Table of Contents

Three Issues

- Use of multiple PEs
- Cooperation among the PEs
- Potential for survival to partial failure

Control Mechanisms

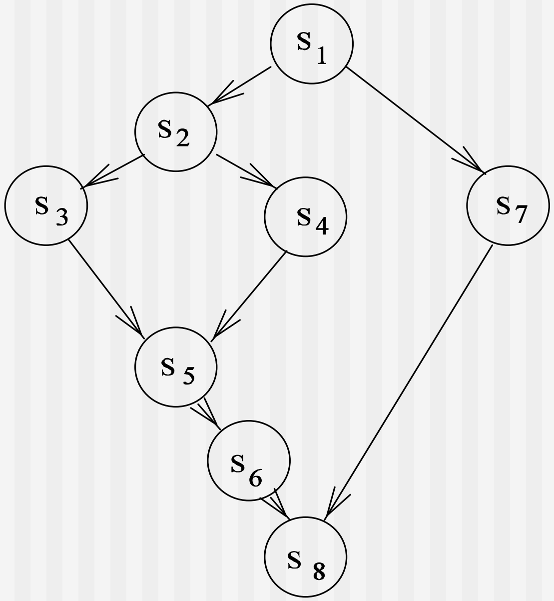
Statement type \ Control type	Sequential control	Parallel Control
Sequential/parallel statement	**Begin** S_1, S_2 **end**	**Parbegin** S_1, S_2 **Parend** **Fork/join**
Alternative statement	**goto**, **case if** C **then** S_1 **else** S_2	Guarded commands: G $\rightarrow$C
Repetitive statement	**for … do**	**doall, for all**
Subprogram	**procedure** **Subroutine**	**procedure** **subroutine**

Four basic sequential control mechanisms with their parallel counterparts.

Focus 6: Expressing Parallelism

parbegin/parend statement

$S_1;[[S_2;[S_3||S_4];S_5;S_6]||S_7];\mathrm{S}_8$

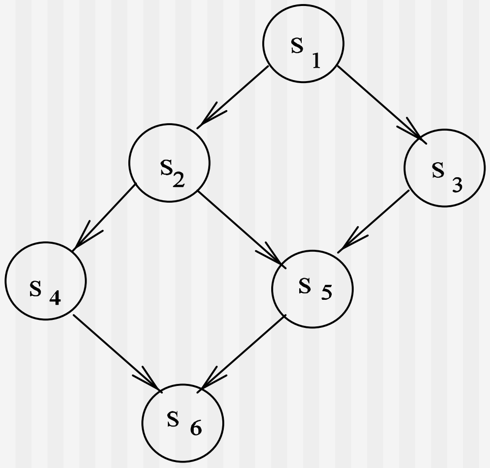


A precedence graph of eight statements.

Focus 6 (Cont'd.)

fork/join statement

s_1;
$c_1 := 2$;
fork $L1$;
s_2;
$c_2 := 2$;
fork $L2$;
s_4;
go to $L3$;
$L1$: s_3;
$L2$: **join** c_1;
s_5;
$L3$: **join** c_2;
s_6;



A precedence graph.

Dijkstra's Semaphore + Parbegin/Parend

S(i): A sequence of P operations; S_i; a sequence of V operations

s: a binary semaphore initialized to 0.

$S(1)$: $S_1;V(s_{12});V(s_{13})$
$S(2)$: $P(s_{12});S_2;V(s_{24});V(s_{25})$
$S(3)$: $P(s_{13});S_3;V(s_{35})$
$S(4)$: $P(s_{24});S_4;V(s_{46})$
$S(5)$: $P(s_{25});P(s_{35});S_5;V(s_{56})$
$S(6)$: $P\ (s_{46});\ P\ (s_{56});\ S_6$

Focus 7: Concurrent Execution

- $R(S_i)$, the **read set** for S_i, is the set of all variables whose values are referenced in S_i.
- $W(S_i)$, the **write set** for S_i, is the set of all variables whose values are changed in S_i.
- **Bernstein conditions**:
  - $R(S_1) \cap W(S_2) = \phi$
  - $W(S_1) \cap R(S_2) = \phi$
  - $W(S_1) \cap W(S_2) = \phi$

