Lesson 9

Concurrency Control in Distributed Environment

Ch 8 [BenA 06]

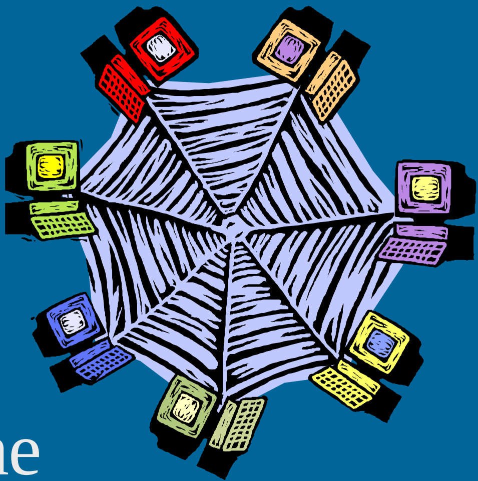
Messages

Channels

Rendezvous

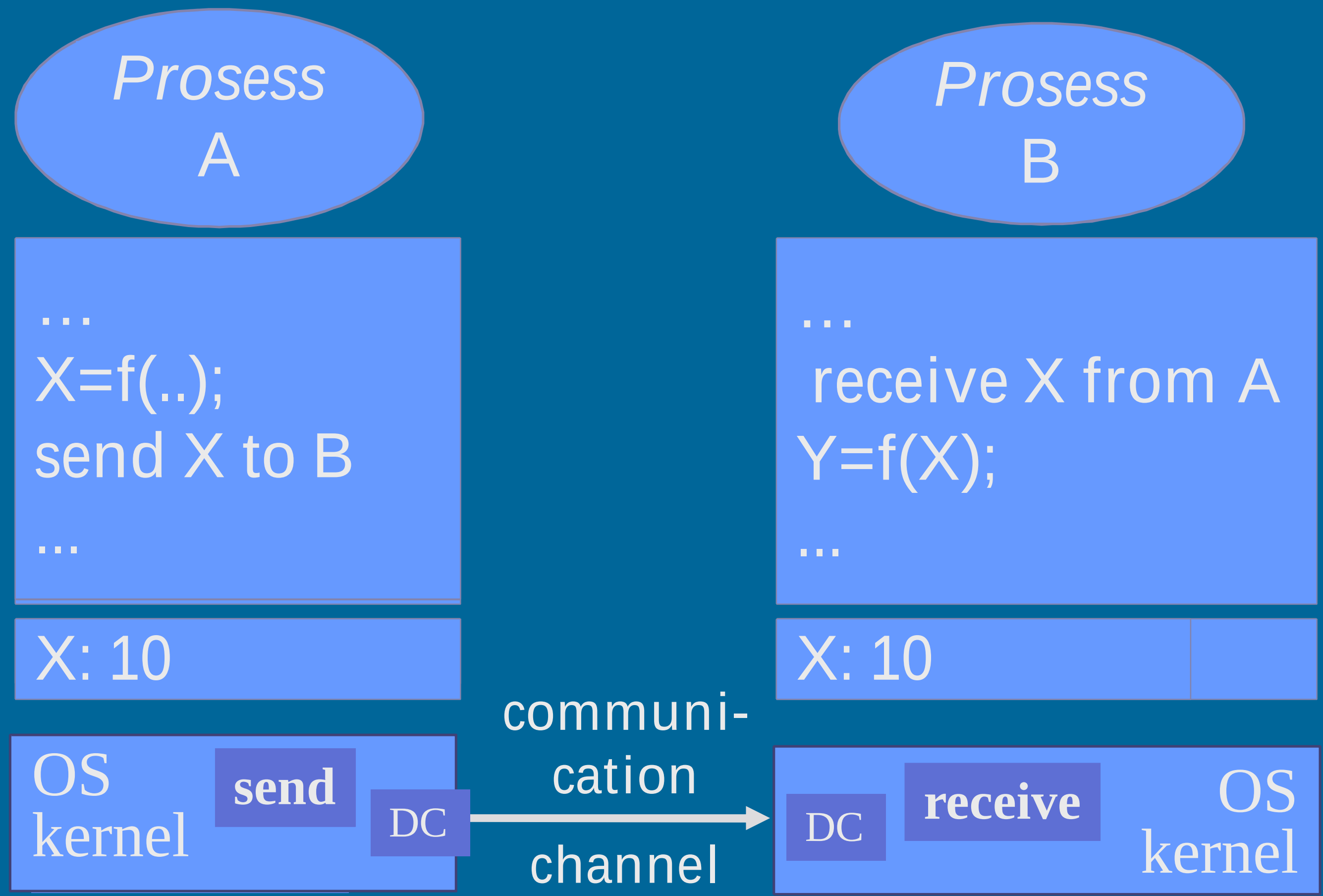
RPC and RMI

Distributed System



- No shared memory
- Communication with messages
- Tighly coupled systems
  - Processes alive at the same time
- Persistent systems
  - Data stays even if processes die
- Fully distributed systems
  - Everything goes

Communication with Messages (4)



- Sender, receiver
- Synchronous/asynchronous communication

Message Passing

- Synchronous communication
  - Atomic action
  - Both wait until communication complete
- Asynchronous communication
  - Sender continues after giving the message to OS for delivery
  - May get an acknowledgement later on
    - Message received or not
- Addressing
  - Some address for receiver process
    - Process name, id, node/name, …
  - Some address for the communication channel
    - Port number, channel name, …
  - Some address for requested service
    - Broker will find out, sooner or later
      - After message has been sent?
    - Service address not known at service request time

