# Message Passing

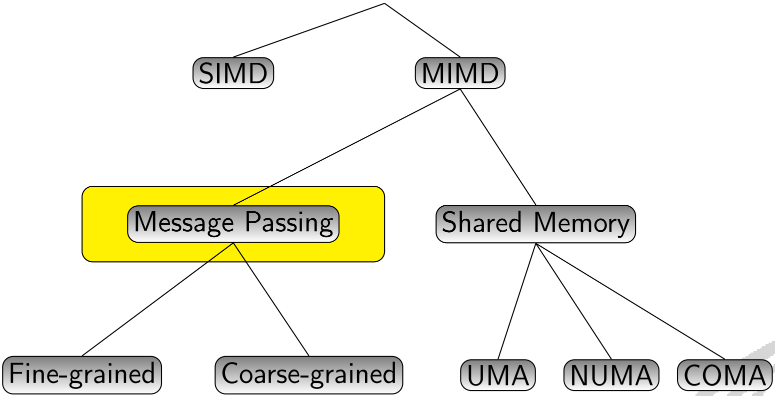
Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems

Uppsala University

Summer 2009

# MultiProcessor world - Taxonomy



# Scenario

Several cars want to drive from point A to point B.

## Sequential Programming

They can compete for space on the same road and end up either:

- following each other
- or competing for positions (and having accidents!).

## Parallel Programming

Or they could drive in parallel lanes,
thus arriving at about the same time without getting in each other's way.

## Distributed Programming

Or they could travel different routes, using separate roads.

# Distributed Programming

**No shared-memory ⇒**

Have to exchange messages with each other.

**Important to define communication interface**

Reads and writes like reads/writes on shared-memory?

⇒

Instead, a better approach is to define special network operations that include synchronization (in the same way as semaphores were special operations on shared variables)

# Distributed Programming

are typically the only objects processes share

⇒ Each variable is

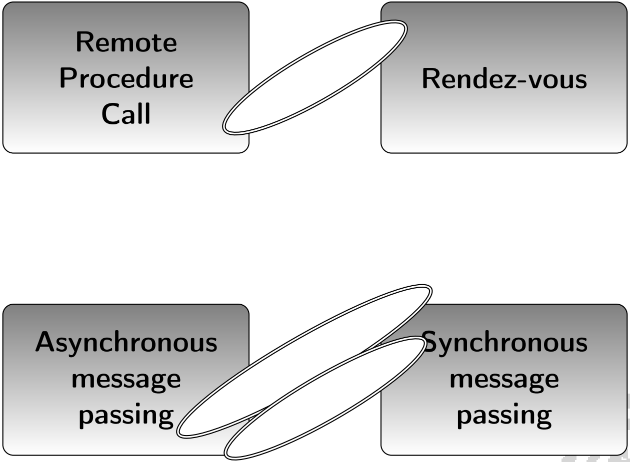
⇒ No concurrent access

⇒

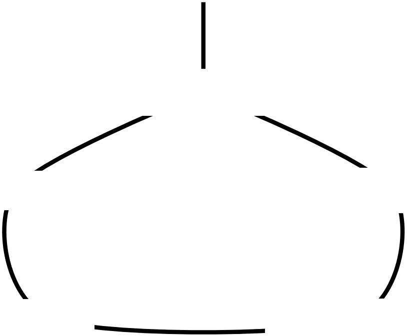
# Communication in the channel

- One-way or two-way information flow
- Asynchronous or synchronous communication
  (non-blocking/blocking)
- Direct or indirect communication
  (mailbox/ports)
- Automatic or explicit buffering

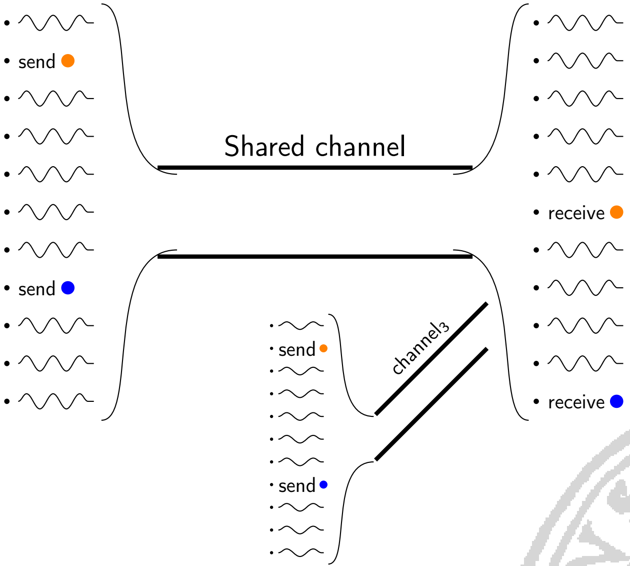
# 4 communication patterns



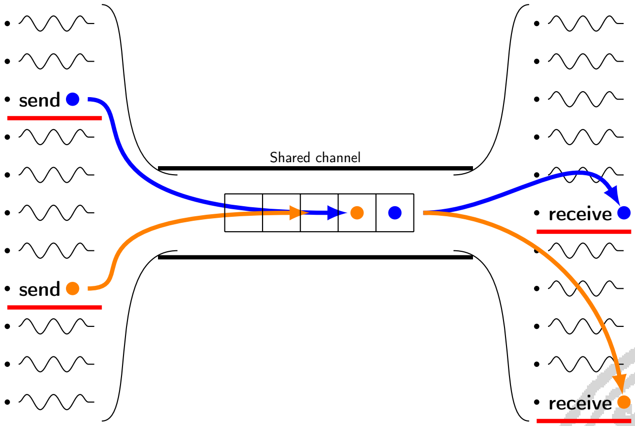
# Relation between concurrent mechanism



# Channels


send
send
Shared channel
receive
receive
send
send
channel$_3$

# send is blocking or non-blocking
# receive is blocking



# `receive` has blocking semantics...

... so the receiving process does not have to use busy-waiting to poll the channel if it has nothing else to do until a message arrives.