Example 7

$S_1 : a := x + y,$

$S_2 : b := x \times z,$

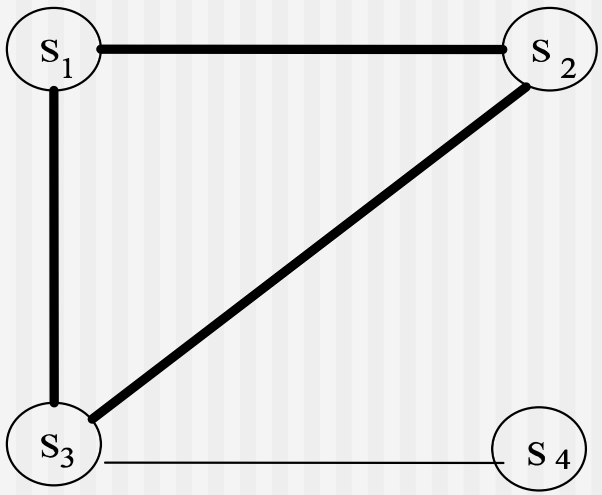
$S_3 : c := y$ - 1, and

$S_4 : x := y + z.$

$S_1 || S_2$, $S_1 || S_3$, $S_2 || S_3$, and $S_3 || S_4$.

Then, $S_1 || S_2 || S_3$ forms a largest complete subgraph.

Example 7 (Cont'd.)



A graph model for Bernstein's conditions.

Alternative Statement

Alternative statement in DCDL (CSP like distributed control description language)

$$[\ G_1 \rightarrow C_1 \,\square\, G_2 \rightarrow \mathrm{C}_2 \;\; \square \ldots \square\, G_n \rightarrow C_n\ ].$$

Example 8

Calculate $m = \max\{x, y\}$:

$$[x \geq y \rightarrow m := x \;\Box\; y \geq \mathrm{x} \rightarrow \mathrm{m} := y]$$

Repetitive Statement

$$*[\ G_1 \rightarrow\ C_1 \ \square\ G_2 \rightarrow\ C_2 \ \square \ldots \square\, \mathrm{G}_\mathrm{n} \rightarrow\ C_n\ ].$$

Example 9

meeting-time-scheduling ::= t := 0;

$*[\ t := a(t)\ \square\ t := \mathrm{b}(t)\ \square\ t := c(t)\ ]$

Non-deterministic schedule brings the issue of fairness

Fairness

A scheduler that allows all possible schedules is *unfair*

Unconditional fairness: every process (enabled or not) gets its turn infinitely often.

$$*[\text{true} \rightarrow x{:=}0 \;\square\; x=0 \rightarrow x{:=}1 \;\square\; x=1 \rightarrow x{:=}2]$$

Unfair: x = 0, 0, 0, 0, 0, 0,…

Unfair: x = 0, 1, 0, 0, 0, 1, …

Fair: x = 0, 1, 2, 0, 1, 0, …

Fairness (cond't)

Weak fairness: every process that is continuously enabled from a certain time, gets its turn (infinitely) often.

Strong fairness: every process enabled infinitely often gets its turn (infinitely) often.

x = T

$* [x \rightarrow y{:=}T \,\square\, x \rightarrow y{:=}F \,\square\, y \rightarrow x{:=}F \,\square\, y \rightarrow y{:=}\text{not } y]$

Weakly fair scheduler: termination is not guaranteed

Strongly fair scheduler: termination is guaranteed

Communication and Synchronization

- One-way communication: **send** and **receive**
- Two -way communication: **RPC**(Sun), **RMI**(Java and CORBA), and **rendezvous** (Ada)
- Several **design decisions**:
  - One-to one or one-to-many
  - Synchronous or asynchronous
  - One-way or two-way communication
  - Direct or indirect communication
  - Automatic or explicit buffering
  - Implicit or explicit receiving

