Concurrent Programming (Part II)
Lecture 11: Concurrency without access to shared memory (i.e. distributed systems)

Dr Kevin Bryson

K.Bryson@cs.ucl.ac.uk

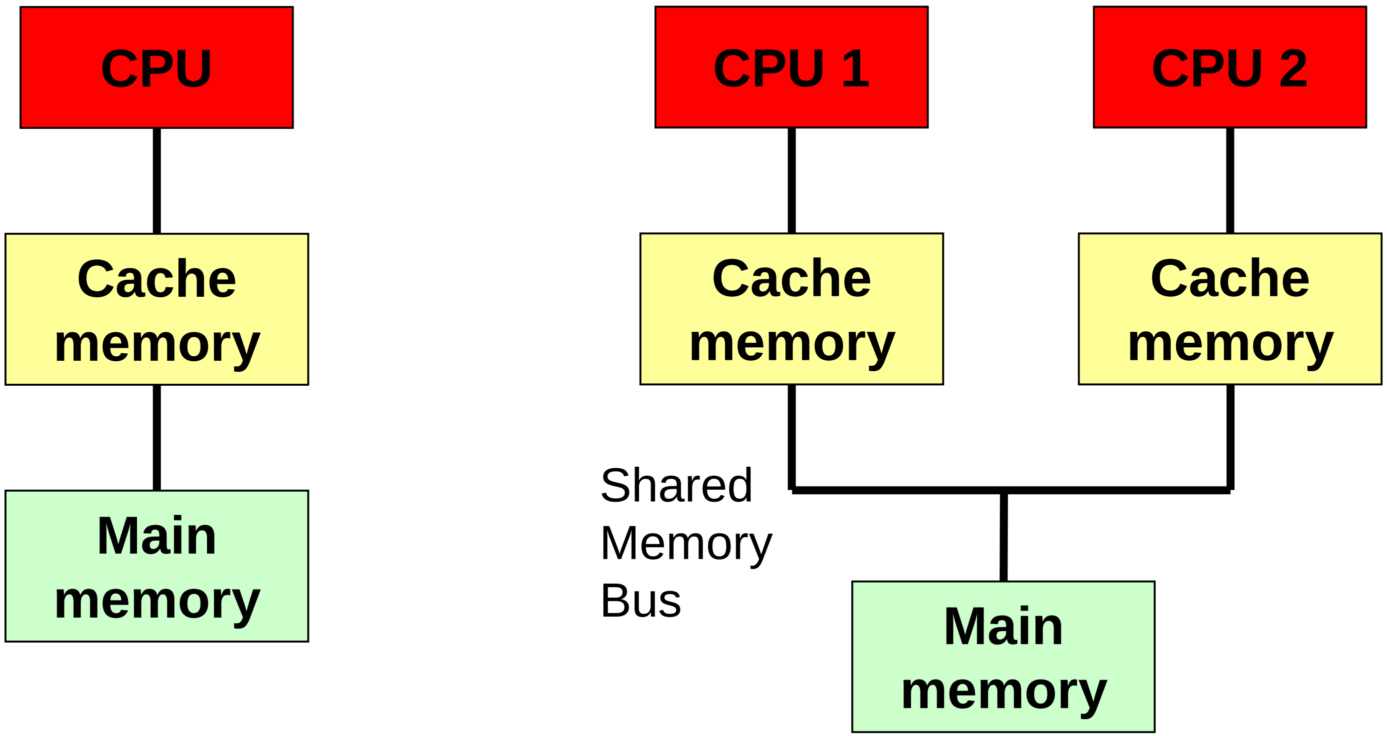
Room 8.04

Course Web Site on Moodle

http://moodle.ucl.ac.uk/course/view.php?id=753

Enrolment Key: ATOMIC

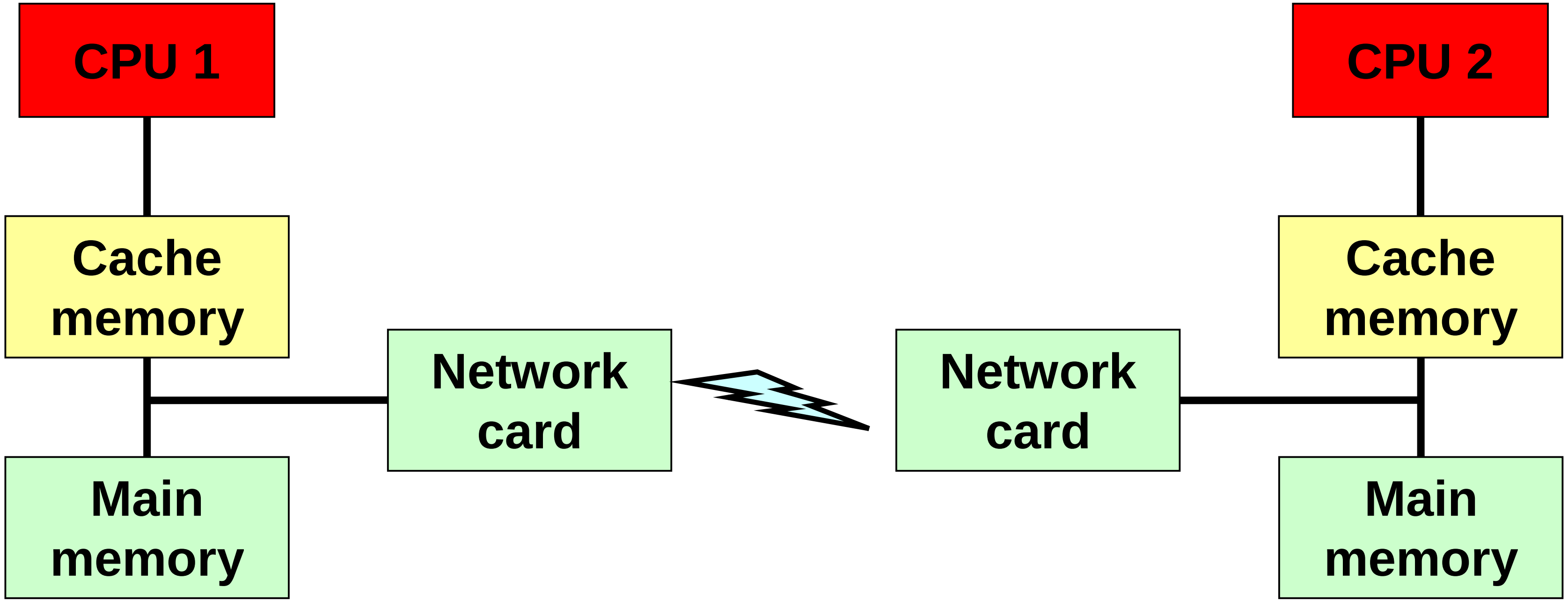
Recall shared memory architectures from Lecture 2 …



Thread communication within a single JVM (within one OS process) using shared memory

- Until now all our programs have assumed threads to be in the same JVM and therefore sharing an address space
- How do these threads communicate?
  - Reading/writing shared variables (generally under restricted concurrent access such as mutual exclusion)
  - Event conditions (wait/notify mechanism)
  - Changing the state of more complex data structures generally structured as Monitors (for instance a buffer)
  - Directly calling thread methods such as interrupt()

Also recall we briefly showed a distributed system in Lecture 2 … what are the key concepts for understanding concurrency in such systems?



Well … what is the relevance of this picture?

- This picture begs the question … how do we communicate with each other? (Given that we do not share direct access to each other's working memories!)
- Distributed threads need to employ a similar mechanisms …

Overview of this lecture

- When processes do not directly share an address space they cannot communicate by means of shared local variables like all the examples we have had so far!
- One example would be when processes sit at different ends of a network (i.e. on different physical machines).
- This lecture is about the *key theoretical abstractions* behind concurrent systems that do not employ shared memory mechanisms.
- This will lead into the final set of lectures that will cover more concrete material about writing distributed concurrent systems using Java.

