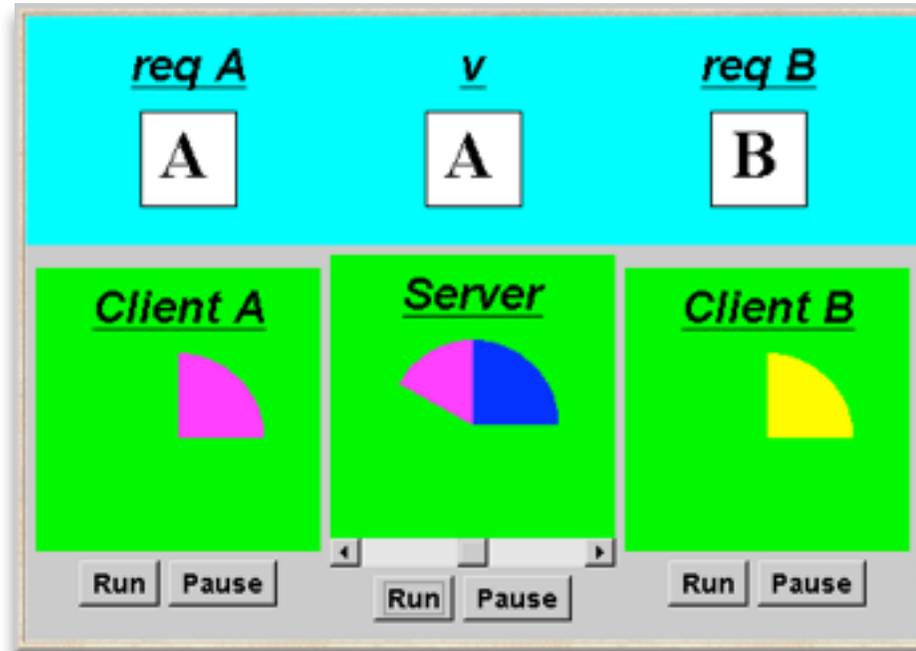
◆  **$req = accept(e)$**  - receive the value of the request message from the entry  **$e$**  into local variable  **$req$** . The calling process is **blocked** if there are no messages queued to the entry.

◆  **$reply(e, res)$**  - send the value  **$res$**  as a reply message to entry  **$e$** .

The model and implementation use a port for one direction and a channel for the other. **Which is which?**

# Rendezvous - Applet

Two clients call a server which services a request at a time.



```
Entry<String,String> entry = new Entry<String,String>();  
clA.start(new Client(entry,clientAdisp,"A"));  
clB.start(new Client(entry,clientBdisp,"B"));  
sv.start(new Server(entry,serverdisp));
```