Primitives	Example Languages
PARALLELISM	
Expressing parallelism	
Processes	Ada, Concurrent C, Lina, NIL Emerald,
Objects	Concurrent Smalltalk
Statements	Occam
Expressions	Par Alfl, FX-87
Clauses	Concurrent PROLOG, PARLOG
Mapping	
Static	Occam, Star Mod
Dynamic	Concurrent PROLOG, ParAlfl
Migration	Emerald
COMMUNICATION	
Message Passing	
Point-to-point messages	CSP, Occam, NIL
Rendezvous	Ada, Concurrent C
Remote procedure call	DP, Concurrent CLU, LYNX
One-to-many messages	BSP, StarMod
Data Sharing	
Distributed data Structures	Lina, Orca
Shared logical variables	Concurrent PROLOG, PARLOG
Nondeterminism	
Select statement	CSP, Occam, Ada, Concurrent C, SR
Guarded Horn clauses	Concurrent PROLOG, PARLOG
PARTIAL FILURES	
Failure detection	Ada, SR
Atomic transactions	Argus, Aeolus, Avalon
NIL	

Message-Passing Library for Cluster Machines (e.g., Beowulf clusters)

- **Parallel Virtual Machine (PVM):**
  www.epm.ornl/pvm/pvm_home.html
- **Message Passing Interface (MPI):**
  www.mpi.nd.edu/lam/
  www-unix.mcs.anl.gov/mpi/mpich/
- **Java multithread programming:**
  www.mcs.drexel.edu/~shartley/ConcProjJava
  www.ora.com/catalog/jenut
- **Beowulf clusters**:
  www.beowulf.org

Message-Passing (Cont'd.)

- **Asynchronous** point-to-point message passing:
  - **send** message list **to** destination
  - **receive** message list {**from** source}
- **Synchronous** point-to-point message passing:
  - **send** message list **to** destination
  - **receive** empty signal **from** destination
  - **receive** message list **from** sender
  - **send** empty signal **to** sender

Relationships Between Models

- Stronger model: a model with more constraints
  - Strong: synchronous systems, Weak: asynchronous systems
  - Strong: bounded-delay channels, Weak: unbounded-delay channels

- *Can model A be simulated using model B*: layers of abstraction
  - FIFO channel for Non-FIFO channel

    sequence number

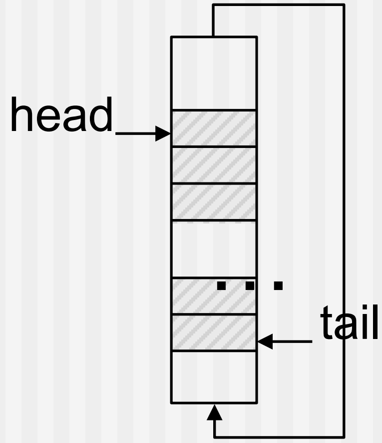
    ack with bounded buffer
  - Asynchronous systems for synchronous systems

    ABD synchronizer (for asynchronous bounded delay)

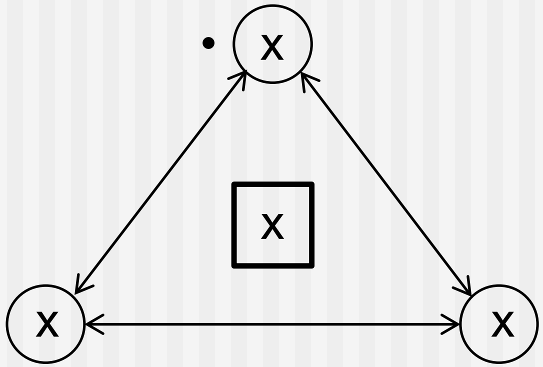
    Awerbuch's three synchronizers (for asynchronous unbounded delay)

Relationships Between Models (cont'd)

- Message Passing for Shared Memory: circular buffer



- Shared Memory for Message Passing: (total order) multicast



Example 10

The squash program replaces every pair of consecutive asterisks "**" by an upward arrow “↑”.

input::= * [**send** *c* **to** squash]

output::= * [**receive** *c* **from** squash]

Example 10 (Cont'd.)

```
squash::=
    *[ receive c from input →
            [ c ≠ *→ send c to output
             □ [ c = *→ receive c from input;
                        [ c ≠ *→ send * to output;
                                 send c to output
                          c = *→ send ↑ to output
                        ] □
                  ]
            ]
    ]
```

