Unit-2

CONCURRENT PROCESSES AND PROGRAMMING

2.1 Processes and threads

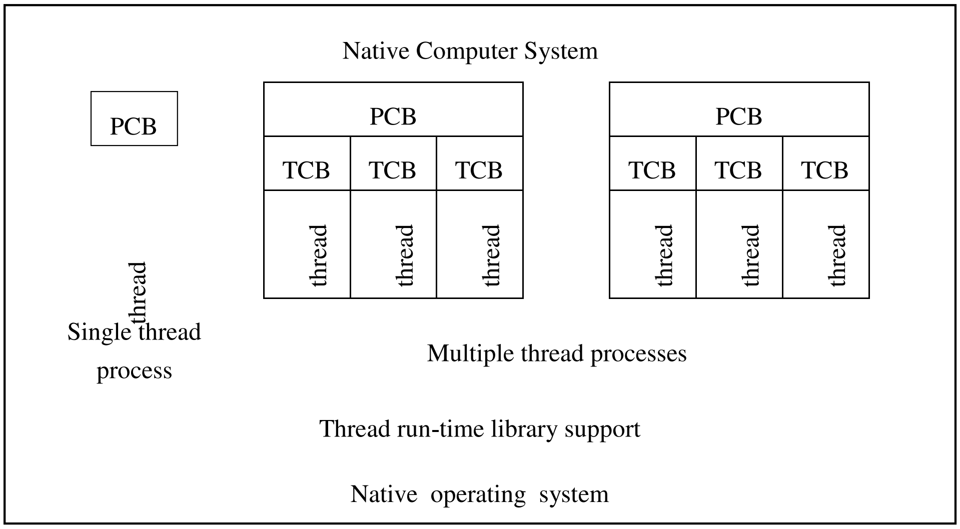
2.1.1 Introduction

Processes: separate logical address space
Threads: common logical address space

Major Issues

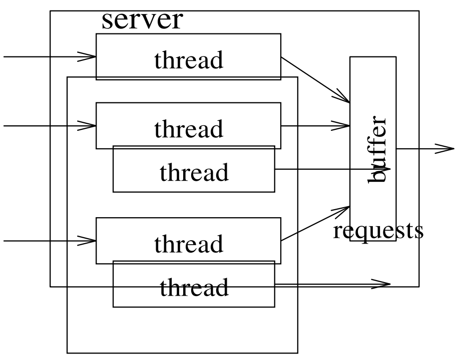
Process/thread creation
Light weight context switching
Blocking and scheduling

Two-level concurrency of processes and threads



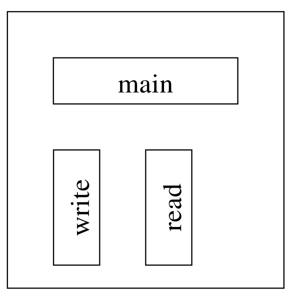
2.1.2 Thread applications

(a) Terminal server



Identical static threads dispatcher

(b) File server



Dynamic threads with

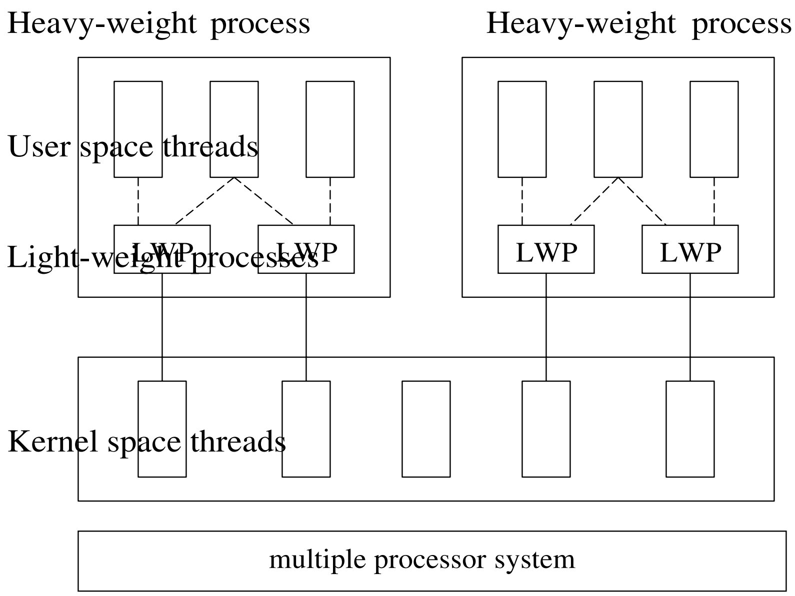
(c) Client

Concurrent and asynchronous requests

Thread implementations

- *User space*: simple but non-preemptable
- *Kernel space*: efficient but not portable