Usual case

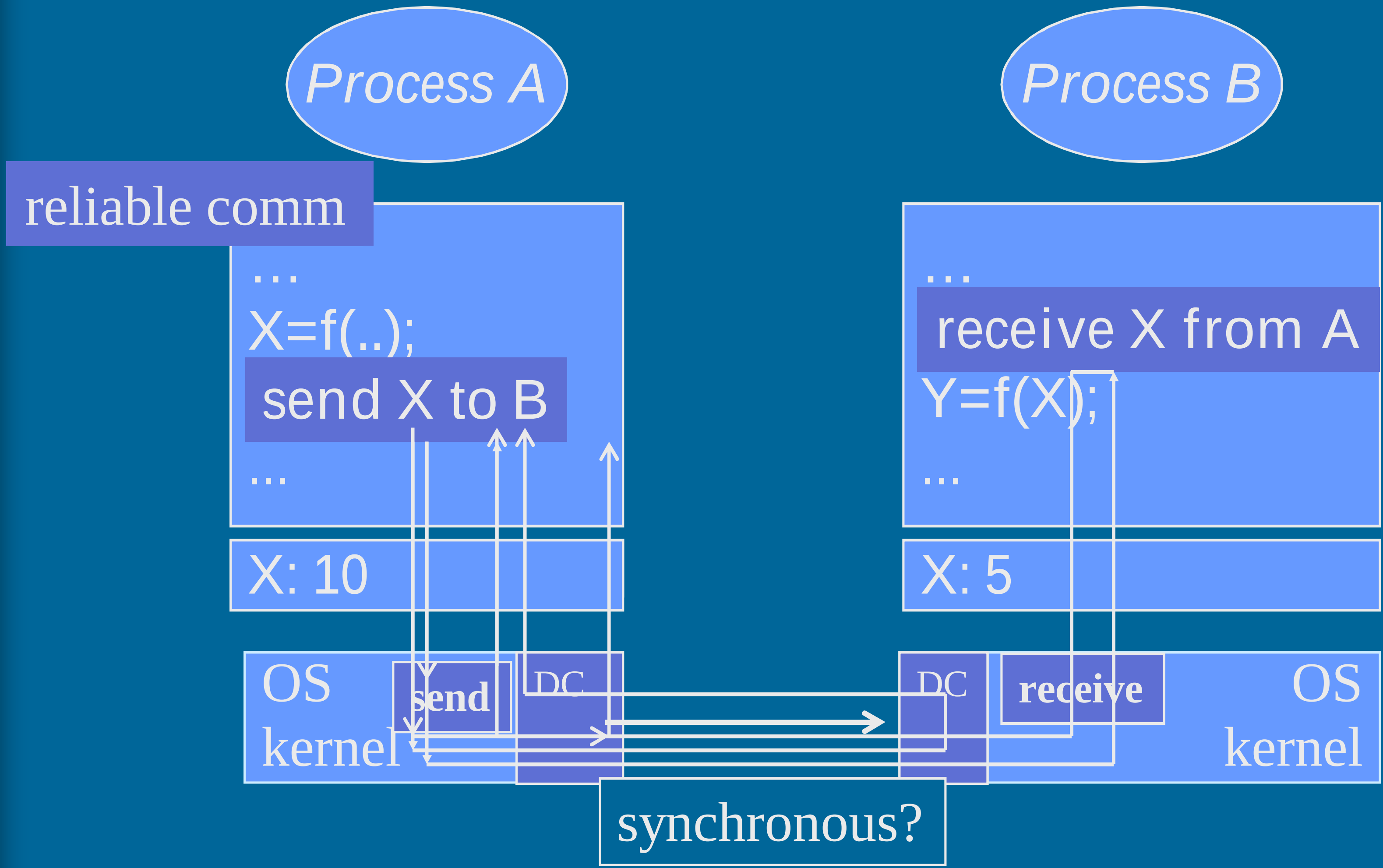
prosessi

kanava

palvelu

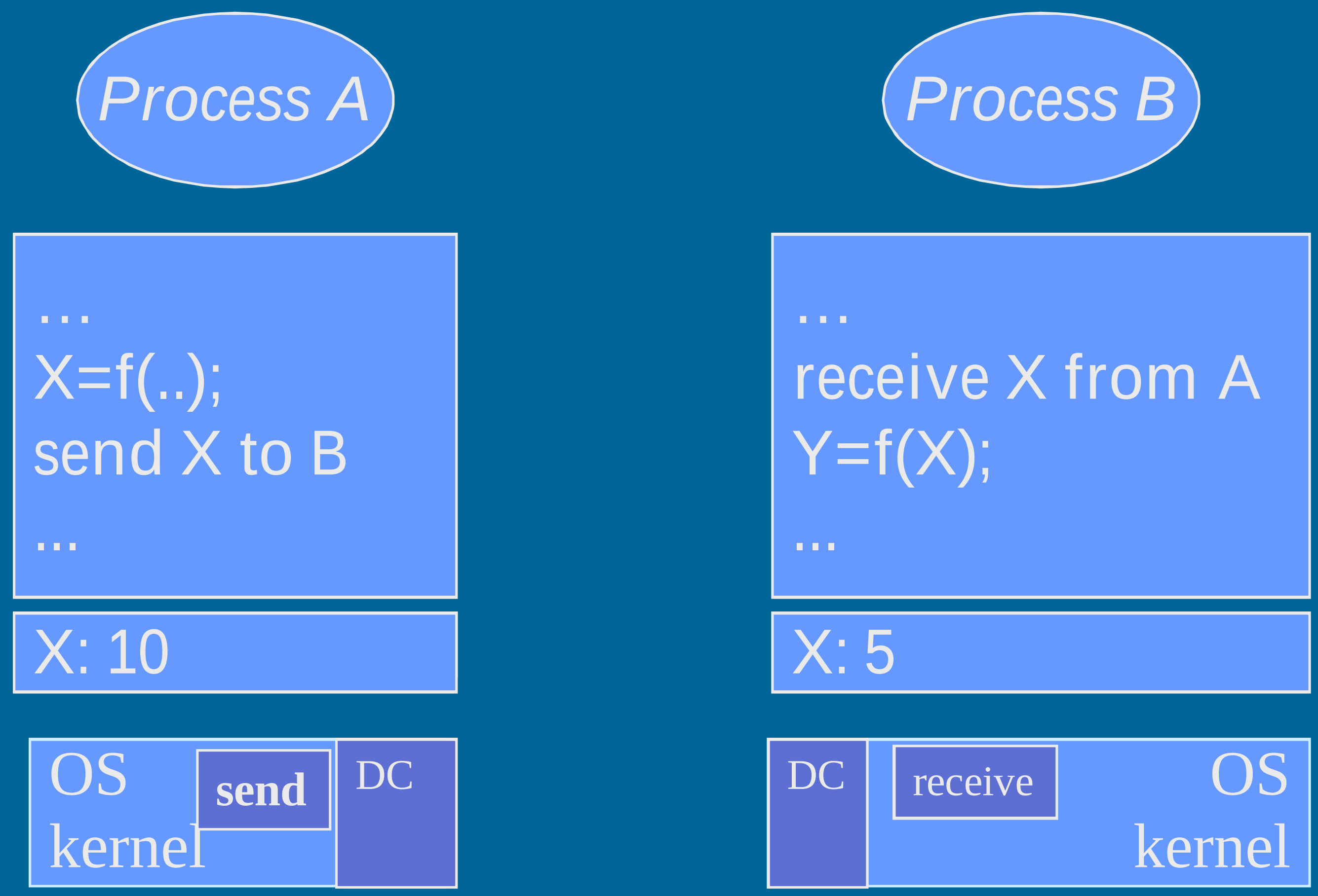
meklari

Synchronization levels (10)