Instances of ThreadPanel

Instances of SlotCanvas

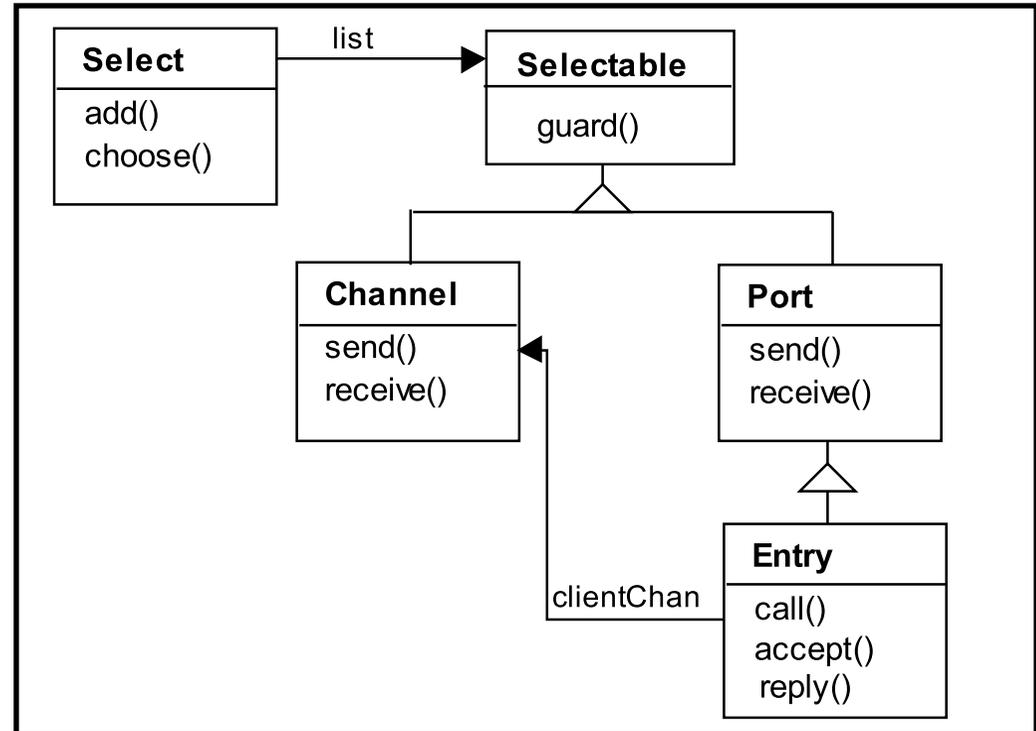


# Java Implementation - Entry

**Entries:** implemented as extensions of ports

**call()** creates a channel object on which to receive the reply and passes a references to this in the message to the server.

It then awaits the reply on the channel.



**accept()** keeps a copy of the channel reference;

**reply()** sends the reply message to this channel.



# Java Implementation - Entry

```
class Entry<R,P> extends Port<CallMsg<R,P>> {
    private CallMsg<R,P> cm;

    public P call(R req) throws InterruptedException {
        Channel<P> clientChan = new Channel<P>();
        this.send(new CallMsg<R,P>(req,clientChan));
        return clientChan.receive();
    }

    public R accept() throws InterruptedException {
        cm = this.receive();
        return cm.request;
    }

    public void reply(P res) throws InterruptedException {
        cm.replychan.send(res);
    }

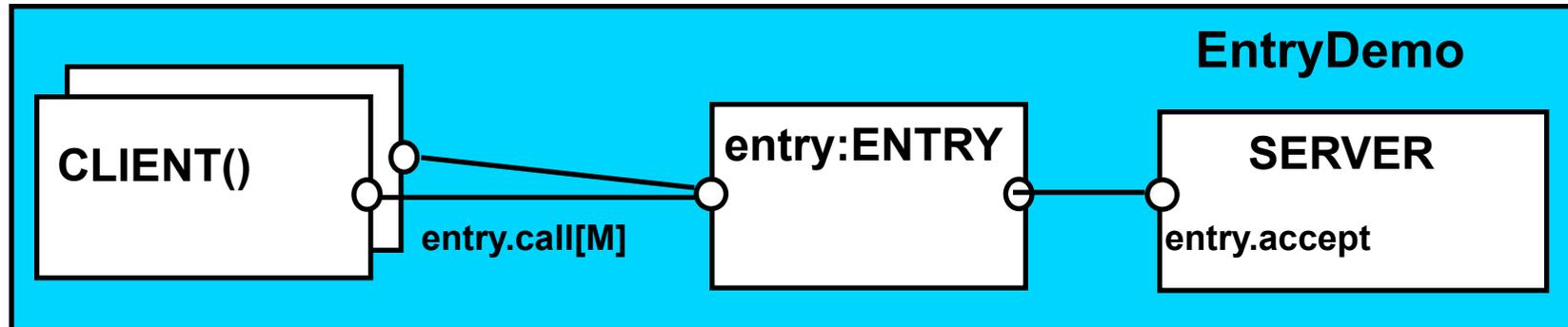
    private class CallMsg<R,P> {
        R request;
        Channel<P> replychan;
        CallMsg(R m, Channel<P> c)
            {request=m; replychan=c;}
    }
}
```

Do **call**, **accept** and **reply** need to be synchronized methods?



# Model Of Entry And Applet

We reuse the models for ports and channels ...



```

set M = {replyA,replyB}           // reply channels
||ENTRY = PORT/{call/send, accept/receive}.
CLIENT(CH='reply) = (entry.call[CH]->[CH]->CLIENT) .
SERVER = (entry.accept[ch:M]->[ch]->SERVER) .
||EntryDemo = (CLIENT('replyA) || CLIENT('replyB)
               || entry:ENTRY || SERVER) .

```

Action labels used in expressions or as parameter values must be prefixed with a single quote.



# Rendezvous **Vs** Monitor Method Invocation

**What is the difference?**

... from the point of view of the **client**?

... from the point of view of the **server**?

... **mutual exclusion**?

**Which implementation is more efficient?**

... in a **local** context (client and server in same computer)?

... in a **distributed** context (in different computers)?

**Concepts:** **synchronous** message passing - **channel**  
**asynchronous** message passing - **port**  
- **send** and **receive** / **selective receive**  
**rendezvous** bidirectional comm. - **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)