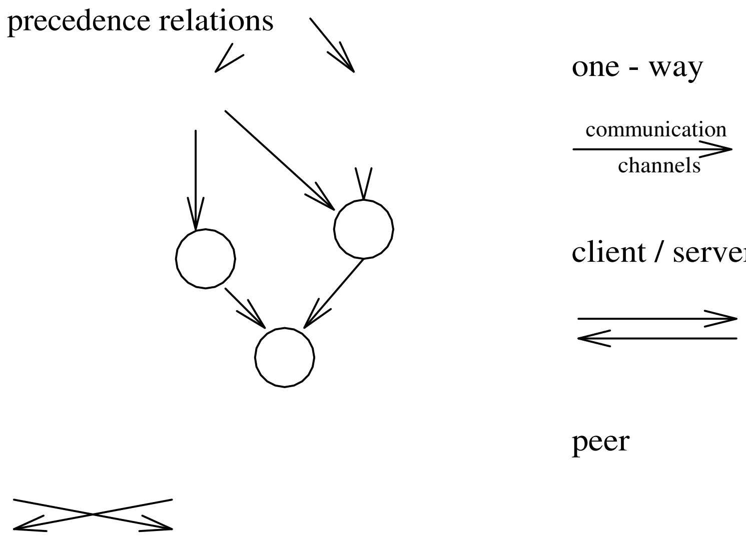
Solaris thread implementation



2.2 Process models

2.2.1 Synchronous Process, Asynchronous Communication, Time-Space

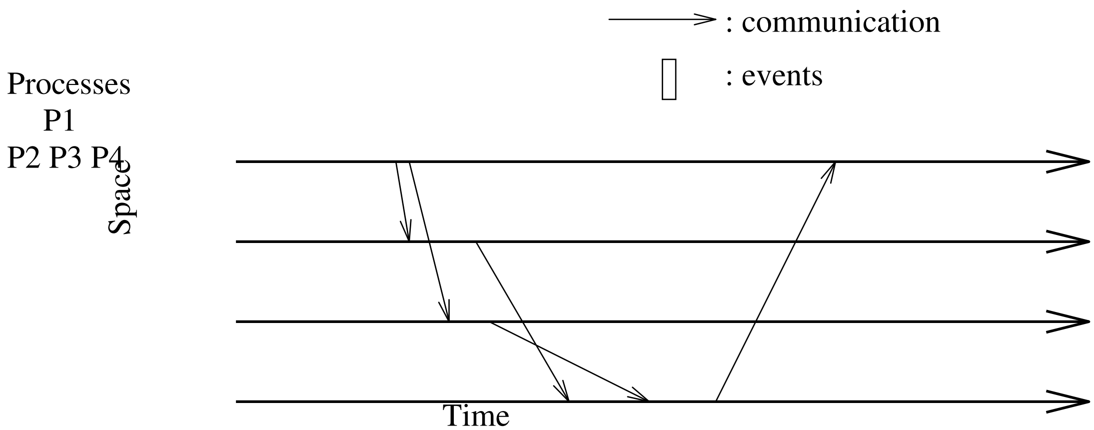
Graph representations



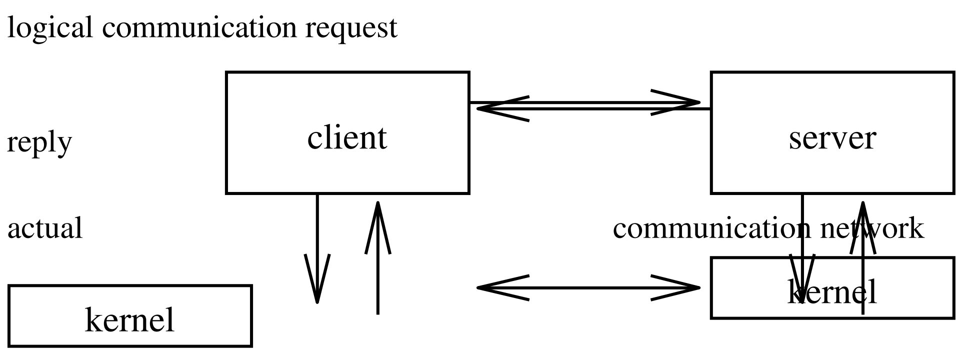
Synchronous process graph

Asynchronous process graph and communication scenarios

Time-space model



2.3 Client/server model



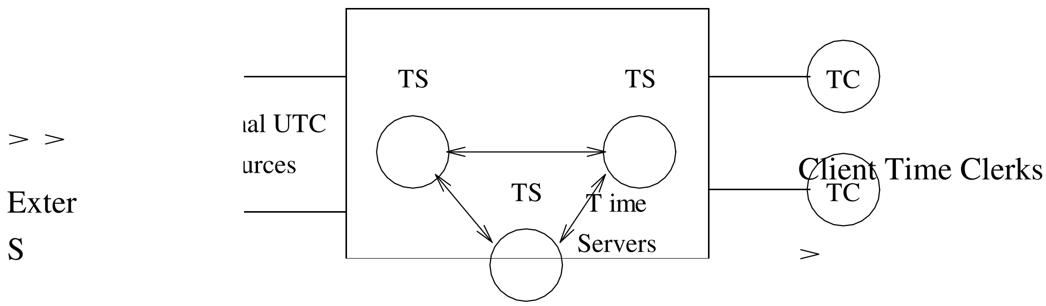
2.3.1 Time services

- time and timer
- physical and logical clocks

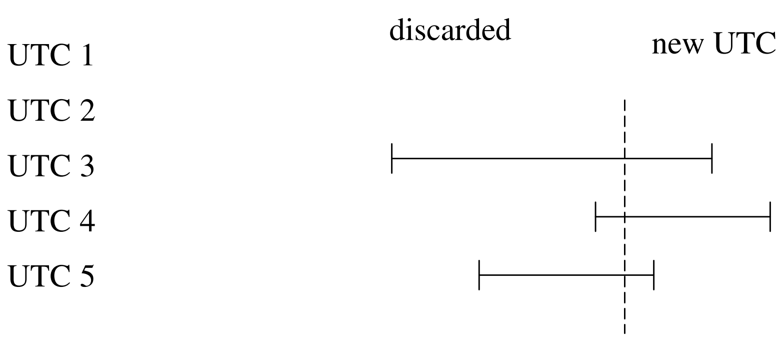
Physical clock

A distributed time service architecture

Distributed Time Service



Time Discrepancies



Lamport Logical Clock

The *happens-before* relationship: $\rightarrow$

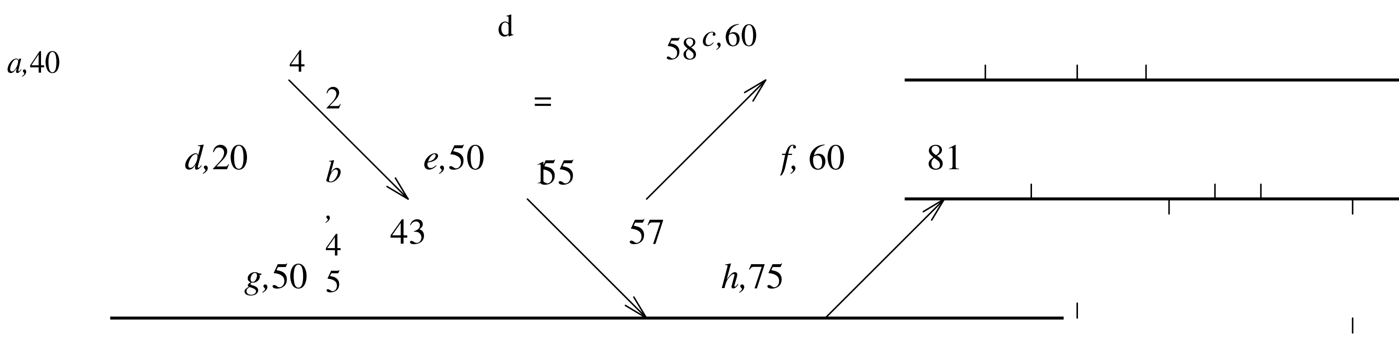
1. If $a \rightarrow b$ within a same process then $C(a) < C(b)$.

2. If a is the sending event of P_i and
b is the corresponding receiving event of P_j, then $a \rightarrow b$ and
$C_i(a) < C_j(b)$.

For it to be possible for a to have an influence on b, then $a \rightarrow b$ must be true.

Implementation:

$C(b) = C(a) + d$ and
$C_j(b) = max(TS_a + d, C_j(b))$,
where TS_a is the timestamp of the sending event and d is a positive number.



56 80

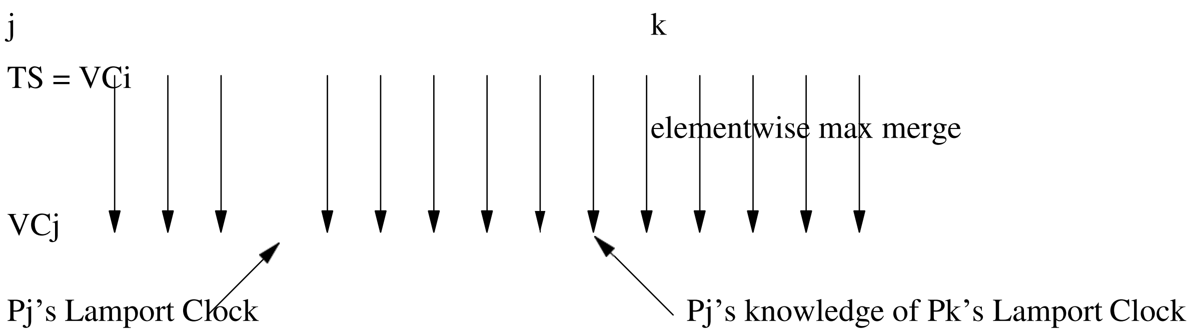
So, $a \rightarrow b \implies C(a) < C(b)$, but $C(a) < C(b) \not\Rightarrow$ $a \rightarrow b$.