Key concept 1 - Message Passing

- Messages can be sent to communicate from one thread to another Thread. These threads may be within:
  - the same OS process/JVM (as before …)
  - different OS processes on the same machine
  - different OS processes on different physical machines, with a communication network between them
- A sender process **sends** the message and a receiver process **receives** the message

- BUT WHY WOULD WE WANT TO SEND MESSAGES BETWEEN THREADS IN THE SAME JVM ???

Key concept 2 – concurrency behaviour

- **Synchronous** message passing: the sender sends the message and then *blocks* – waiting for the receiver to accept it. The sender only continues once the message has been received.
- **Asynchronous** message passing: the sender sends the message and then continues executing. This is a *non-blocking* protocol.

Key concept 3 – communication path

- **One-to-one over a dedicated 'network'.** A dedicated channel between the sender and receiver exists (for instance in Occam where the physical transputer chips are connected in a North/South/East/West fashion to other transputer chips ***in hardware***).
- **One-to-one over a switching 'network'.** In this case the receiver processes need unique identifiers so that messages can be corrected routed to them.
- **One-to-many messaging (Broadcast messaging).** The sender does not need an identifier for the receiver since the message gets sent to all processes current listening.

- THESE ARE ABSTRACT CONCEPTS – communication could be between threads in the same JVM using a message passing package like Java Message Service (JMS), i.e. in software without a physical network. The above concepts still apply!

Key concept 4 – messaging protocols

- **One-way communication.** A communication event only consists of the sender sending a message to the receiver.
- **Two-way communication: *Rendezvous* (or request-reply)**: the sender sends the message and then *blocks* while it waits for the receiver to send back a message back. The receiver typically carries out processing on the message before replying.

Examples of message passing

- **When thinking about examples we need to identify the 'processes' doing the communication and what constitutes the 'message' (i.e. the level of abstraction)**

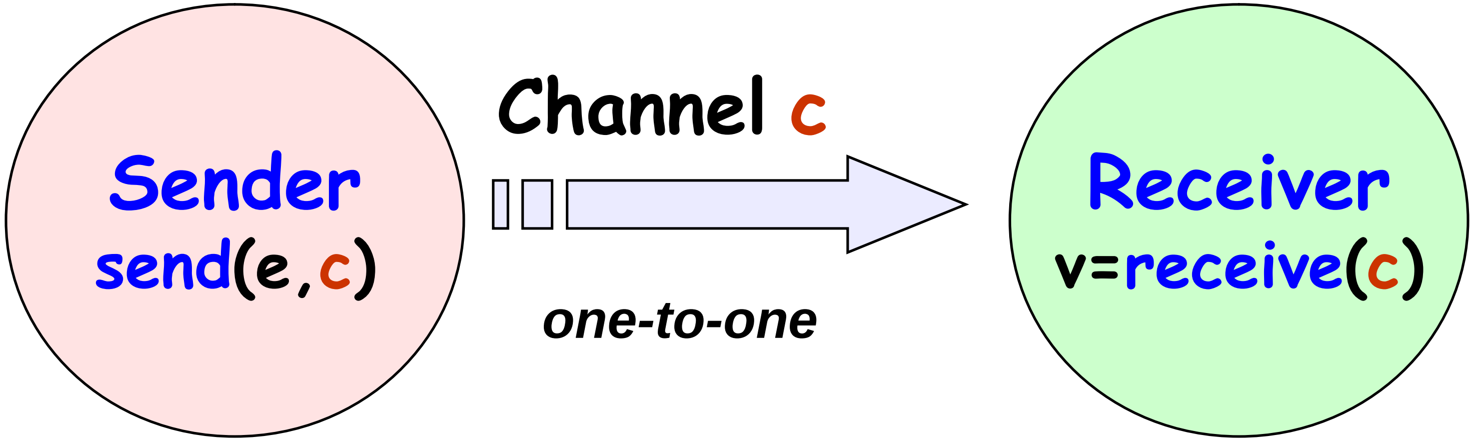
- There are different ways in which messages can be sent.
  Can you think of some from real life?
  Can you think of key differences in their mechanisms?