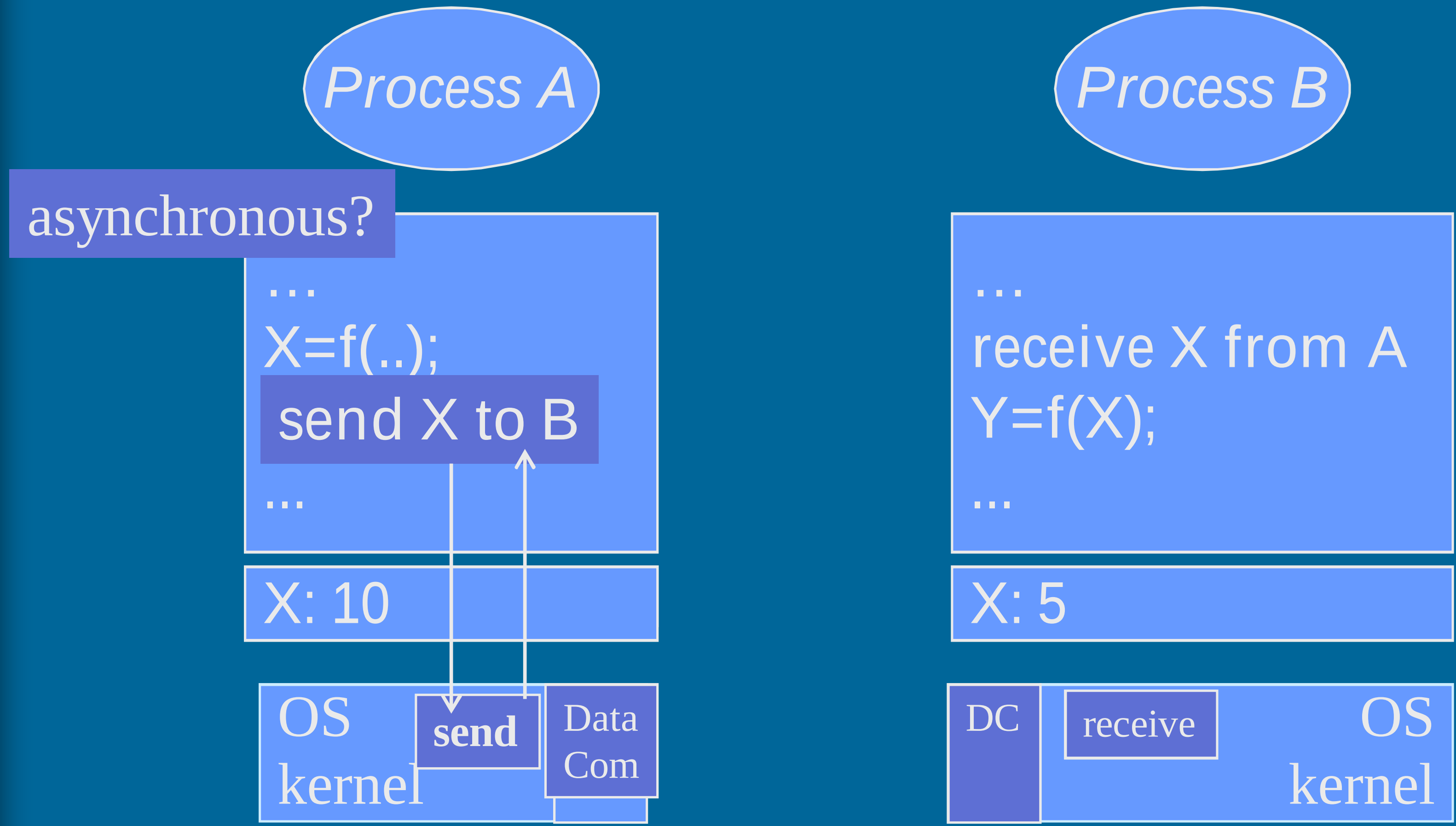


Discussion 0

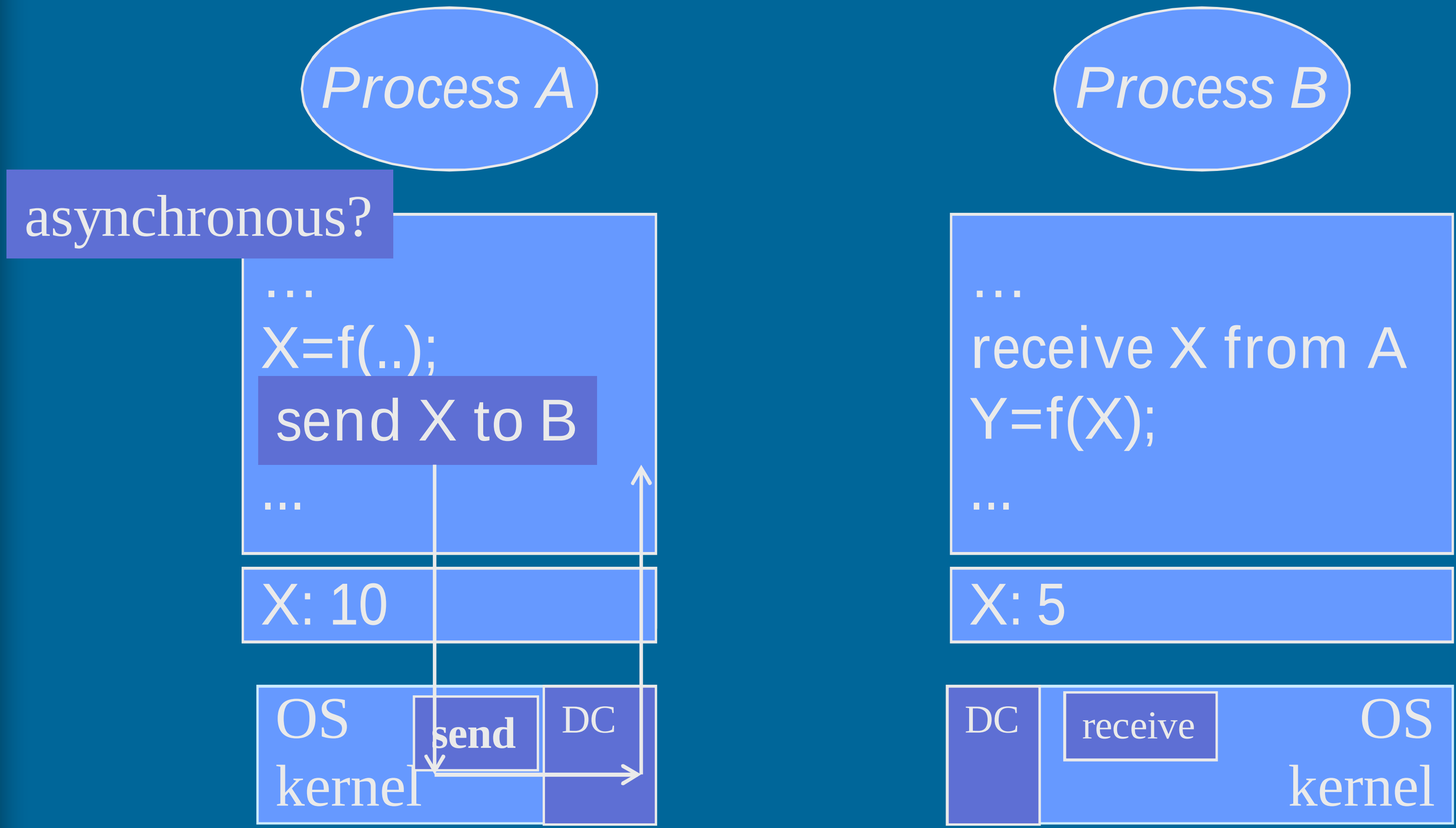
Synchronization levels (1/5)



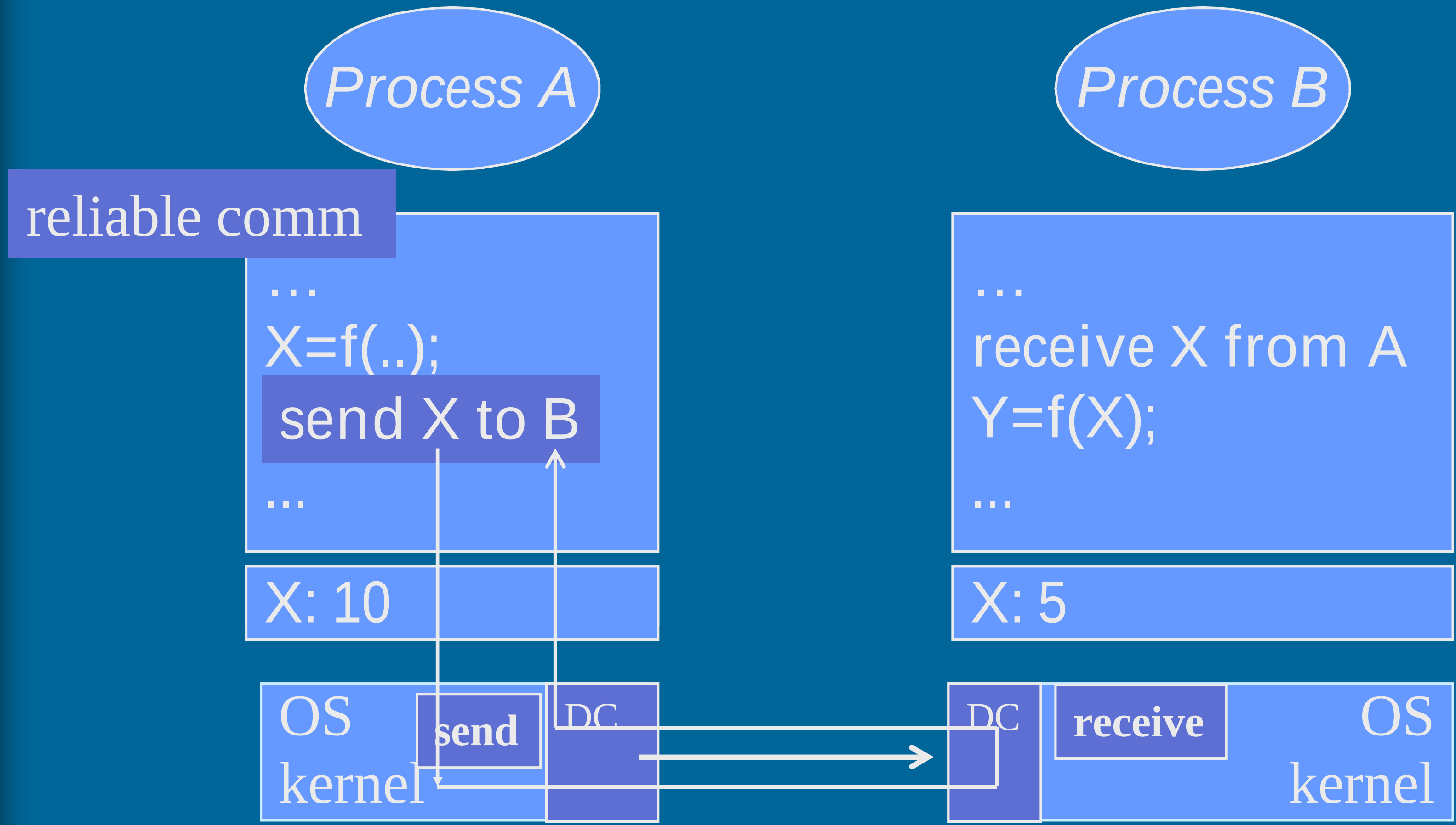
Synchronization levels (2/5)