Vector Logical Clock

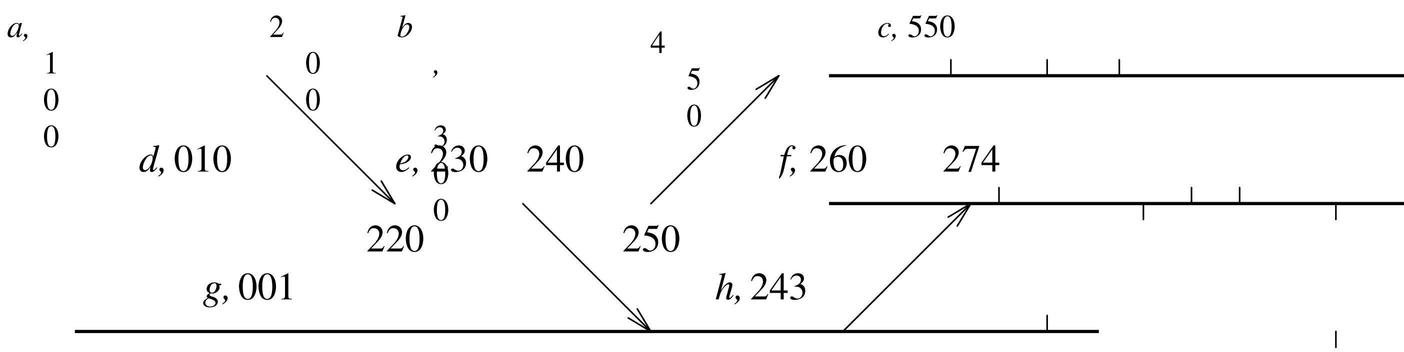
Used so that if $C_i(a) < C_j(b)$ then $a \rightarrow b$. Define $VC_i = [TS_1, TS_2, ..., C_i, ..., TS_n]$, where n is the number of cooperating processes. On message receipt, use *pair-wise maximum*.

$VC_j[j] = VC_j[j] + d$

$VC_j[k] = \max(VC_j[k], TS_i[k]) : l = 1..n$



$VC_j[j]$ is P_j's count of events that have occured at P_j,
$VC_j[k]$ is P_j's knowledge of events that have occured at P_k.



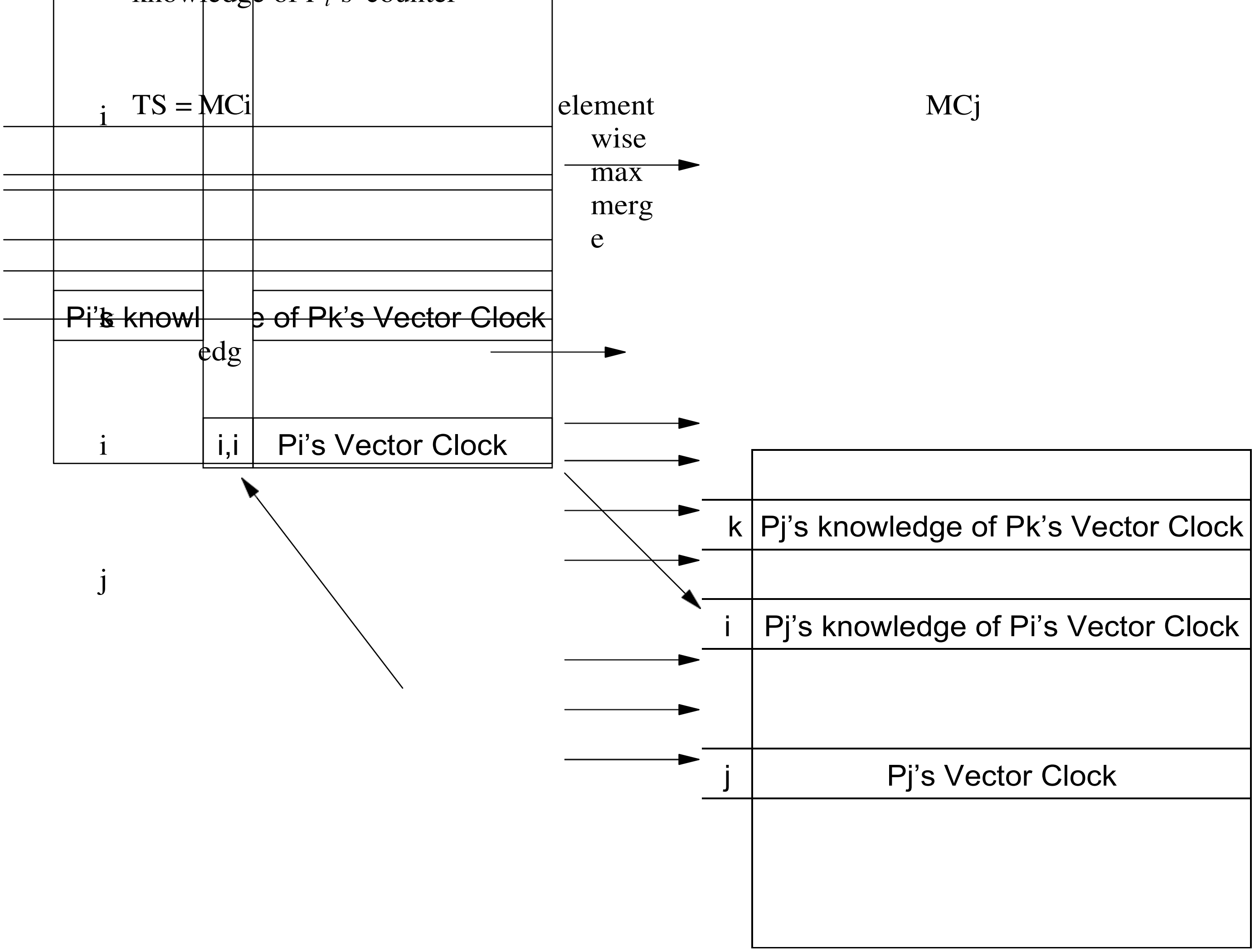
242 244

Matrix Logical Clock

MC_i represents
P_i's knowledge of its local events ($MC_i[i, i]$),
its knowledge of the events that P_j knows about ($MC_i[i, j]$), and its knowledge of P_j's knowledge of events at P_k ($MC_i[j, k]$).
$MC_i[i, i] = MC_i[i, i] + d$ – P_i updates local event counter on send

When P_j receives a message from P_i with timestamp TS, $MC_j[j, l] = \max(MC_j[j, l], TS_i[i, l]) : l = 1..n$ update vector clock, and

$MC_j[k, l] = \max(MC_j[k, l], TS_i[k, l])$: $k = 1..n$, $l = 1..n$ update P_k's knowledge of P_l's counter