- **Synchronous**: fax -> fax (message on bit of paper)
  person -> person on phone
  person -> answer-phone (message what is said)

- **Asynchronous**: person -> person via a letter, email, …
  person -> person **via answer-phone**

- ***Rendezvous***: person -> person (telephone call)
  person -> person (meeting and chatting …)

10.1 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.

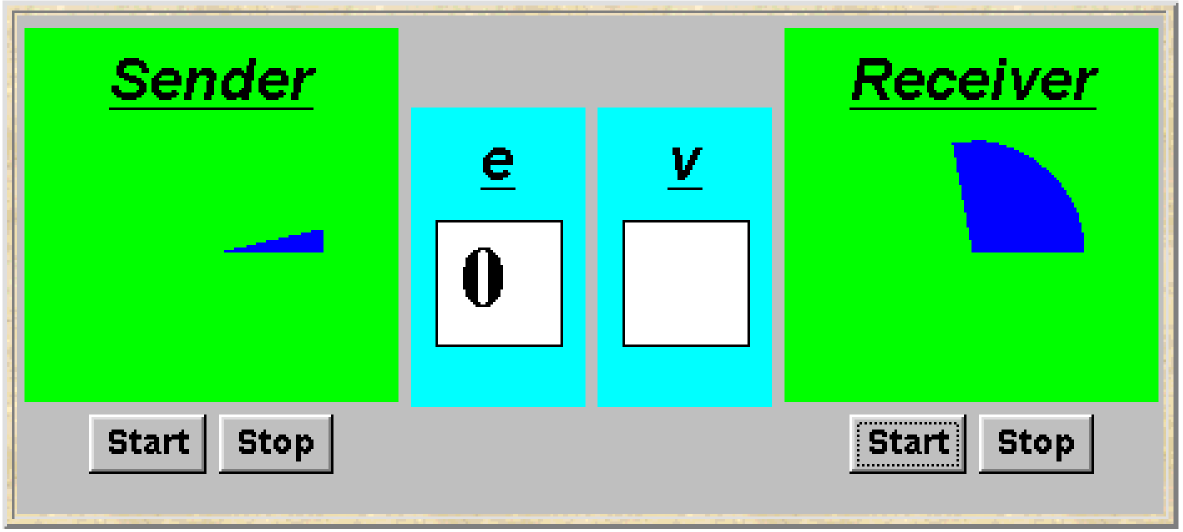
♦ ***v*** **= receive(*c*)** - receive a value into local variable *v* from channel *c*. The process calling the receive operation is ***blocked*** waiting until a message is sent to the channel.

Synchronous message passing - demo

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.

See the demo …




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

But does this not just implement the Channel as a monitor using shared variables ?

- YES !

- … but it is still message passing since it is the approach that is important … not the implementation (it is an abstract concept).

- Messages are written by a sender process … and then they are received by a receiver process … the ‘ownership’ of the message is transferred.
(The Java Messaging Service (JMS API) uses shared memory when it is possible/efficient to do so …)

- **In theory a hardware communication channel could have existed between these processes …**

Car Park Demo – key difference between shared memory & message passing versions

<u>Shared Memory</u>

Calling Methods

Arrive **Process** → Car Park Controller **MONITOR** ← Depart **Process**

<u>Message Passing</u>

Transmitting Messages – could be distributed

Arrive **Process** → Car Park Controller **Process** ← Depart **Process**

PROBLEM - WHAT HAPPENS IF THE CAR PARK CONTROLLER HAD BLOCKED WAITING ON THE ‘WRONG’ MESSAGE ???