Partial Correctness

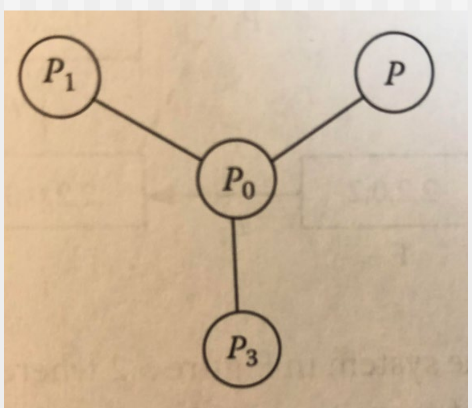
- Neighbors are in different colors

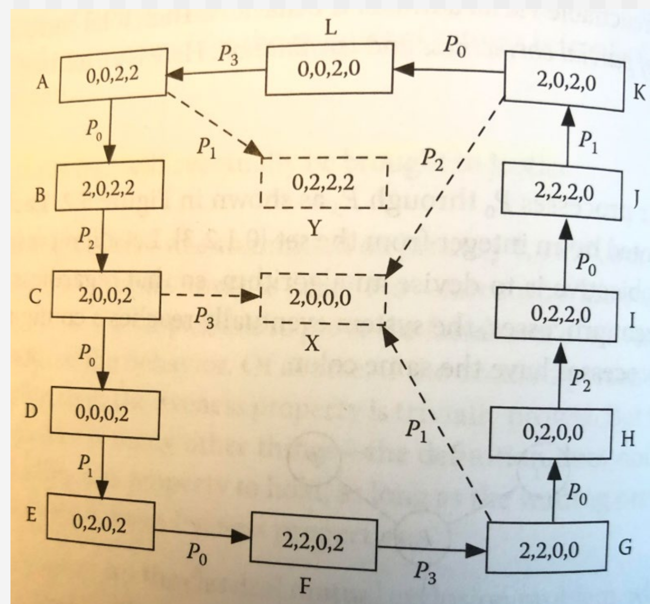
Program produces correct results if the program terminates

Color set $\{0, 1, 2, 3\}$

$Colorme ::= *[c[i] = c[j] \wedge j \in N[i] \rightarrow c[i] := c[i] + 2 \bmod 4]$

Colorme is partial correct



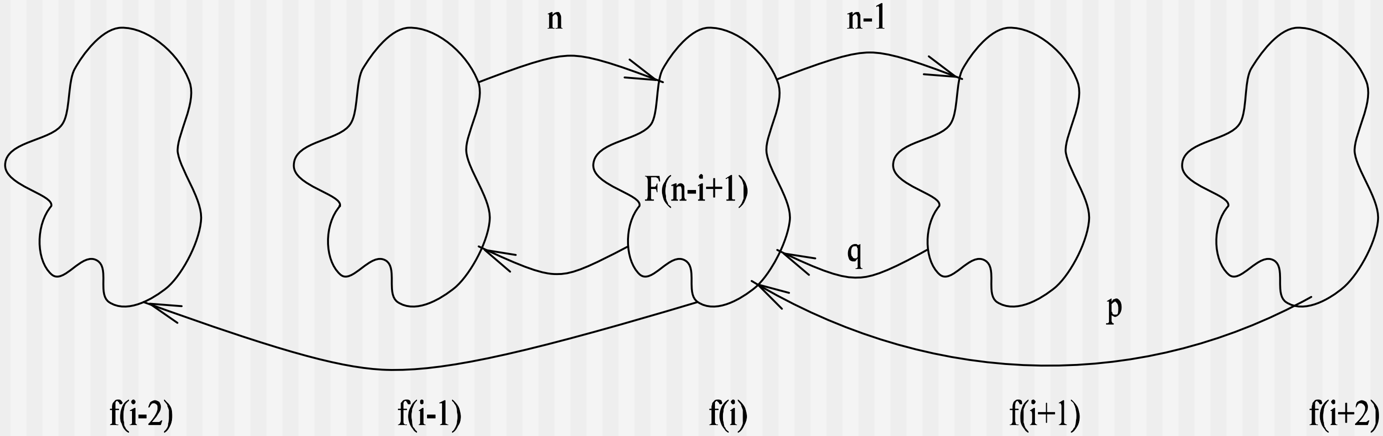


Focus 8: Fibonacci Numbers

- $F(i) = F(i\text{-}1) + \mathrm{F}\ (i - 2)$ for i > 1, with initial values F(0) = 0 and $F(1) = 1$.
- $F(i) = (\phi^{\,i} - \phi'^{i})/(\phi - \phi')$,where $\phi = (1+5^{0.5})/2$ (golden ratio) and $\phi' = (1\text{-}5^{0.5})/2$.