Pi's knowledge of Pj's knowledge of Pi's Lamport Clock

2.4 Language constructs for synchronization

Concurrent languages

- Specification of concurrent activities
- Synchronization of processes
- Interprocess communication
- Nonderterministic execution of processes

Language constructs

- Program structure
- Data structure
- Control structure
- Procedure and system call
- Input and output
- Assignment

Synchronization mechanisms and language facilities

Synchronization Methods	*Language Facilities*
Shared-Variable Synchronization	
semaphore	shared variable and system call
monitor	data type abstraction
conditional critical region	control structure
serializer	data type and control structure
path expression	data type and program structure
Message Passing Synchronization	
communicating sequential processes	input and output
remote procedure call	procedure call
rendezvous	procedure call and communication

Shared-variable synchronization

- *Semaphore* and *conditional critical region*
- *Monitor* and *serializer*
- *Path expression*

Classic Problems

- Critical Section
- Dining Philosophers
- Readers/Writers
- Producer-Consumer

Example: the Reader/Writer Problems synchronization + concurrency

Basics

- if DB empty, allow anyone in
- if reader in DB, writer not allowed in
- if writer in DB, nobody allowed in

Lock Requested	Lock Held	
	Read Lock	Write Lock
Read Lock	✓	✗
Write Lock	✗	✗

Variations

- *reader preference*

Allow a reader in if other readers are in

- *strong reader preference*

Allow readers in when writer leaves

- *weak reader preference*

When writer leaves, select a process at random

- *weaker reader preference*

Allow a writer in when writer leaves

- *writer preference*

Do not allow readers in if writer is waiting

Semaphore solution to the weak reader preference problem

var mutex=1, db=1: semaphore; rc=0: integer

reader processes	writer processes
do(forever)	do (forever)
BEGIN	BEGIN
otherStuff()	otherStuff()
P(mutex)	
rc := rc + 1	
if rc = 1 then P(db)	P(db) V(mutex)
read database	write database
P(mutex) rc := rc-1	
if rc = 0 then V(db)	V(db) V(mutex)
END	END

Monitor solution

```
rw : monitor
var rc : integer; busy : boolean; toread, towrite : condition;

procedure startread
begin
if busy then toread.wait;
rc := rc + 1;
toread.signal;
end

procedure endread
begin

rc := rc - 1;
if rc = 0 then towrite.signal;
end

procedure startwrite
begin
if busy or rc f= 0
then towrite.wait;
busy := true;
end

procedure endwrite
begin

busy := false;
toread.signal or towrite.signal;
end

begin rc := 0; busy := false end
```

---

reader processes

```
do (forever) BEGIN

otherStuff()

rw.startread
read database
rw.endread

END
```

writer processes

```
do (forever) BEGIN

otherStuff()

rw.startwrite
write database
rw.endwrite

END
```

CCR solution

var db: shared; rc: integer;

readerprocesses

region db begin rc := rc + 1 end;
begin write dat abase end region db begin rc := rc - 1 end;

writer processes

region db when rc = 0 readdatabase

Serializer solution

rw : serializer
var readq, writeq: queue; rcrowd, wcrowd: crowd;

procedure read
begin
enqueue(readq) until empty(wcrowd); joincrowd(rcrowd) then begin read database
end; end

procedure write
begin
enqueue(writeq) until (empty(wcrowd) and empty(rcrowd));
joincrowd(wcrowd) then begin write database end; end

Path Expression solution

path 1:([read],write) end

Message Passing Synchronization

- *Asynchronous*: non-blocking send, blocking receive
- *Synchronous*: blocking send, blocking receive

Mutual exclusion using asyn. msg. passing

process P_i	channel server	process P_j
begin	begin	begin
receive(channel)	create channel	receive(channel)
critical section	send(channel)	critical section
send(channel)	manage channel	send(channel)
end	end	end

Mutual exclusion using syn. msg. passing

process P_i	semaphore server	process P_j
begin	loop	begin
send(sem,msg)	receive(pid,msg)	send(sem,msg)
critical section	send(pid,msg)	critical section
receive(sem,msg)	end	receive(sem,msg)
end		end

Communicating Sequential Processes (CSP)

P: $Q!exp$, Q: $P?var$, and *guarded* commands Process P executes $Q!(x+y)$,

. then expression $x + y$ is evaluated and sent to process Q. Process Q executes P $?z$,

. then process Q sets variable z to the value received from process P

ADA rendezvous task rw is
entry startread; entry endread; entry startwrite; entry endwrite;
end

task body rw is
rc: integer := 0;

```
busy: boolean := false;
begin loop
select
when busy = false →
accept startread do rc := rc + 1 end;
or
                                 →
                                 accept endread do rc := rc - 1 end;
or or
end                              when rc = 0 and busy = false →
   loo                           accept startwrite do busy = true end;
   p
   en                            →
   d;                            accept endwrite do busy = false end;
```

2.5 Concurrent Programming Languages

A taxonomy

2.5.1 Coordination languages

- *OCCAM*: based on CSP process model, use PAR, ALT, and SEQ con- structors, use explict global links for communication.

- *SR*: based on resource (object) model, use synchronous CALL and asyn- chronous SEND and rendezvous IN, use *capability* for channel naming.

- *LINDA*: based on distributed data structure model, use tuples to repre- sent both process and object, use blocking IN and RD and non-blocking OUT for communication.

	System	Object model	Channel naming
OCCAM	concurrent programming language	processes	static global channels
SR	concurrent programming language	resources	dynamic capabilities
LINDA	concurrent programming paradigm	distributed data structures	associative tags

2.5.2 Concurrent Programs

Processes and concurrent programs:

basic definitions A sequential program specifies sequential execution of a list of statements; its execution is called a process. A concurrent program specifies two or more sequential programs that may be executed concurrently as parallel processes. In many languages, process is also the name of the construct used to describe process behavior ; one notable exception is Ada, which uses the name task for this purpose.

Distinguishing concurrent, parallel, and distributed programs:

A concurrent program is commonly discussed in the same context as parallel or distributed programs. Unfortunately, few authors give precise meanings to these terms and the meanings that are offered tend to conflict. On balance, the following definitions seem appropriate:

- A concurrent program defines actions that may be performed simultaneously.
- A parallel program is a concurrent program that is designed for execution on parallel hardware.
- A distributed program is a parallel program designed for execution on a network of autonomous processors that do not share main memory

Distinguishing concurrent programs and concurrent systems:

A concurrent program is primarily a coherent unit of software. If two pieces of communicating software run concurrently, the result is a concurrent program when the two pieces form a conceptual whole; otherwise, the situation is viewed as two programs communicating through an agreed protocol. The communicating programs do, however, constitute a concurrent system (or parallel system or distributed system, as appropriate).