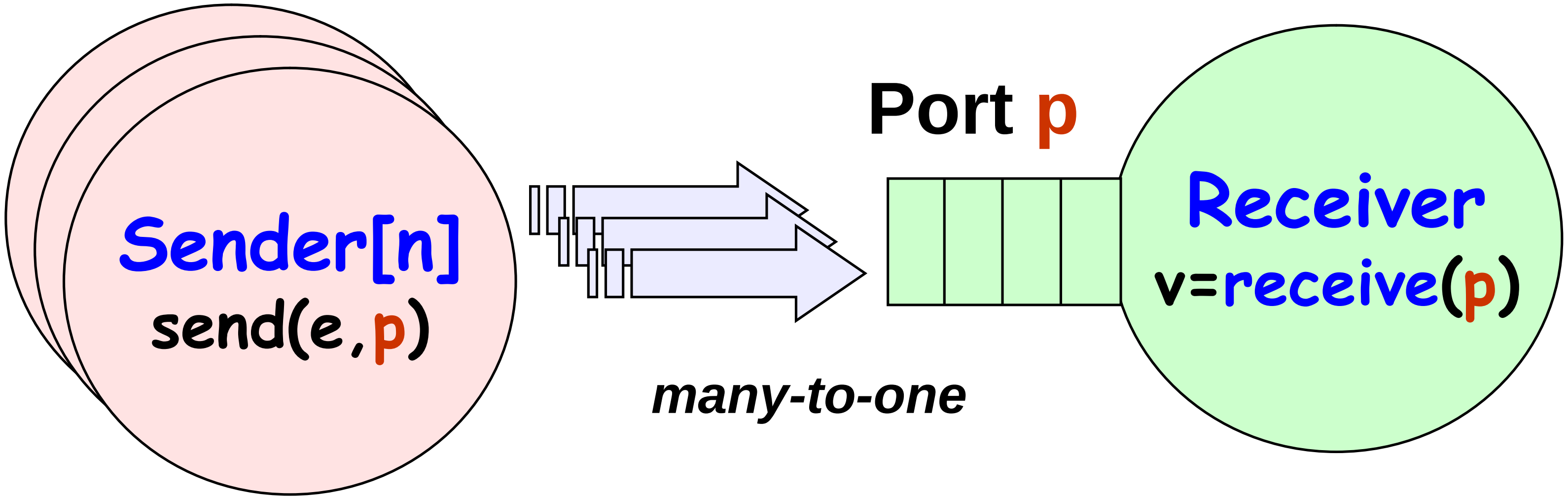
Selective Receives

- What happens if a process wants to receive from more than one channel?
  - It blocks if it chooses the wrong one!
- Selective receive is a mechanism employed in languages such as Occam and Ada to solve this problem.
- The mechanism is similar to a ‘select’ statement which blocks until it can receive data on a particular channel.
- G1, G2, G3 are Boolean *guards* which must be true for the receive to be eligible.
- S1, S2, S3 are statements which are carried out depending on which channel it receives data on.

```
select

      when G1 and v1 = receive(chan1) => S1;
      when G2 and v2 = receive(chan1) => S2;
      when G3 and v3 = receive(chan1) => S3;
end
```

10.2 Asynchronous Message Passing - port



♦ **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.

