Distributed Systems

An operating system (OS) is basically a collection of software that manages computer hardware resources and provides common services for computer programs. Operating system is a crucial component of the system software in a computer system.

Click to add text

Distributed Operating System is one of the important type of operating system.

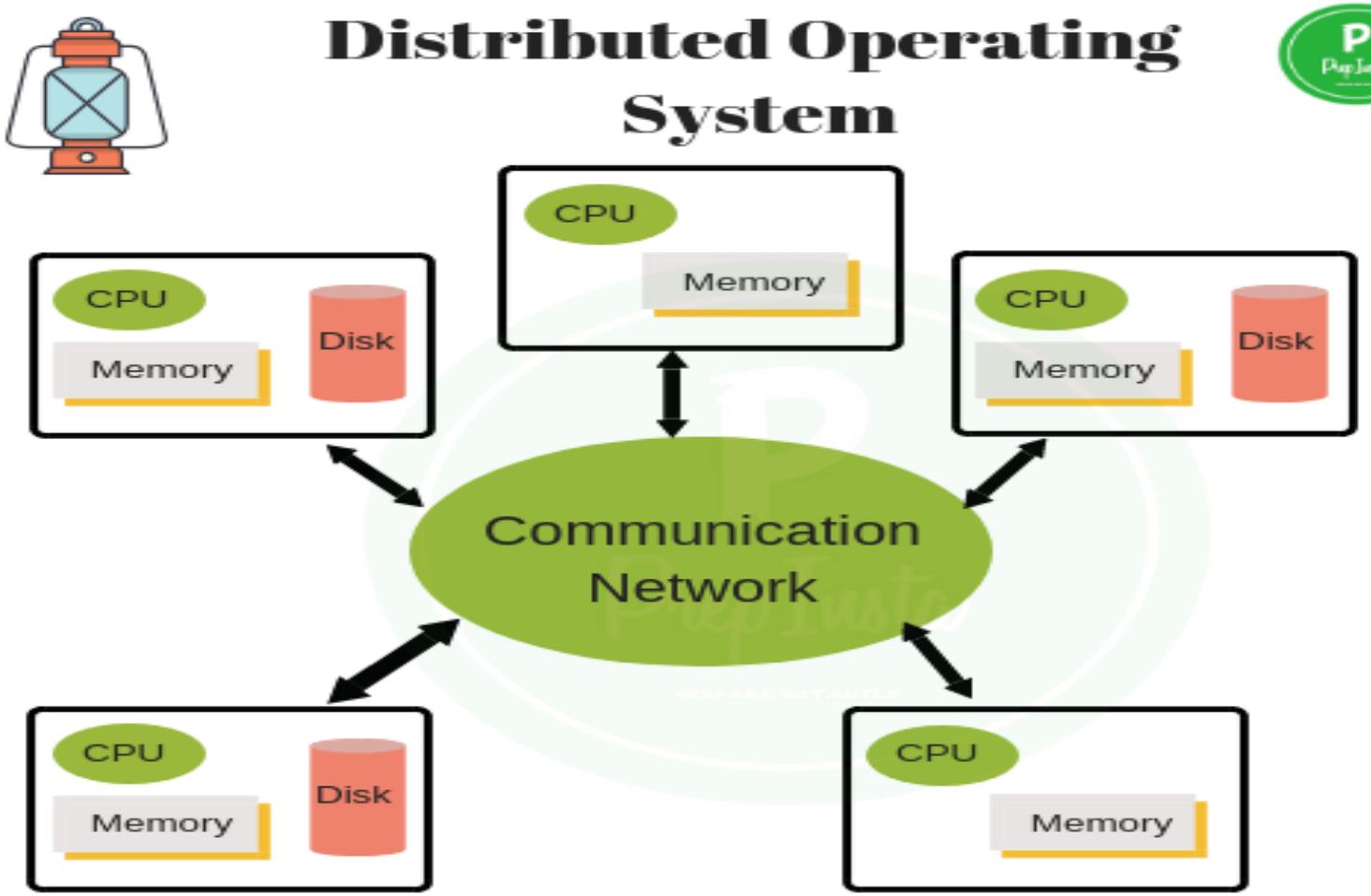
Definition:

Multiple central processors are used by Distributed systems to serve multiple real-time applications and multiple users. Accordingly, Data processing jobs are distributed among the processors.

Click to add text

Processors communicate with each other through various communication lines (like high-speed buses or telephone lines). These are known as loosely coupled systems or distributed systems. Processors in this system may vary in size and function. They are referred as sites, nodes, computers, and so on.

Architecture of Distributed OS:



Advantages of Distributed OS:

1] With resource sharing facility, a user at one site may be able to use the resources available at another.

2] Speedup the exchange of data with one another via electronic mail.

3] Failure of one site in a distributed system doesn't affect the others, the remaining sites can potentially continue operating.

Click to add text

4] Better service to the customers.

5] Reduction of the load on the host computer.

6]Reduction of delays in data processing.

The Rise of Distributed Systems

- Computer hardware prices are falling and power increasing.
- Network connectivity is increasing.
  - Everyone is connected with fat pipes.
- It is *easy* to connect hardware together.
- Definition: a *distributed system* is
  - A collection of independent computers that appears to its users as a single coherent system.

Forms of Transparency in a Distributed System

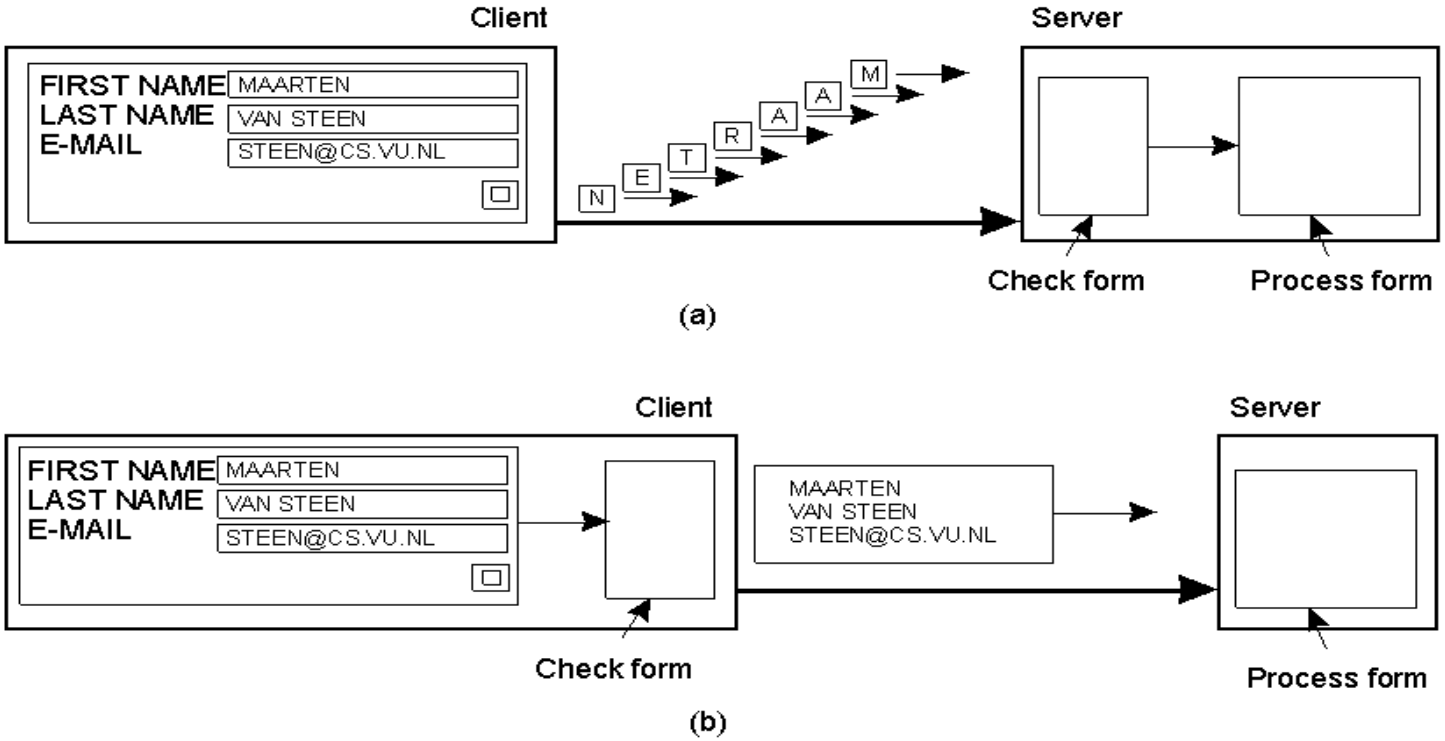
Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be shared by several competitive users
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

Scalability Problems

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

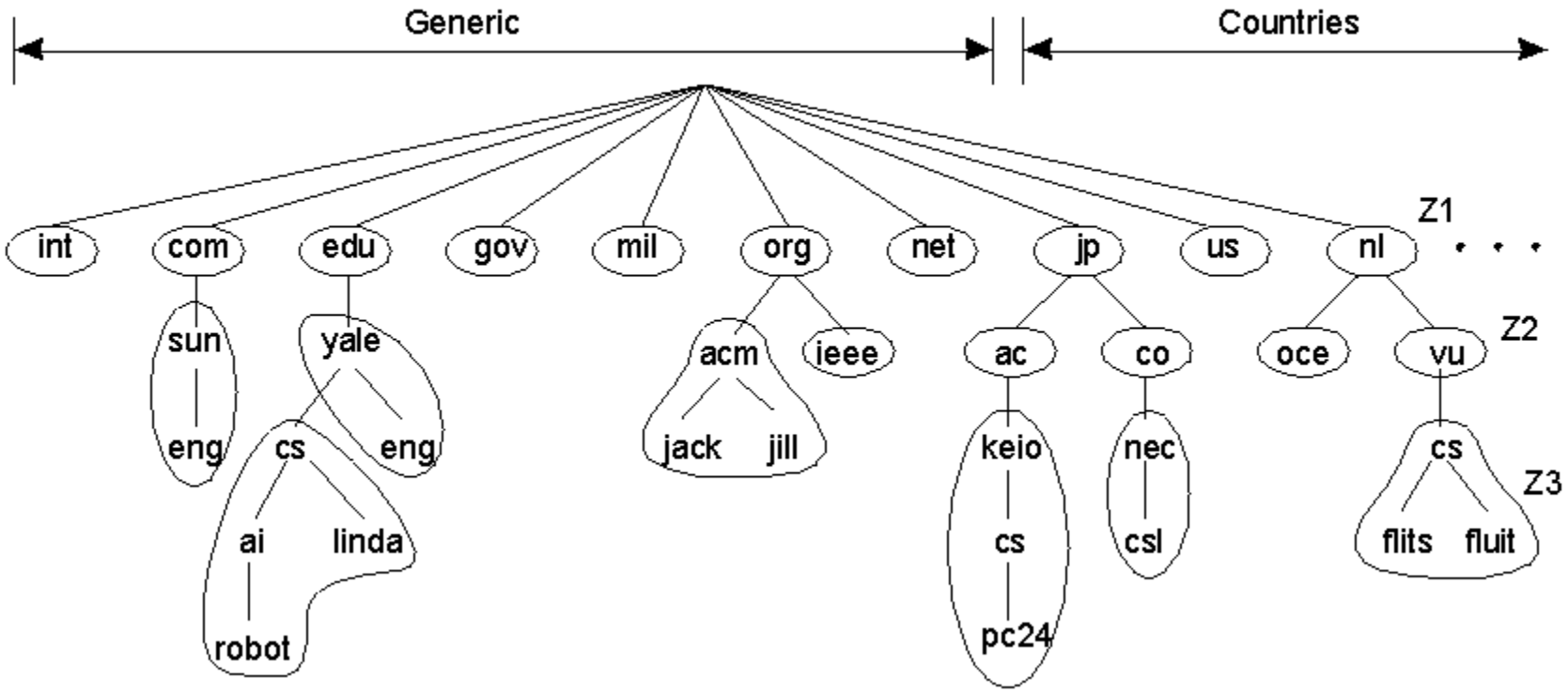
- As distributed systems grow, centralized solutions are limited.

Hiding Communication Latency



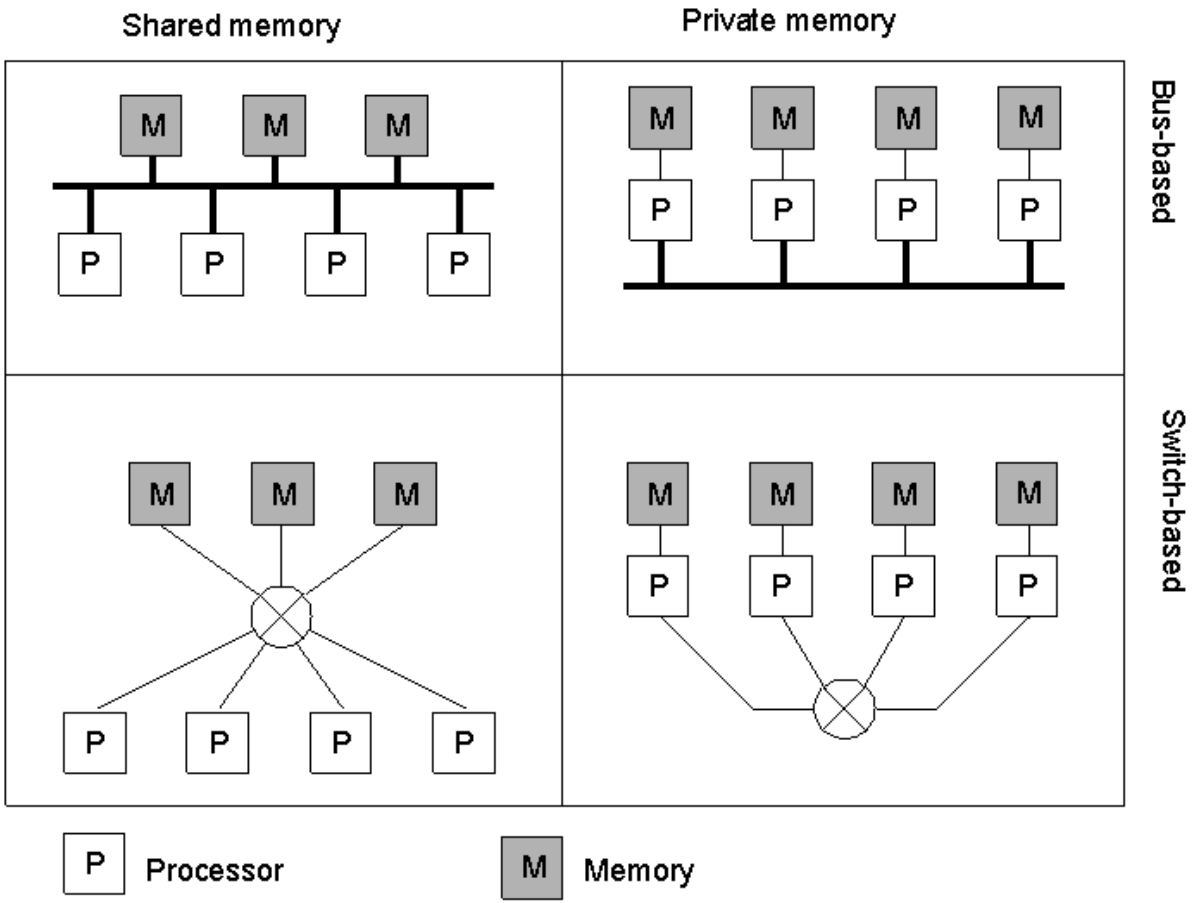
- This is especially important for interactive applications
- If possible, system can do *asynchronous communication*.
- The system can hide latencies.