2.5.3 Problems in concurrent programs

Violating mutual exclusion:

Some operations in a concurrent program may fail to produce the desired effect if they are performed by two or more processes simultaneously. The code that implements such operations constitutes a critical region or critical section. If one process is in a critical region, all other processes must be excluded until the first process has finished. When constructing any concurrent program, it is essential for software developers to recognize where such mutual exclusion is needed and to control it accordingly

Most discussions of the need for mutual exclusion use the example of two processes attempting to execute a statement of the form: x := x + 1

Assuming that x has the value 12 initially, the implementation of the statement may result in each process taking a local copy of this value, adding one to it and both returning 13 to x (unlucky!). Mutual exclusion for individual memory references is usually implemented in hardware. Thus, if two processes attempt to write the values 3 and 4, respectively, to the same memory location, one access will always exclude the other in time leaving a value of 3 or 4 and not any other bit pattern.

Deadlock:

A process is said to be in a state of deadlock if it is waiting for an event that will not occur. Deadlock usually involves several processes and may lead to the termination of the program. A deadlock can occur when processes communicate (e.g., two processes attempt to send messages to each other simultaneously and synchronously) but is a problem more frequently associated with resource management. In this context there are four necessary conditions for a deadlock to exist .

1. Processes must claim exclusive access to resources.
2. Processes must hold some resources while waiting for others (i.e., acquire resources in a piecemeal fashion). 3. Resources may not be removed from waiting processes (no preemption).
4. A circular chain of processes exists in which each process holds one or more resources required by the next process in the chain.

Busy waiting

Regardless of the environment in which a concurrent program is executed, it is rarely acceptable for any of its processes to execute a loop awaiting a change of program state. This is known as busy waiting. The state variables involved constitute a spin lock. It is not in itself an error but it wastes processor power, which in turn may lead to the violation of a performance requirement. Ideally, the execution of the process concerned should be suspended and continued only when the condition for it to make progress is satisfied.

2.5.4 Properties of Concurrent Programs:

Safety

Safety properties assert what a program is allowed to do, or equivalently, what it may not do. Examples include:

- Mutual exclusion: no more than one process is ever present in a critical region.
- No deadlock: no process is ever delayed awaiting an event that cannot occur.
- Partial correctness: if a program terminates, the output is what is required.

Liveness

Liveness (or progress)properties assert what a program must do; they state what will happen (eventually) in a computation.

Examples include:

- Fairness (weak): a process that can execute will be executed.
- Reliable communication: a message sent by one process to another will be received.
- Total correctness: a program terminates and the output is what is required.

2.5.5 Executing Concurrent Programs

1. Measures of concurrency

Concurrent behavior can be measured in several ways. In practice, the measures are merely rough classifications of behavior that help characterize a program. These measures are given names here, for convenience, but there is no consensus on naming. In particular, references to the term grain or granularity of concurrency in the literature may mean any of the following measures:

- **The unit of concurrency** is the language component on which process behavior is defined. It may be an element in an expression; it may be a program statement; but most commonly it is a program block.
- **The level of concurrency** is the mean number of active processes present during the execution of a program.
- **The scale of concurrency** is the mean duration (or lifetime) of processes in the execution of a

program ; there is an overhead in initiating a concurrent activity, and so ideally its duration should be sufficiently long to make that overhead negligible.

- **The grain of concurrency** is the mean computation time between communications in the execution of a program ; this should be relatively large if a physical distribution of processes is required.

2. Execution environments

Programs involving large-scale concurrent behavior (comprising processes of relatively long duration) are executed most commonly on a single processor computer in which the processor is shared among the active processes. This is known as multiprogramming (or multitasking). Multiprocessing occurs on a multiprocessor, a computer in which several (usually identical) processors share a common primary memory.

Multicomputers use separate primary memory, and their execution of processes is known as distributed processing. Closely coupled multicomputers have fast and reliable point-to-point interprocessor links; loosely coupled systems communicate over a network that is much slower and much less reliable. Components of a multicomputer may be in the same vicinity or physically remote from each other.

In these are referred to as workstation-LANs (Local Area Networks) and workstation-WANs (Wide Area Networks), respectively.

Small-scale concurrent programs are usually executed by array or vector processor computers that apply the same operation to a number of data items at the same time. This is known as synchronous processing .

Dataflow and reduction machines apply different operations to different data items simultaneously . These latter machines are still largely experimental. A detailed presentation of the hardware available for parallel processing is given in . A collection of early papers on parallel processing may be found in .

3. Patterns of execution

Most commonly, a concurrent program starts as a single process and subdivides into multiple processes at some point in its execution. The spawned processes may be activated individually or in sets. The processes thus activated may be able to subdivide in the same way.

There are two main models of execution:

1. The spawned processes, when activated, execute independently of the process that triggers their execution.
2. The triggering process forks into multiple processes which, when complete, join to form a single process again.

Most programming languages support the fork-and-join model.

3. Process states

A process exists in one of three states (there is no agreement on the names used):

1. Awake, meaning that the process is able to execute.
2. Asleep (or blocked), meaning that the process is suspended awaiting a particular event (e.g., message arrival or resource available).
3. Terminated, meaning that the execution of the process has finished.

Processes that are awake can be further divided into those that are running (executing) and those that are ready to run as soon as a processor becomes available.

4. **Process scheduling**

In exceptional circumstances, a concurrent program may run directly on bare hardware.

More usually, however, it will execute on top of support software that provides a more abstract interface to that hardware. This is known as the system kernel or nucleus.

One component of the nucleus is the scheduler, which is responsible for the allocation of processors to processes, that is, the resolution of the mismatch between the number of processes that can execute and the number of processors available to execute them. In distributed systems, the scheduler itself may be distributed .

The processes in some concurrent programs are assigned explicitly to particular processors by the program designer. More commonly, however, the mapping is handled implicitly by the scheduler. Processes often execute with different priorities.

One objective of the scheduler is to ensure that all running processes have no lower a priority than those that are in a ready state. Priorities may be assigned explicitly by the program designer or be set and adjusted implicitly by the scheduler.

The compilation of a concurrent program results in the generation of calls to the kernel that may trigger scheduling operations. Any entry to the kernel provides an opportunity to suspend the process involved and select another for execution. In some cases, normal program behavior may result in an acceptably even distribution of processor power over the competing processes. However, when processing power is scarce it is desirable to implement some form of time slicing to ensure that all processes make steady progress. This is often implemented with the assistance of a system clock that interrupts at least one processor at regular intervals.