0, 1, 2, 3, 5, 8, 13, 21, 35, 54, …

Focus 8 (Cont'd.)



A solution for F (n).

Focus 8 (Cont'd.)

- f(0) ::=

  send n **to** f(1);

  receive p **from** f(2);

  receive q **from** f(1);

  ans := q
- f(-1) ::=

  receive p **from** f(1)

Focus 8 (Cont'd.)

- f(i) ::=

  receive n **from** f(i - 1);

  [n > 1 → [**send** n - 1 **to** f(i + 1);

  receive p **from** f(i + 2);

  receive q **from** f(i + 1);

  send p + q **to** f(i - 1);

  send p + q **to** f(i - 2)]

  □ n = 1 → [**send** 1 **to** f(i - 1);

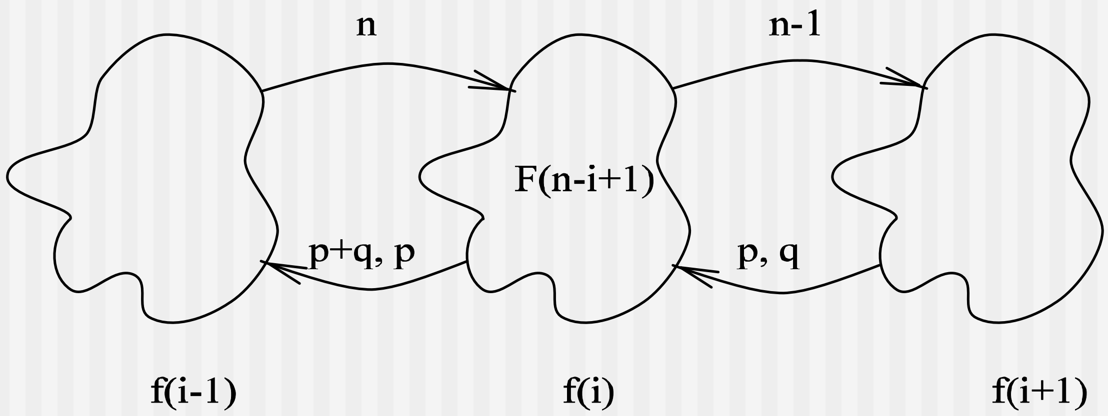
  send 1 **to** f(i - 2)]

  □ n = 0 → [**send** 0 **to** f(i - 1);

  send 0 **to** f(i - 2)]

  ]

Focus 8 (Cont'd.)



Another solution for F (n).

Focus 8 (Cont'd.)

- f(0)::=

```
[ n > 1 → [ send n to f(1);
             receive p from f(1);
             receive q from f(1);
             ans := p
             ]
 □ n = 1 → ans := 1
 □ n = 0 → ans := 0
]
```

Focus 8 (Cont'd.)