Dividing the DNS name space into zones



Hardware Concepts

Basic organizations and memories in distributed computer systems



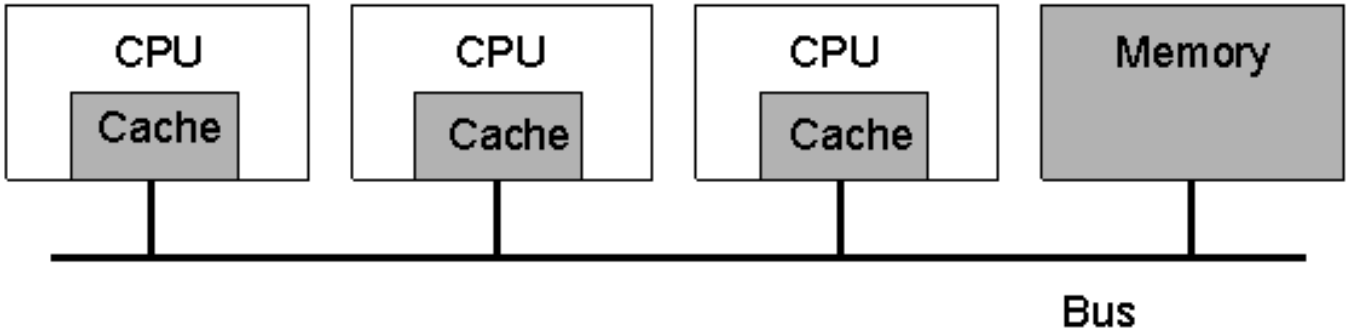
Hardware Considerations

- General Classification:
  - Multiprocessor – a single address space among the processors
  - Multicomputer – each machine has its own private memory.
- OS can be developed for either type of environment.

Multiprocessor Organizations

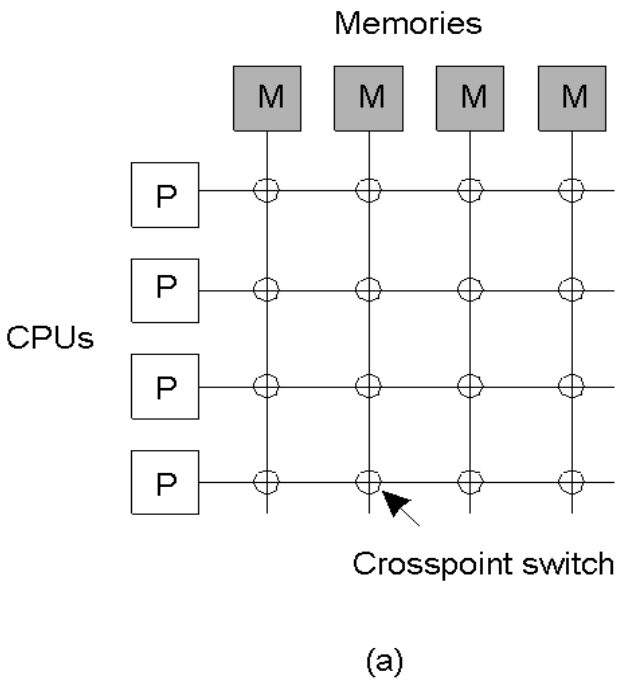
- ***Uniform Memory Access* [UMA]**
  - Caching is vital for reasonable performance (e.g., caches on a shared memory multiprocessor).
  - Want to maintain cache coherency
    - **Write-through cache ::** any changes to cache are written through to memory.

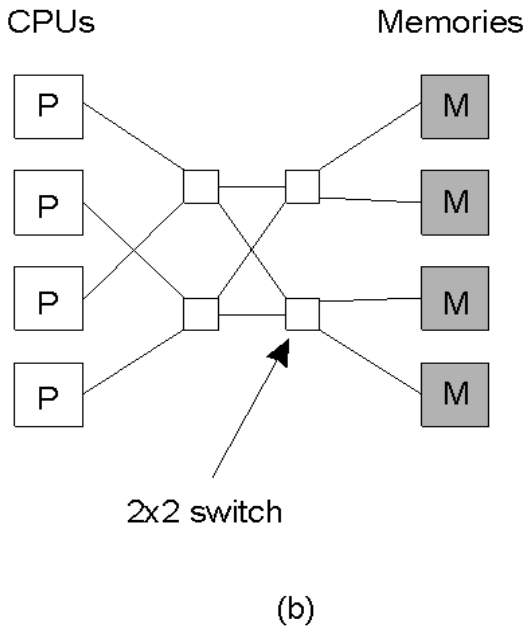
Multiprocessors



A bus-based multiprocessor.

Multiprocessors





A crossbar switch

An omega switching network

Multiprocessor Organizations

- ***Non-Uniform Memory Access*** **[NUMA]**
  - A hierarchy where CPUs have their own memory (not the same as a cache).
  - Access costs to memory is non-uniform.

Replication

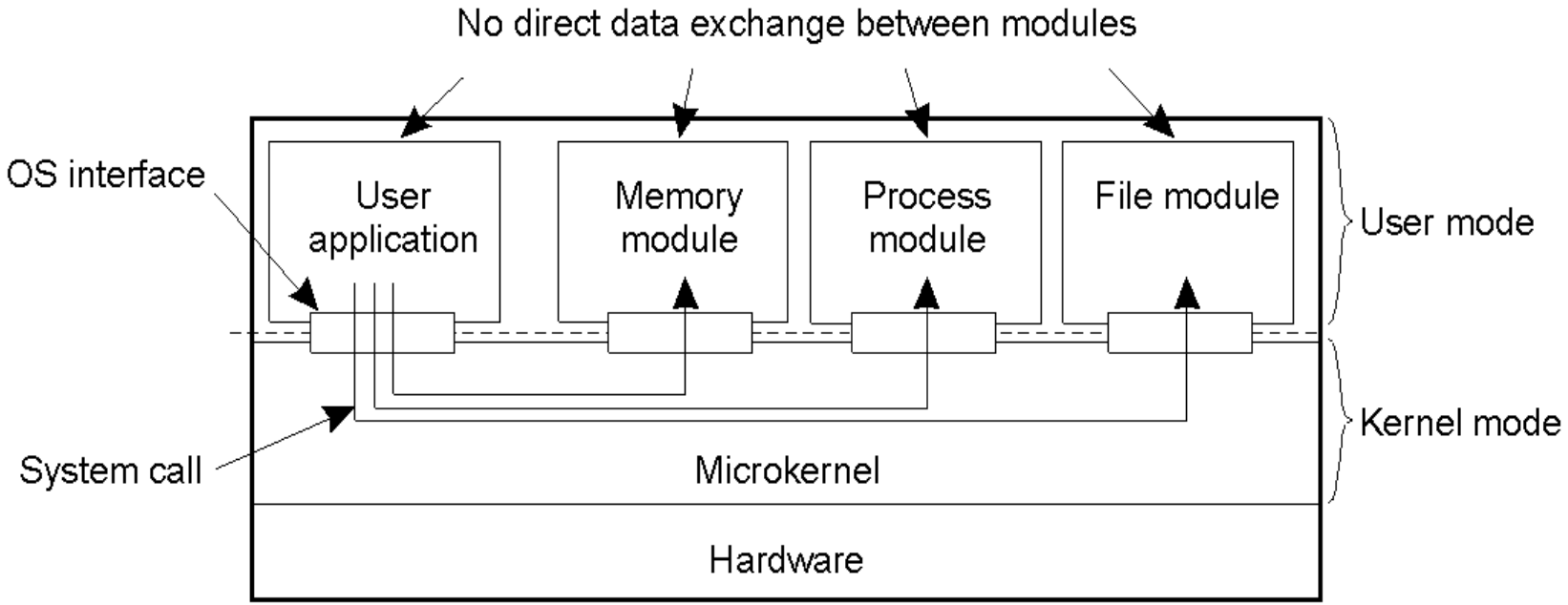
- Make a copy of information to increase availability and decrease centralized load.
  - Example: P2P networks (Gnutella +) distribute copies uniformly or in proportion to use.
  - Example: CDNs (Akamai)
  - Example: Caching is a replication decision made by client.
- Issue: Consistency of replicated information
  - Example: Web Browser cache

Software Concepts

- DOS (Distributed Operating Systems)
- NOS (Network Operating Systems)
- Middleware

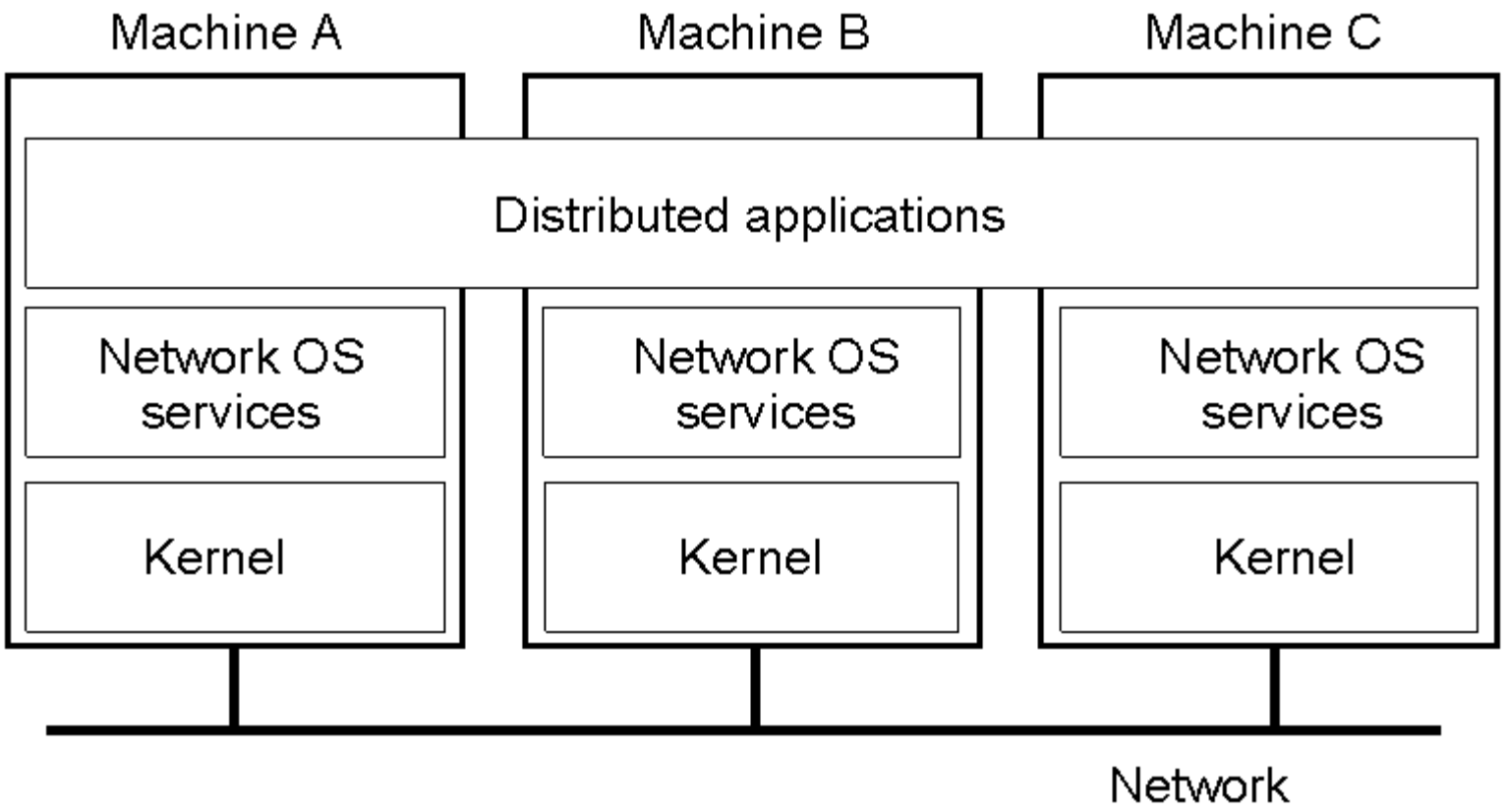
System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

Uniprocessor Operating Systems



- Separating applications from operating system code through a microkernel
  - Can extend to multiple computers

Network Operating System

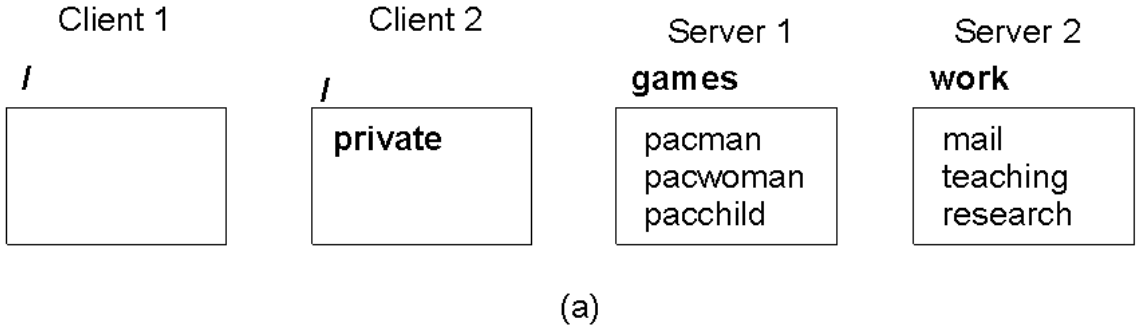


- OSes can be different (Windows or Linux)
- Typical services: rlogin, rcp
  - Fairly primitive way to share files

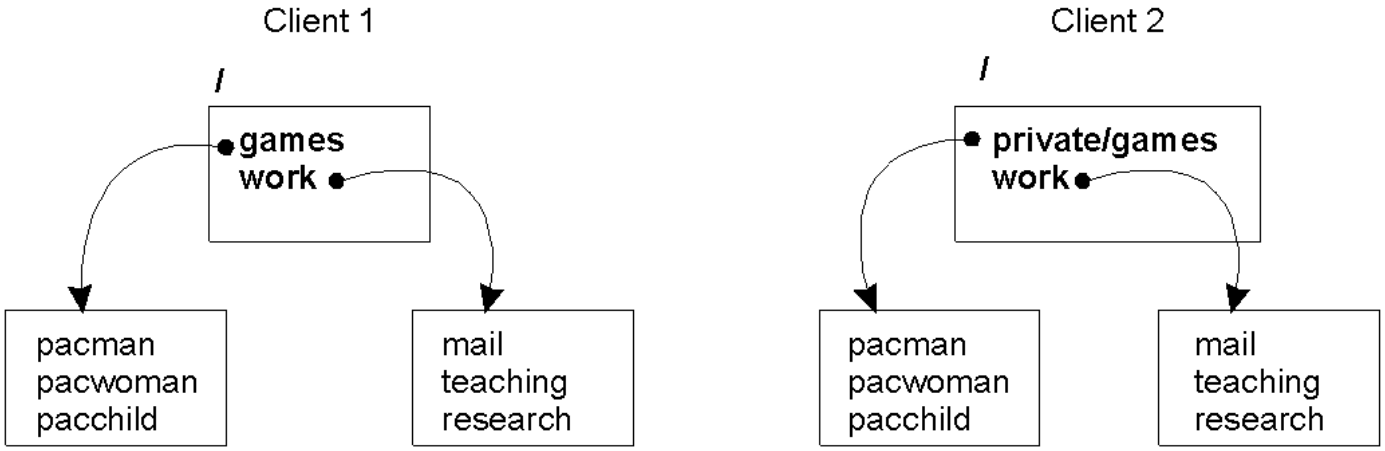
Network Operating System

- Can have one computer provide files transparently for others (NFS)
  - (try a "`df`" on the WPI hosts to see. Similar to a "mount network drive" in Windows)