2.6 INTERPROCESS COMMUNICATION AND COORDINATION

Basic message passing communication

Communication primitives:

send(destination, message)

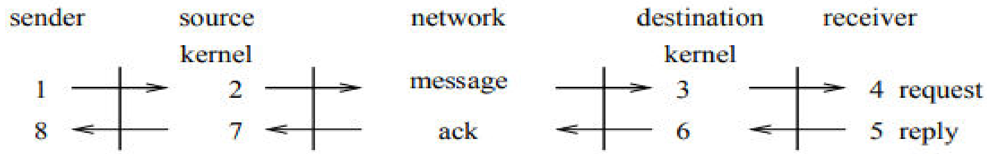
receive(source, message)

channel naming = process name, link, mailbox, port

- direct communication: symmetric/asymmetric process naming, link
- indirect communication: many-to-many mailbox, many-to-one port

2.6.1Message Passing

Message buffering and synchronization

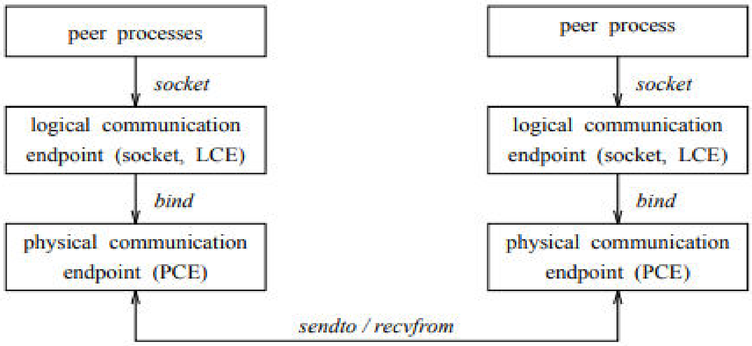


1. **Nonblocking send,** *1+8* : Sender process is released after message has been composed and copied into sender's kernel (local system call).
2. **Blocking send,** *1+2+7+8* : Sender process is released after message has been transmitted to the network (NIC interrupt OS).
3. **Reliable blocking send,** *1+2+3+6+7+8* : Sender process is released after message has been received by the receiver's kernel (kernel receives network ACK).
4. **Explicit blocking send,** *1+2+3+4+5+6+7+8* : Sender process is released after message has been received by the receiver process (kernel receives kernel delivery ACK).
5. **Request and reply,** *1-4, service, 5-8* : Sender process is released after message has been processed by the receiver and response returned to the sender.

Message passing API

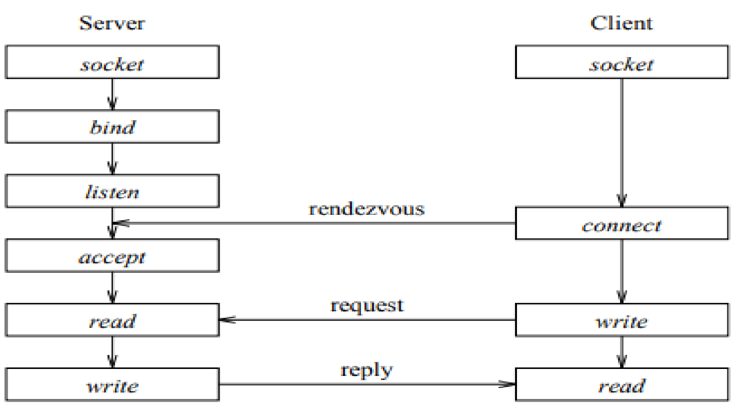
- **Pipe**: A FIFO byte-stream unidirectional link for related processes (set up at process invocation)
- **Message queue**: A structured variable length message queue
- **Named Pipe**: A special FIFO file pipe using path name for unrelated processes under the same domain (explicitly created and accessed)
- **Socket:** A logical communication endpoint for communication between autonomous domains (bound to physical communication endpoint)

Connectionless socket communication



- *peer process*: application level processes - application protocol
- *LCE*: Logical Communication Endpoint - established with socket call
- *PCE*: Physical Communication Endpoint - (Transport TSAP/L4SAP, Network NSAP/L3SAP) bound to LCE with bind call
- *Network*: Accessed by sendto/recvfrom primitives

Connection-oriented socket communication



2.6.2 Request/Reply and Transaction Communication

Asymmetric - Client and Server Server starts first:

- **Server process**: application level process

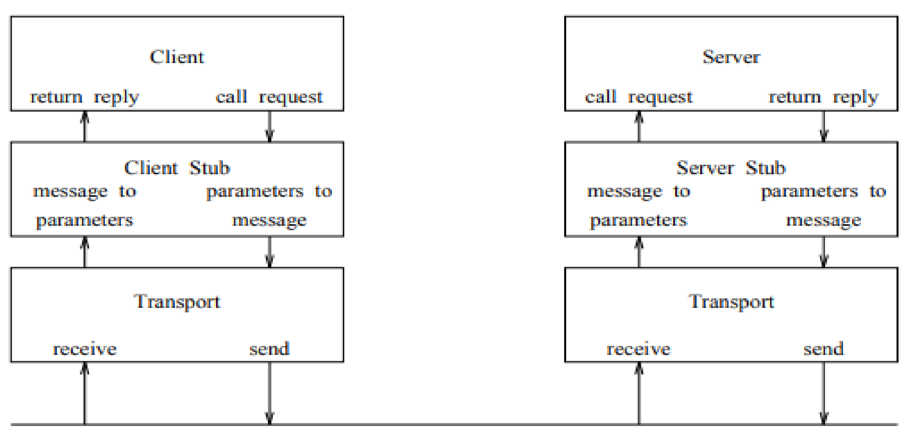
- server protocol

- **LCE**: Logical Communication Endpoint - established with socket call
- **PCE:** Physical Communication Endpoint - (Transport TSAP/L4SAP, Network NSAP/L3SAP) bound to LCE with bind call
- **Listen**: Server waits for incoming connection request
- **Accept**: Server accepts connection request, initializes connection
- **Read:** Server reads incoming segment(s) of request • Write: Server writes reply segment(s)
- **Close**: Server terminates connection when reply is received

Client starts after Server:

- **Client process**: application level process - runs server protocol
- **LCE**: Logical Communication Endpoint - established with socket call
- **PCE**: Physical Communication Endpoint - (Transport TSAP/L4SAP, Network NSAP/L3SAP) bound to LCE with connect call, which also initialized connection to server PCE
- **Write**: Client writes request segment(s)
- **Read**: Client reads incoming segment(s) of reply
- **Close**: Client terminates connection when reply is received and acknowledged

Remote Procedure Calls (RPCs)

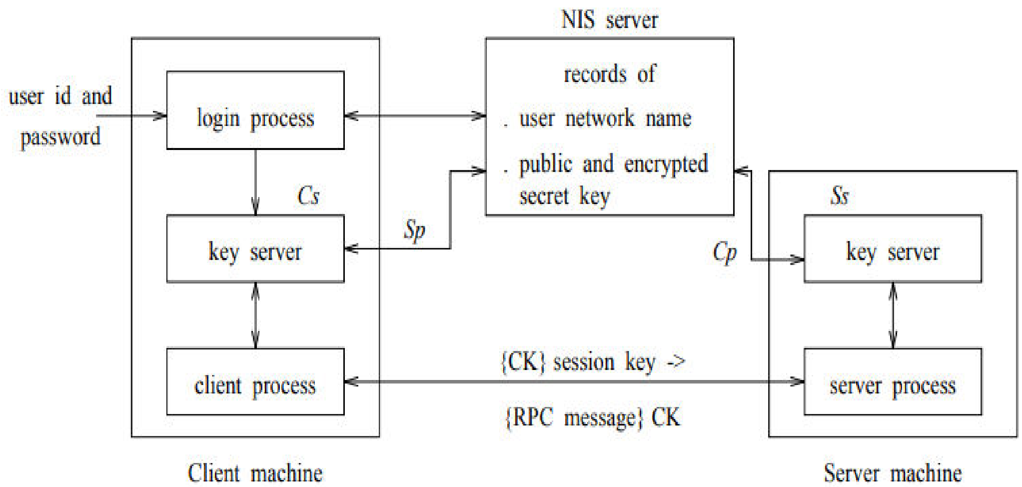


- *Parameter passing and data conversion*
- *Binding*
- *Compilation*
- *Exception and failure handling*
- *Security*