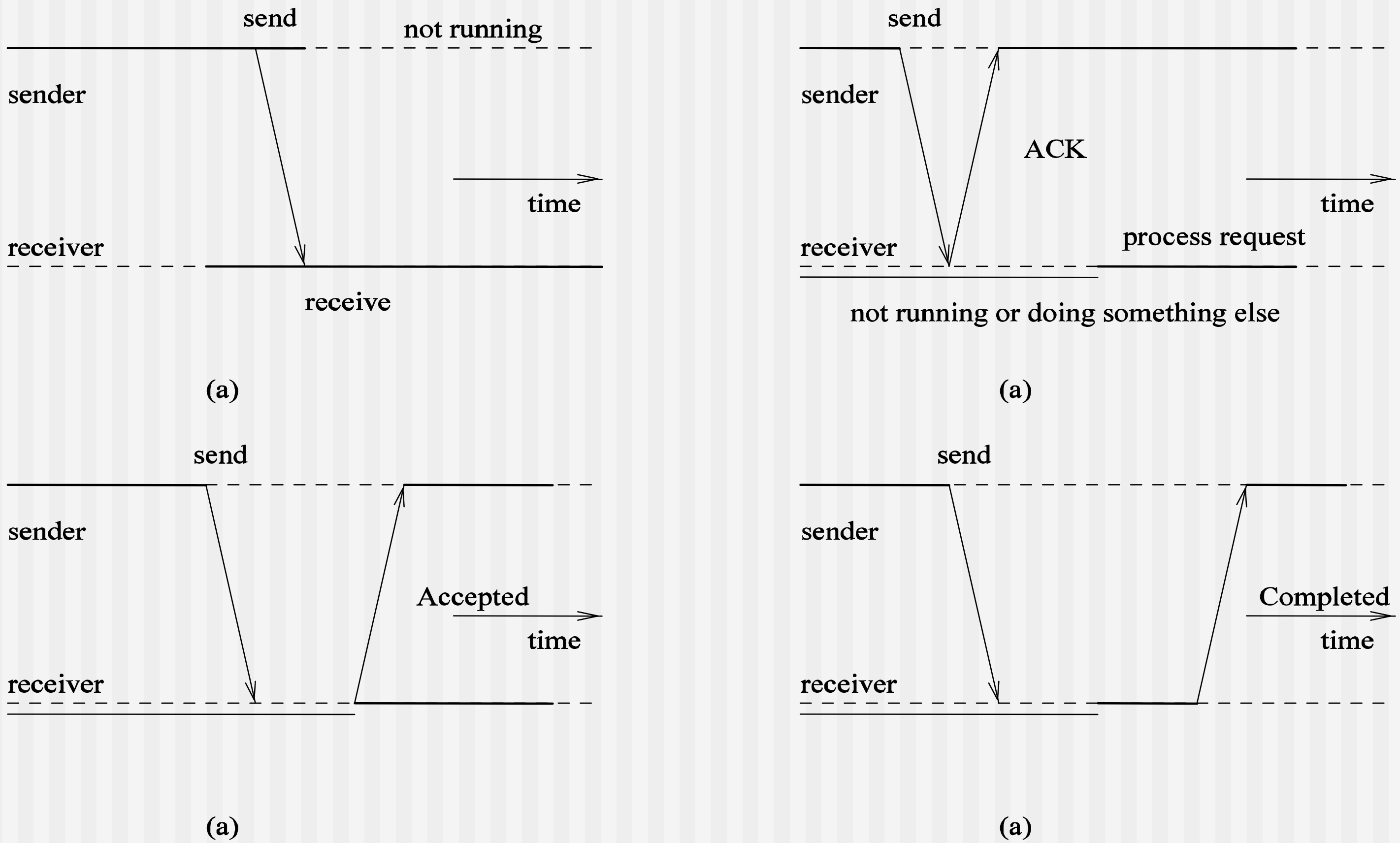
- f(i)::=

```
receive n from f(i - 1);
[ n > 1 → [ send n - 1 to f(i + 1);
            receive p from f(i + 1);
            receive q from f(i + 1);
            send p + q to f(i - 1);
            send p to f(i - 1)
            ]
  □ n = 1 → [ send 1 to f(i - 1);
              send 0 to f(i - 1)
            ]
]
```

Focus 9: Message-Passing Primitives of MPI

- MPI_Isend: asynchronous communication
- MPI_send: receipt-based synchronous communication
- MPI_ssend: delivery-based synchronous communication
- MPI_sendrecv: response-based synchronous communication

Focus 9 (Cont'd.)



Message-passing primitives of MPI: Isend, send, ssend, sendrecv.

Focus 10: Interprocess Communication in UNIX

- Socket: int socket (int domain, int type, int protocol).
  - domain: normally internet.
  - type: datagram or stream.
  - protocol: TCP (Transport Control Protocol) or UDP (User Datagram Protocol)
- Socket address: an Internet address and a local port number.

Focus 10 (Cont'd.)