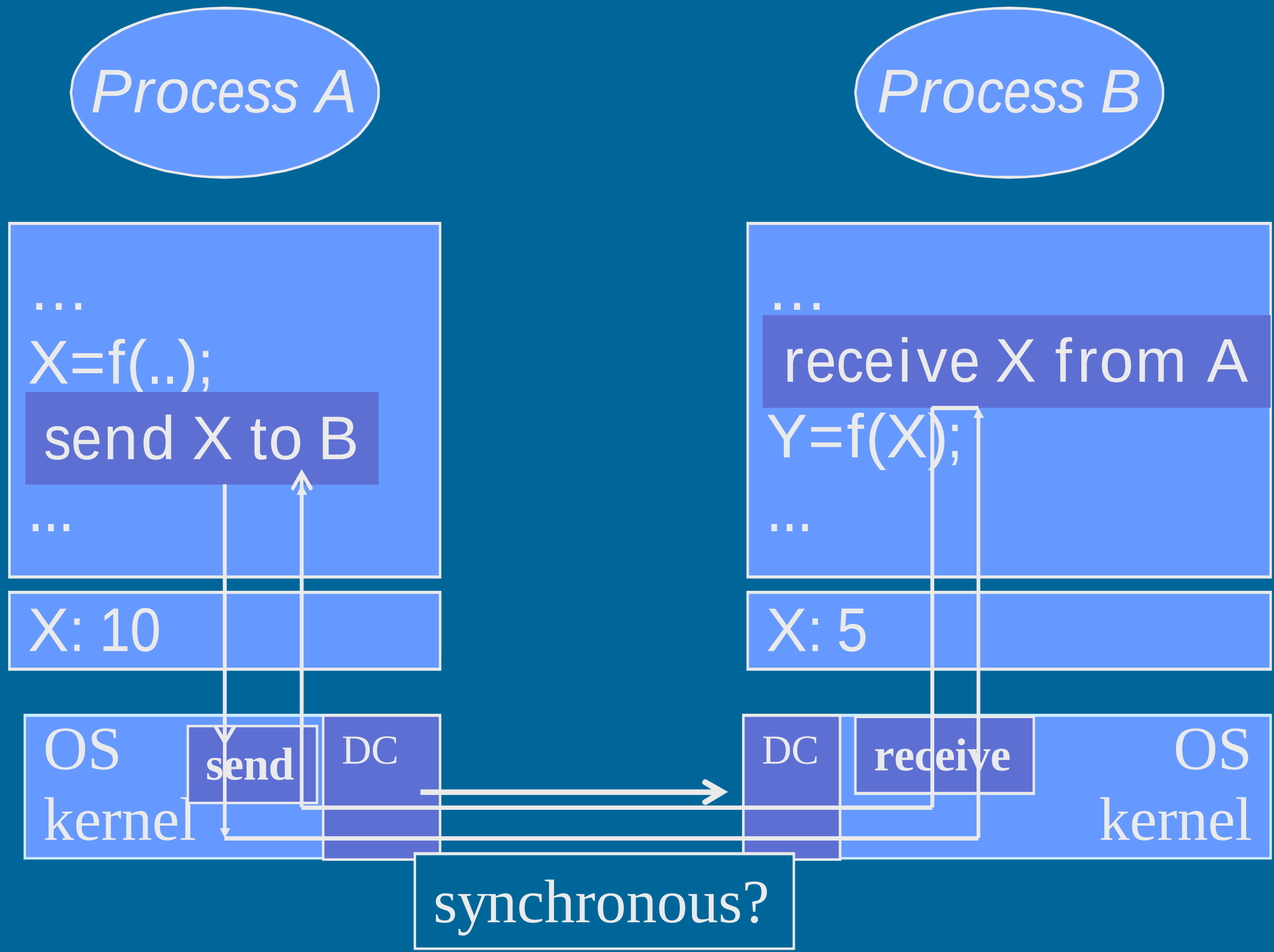
Synchronization levels (3/5)



Synchronization levels (4/5)



Synchronization levels (5/5)



Message Passing

- Symmetric communication
  - Cooperating processes at same level
  - Both know about each others address
  - Communication method for a fixed channel

- Asymmetric communication
  - Different status for communicating processes
  - Client-server model
    - Server address known, client address given in request

- Broadcast communication
  - Receiver not addressed directly
  - Message sent to everybody (in one node?)
  - Receivers may be limited in number
    - Just one?
    - Only the intended recipient will act on it?

Wait Semantics

- Sender
  - Continue after OS has taken the message
    - Non-blocking send
  - Continue after message reached receiver node
    - Blocking send
  - Continue after message reached receiver process
    - Blocking send
- Receiver
  - Continue only after message received
    - Blocking receive
  - Continue even if no message received
    - Status indicated whether message received or not
    - Non-blocking receive

Usual case

Usual case

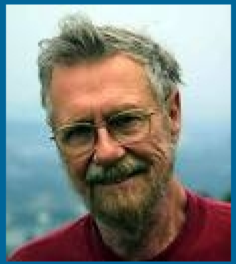
Message Passing

- Data flow
  - One-way
    - Synchronous may be one-way
    - Asynchronous is always one-way
  - Two-way
    - Synchronous may be two-way
    - Two asynchronous communications
- Primitives
  - One message at a time
  - Need addresses for communicating processes
  - Operating system level service
  - Usually not programming language level construct
    - Too primitive: need to know node id, process id, port number,…

data flow
vs.
control flow!

Channels

- History of languages utilizing channels
  - Guarded Commands
    - Dijkstra, 1975
  - Communicating Sequential Processes
    - CSP, Hoare, 1978
  - Occam
    - David May et al, 1983
    - Hoare as consultant
    - Inmos Transputer



Edsger Dijkstra

C.A.R. Hoare

David May

Guarded Commands (Dijkstra)

- Way to describe predicate transformer semantics
- Communication not really specified
- Guarded command
  - Condition or guard
  - Statement

C → S

predikaatti-muunnos-semantiikka

greatest common divisor

```
x, y = X, Y      -- statement (unguarded)
do   -- loop command, loop terminates when x = y
  x ≠ y →
     if      -- conditional command (itself guarded)
          x > y → x := x-y      -- guarded statement in the if
          y > x → y := y-x
     fi
od
print x ; -- another statement, also unguarded
```

vartioitu lauseke

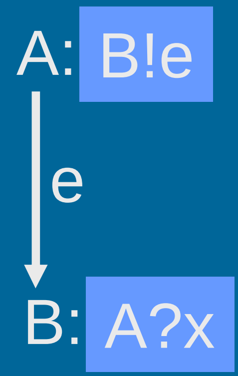
guard

can be also input/output statement

http://en.wikipedia.org

Communicating Sequential Processes – CSP (Hoare)