Network Operating System



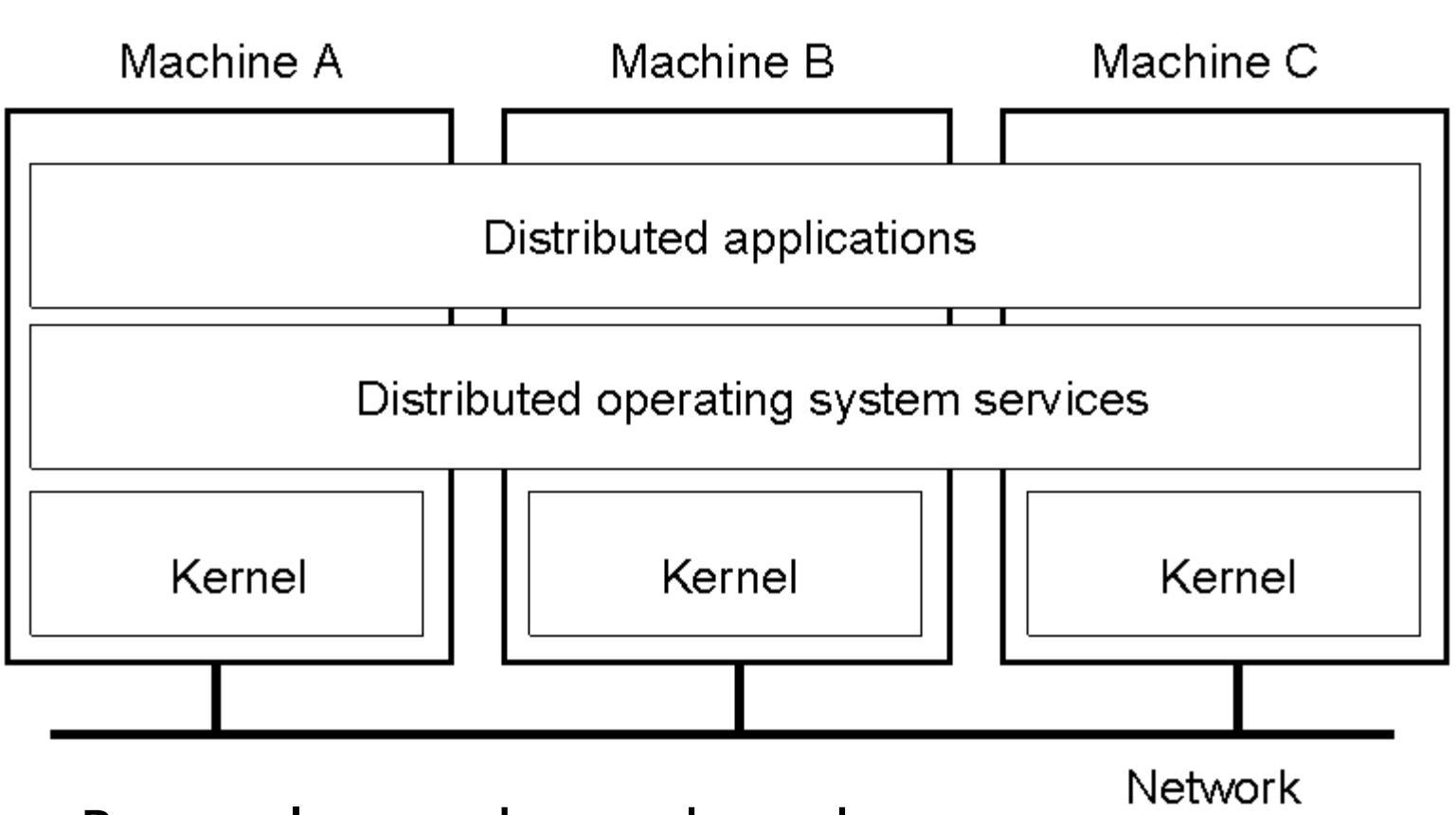
(a)



(b) (c)

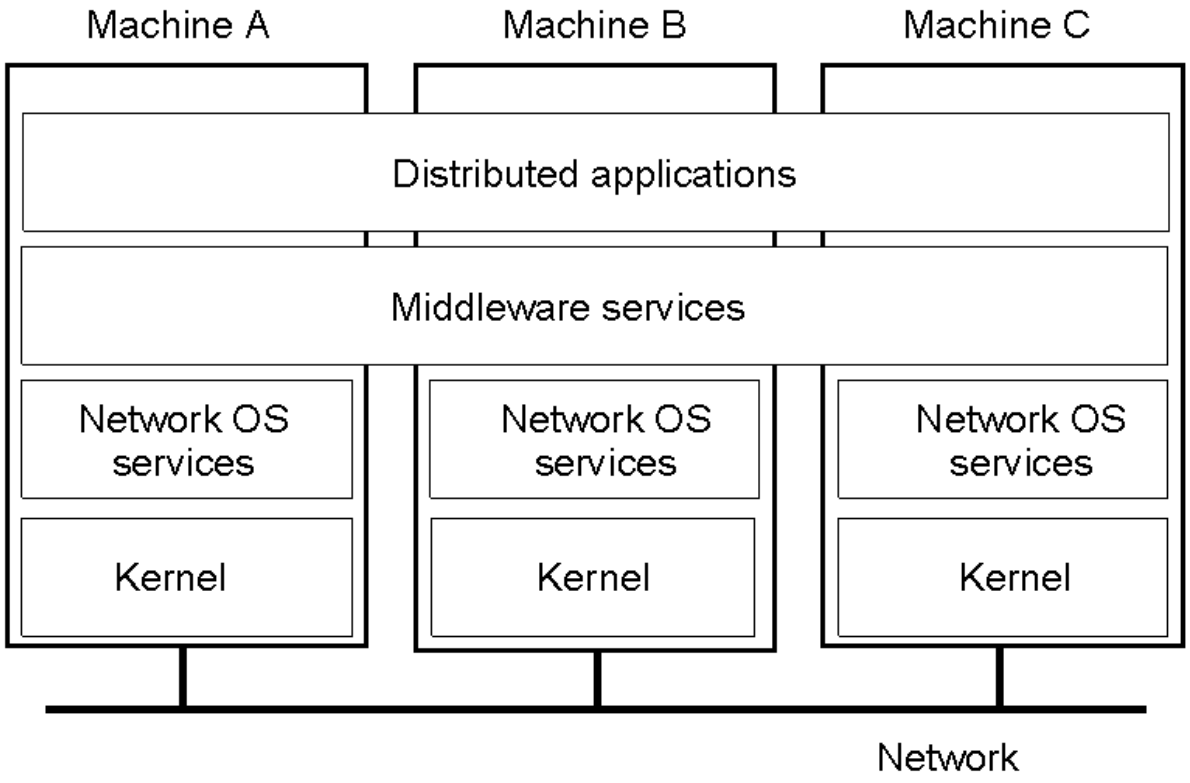
- Different clients may mount the servers in different places
- Inconsistencies in view make NOS's harder, in general for users than DOS's.
  - But easier to scale by adding computers

Distributed Operating Systems



- But no longer have shared memory
  - Provide *message passing*
  - Can try to provide *distributed shared memory*
    - But tough to get acceptable performance

Distributed System as Middleware



Positioning Middleware

- Network OS's are not transparent.
- Distributed OS's are not independent of computers.
- Middleware can help.

Middleware Models

- View everything as a file - *Plan 9.*
- Less strict – *distributed file systems.*
- Make all procedure calls appear to be local – *Remote Procedure Calls (RPC).*
- *Distributed objects (oo model).*
- The Web – *distributed documents.*

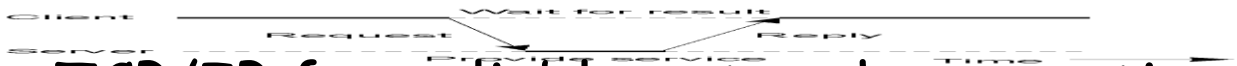
Middleware and Openness

- In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.
  - If different, there will be compatibility issues
  - If incomplete, then users will build their own or use lower-layer services (frowned upon)

Comparison between Systems

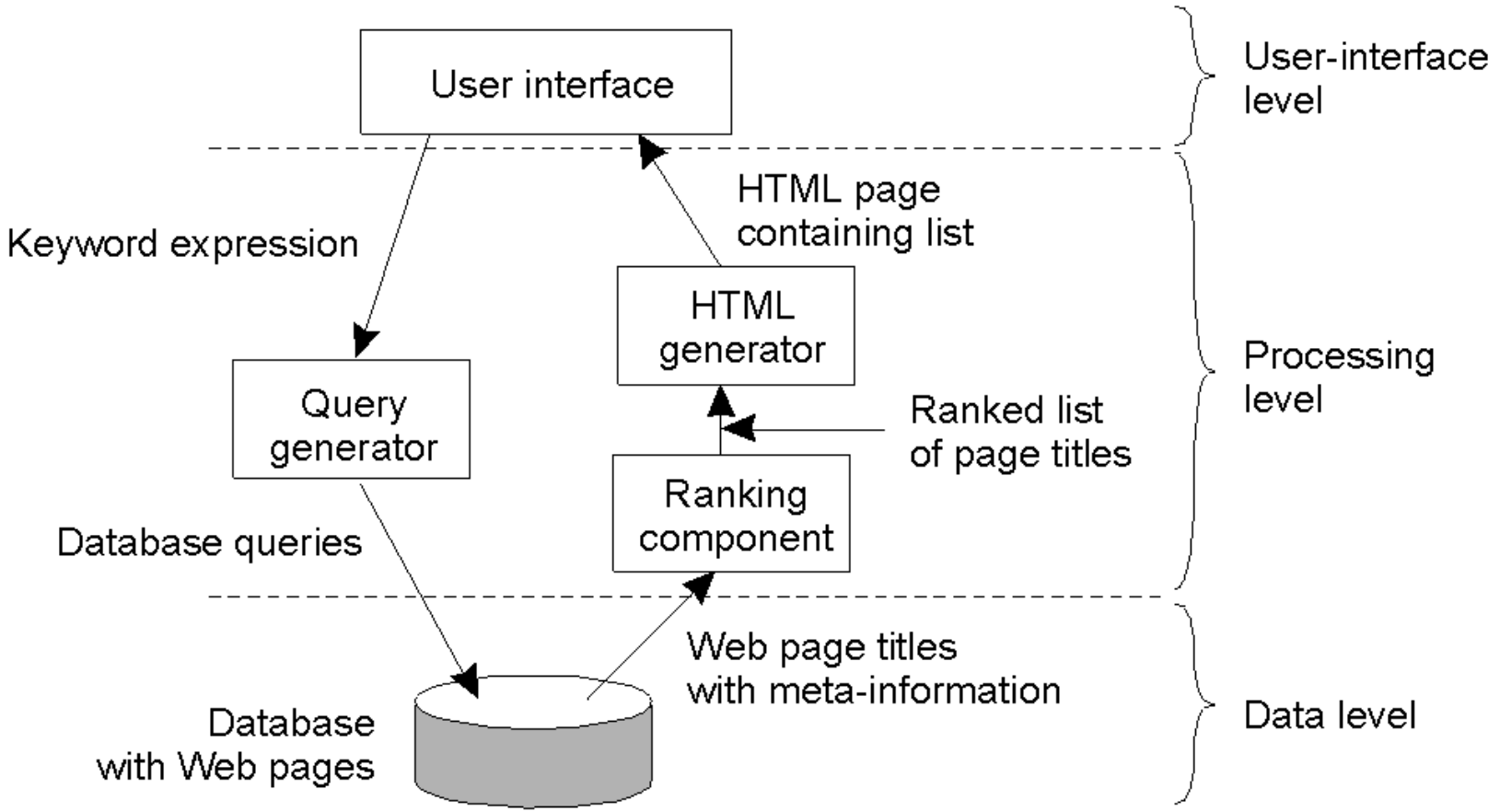
Item	Distributed OS		Network OS	Middleware -based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

Client-Server Model

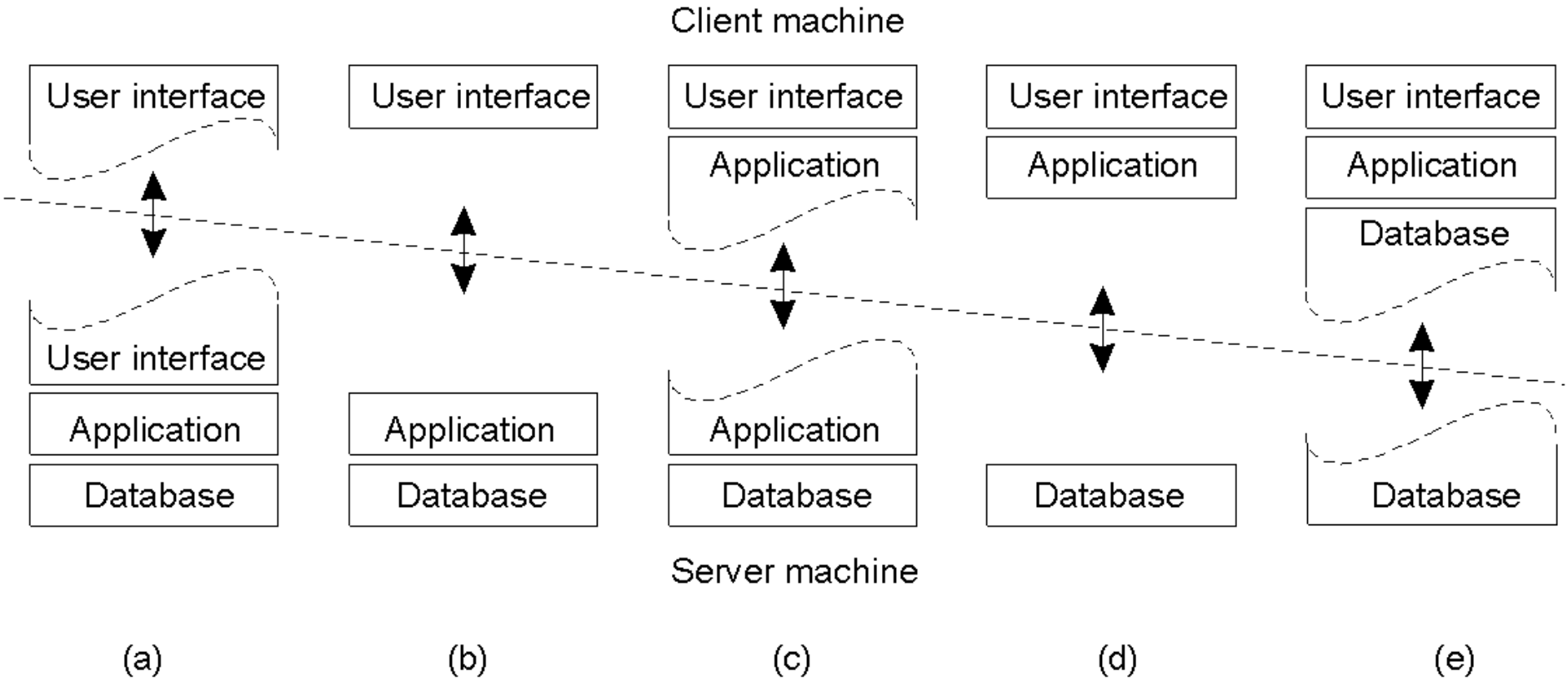


- Use TCP/IP for reliable network connection.
  - This implies the client must establish a connection before sending the first request.