Sender

```
s= socket(AF_INET, SOCK_DGRAM, 0)
. . .
bind(s, ClientSocketAddress)
. . .
sendto(s, "message", ServerSocketAddress)
```

Receiver

```
t= socket(AF_INET, SOCK_DGRAM, 0)
. . .
bind(t, ServerSocketAddress)
. . .
amount = recvfrom(t, buffer, from)
```

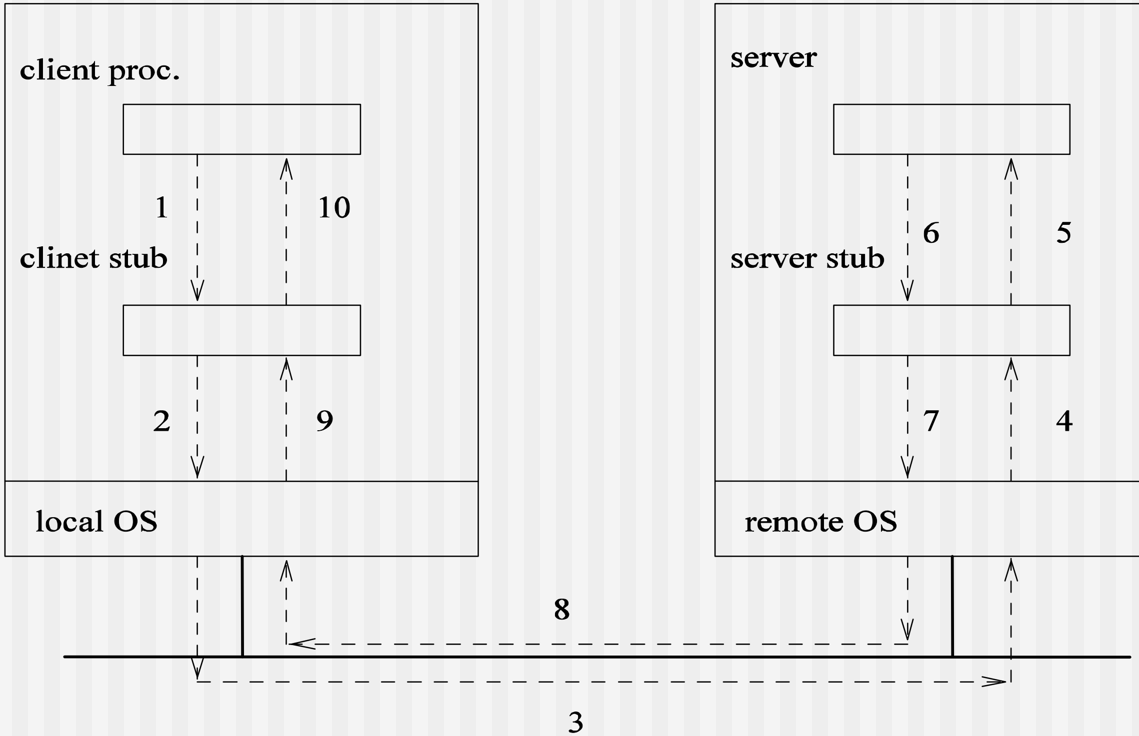
Sockets used for datagrams

High-Level (Middleware) Communication Services

- Achieve access transparency in distributed systems
  - Remote procedure call (RPC)
  - Remote method invocation (RMI)

Remote Procedure Call (RPC)