- Language for modeling and analyzing the behavior of concurrent communicating systems
- A known group of processes A, B, …
- Communication:
  - output statement: B!e
    - evaluate e, send the value to B
  - input statement: A?x
    - receive the value from A to x
  - input, output: blocking statements
  - output & input: "distributed assignment"
    - Communicate value from one process to a variable in some other process



CSP communication

- Input/output statements
  - Destination!port (e_1, …, e_n) ;
  - Source?port (x_1, …, x_n) ;
- Binding
  - Communication with **named processes**
  - Matching types for communication
- Example: **Copy** (West => Copy => East)

West:

```
do true ->
    Copy!c;
    …
od
```

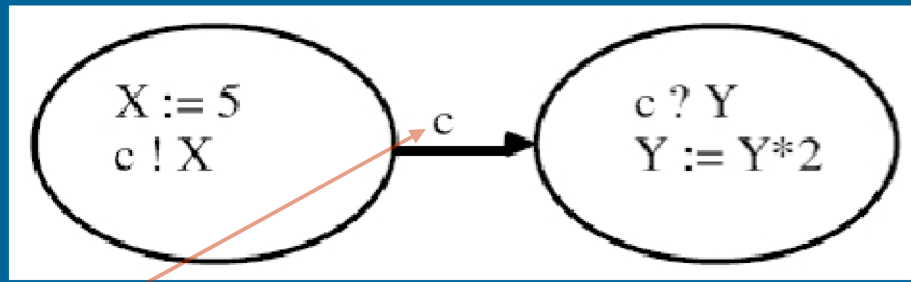
Copy:

```
do true ->
    West?c;
    East!c ;
od
```

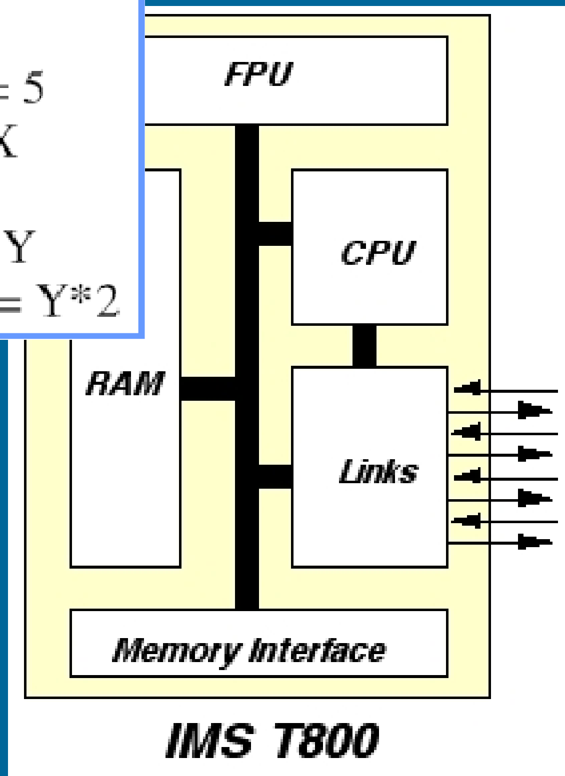
East:

```
do true ->
    Copy?c;
    …
od
```

OCCAM Language




```
PAR
  SEQ
    X := 5
    c ! X
  SEQ
    c ? Y
    Y := Y*2
```

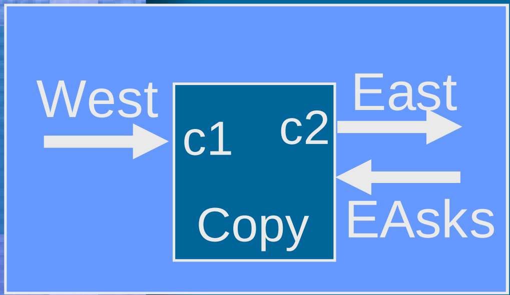


- Communication through **named channels**
  - Globally defined
    - Somewhere, in advance
  - Each channel has one sender and one receiver
    - Process in some node
- Transputer
  - Multicomputer
    - E.g., 100 node Hathi-2 in ÅA
  - Automatic message routing for channels
  - Programmed with OCCAM

http://www.embedded.com.au/reference/transputers.html

OCCAM Example

(Andrews, p 331)

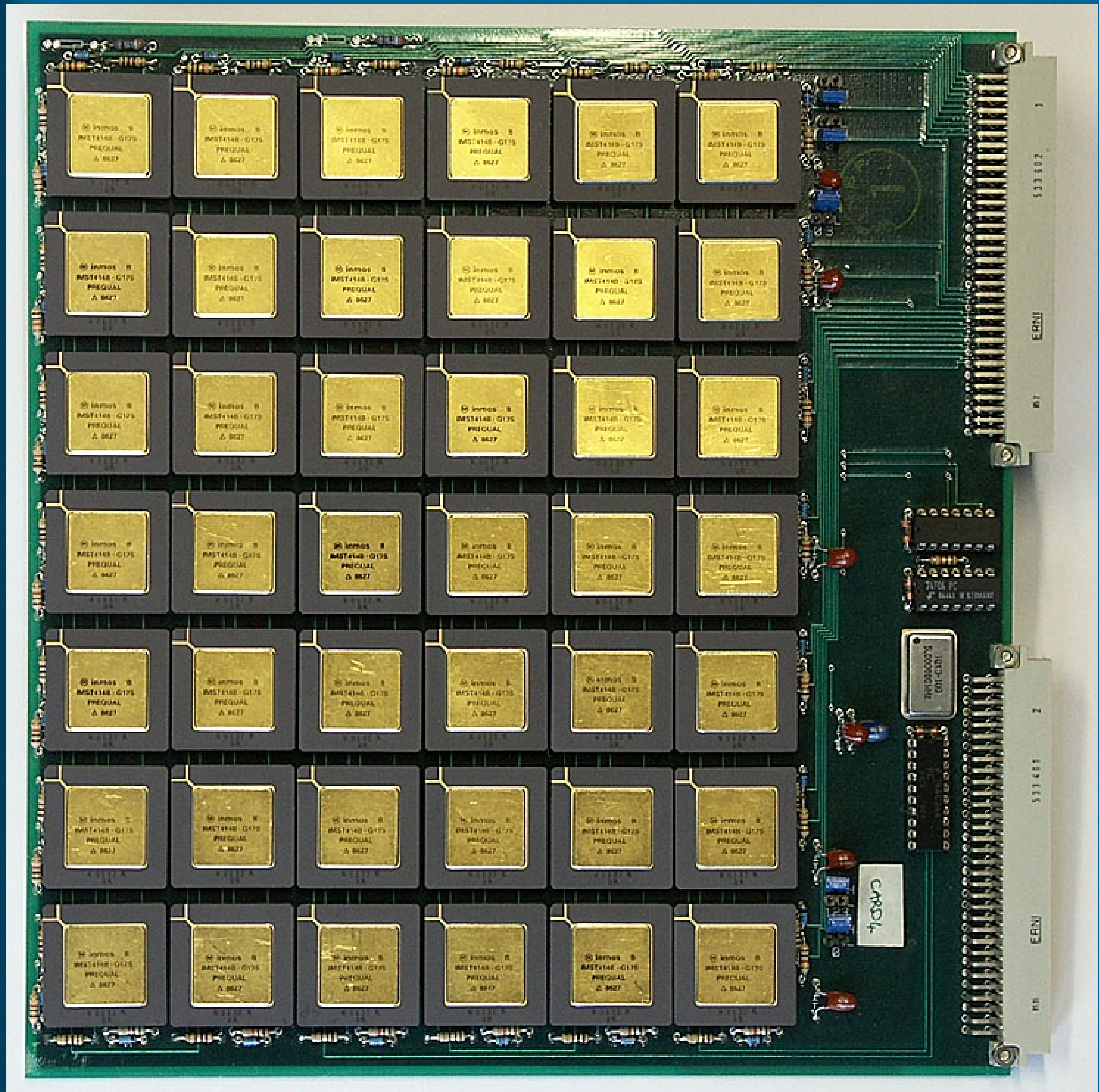



```
PROC Copy (CHAN OF BYTE West, EAsks, East)
  BYTE  c1, c2, dummy;  -- buffer size = 2
  SEQ
     West ? c1                         - - West has 1st byte
     WHILE TRUE
        ALT
           West ? c2                   - - West has new byte
              SEQ
                 East ! c1     - - send previous byte
                 c1 := c2      - - copy to buffer c1
           EAsks ? dummy  - - East wants a byte
              SEQ
                 East ! c1     - - send previous byte
                 West ? c1  - - receive next one
```

block here, until other end ready ("guards")