Internet Search Engine

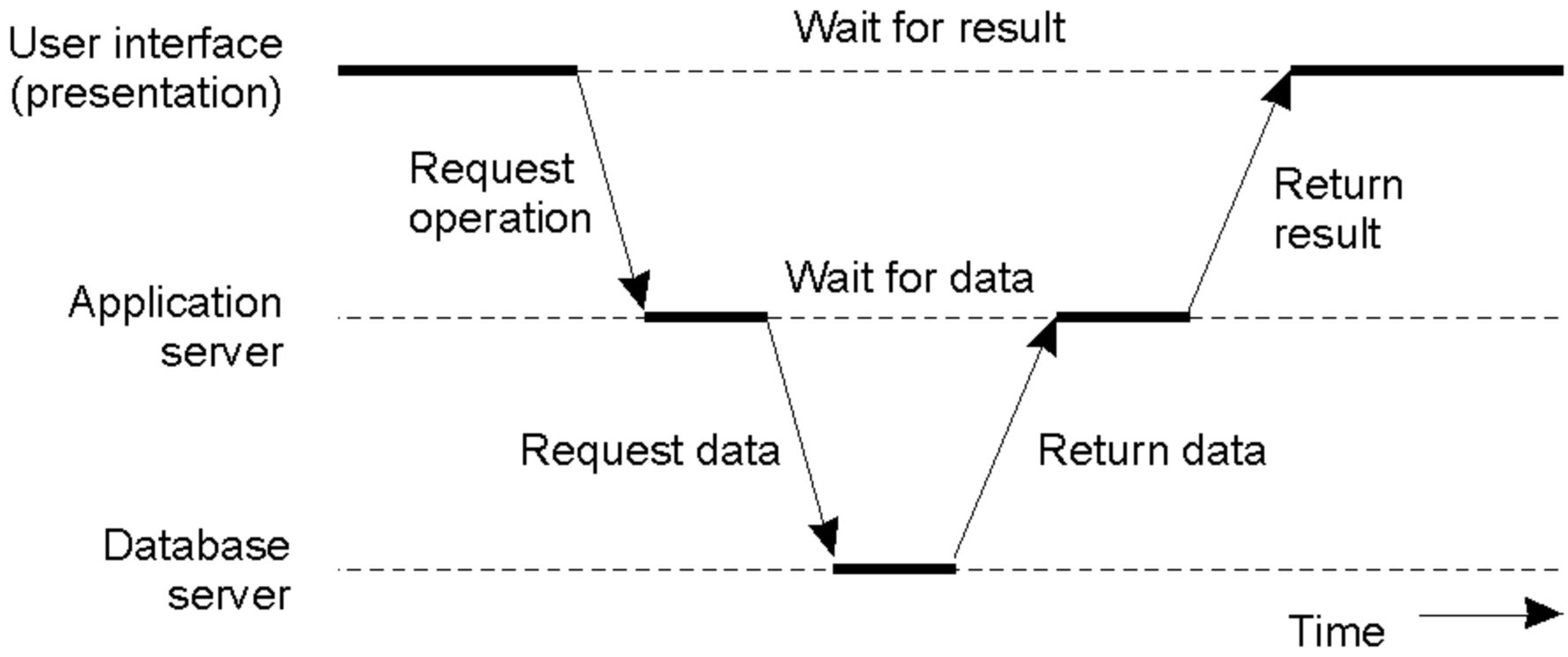


Multitiered Architectures



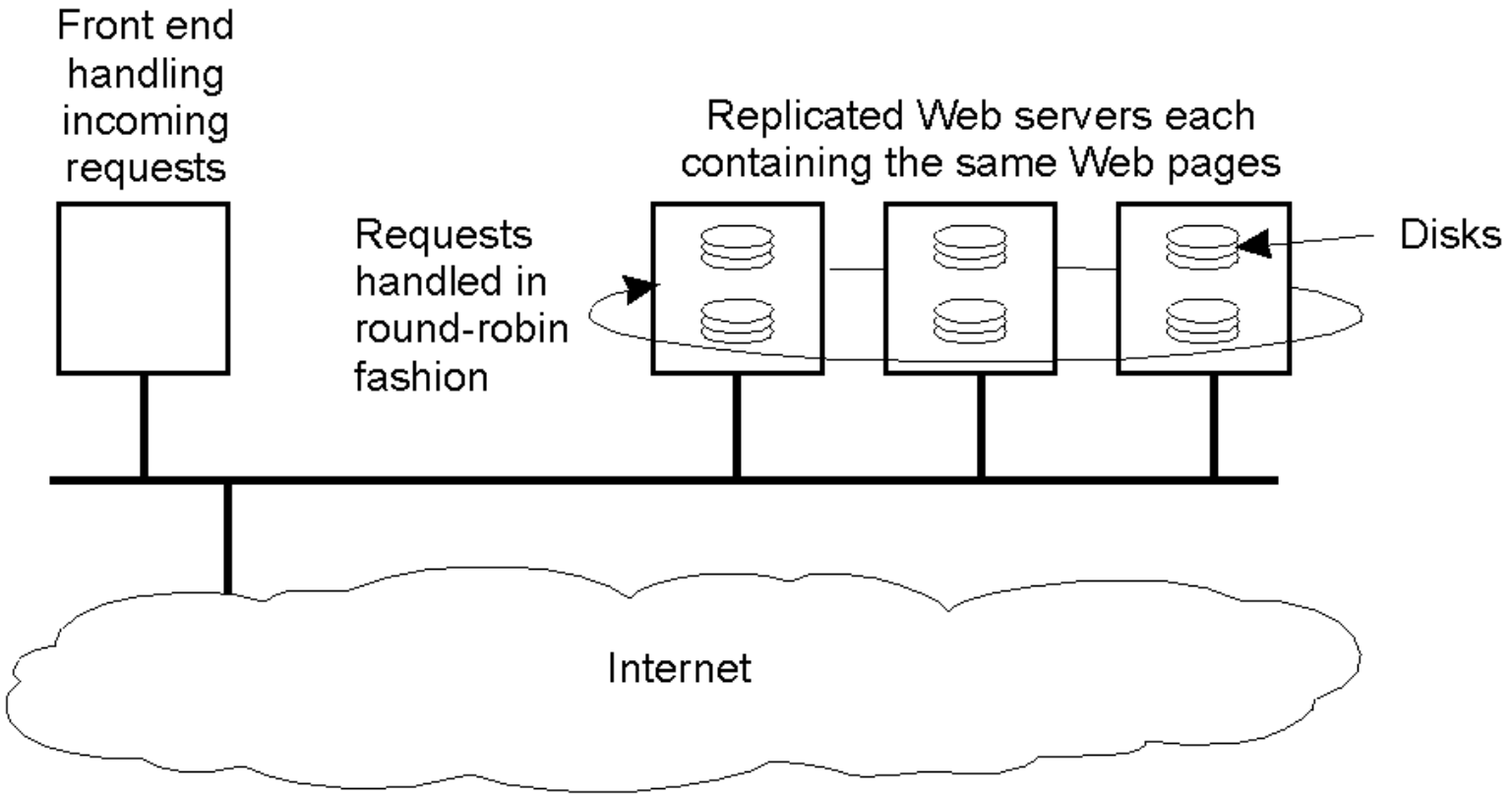
- Thin client (a) to Fat client (e)
  - (d) and (e) popular for NOS environments

Multitiered Architectures: 3 tiers



- Server may act as a client
  - Example would be transaction monitor across multiple databases

Horizontal Distribution



- Distribute servers across nodes
  - E.g., Web server "farm" for load balancing
- Distribute clients in peer-to-peer systems.

Communication in Distributed Systems

Distributed Systems

Sistemi Distribuiti

Andrea Omicini

andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2014/2015

Outline

These Slides Contain Material from [TvS07]

Slides were made kindly available by the authors of the book

- Such slides shortly introduced the topics developed in the book [TvS07] adopted here as the main book of the course
- Most of the material from those slides has been re-used in the following, and integrated with new material according to the personal view of the teacher of this course
- Every problem or mistake contained in these slides, however, should be attributed to the sole responsibility of the teacher of this course

What You Are Supposed to Know...

... from the Courses of Computer Networks, Telecommunication Networks and Foundations of Informatics

Basics about protocols

- ISO/OSI
- Protocols and reference model
- Main network and Internet protocols

Basics about communication

- Procedure call
- Representation formats and problems – e.g., little endian vs. big endian
- Sockets

Outline

The Role of Interaction in Distributed System I

Interaction vs. Computation

- Talking of processes, threads, LWP, and the like, is just half of the story
- Maybe, not even the most important half...

→ They represent the *computational* components of a (distributed) system

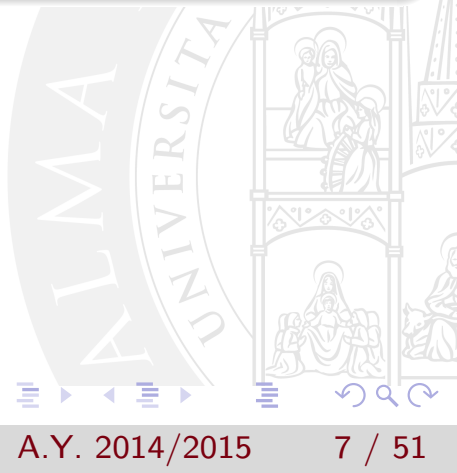
- Components of a system actually make a system only by interacting with each other

→ *Interaction* represents an orthogonal dimension with respect to *computation*

The Role of Interaction in Distributed System II

Engineering Interaction

- Methodologies and technologies for engineering communication are not the same as those for engineering computation
- New models and tools are required
- which could be seamlessly integrated with those for engineering computational components



Interaction vs. Communication

Interaction is more general than communication

- Communication is a form of interaction
- Communication is interaction where information is exchanged
- Not every interaction is communication
- E.g., sharing the same space is a way of interacting without communicating

Whereas such a distinction is not always evident from the literature...

- On the one hand, we should keep this in mind
- On the other hand, in the classical field of inter-process communication, this distinction is often not essential

Communication in Non-distributed Settings

Communication does not belong to distributed systems only

- Communication mechanisms like procedure call and message-passing just require a plurality of interacting entities, not distributed ones
- However, communication in distributed systems presents more difficult challenges, like unreliability of communication and large scale
- Of course, communication in distributed systems first of all deals with distribution / location transparency

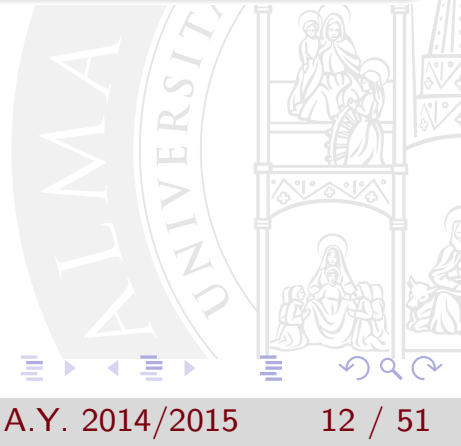
Outline

Outline

Layered Communication I

Communication involves a number of problems at many different levels

- From the physical network level up to the application level
- Communication can be organised on layers
- A *reference model* is useful to understand how protocols, behaviours and interactions



Layered Communication II

OSI model

- Standardised by the International Standards Organization (ISO)
- Designed to allow open systems to communicate
- Rules for communication govern the format, content and meaning of messages sent and received
- Such rules are formalised in *protocols*
- The collection of protocols for a particular system is its *protocol stack*, or *protocol suite*

Types of Protocols

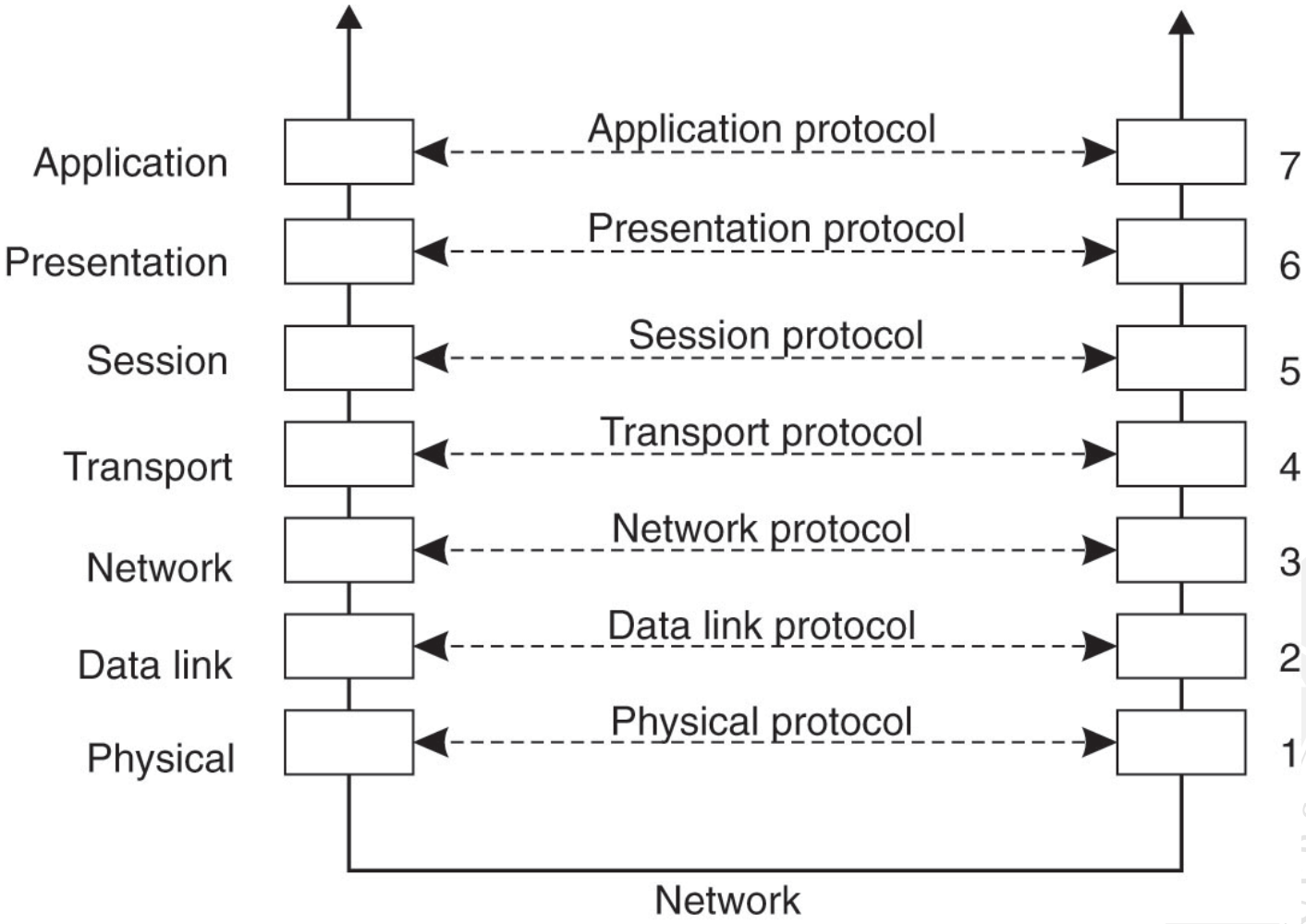
Connection-oriented protocols

- First of all, a connection is established between the sender and the receiver
- Possibly, an agreement over the protocol to be used is reached
- Then, communication occurs through the connection
- Finally, the connection is terminated