**Assumption**

Access to the content of each channel is atomic and that message deliver y is reliable and error-free.

# Naming convention

- Sending to and receiving from any channel

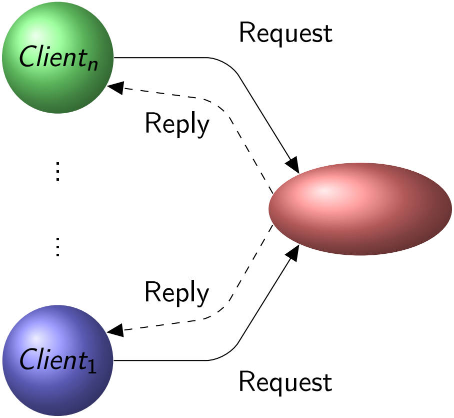
---

- exactly one receiver
- eventually many senders

---

- exactly one receiver
- exactly one sender

# Clients / Server



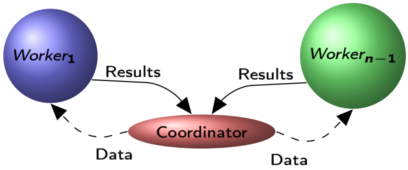
# Clients/Server with one operation op

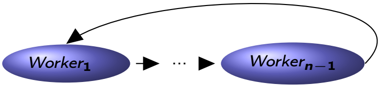
```
channel request(int clientID, types of input values);
channel reply[n](types of results);

 process Client  {    ▷i= 0,...,n-1
     send request(i, value arguments);
     receive reply[i](result arguments);
}

 process Server  {
     int clientID;
      ▷declaration of other permanent variables
      ▷initialization code;
     while ( true )  {
          receive request(clientID, input values);
           ▷code from body of operation op;
          send reply[clientID](results);
     }
}
```

# Clients/Server with multiple operation

```
type op_kind, arg_type, result_type;

channel request(int clientID, op_kind, arg_type);
channel reply[n](res_type);


 process Client  {   ▷i= 0,...,n-1
       arg_type myargs; result_type myresults;
        ▷ place value arguments in myargs;
       send request(i, op_j, myargs);  ▷"call" op_j
       receive reply[i](myresults);   ▷wait for reply
}


 process Server  {
       int clientID; op_kind kind; arg_type args; res_type results;
        ▷ declaration of other permanent variables;
        ▷ initialization code;
       while ( true )  {
              receive request(clientID, kind, args);
              if (kind == op_1){body of op_1}
              ...
              else if (kind == op_n){body of op_n}
              send reply[clientID](results);
       }
}
```

# Interacting Peers



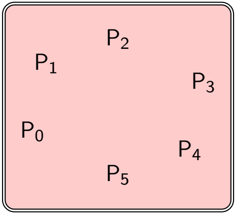

Each worker has a local value.
Task: Sort the smallest and biggest values among the workers.

# Interacting peers – Workers/Coordinator

```
channel values(int);
channel results[n](int smallest, int largest);

process P_{i=1,...,n-1} {
    int val;  ▷Assume val has been initialized
    int smallest, int largest;
    send values(val);
    receive results[i](smallest,largest);
}

process                         {
    int val;  ▷Assume val has been initialized
    int new, smallest = val, largest = val;  ▷initial state

    for [i = 1 to n-1] {   ▷gather values and save the smallest and largest



    }

    for [i = 1 to n-1] {   ▷Send the result to the other processes

    }
}
```

# Interacting peers – Symmetric solution

```
channel values[k](int);        ▷k = n*(n+1)/2

    process P_{i=0,...,n-1}  {
        int val;  ▷Assume val has been initialized
        int new, smallest = val, largest = val;  ▷initial state

        ▷send my value to the other processes
        for [j = 0 to n-1 but j ≠ i]  {
            send values[j](val);
        }

        ▷gather values and save the smallest and largest
        for [j = 0 to n-1 but j ≠ i]  {


        }
    }
```



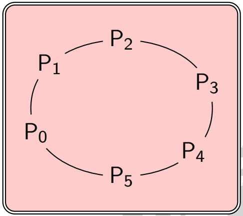
# Interacting peers – Circular pipeline

channel values[n](int smallest, int largest);

process $P_{1,\ldots,n-1}$ {
  int val; ▷*Assume val has been initialized*
  int smallest, int largest; ▷*initial state*
  ▷*receive smallest and largest so far then update them by comparing their value to val*

  ▷*send the result to the next process and then wait to get the global result*

}




process { ▷*initiates the exchanges*
  int val; ▷*Assume val has been initialized*
  int smallest = val, largest = val; ▷*initial state*

  ▷*send val to the next process, $P_1$*

  ▷*get global smallest and largest from $P_{n-1}$ and pass them on to $P_1$*

}

# Conclusion

Tools:

- MPI
- Java RMI
- CORBA
- SOAP
- RPC
- used in Microkernels
- Erlang

Message passing systems have been called *"shared nothing"* systems because the message passing abstraction hides underlying state changes that may be used in the implementation of sending messages.

Message passing model based programming languages typically define messaging as the (usually asynchronous) sending (usually by copy) of a data item to a communication endpoint (Actor, process, thread, socket, etc...)