- How to bind processes to nodes? 8 vs. 100 nodes?
- How to bind channels to processes, physical system?
  - 4 physical ports (N, S, E, W) in each processor

Inmos Trans-puter



- B0042
- 2D array
- 10 boards
  420 cpu's
- 30 boards
  1260 cpu's

http://www.cs.bris.ac.uk/~dave/transputer.html

Channels

- Communication through named channels
  - Typed, global to processes
  - Programming language concept
  - Any one can read/write (usually limited in practice)
- Pipe or mailbox
- Synchronous, one-way (?)
- How to tie in with many nodes?
  - Not really thought through! Easy with shared memory!

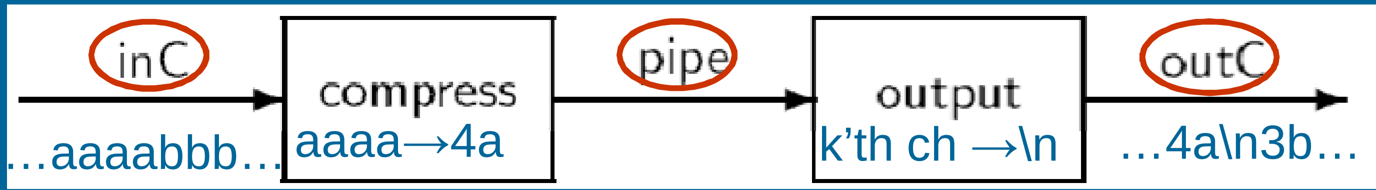
many readers/writers? same process writes and reads?

Algorithm 8.1: Producer-consumer (channels)

channel of integer ch			
producer		**consumer**	
	integer x		integer y
	loop forever		loop forever
p1:	x ← produce	q1:	ch ⇒ y
p2:	ch ⇐ x	q2:	consume(y)

buffer size?

Filtering Problem



- Compress many (at most MAX) similar characters to pairs …
  - {nr of chars, char}

"compress"

- … and place newline (\n) after every K'th character in the compressed string

"output"

- Why is it called "Conway's problem"?
  - "Classic coroutine example"

vuorottaisrutiinit

Conway, M. "Design of a separable transition-diagram compiler," *CACM* 6, 1963, pages 396–408.

Filtering Problem with Channels

Algorithm 8.2: Conway's problem

constant integer MAX ← 9
constant integer K ← 4
channel of integer inC, pipe, outC

	compress		**output**
	char c, previous ← 0		char c
	integer n ← 0		integer m ← 0
	inC ⇒ previous		
	loop forever		loop forever
p1:	inC ⇒ c	q1:	pipe ⇒ c
p2:	if (c = previous) and (n < MAX − 1)	q2:	outC ⇐ c
p3:	n ← n + 1	q3:	m ← m + 1
	else		
p4:	if n > 0	q4:	if m >= K
p5:	pipe ⇐ intToChar(n+1)	q5:	outC ⇐ newline
p6:	n ← 0	q6:	m ← 0
p7:	pipe ⇐ previous	q7:	
p8:	previous ← c	q8:	

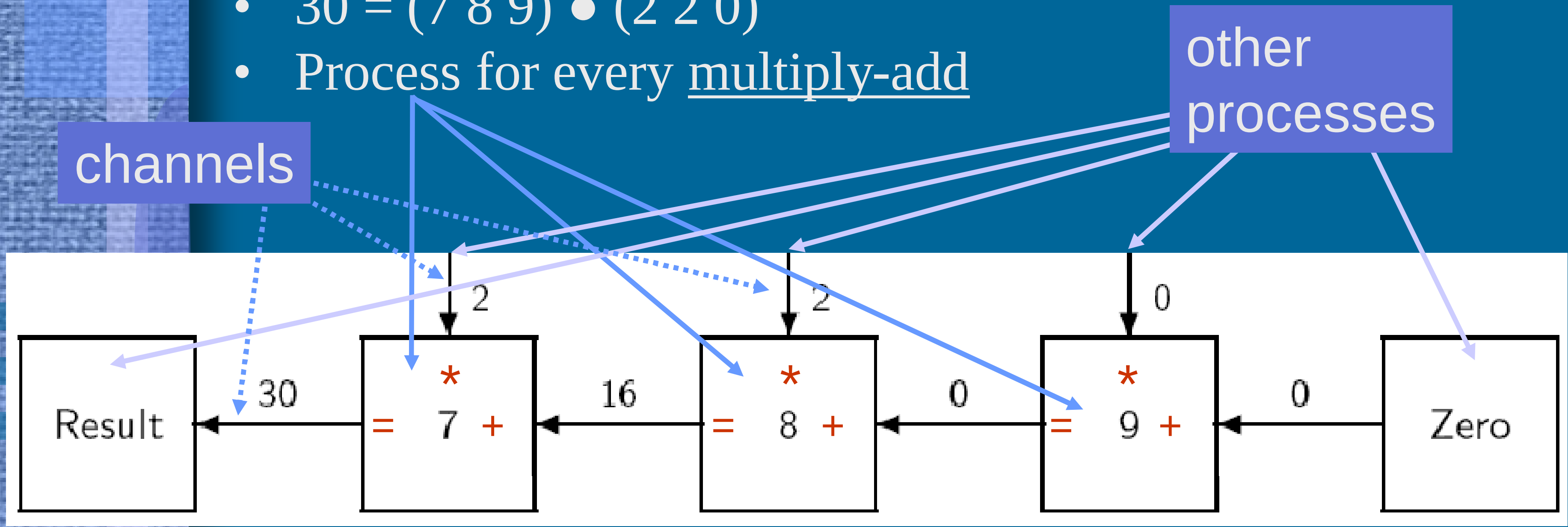
no last char?