2.6.2 Name and Directory services

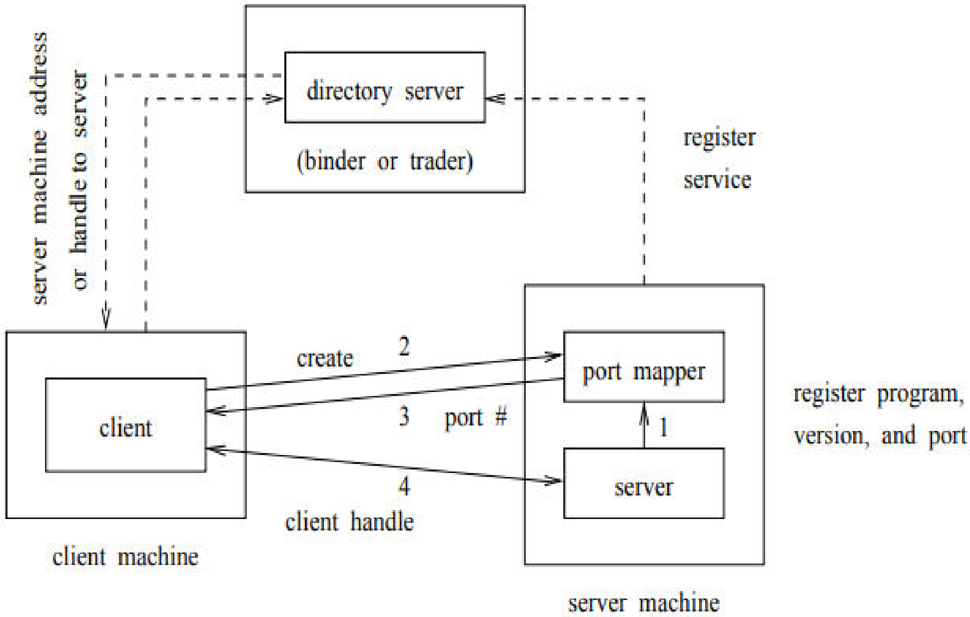
Name and Directory Services

Object attributes and name structures

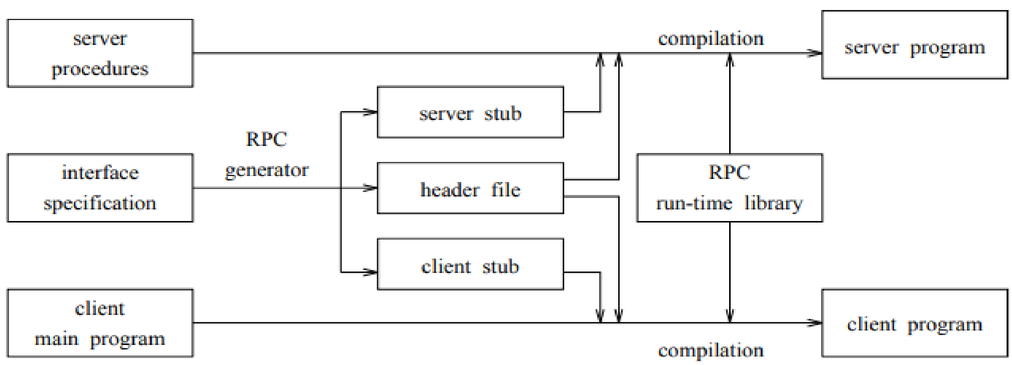


2.6.3 RPC and RMI case studies

RPC Binding


server machine address
or handle to server
directory server
(binder or trader)
register
service
create
2
port mapper
client
3
port #
1
register program,
version, and port
4
server
client handle
client machine
server machine

RPC compilation



RPC exception and failure

• **Exception**: in-band or out-band signaling

• **Link failure**: retransmission, sequence number and idempotent requests, use of transaction id xid

• **Server crash**: –

at least once: server raises an exception and client retries

– **at most once**: server raises an exception and client gives up

– **maybe**: server raises no exception and client retries

• Client crash:

– orphan killed by client

– orphan killed by server

– orphan killed by expiration

Secure RPC

- Cs and Ss are 128-bit random numbers.
- Cp = α Csmod M, and Sp = α Ssmod M, where α and M are known constants.

SKcs = S Cs p = (α Ss) Cs = α Ss∗Cs
SKsc = C Ss p = (α Cs) Ss = α Cs∗Ss

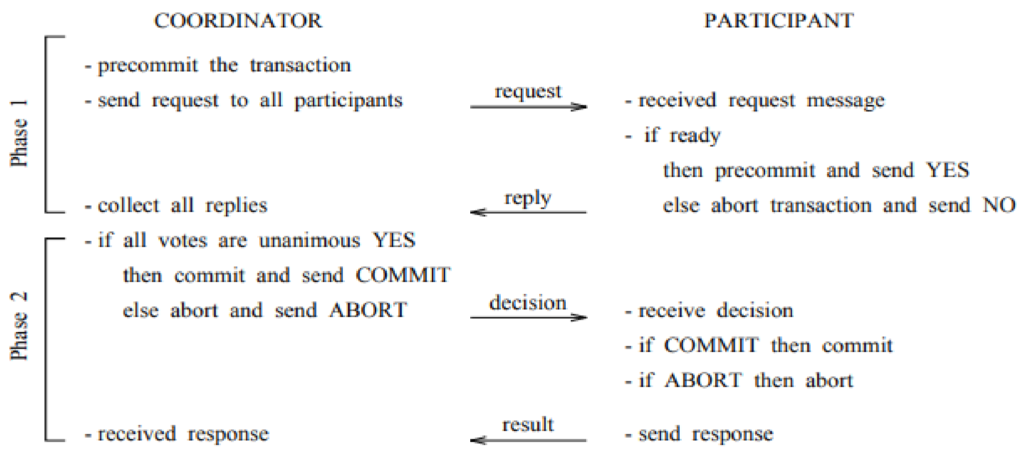
2.7 Transaction Communication ACID properties