Connectionless protocols

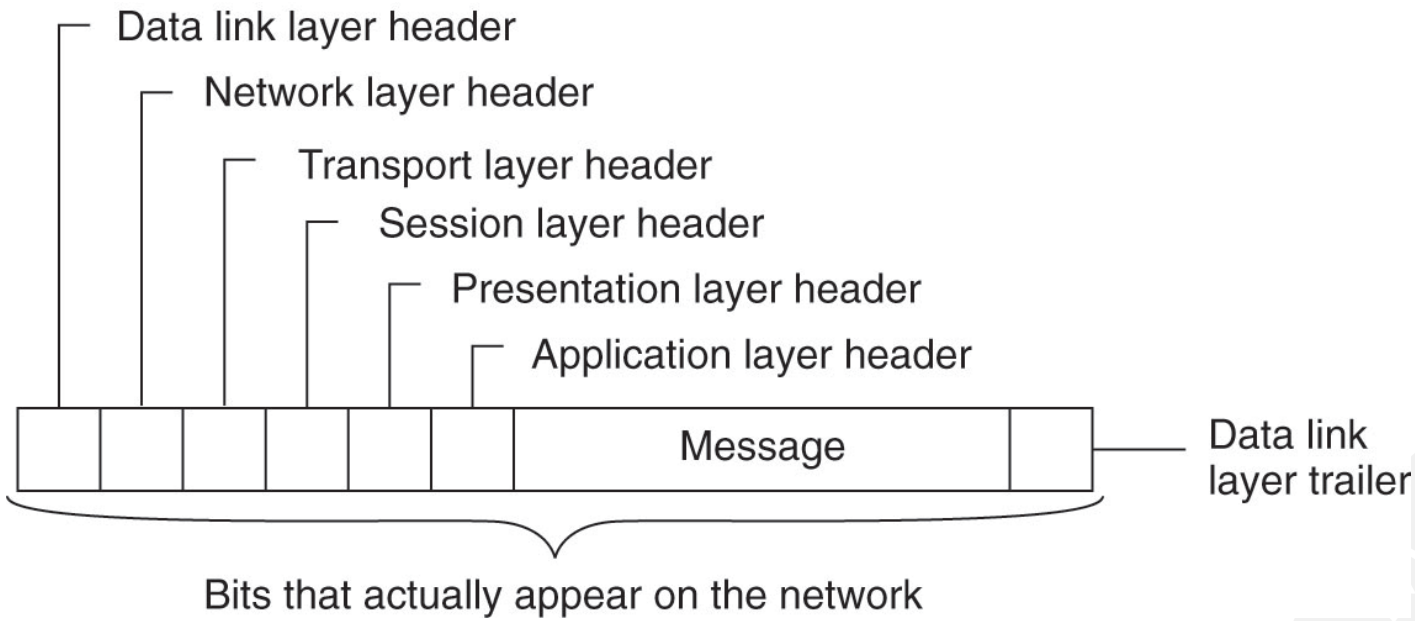
- No setup is required
- The sender just send a message when it is ready

The OSI Reference Model



Layers, interfaces, and protocols in the OSI Model [TvS07]

A Message in the OSI Reference Model



A typical message as it appears on the network [TvS07]

OSI Model $\neq$ OSI Protocols

OSI protocols

- Never successful
- TCP/IP is not an OSI protocol, and still dominates its layers

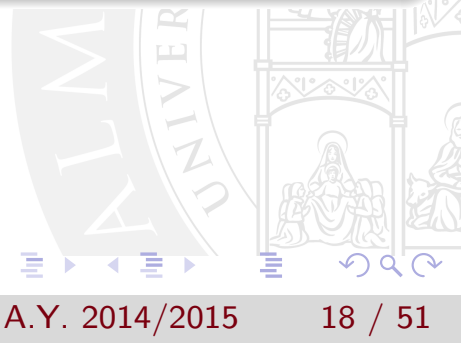
OSI model

- Perfect to understand and describe communication systems through layers
- However, some problems exist when middleware comes to play

Middleware Protocols I

The problem

- Middleware mostly lives at the application level
- Protocols for middleware services are different from high-level application protocols
- ← Middleware protocols are application-independent, application protocols are obviously application-dependent
- How can we distinguish between the two sorts of protocols at the same layer?

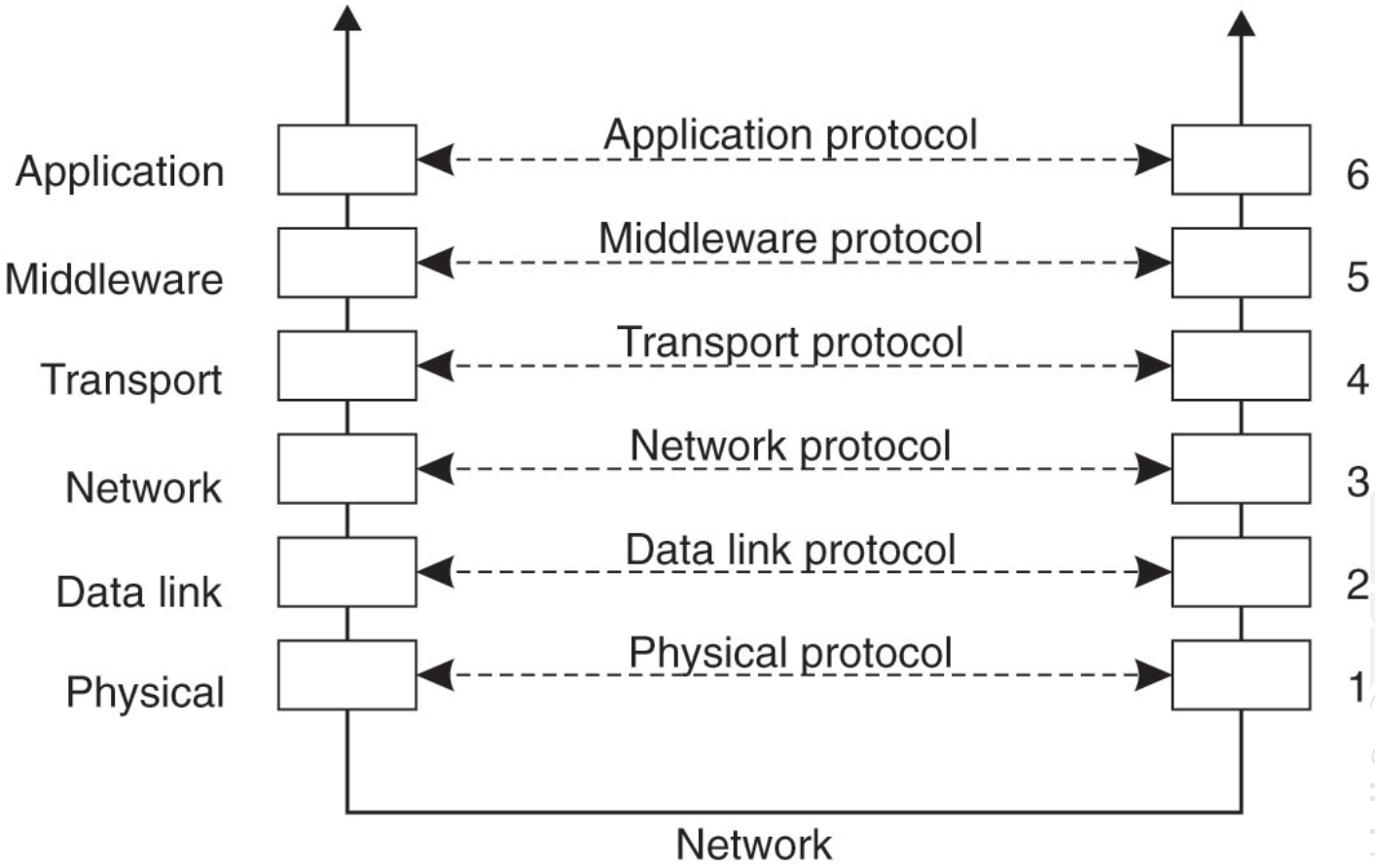


Middleware Protocols II

Extending the reference model for middleware

- Session and presentation layers are replaced by a *middleware layer*, which includes all application-independent protocols
- Potentially, also the transport layer could be offered in the middleware one

Middleware as an Additional Service in Client-Server Computing



Adapted reference model for network communication [TvS07]

Outline

Types of Communication I

Persistent vs. transient communication

- *Persistent communication* — A message sent is stored by the communication middleware until it is delivered to the receiver

→ No need for time coupling between the sender and the receiver

- *Transient communication* — A message sent is stored by the communication middleware only as long as both the receiver and the sender are executing

→ Time coupling between the sender and the receiver

Types of Communication II

Asynchronous vs. synchronous communication

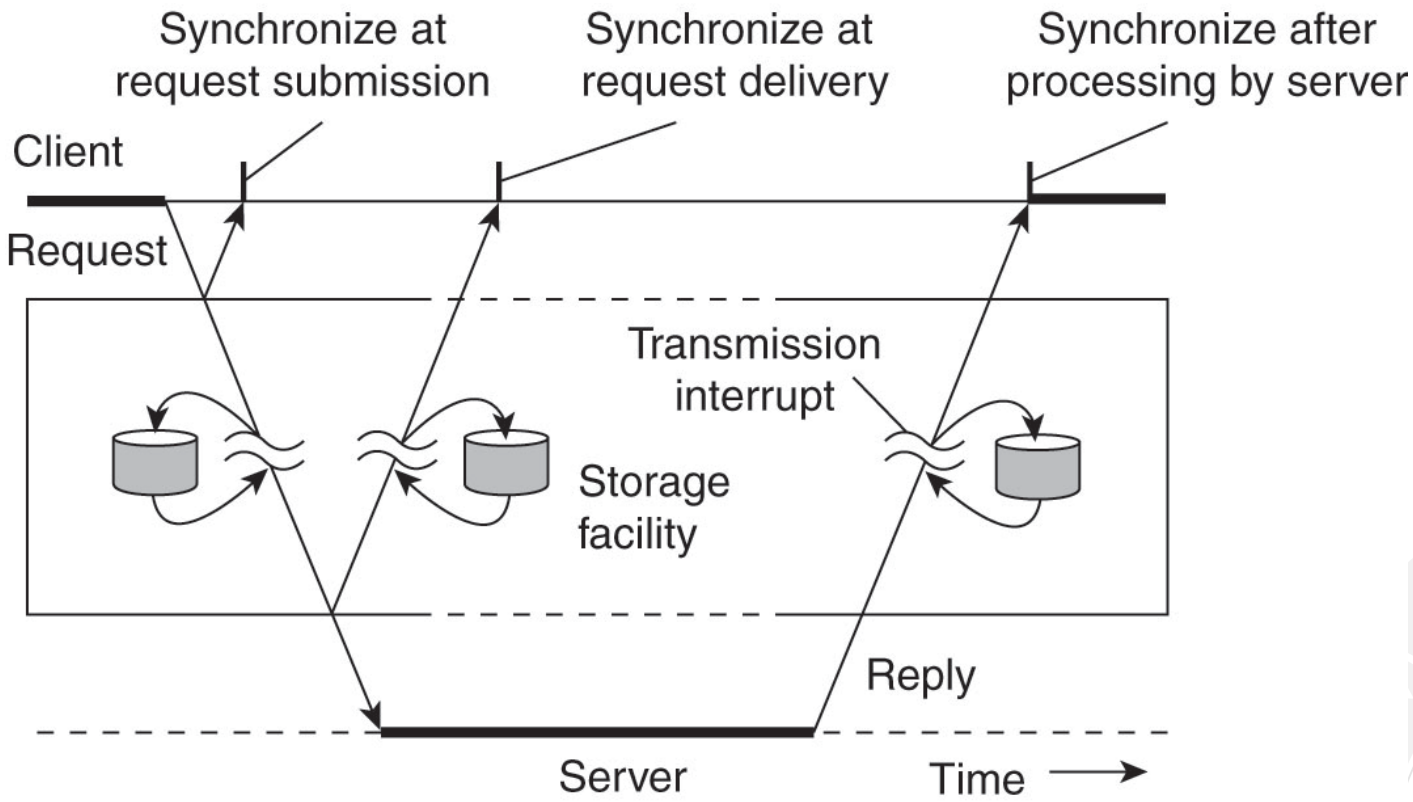
- *Asynchronous communication* — The sender keeps on executing after sending a message

→ The message should be stored by the middleware

- *Synchronous communication* — The sender blocks execution after sending a message and waits for response – until the middleware acknowledges trasmission, or, until the receiver acknowledges the reception, or, until the receiver has completed processing the request

→ Some form of coupling in control between the sender and the receiver

Communications with a Middleware Layer



Viewing middleware as an intermediate (distributed) service in application-level communication
[TvS07]

Actual Communication in Distributed Systems I

Persistency & synchronisation in communication

- In the practice of distributed systems, many combinations of persistency and synchronisation are typically adopted
- Persistency and synchronisation should then be taken as two dimensions along which communication and protocols could be analysed and classified

Actual Communication in Distributed Systems II

Discrete vs. streaming communication

- Communication is not always *discrete*, that is, it does not always happen through complete units of information – e.g., messages
- *Discrete communication* is then quite common, but not the only way available – and does not respond to all the needs
- Sometimes, communication needs to be continuous – through sequences of messages constituting a possibly unlimited amount of information
- *Streaming communication* — The sender delivers a (either limited or unlimited) sequence of messages representing the *stream* of information to be sent to the receiver

→ Communication may be *continuous*

Outline

Remote Procedure Call (RPC)

Basic idea

- Programs can call procedures on other machines
- When a process A calls a procedure on a machine B, A is suspended, and execution of procedure takes place on B
- Once the procedure execution has been completed, its completion is sent back to A, which resumes execution

Information in RPC

- Information is not sent directly from sender to receiver
- Parameters are just packed and transmitted along with the request
- Procedure results are sent back with the completion
- *No message passing*