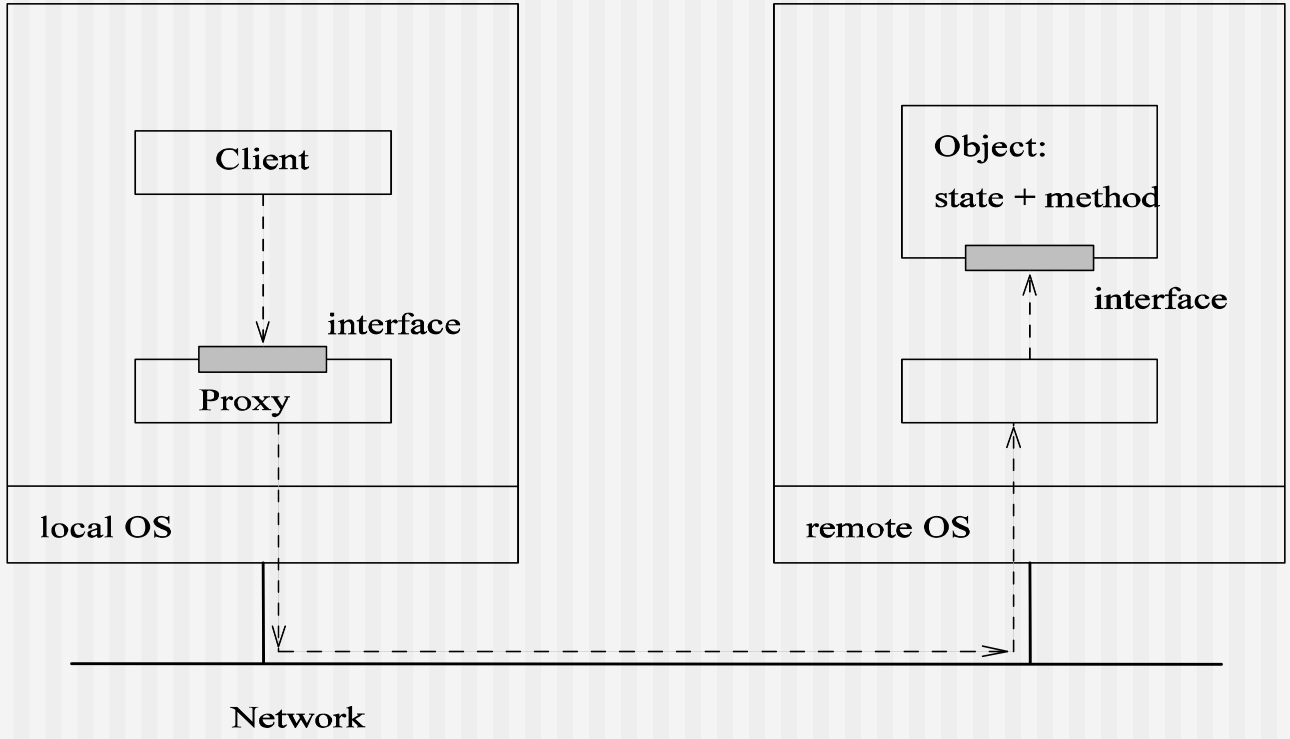
- Allow programs to call procedures located on other machines.
- Traditional (synchronous) RPC and asynchronous RPC.



RPC.

Remove Method Invocation (RMI)

RMI is a generalization of RPC in an object-oriented system



RMI.

Robustness

- Exception handling in high level languages (Ada and PL/1)
- Four Types of Communication Faults
  - A message transmitted from a node does not reach its intended destinations
  - Messages are not received in the same order as they were sent
  - A message gets corrupted during its transmission
  - A message gets replicated during its transmission

Failures in RPC

If a **remote procedure call** terminates abnormally (the time out expires) there are four possibilities.

- The receiver did not receive the call message.
- The reply message did not reach the sender.
- The receiver crashed during the call execution and either has remained crashed or is not resuming the execution after crash recovery.
- The receiver is still executing the call, in which case the execution could interfere with subsequent activities of the client.

Exercise 3

1. (The Welfare Crook by W. Feijen) Suppose we have three long magnetic tapes each containing a list of names in alphabetical order. The first list contains the names of people working at IBM Yorktown, the second the names of students at Columbia University and the third the names of all people on welfare in New York City. All three lists are endless so no upper bounds are given. It is known that at least one person is on all three lists. Write a program to locate the first such person (the one with the alphabetically smallest name). Your solution should use three processes, one for each tape.

Exercise 3 (Cont'd.)

2.Convert the following DCDL expression to a precedence graph.

$$[S_1 \| [[S_2 \| S_3]; S_4]]$$

Use **fork** and **join** to express this expression.

3.Convert the following program to a precedence graph:

$$S_1;[[S_2;S_3\|S_4;S_5\|S_6]\|S_7];S_8$$

Exercise 3 (Cont'd.)

4. G is a sequence of integers defined by the recurrence $G_i = G_{i\text{-}1} + G_{i\text{-}3}$ for $i > 1$, with initial values $G_0 = 0$, $G_1 = 1$, and $G_2 = 1$. Provide a DCDL implementation of G_i and use one process for each G_i.

5. Using DCDL to write a program that replaces $a{*}b$ by $a \uparrow b$ and $a{*}{*}b$ by $a \downarrow b$, where a and b are any characters other than *. For example, if $a_1a_2{*}a_3{*}{*}a_4{*}{*}{*}a_5$ is the input string then $a_1a_2\uparrow a_3\downarrow a_4{*}{*}{*}a_5$ will be the output string.