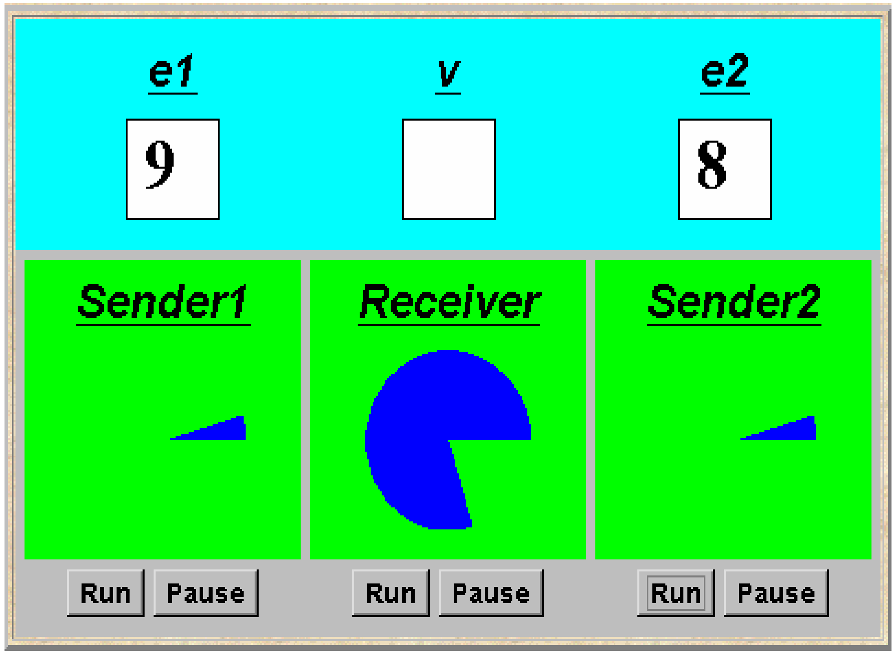
♦ ***v* = receive(*p*)** - receive a value into local variable *v* from port *p*. The process calling the receive operation is ***blocked*** if there are no messages queued to the port.

Asynchronous message passing - demo

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.