Issues of RPC

Main problems

- The address space of the caller and the callee are separate and different

→ Need for a common reference space

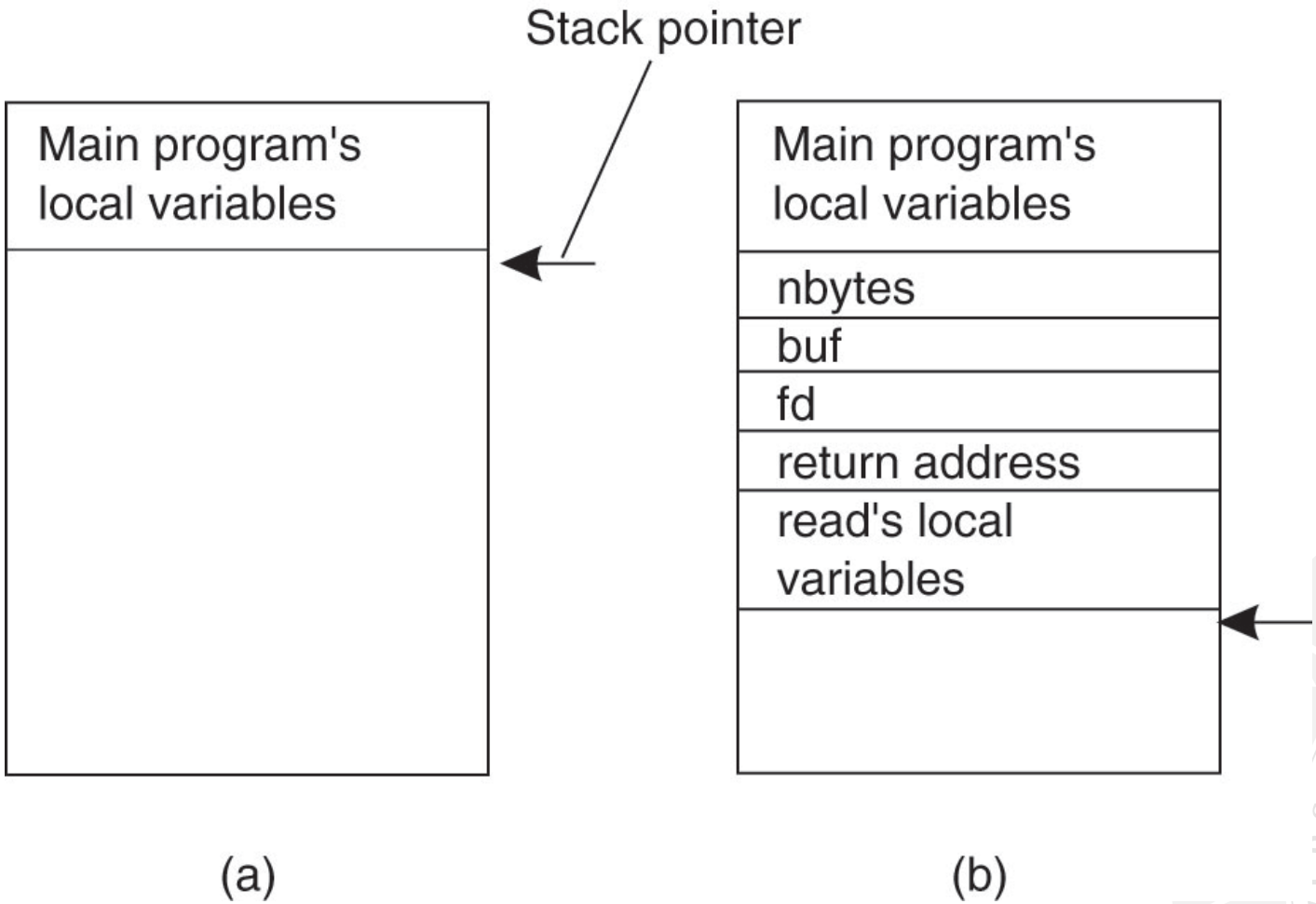
- Parameters and results have to be passed and handled correctly

→ Need for a common data format

- Either / both machines could unexpectedly crash

→ Need for suitable fault-tolerance policies

Conventional Procedure Call



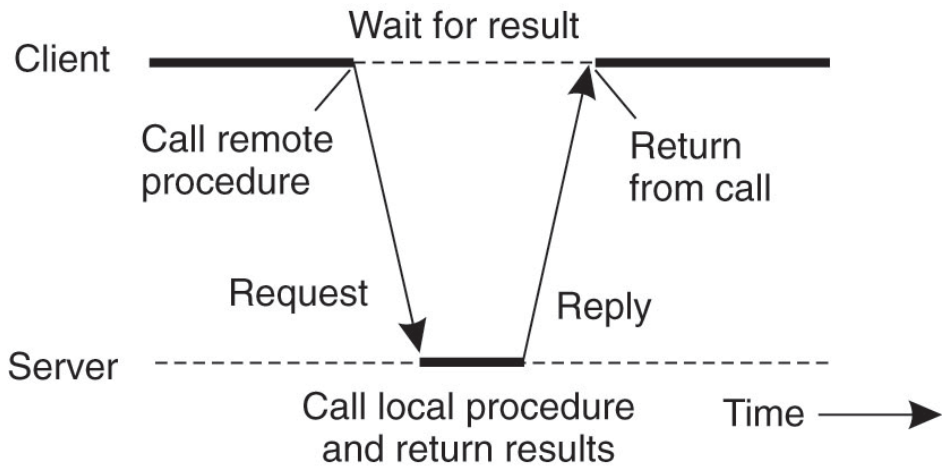
Parameter passing in a local procedure call [TvS07]

Client & Server Stubs

Main goal: transparency

- RPC should be like local procedure call from the viewpoint of both the caller and the callee

→ Procedure calls are sent to the *client stub* and transmitted to the *server stub* through the network to the called procedure



Principle of RPC between a client and server program [TvS07]

Steps for a RPC

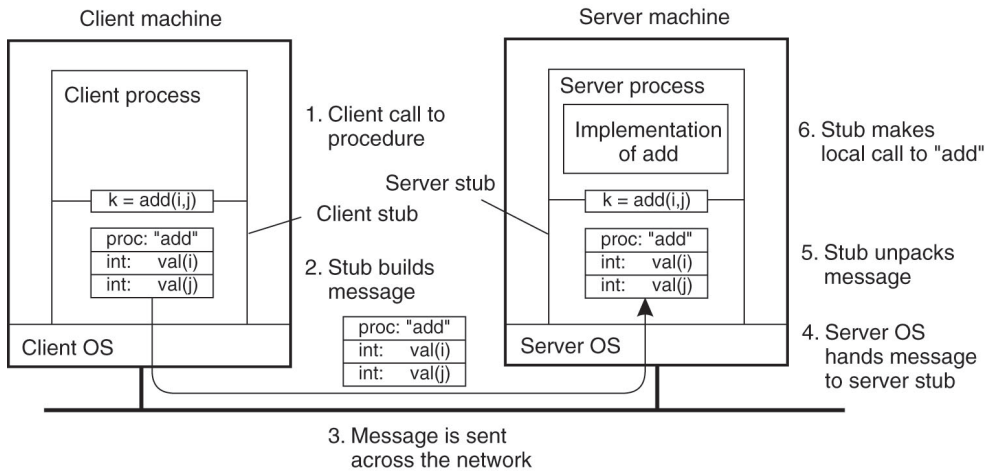
- The client procedure calls the client stub in the normal way
- The client stub builds a message and calls the local operating system
- The client's OS sends the message to the remote OS
- The remote OS gives the message to the server stub
- The server stub unpacks the parameters and calls the server
- The server does the work and returns the result to the stub
- The server stub packs it in a message and calls its local OS
- The server's OS sends the message to the client's OS
- The client's OS gives the message to the client stub
- The stub unpacks the result and returns to the client

Parameter Passing

Passing value parameters

- Parameters are *marshalled* to pass across the network

→ Procedure calls are sent to the *client stub* and transmitted to the *server stub* through the network to the called procedure

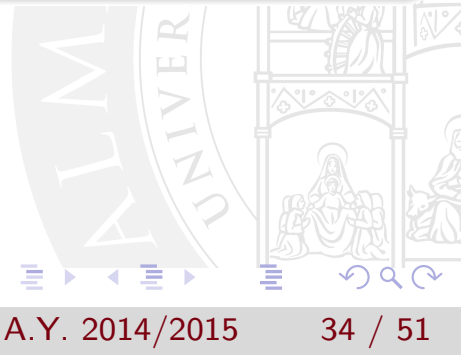


Steps of a remote computation through a RPC [TvS07]

Issues in Parameter Passing I

Passing value parameters

- Problems of representation and meaning
  - e.g., little endian vs. big endian
- In order to ensure transparency, stubs should be in charge of the mapping & translation
- A possible approach: interfaces described through and IDL (Interface Definition Language), and consequent handling compiled into the stubs



Issues in Parameter Passing II

Passing reference parameters

- Main problem: reference space is local
- First solution: forbidding reference parameters
- Second solution: copying parameters (suitably updating the reference), then copying them back (according to the original reference)

→ Call-by-reference becomes copy&restore

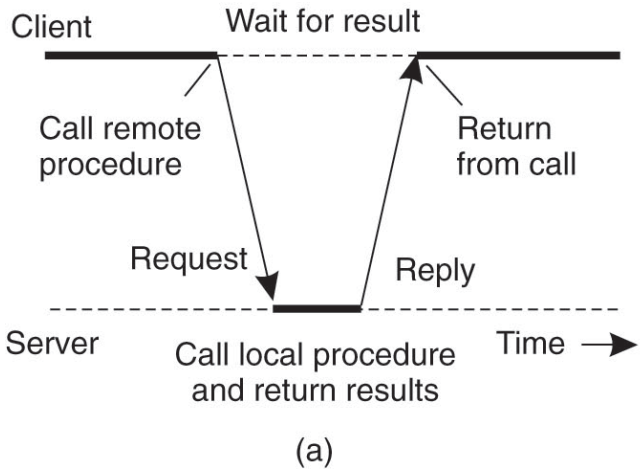
- Third solution: creating a global/accessible reference to the caller space from the callee

Asynchronous RPC

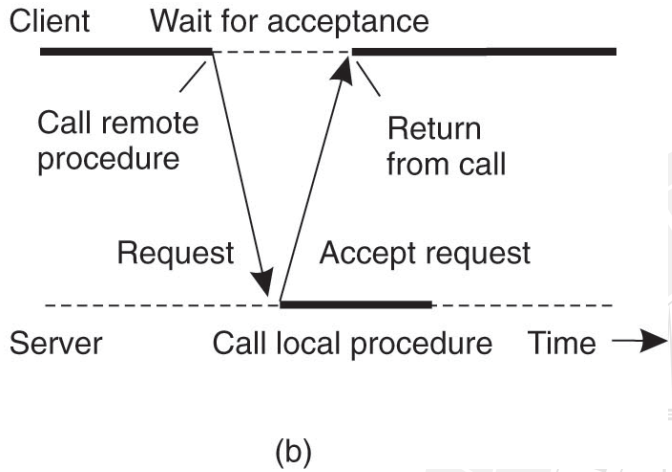
Synchronicity might be a problem in distributed systems

- Synchronicity is often unnecessary, and may create problems

→ *Asynchronous RPC* is an available alternative in many situations



Traditional RPC



Asynchronous RPC

[TvS07]

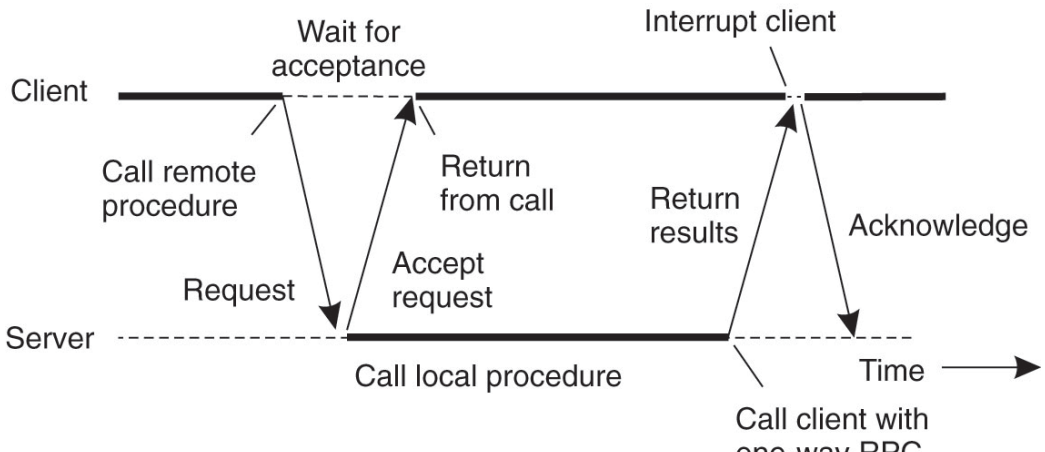
Deferred Synchronous RPC

Combining asynchronous RPCs

- Sometimes some synchronicity is required, but too much is too much

→ *Deferred Synchronous RPC* combines two asynchronous RPC to provide an *ad hoc* form of synchronicity

- The first asynchronous call selects the procedure to be executed and provides for the parameters
- The second asynchronous call goes for the results
- In between, the caller may keep on computing



Limits of RPC

Coupling in time

- Co-existence in time is a requirement for any RPC mechanism
- Sometimes, a too-hard requirement for effective communication in distributed systems
- An alternative is required that does not require the receiver to be executing when the message is sent

The alternative: messaging

- Please notice: message-oriented communication is not synonym of uncoupling
- However, we can take this road toward uncoupled communication

Outline

Message-oriented Transient Communication

Basic idea

- Messages are sent through a channel abstraction
- The channel connects two running processes
- Time coupling between sender and receiver
- Transmission time is measured in terms of milliseconds, typically

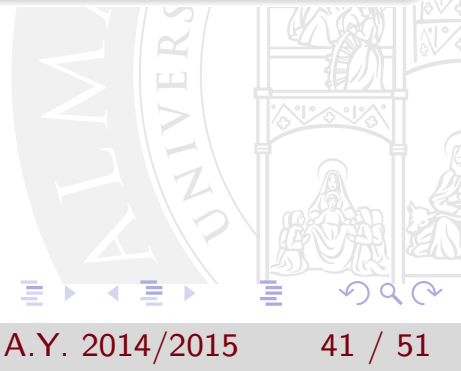
Examples

- Berkeley Sockets — typical in TCP/IP-based networks
- MPI (Message-Passing Interface) — typical in high-speed interconnection networks among parallel processes