- Atomicity

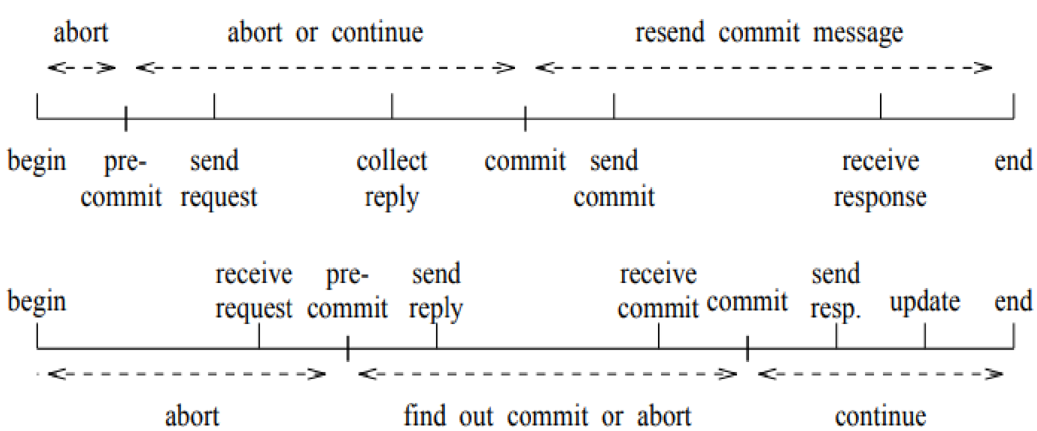
Consistency

- Isolation
- Durability

Two-phase commit protocol



Coordinator failure recovery actions

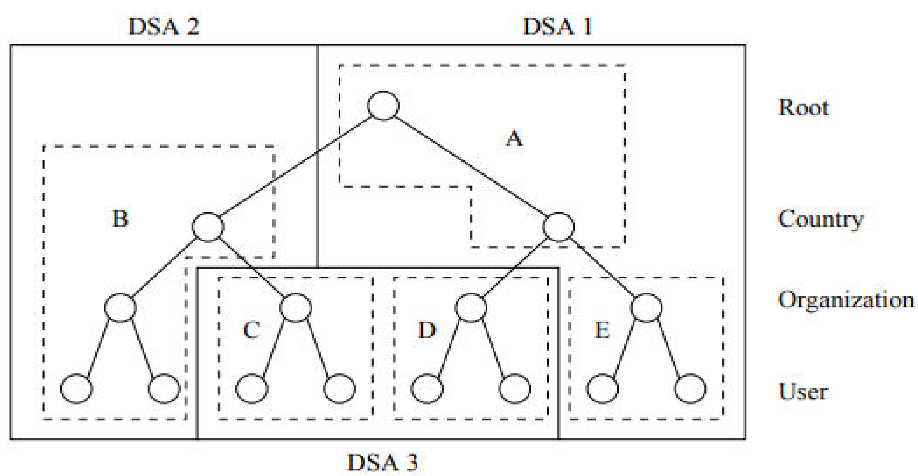


Participant failure recovery actions

Object attributes and name structures

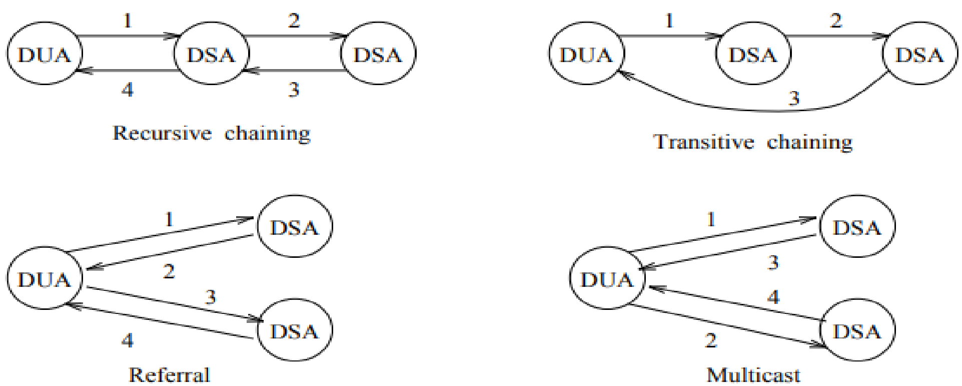
Service /object Attributes	Name Structures	Attribute Partitioning
< attributes > < name, attributes, address > < name, type, attributes, address >	flat structure hierarchical structure name-based resolution (e.g., white pages) structure-free attribute-based resolution (e.g., yellow pages)	physical organizational functional

Name space and information base



Five naming contexts of Directory Info Tree in three Directory Service Agents

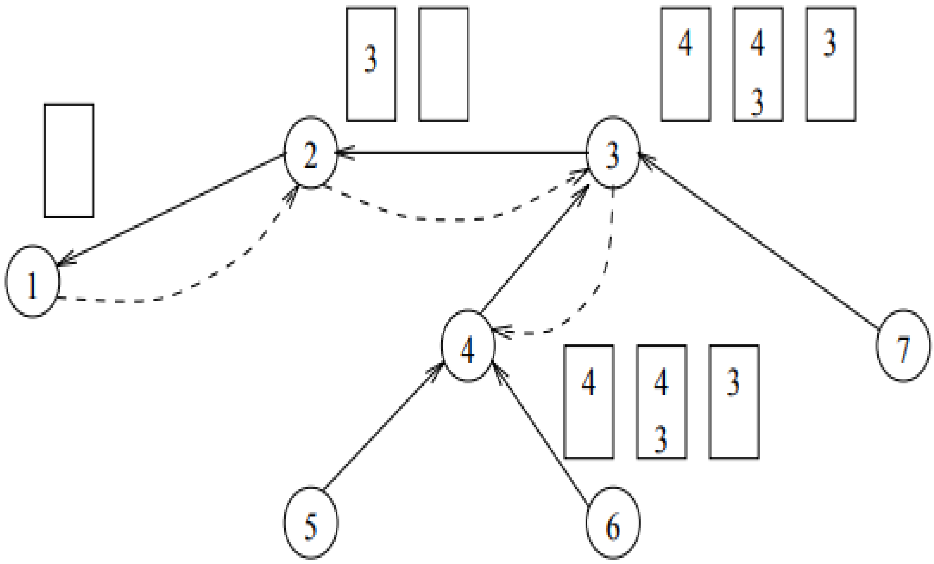
Name resolution



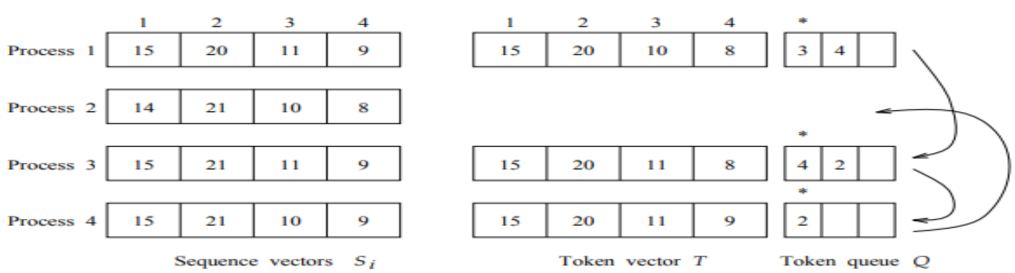
Distributed Mutual Exclusion

- Contention-based:
  - Timestamp prioritized
  - Voting
- Control (Token)-based:
  - Ring structure
  - Tree structure
  - Broadcast structure

Tree-structure token passing



Broadcast structure token passing



Leader Election Complete topology
Complete topology

The Bully algorithm

- Are-U-Up to higher numbered nodes
- If highest alive, Enter-Election to lower nodes
- When ACK or TRO for all lower nodes, send Result
- Enter-Election received: transient state until Result

Logic ring topology

```
The initiator node sets partcipating = true and
send (id) to its successor node;

For each process node ,

    receive (value);
    case
        value > id : participating := true,    send (value);
        value < id and participating == false : participating := true,    send (id);
        value == id : announce leader;
    end case
```