Matrix Multiplication with Channels

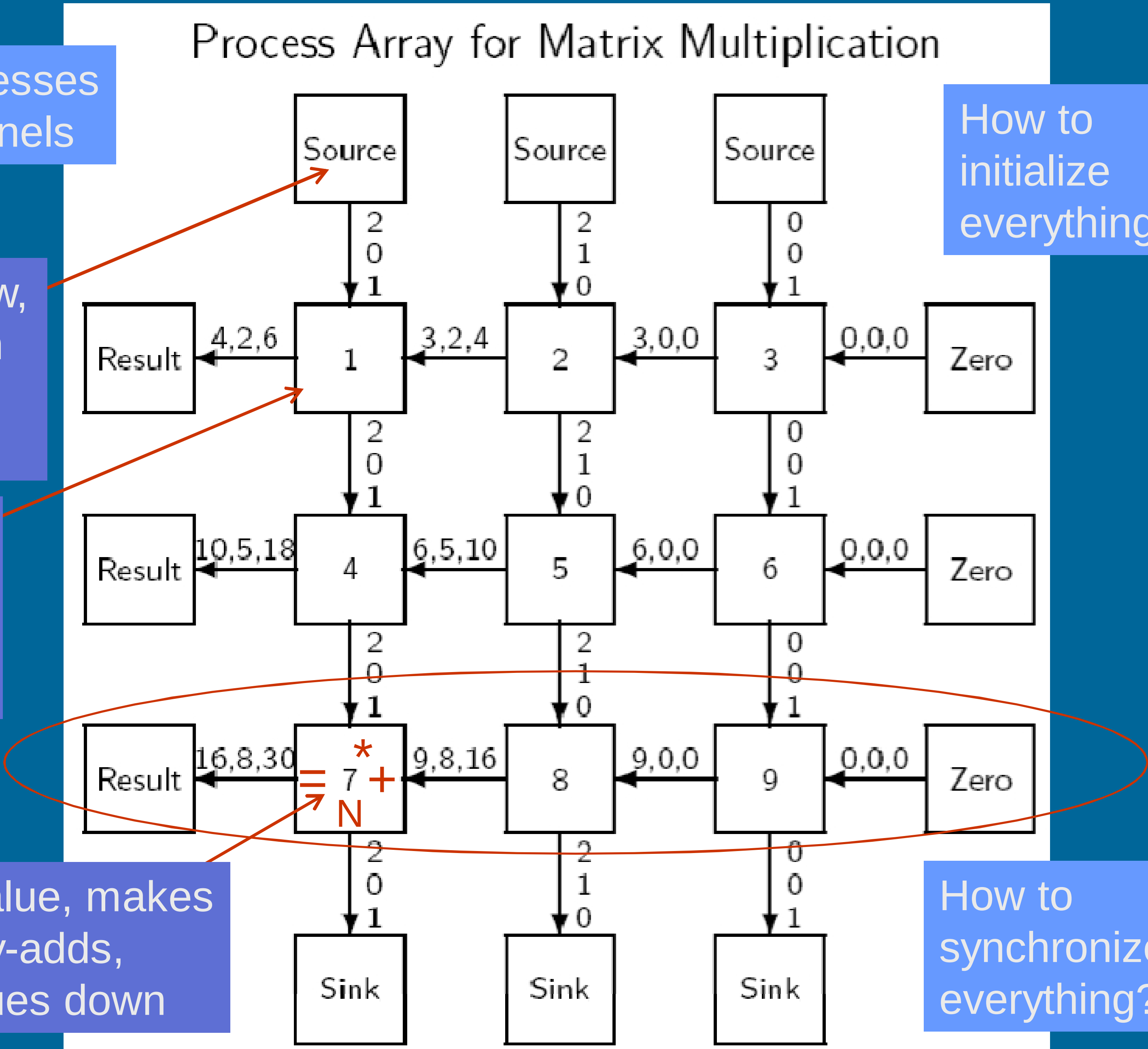
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 4 & 2 & 6 \\ 10 & 5 & 18 \\ 16 & 8 & 30 \end{pmatrix}$$

- 16 = (7 8 9) ● (1 0 1)
- 30 = (7 8 9) ● (2 2 0)
- Process for every multiply-add

7*2+ 8*2+ 9*0+ 0



27 processes
24 channels



How to initialize everything?

contains 1 row, sends it down one element at a time

West-bound multiply-add, South-bound copy North

contains 1 value, makes three multiply-adds, forwards values down

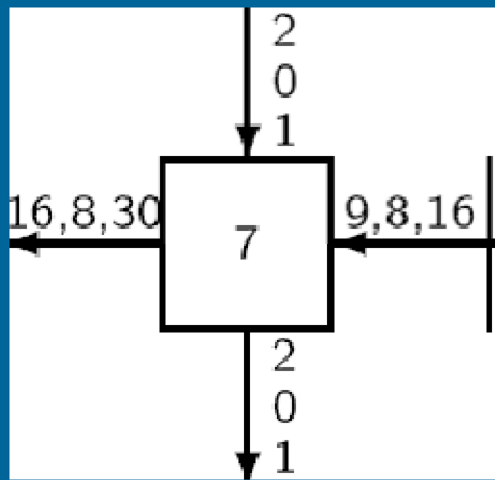
How to synchronize everything?

Algorithm 8.3: Multiplier process with channels

integer FirstElement
channel of integer North, East, South, West
integer Sum, integer SecondElement

Relative names?

```
    loop forever
p1:     North ⇒ SecondElement        ← wait 1st for this (*)
p2:     East ⇒ Sum                   ← and then for this
p3:     Sum ← Sum + FirstElement · SecondElement
p4:     South ⇐ SecondElement
p5:     West ⇐ Sum
```



- How to map processes to nodes?
- How to map channels to processes?
  - North channel of one process the South channel of some other
- North-South data flow has priority (*)
  - Waiting even when data-flow East-West available
  - Node on East may be blocked unnecessarily

Discussion 2

Algorithm 8.4: Multiplier with channels and selective input

```
integer FirstElement
channel of integer North, East, South, West
integer Sum, integer SecondElement

    loop forever
       either
p1:          North ⇒ SecondElement
p2:          East ⇒ Sum
       or
p3:          East ⇒ Sum
p4:          North ⇒ SecondElement
p5:       South ⇐ SecondElement
p6:       Sum ← Sum + FirstElement · SecondElement
p7:       West ⇐ Sum
```

If message from North available, do this

If message from East available, do this

sequential block

- Guarded statement
  - Execute one selective input statement
    - Nondeterministic selection (if both available)
    - p2 follows p1, it does not compete with p3

Discussion 3

Dining Philosophers with Channels

- Each fork i is a process, forks[i] is a channel
- Each philosopher i is a process

Algorithm 8.5: Dining philosophers with channels			
channel of boolean forks[5]			
	philosopher i		**fork i**
	boolean dummy		boolean dummy
	loop forever		loop forever
p1:	think	q1:	forks[i] ⇐ true
p2:	forks[i] ⇒ dummy	q2:	forks[i] ⇒ dummy
p3:	forks[i⊕1] ⇒ dummy	q3:	
p4:	eat	q4:	mutex?
p5:	forks[i] ⇐ true (would *false*	q5:	deadlock?
p6:	forks[i⊕1] ⇐ true be ok?)	q6:	

- Would it be enough to initialize each *forks[i] <= true* ?
  - Do you really need *forks[i] => dummy* in fork i? Why?

Rendezvous (1978, Abrial & Andrews)

- Synchronization with communication
  - No channels, usage similar to procedure calls
  - One (*accepting*) process waits for one of the (*calling*) processes
    - One request in service at a time

asymmetric

  - Calling process must know id of the accepting process
  - Accepting process does not need to know the id of calling process
  - May involve parameters and return value
- Good for client-server synchronization
  - Clients are calling processes `server.service(parm, result)`
  - Server is accepting process `accept service(p, r)`
  - Server is active process
  - Language construct, no mapping for real system nodes

Algorithm 8.6: Rendezvous

client		server	
	integer parm, result		integer p, r
	loop forever		loop forever
p1:	parm ← . . .	q1:	
p2:	server.service(parm, result)	q2:	accept service(p, r)
p3:	use(result)	q3:	r ← do the service(p)

- Can have many similar clients
- Implementation with messages (e.g.)
  - Service request in one message
    - Arguments must be marshalled (make them suitable for transmission)
  - Wait until reply received
  - Reply result in another message

Guards in Rendezvous

- Additional constraint for accepting given service call
- Accept service call, if
  - Someone requests it and
  - Guard for that request type is true
    - Guard is based on local state
- If many such requests (with open guards) available, select one randomly
- Complete one request at a time
  - Implicit mutex

Ada Rendezvous

Bounded Buffer in Ada

Export public ops defined before task body

```
task body Buffer is
  B: Buffer_Array;
  In_Ptr, Out_Ptr, Count: Index := 0;
begin
  loop
    select
      when Count < Index'Last =>
        accept Append(I: in Integer ) do
            B(In_Ptr) := I;
        end Append;
      Count := Count + 1; In_Ptr := In_Ptr + 1;
    or
      when Count > 0 =>
        accept Take(I: out Integer ) do
            I := B(Out_Ptr);
        end Take;
      Count := Count – 1; Out_Ptr := Out_Ptr + 1;
    or
        terminate;
    end select;
  end loop;
end Buffer;
```

```
…
Buffer.Append (456);
Buffer.Append (333);
…
```

```
…
Buffer.Take(x);
Buffer.Take(y);
…
```

Terminates when no rendezvous processes available? Tricky!
How to know?
No concurrent operations!