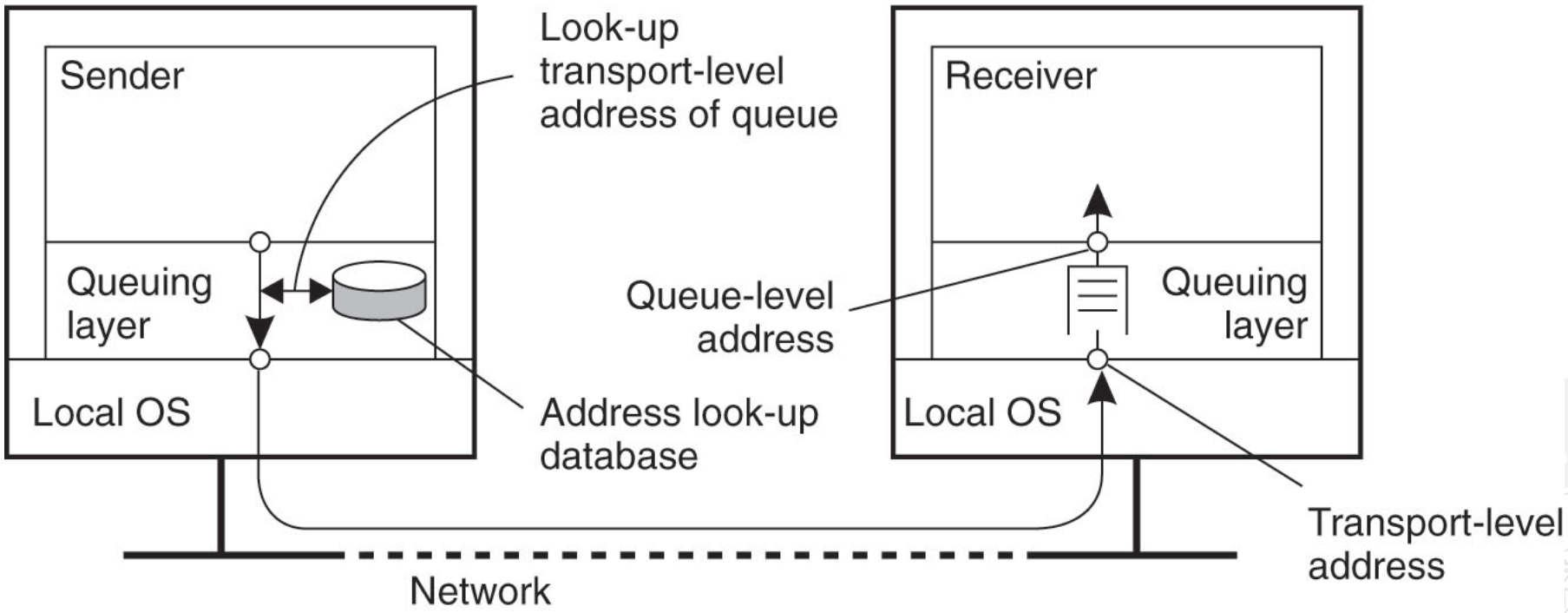
Message-Oriented Persistent Communication I

Message-queuing systems / Message-Oriented Middleware (MOM)

- Basic idea: MOM provides message storage service
- A message is put in a queue by the sender, and delivered to a destination queue
- The target(s) can retrieve their messages from the queue
- Time uncoupling between sender and receiver
- Example: IBM's WebSphere



Message-Oriented Persistent Communication II



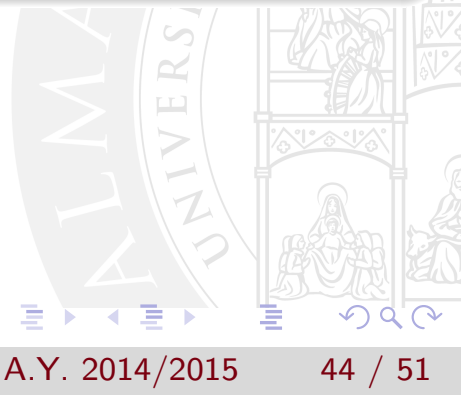
General architecture of a message-queuing system [TvS07]

Outline

Streams

Sequences of data

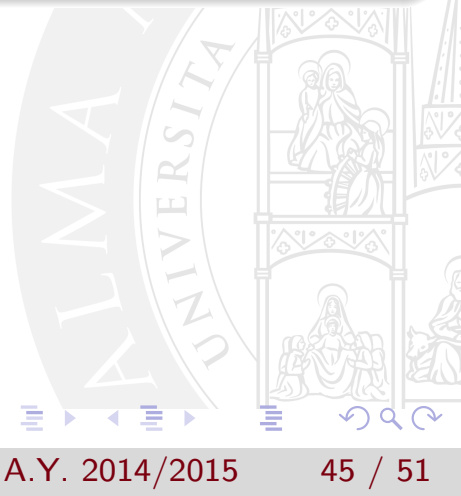
- A stream is transmitted by sending sequences of related messages
- Single vs. complex streams: a single sequence vs. several related simple streams
- Data streams: typically, streams are used to represent and transmit huge amounts of data
- Examples: JPEG images, MPEG movies



Streams & Time I

Continuous vs. discrete media

- In the case of *continuous (representation) media*, time is relevant to understand the data – e.g., audio streams
- In the case of *discrete (representation) media*, time is not relevant to understand the data – e.g., still images



Streams & Time II

Transmission of time-dependent information

Asynchronous transmission mode data items of a stream are transmitted in sequence without further constraints—e.g., a file representing a still image

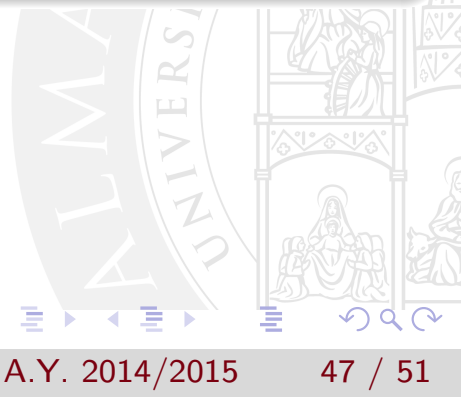
Synchronous transmission mode data items of a stream are transmitted in sequence with a maximum end-to-end delay—e.g., data generation by a pro-active sensor

Isochronous transmission mode data items of a stream are transmitted in sequence with both a maximum and a minimum end-to-end delay—e.g., audio & video

Streams & Quality of Service I

Quality of service

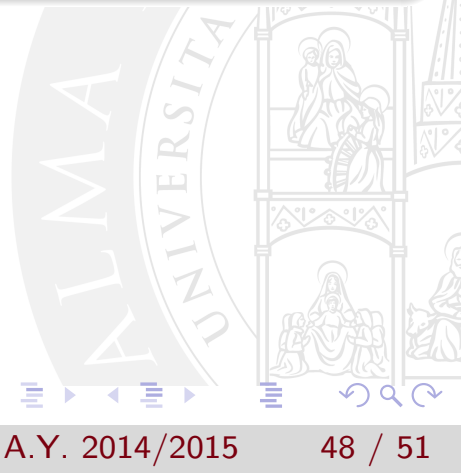
- Timing and other non-functional properties are typically expressed as *Quality of Service* (QoS) requirements
- In the case of streams, QoS typically concerns *timing*, *volume*, and *reliability*
- In the case of middleware, the issue is how can a given middleware ensure QoS to distributed applications



Streams & Quality of Service II

A practical problem

- Whatever the theory, many distributed systems providing streaming services rely on top of the IP stack
- IP specification allow for a protocol implementation dropping packets when needed
- QoS should be enforced at the higher levels



Summing Up

Interaction & communication

- Interaction as an orthogonal dimension w.r.t. computation
- Communication as a form of interaction

High-level abstractions for process-level communication

- Remote Procedure Call
- Message-oriented models
- Streaming
- Other forms like multicasting and epidemic protocols are important, but are not a subject for this course

References


Andrew S. Tanenbaum and Marteen van Steen.
Distributed Systems. Principles and Paradigms.
Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2007.

Communication in Distributed Systems

Distributed Systems

Sistemi Distribuiti

Andrea Omicini

andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2014/2015

Synchronization
in
Distributed Systems

Introduction

- Semaphores require processes to access a shared variable. Other synchronization mechanisms may also be based on shared memory (monitors, for example)
- In distributed systems, interprocess communication is via messages (or RPC) so mutual exclusion mechanisms must be based on this approach. Logical clocks are an example.

Introduction

- Semaphores require processes to access a shared variable. Other synchronization mechanisms may also be based on shared memory (monitors, for example)
- In distributed systems, interprocess communication is via messages (or RPC) so mutual exclusion mechanisms must be based on this approach. Logical clocks are an example.

Message Passing

- Processes may use broadcast (transmit to any interested process) or multicast (transmit to all members of a specific group) message passing.
- Blocking send/receive protocols are often used, because it can be guaranteed that the sender and receiver are synchronized when the message is exchanged
  - Sender will not continue until message has reached receiver

Distributed Mutual Exclusion Characterized

- Processes communicate only through messages – no shared memory or clock.
- Processes must expect unpredictable message delays.
- Processes coordinate access to shared resources (printer, file, etc.) that should only be used in a mutually exclusive manner.

Example: Overlapped Access to Shared File

- Airline reservation systems maintain records of available seats.
- Suppose two people buy the same seat, because each checks and finds the seat available, then each buys the seat.
- Overlapped accesses generate different results than serial accesses – race condition.

Desirable Characteristics for Distributed Mutex Solutions

(1) no deadlocks – no set of sites should be permanently blocked, waiting for messages from other sites in that set.
(2) no starvation – no site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
(3) fairness - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation.)
(4) fault tolerance – the algorithm is able to survive a failure at one or more sites.

Distributed Mutex – Overview

- **Token-based solution**: processes share a special message known as a token
  - Token holder has right to access shared resource
  - Wait for/ask for (depending on algorithm) token; enter Critical Section when it is obtained, pass to another process on exit or hold until requested (depending on algorithm)
  - If a process receives the token and doesn't need it, just pass it on.

Overview - Token-based Methods

- Advantages:
  - Starvation can be avoided by efficient organization of the processes
  - Deadlock is also avoidable
- Disadvantage: token loss
  - Must initiate a cooperative procedure to recreate the token
  - Must ensure that only one token is created!

Overview

- **Permission-based** solutions**:** a process that wishes to access a shared resource must first get permission from one or more other processes.
- Avoids the problems of token-based solutions, but is more complicated to implement.

Basic Algorithms

- Centralized
- Decentralized
- Distributed
  - Distributed with “voting” – for increased fault tolerance
- Token Ring

Centralized Mutual Exclusion

Central coordinator manages requests

FIFO queue to guarantee no starvation

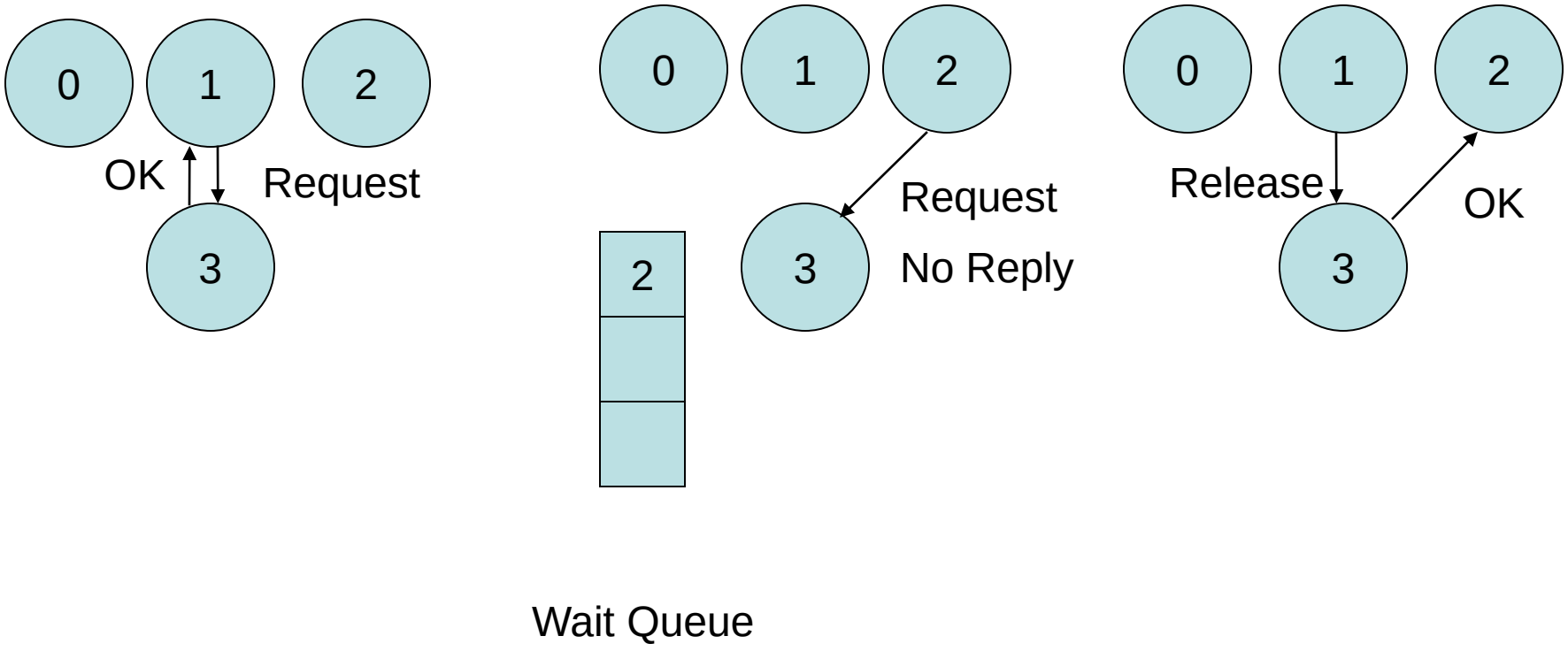


Figure 6-14

Performance Analysis

- Guarantees mutual exclusion
- No starvation/fair
  - If requests are honored in order
- No deadlock
- Fault tolerant?
  - Single point of failure
  - Blocking requests mean client processes have difficulty distinguishing crashed coordinator from long wait
  - Bottlenecks
- Simplicity is a big plus

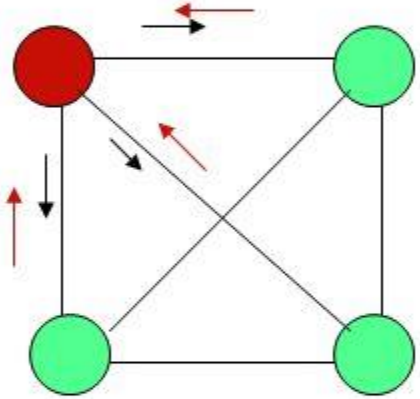
Decentralized algorithm 2

{Ricart & Agrawala's algorithm}

What is new?

1. Broadcast a timestamped ***request*** to all.
2. Upon receiving a request, send ***ack*** if
   - -You do not want to enter your CS, or
   - -You are trying to enter your CS, but your timestamp is **larger** than that of the sender.

   (If you are already in CS, then buffer the request)



3. **Enter CS**, when you receive ***ack*** *from all.*
4. Upon **exit from CS**, send ***ack*** to each pending request before making a new request.

(No release message is necessary)

Analysis

- More robust than the central coordinator approach. If one coordinator goes down others are available.
  - If a coordinator fails and resets then it will not remember having granted access to one requestor, and may then give access to another. According to the authors, it is highly unlikely that this will lead to a violation of mutual exclusion. (See the text for a probabilistic argument.)

Analysis

- If a resource is in high demand, multiple requests will be generated by different processes.
- High level of contention
- Processes may wait a long time to get permission - Possibility of starvation exists
- Resource usage drops.