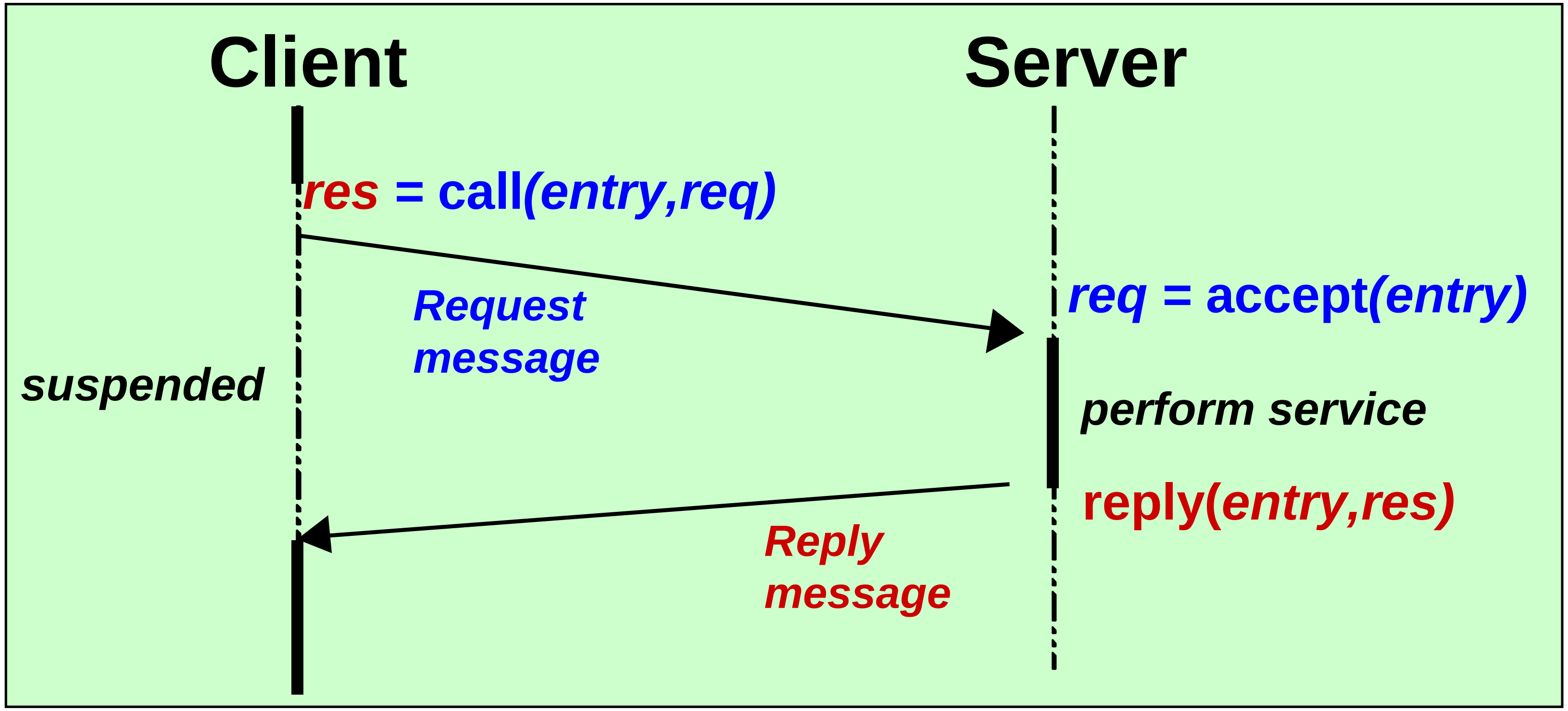
See the demo …




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

Rendezvous

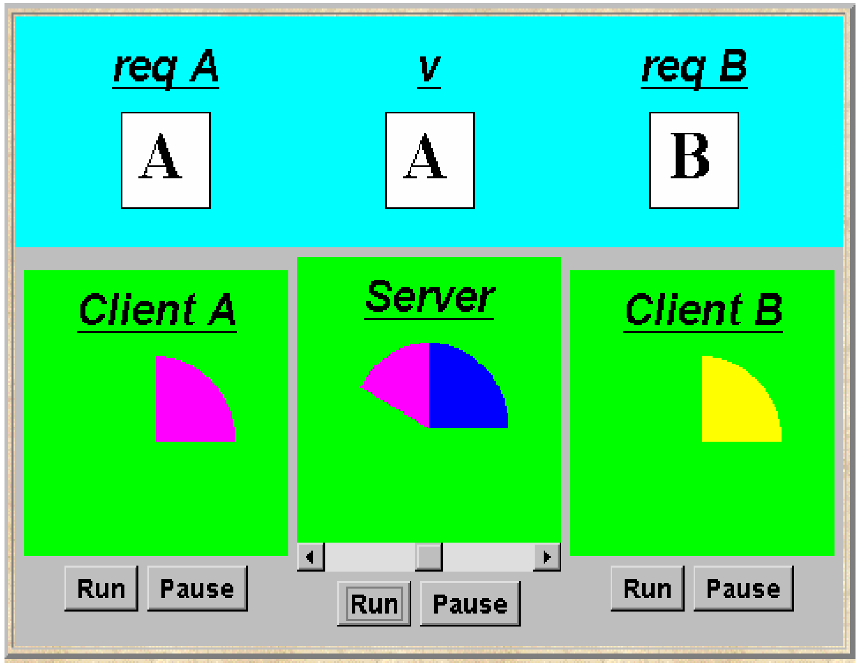
Rendezvous is a form of request-reply to support client-server communication. Many clients may request service, but only one is serviced at a time.



Rendezvous - demo

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

See the demo …




```
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));
```

Summary

- When threads are not in the same address space they cannot communicate using shared memory
- They can communicate using message passing
- Message passing *can be* used even when threads do share a common address space … to facilitate 'de-coupling' of the components. But it *is essential* when threads/processes are physically distributed.
- We have seen different types of message passing:
  - Synchronous
  - Asynchronous
  - Rendezvous
- **Background reading** – Magee & Kramer, Chapter 10.