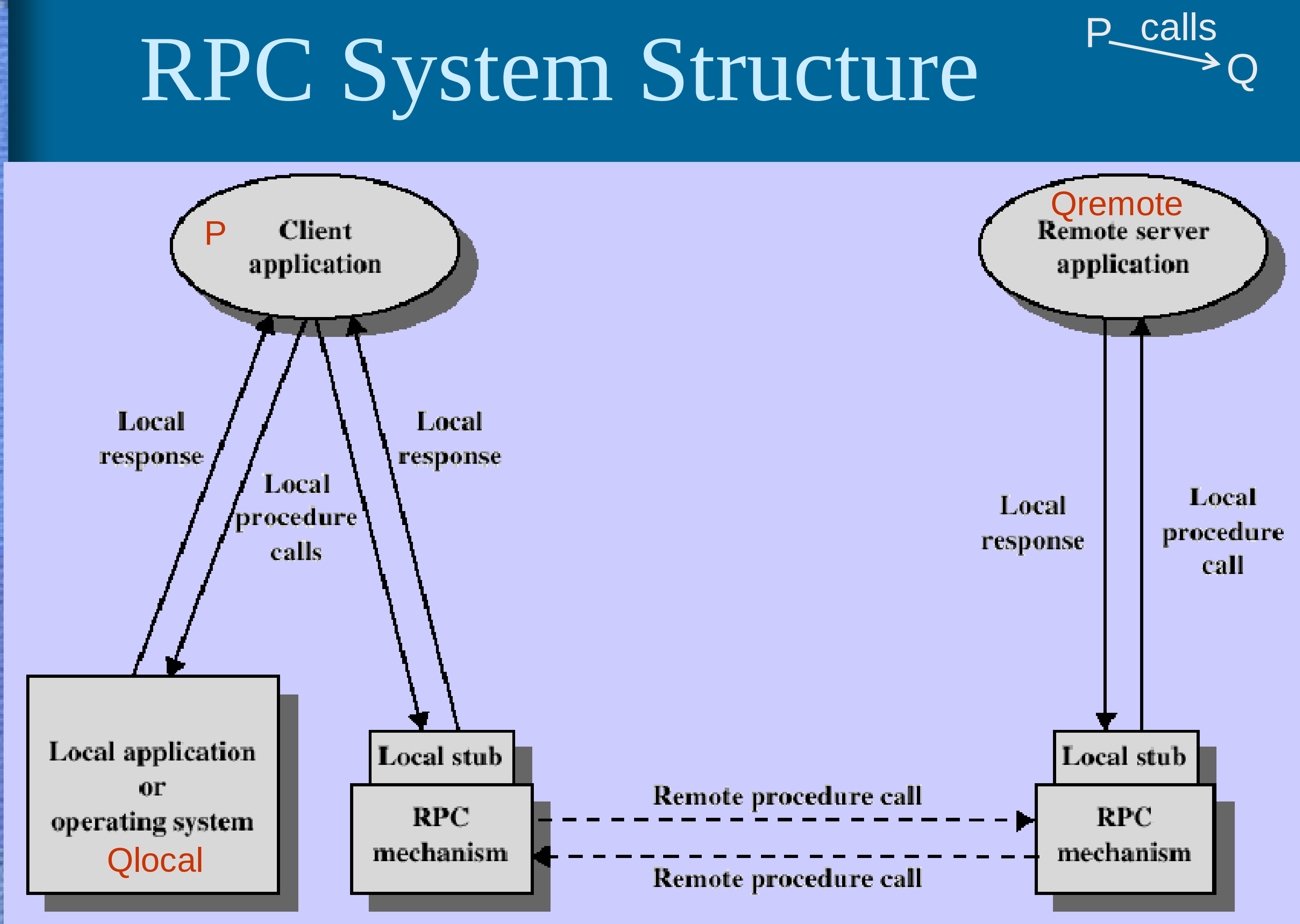
Remote Procedure Call

- Common <u>operating system service</u> for client-server model synchronization
  - Implemented with messages
  - Parameter marshalling
    - Semantics remain, implementation may change
  - Mutex problem
    - Combines monitor and synchronized messages?
      - Automatic mutex for service
    - Multiple calls active simultaneously?
      - Mutex problems solved within called service

Usual case

  - Semantics similar to ordinary procedure call
    - But no global environment (e.g., shared array)
  - Two-way synchronized communication channel
    - Client waits until service completed (usually)

RPC System Structure



(Sta05 Fig 14.12)

RPC Module

```
module mname
   op opname(formals)  [returns  result]
body
    variable declarations;
    initialization code;
    proc opname(formal identifiers)  returns  result identifier
       declarations of local variables;
       statements
    end
    local procedures and processes;
end mname
```

Export public ops

Call:

```
call mname.opname(arguments)
```

RPC Example: Time Server

```
module TimeServer
  op get_time() returns int;   # retrieve time of day
  op delay(int interval);      # delay interval ticks
body
  int tod = 0;               # the time of day
  sem m = 1;                 # mutual exclusion semaphore
  sem d[n] = ([n] 0);        # private delay semaphores
  queue of (int waketime, int process_id) napQ;
  ## when m == 1, tod < waketime for delayed processes

  proc get_time() returns time {
    time = tod;
  }

  proc delay(interval) {     # assume interval > 0
    int waketime = tod + interval;
    P(m);
    insert (waketime, myid) at appropriate place on napQ;
    V(m);
    P(d[myid]);    # wait to be awakened
  }
```

mutex

(And00 Fig 8.1)

(process Clock{} on next slide)

```
process Clock {
   start hardware timer;
   while (true) {
      wait for interrupt, then restart hardware timer;
      tod = tod+1;
      P(m);
      while (tod >= smallest waketime on napQ) {
         remove (waketime, id) from napQ;
         V(d[id]);  # awaken process id
      }
      V(m);
   }
}
end TimeServer
```

- Internal process
  - Keeps the time
  - Wakes up delayed clients
- Service RPC's:

```
time = TimeServer.get_time();
TimeServer.delay(10);
```

Linux machine>> man rpc

```
RPC(3)                                                    RPC(3)

NAME
     rpc - library routines for remote procedure calls

SYNOPSIS AND DESCRIPTION
     These routines allow C programs to make procedure calls on other
     machines across the network.  First, the client calls a procedure to
     send a data packet to the server.  Upon receipt of the packet, the
     server calls a dispatch routine to perform the requested service, and
     then sends back a reply.  Finally, the procedure call returns to the
     client.
```

```
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
     char *host;
     u_long prognum, versnum, procnum;
     char *in, *out;
     xdrproc_t inproc, outproc;
```

remote process

decode/encode parameters/results

Remote Method Invocation (RMI)

```
package example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
        String sayHello() throws RemoteException;
}
```

rmi server

http://java.sun.com/j2se/1.5.0/docs/guide/rmi/hello/hello-world.html

- Java RPC
- Start rmiregistry
  - Stub lookup (default at port 1099)
- Start rmi server
  - Server runs until explicitly terminated by user

rmiregistry &

start rmiregistry

java -classpath *classDir* example.hello.Server &

start java -classpath *classDir* example.hello.Server

rmi server

```
package example.hello;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Server implements Hello {
    public Server() {}
    public String sayHello() {
        return "Hello, world!"; }
    public static void main(String args[]) {
        try { Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
                    // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);
            System.err.println("Server ready");
        }   catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
}
```

Output: Server ready

rmi client

```java
package example.hello;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    private Client() {}
    public static void main(String[] args) {
        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Output: response: Hello, world!

Summary

- Distributed communication with messages
  - Synchronization and communication
  - Computation time + communication time = ?
- Higher level concepts
  - Guarded commands (theoretical background)
  - CSP (idea) & Occam (application)
  - Named Channels (ok without shared memory?)
  - Rendezvous
  - RPC & RMI (Java)