Distributed Mutual Exclusion

- Probabilistic algorithms do not guarantee mutual exclusion is correctly enforced.
- Many other algorithms do, including the following.
- Originally proposed by Lamport, based on his logical clocks and total ordering relation
- Modified by Ricart-Agrawala

The Algorithm

- Two message types:
  - Request access to shared resource: sent to all processes in the group
  - Reply/OK: A message eventually received at the request site, S_i, from all other sites.
- Messages are time-stamped based on Lamport's total ordering relation, with logical clock.process id. Reason: to provide an unambiguous order of all relevant events.

Requesting

- When a process P_i wants to access a shared resource it builds a message with the resource name, pid and current timestamp: *Request* (r_a, ts_i, i)
  - A request sent from P_3 at “time” 4 would be time-stamped (4.3). Send the message to all processes, including yourself.
- Assumption: message passing is reliable.

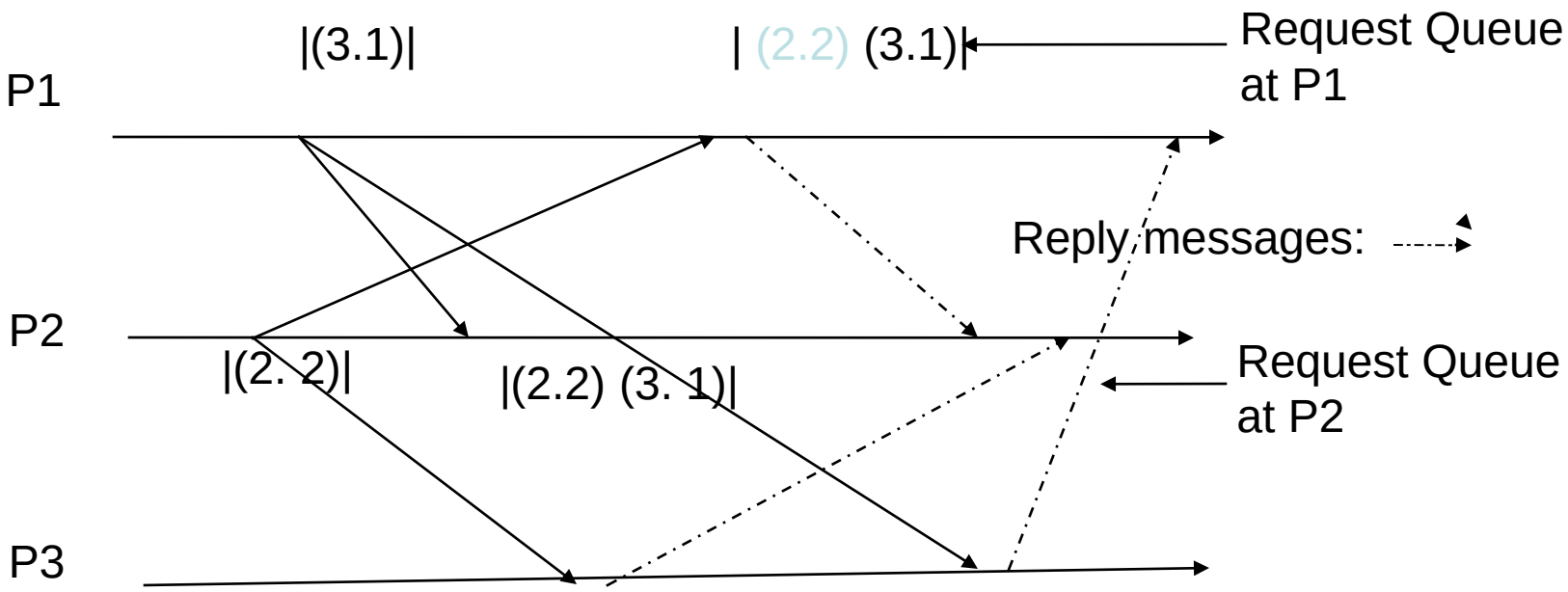
Processing a Request

- P_i sends a Request (r_a, ts_i, i) to all sites.
- When P_k receives the request it acts based on its own status relative to the critical section.
  - sends a Reply (OK) if it (P_k) is not in the critical section and doesn't want the critical section
  - queues the request locally if it is in its critical section, but does not reply
  - if it isn't in the CS but would like to be, sends a Reply (OK) if the incoming request has a lower timestamp than its own, otherwise queues the request and does not reply. In this case the incoming request has lower priority than P_k's request priority.

Executing the Critical Section

- P_i can enter its critical section when it has received an OK Reply from every other process.
- No undelivered higher priority request can exist. It would have arrived before the OK reply.

Based on Figure 6.4 – Singhal and Shivaratri



Process 2 can enter the critical section.
It has received messages from site 1 and site 3
P2 did not reply to P1 because P1's Request has a larger timestamp.

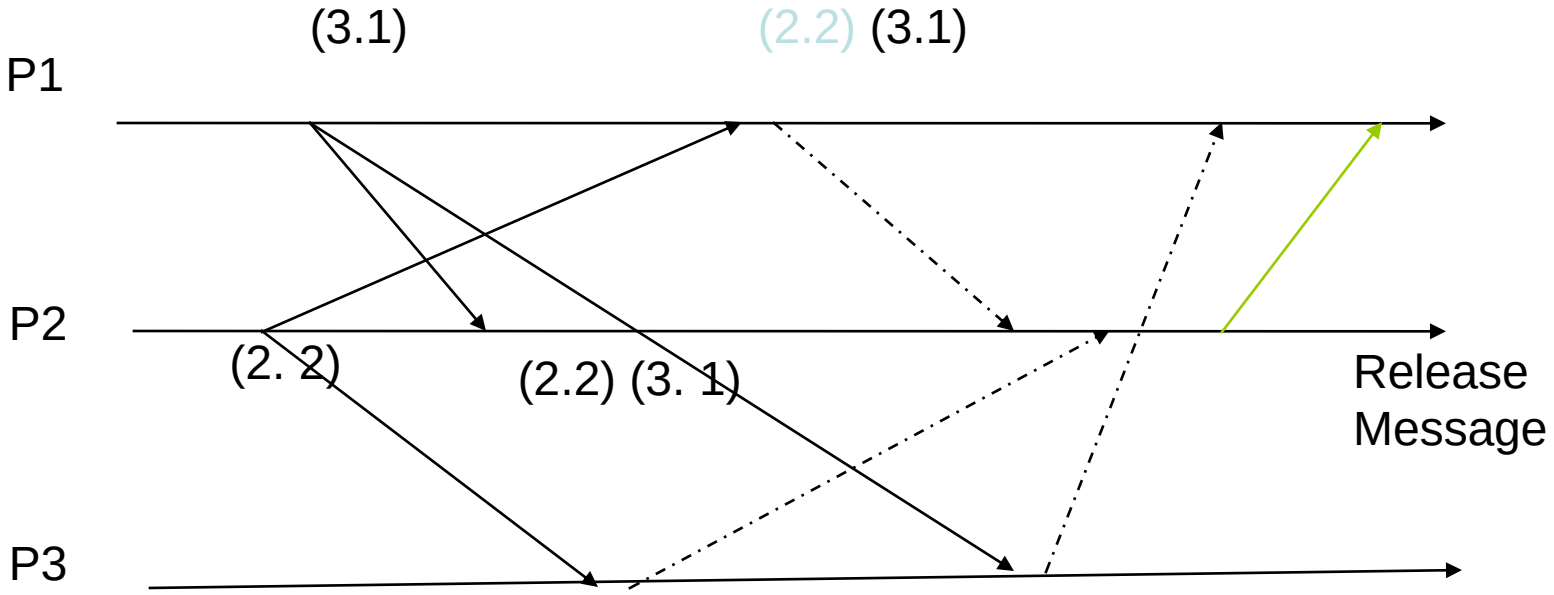
Releasing the Critical Section

- When a processor completes its CS, it sends a Release message to all processors. (the Release message acts as an OK Reply to any process waiting for one.)
- Its request is removed from all queues at this time.
- If other processes are waiting to execute CS's, one of them will now be able to proceed.

Comments

- Purpose of REPLY from node *i* to *j:* ensures that *j* has seen all requests from *i* prior to sending the REPLY (and therefore, possibly any request of *i* with timestamp lower than *j*'s 's request)
- Requires FIFO channels.
- 2(*n* – 1) messages per critical section
- Synchronization delay = one message transmission time
- Requests are granted in order of increasing timestamps

Based on Figure 6.4



When Process 2 leaves the critical section it sends a Release Message to Process 1 and now the next site can enter mutual exclusion.

Analysis

- Deadlock and starvation are not possibilities since the queue is maintained in order, and OKs are eventually sent to all processes with requests.
- Message passing overhead is not excessive: 2(n-1) per critical section
- However – Consider fault tolerance.
  - What happens if a processor in the system fails?

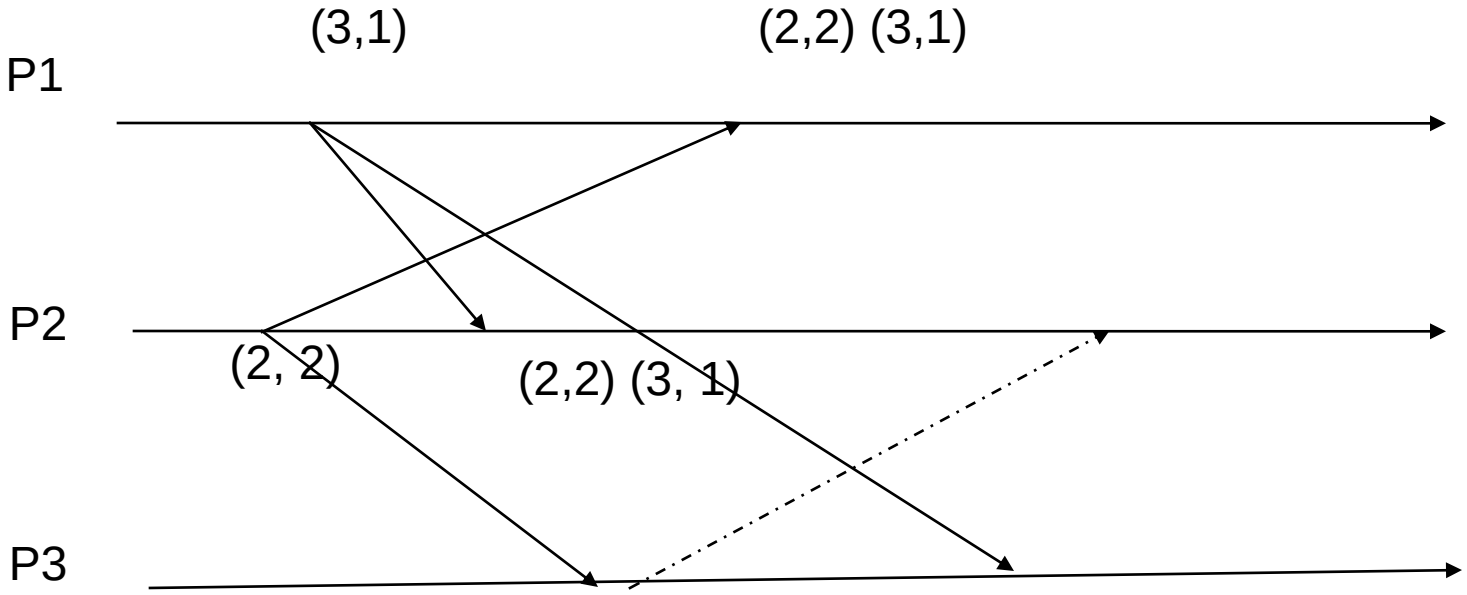
Dealing With Failure – Additional Messages

- When a request comes in, the receiver always sends a reply (Yes or No)
- If a reply doesn't come in a reasonable amount of time, repeat the request.
  - Continue until a reply arrives or until the sender decides the processor is dead.
- Lots of message traffic, *n* bottlenecks instead of one, must keep track of all group members, etc.

Dealing with Failure - A Voting Scheme Modification

- But it really isn’t necessary to get 100% permission
- When a processor receives a Request it sends a reply (vote) only if it hasn't "voted" for some other process. In other words, **you can only send out one OK Reply at a time**.
- There should be at most N votes in the system at any one time.

Voting Solution to Mutual Exclusion



Process 3 receives two requests; it “votes” for P2 because it received P2’s request first. P2 votes for itself.

A Further Problem

- Voting improves fault tolerance but what about deadlock?
- Suppose a system has 10 processes
- Also assume that three Requests are generated at about the same time and that two get 3 votes each, one gets 4 votes, so no process has a majority.

Tweaking the Voting Solution to Prevent Deadlock

- A processor can change its vote if it receives a Request with an earlier time stamp than the request it originally voted for.
  - Additional messages: Retract and Relinquish.
- If P_i receives a Retract message from P_k and has not yet entered its critical section, it sends P_k a Relinquish message and no longer counts that vote.

Tweaking the Voting Solution

- When P_k gets a Relinquish message, it can vote again.
- Eventually, the processor with the earliest timestamp will get enough votes to begin executing the critical section.
- This is still an O(N) algorithm although the message traffic is increased.

A Token Ring Algorithm

- Previous algorithms are permission based, this one is token based.
- Processors on a bus network are arranged in a logical ring, ordered by network address, or process number (as in an MPI environment), or some other scheme.
- Main requirement: that the processes know the ordering arrangement.

Algorithm Description

- At initialization, process 0 gets the token.
- The token is passed around the ring.
- If a process needs to access a shared resource it waits for the token to arrive.
- Execute critical section & release resource
- Pass token to next processor.
- If a process receives the token and doesn’t need a critical section, hand to next processor.

Analysis

- Mutual exclusion is guaranteed trivially.
- Starvation is impossible, since no processor will get the token again until every other processor has had a chance.
- Deadlock is not possible because the only resource being contended for is the token.
- The problem: lost token

Lost Tokens

- What does it mean if a processor waits a long time for the token?
  - Another processor may be holding it
  - It’s lost
- No way to tell the difference; in the first case continue to wait; in the second case, regenerate the token.

Crashed Processors

- This is usually easier to detect – you can require an acknowledgement when the token is passed; if not received in bounded time,
  - Reconfigure the ring without the crashed processor
  - Pass the token to the new “next” processor

Comparison

Algorithm	Messages per entry/exit	Delay before entry	Synch ‡ ‡ Delay	Problems
Centralized	3	2	2	coordinator crash
Decentralized	3mk*	2m‡ (if no competition)	2m	starvation, low efficiency
Distributed process	2(n – 1)	2(n – 1)**	1***	crash of any
Token ring	1 to ∞	0 to n -1	1 to n-1	lost token, process crash

* m: number of coordinators contacted; k: number of attempts

** (n-1) requests and (n-1) replies where n is the number of processes

** n-1 release messages; sent one after the other

*** 1 message to the next process,

‡ Textbook is inconsistent: 2m in the figure, 3mk in the discussion

‡ ‡ Delay Before Entry: Assumes no other process is in critical section
Synchronization Delay: After one process leaves, how long before next enters
DBE and SD figures assume messages are sent one after another, not broadcast

Summary

- The centralized method is the simplest. If crashes are infrequent, probably the best.
- Also … can couple with a leader election algorithm to improve fault tolerance.
- All of these algorithms work best (and are most likely to be needed) in smallish systems.

Election Algorithms

Election Algorithms

- If we are using one process as a coordinator for a shared resource …
- …how do we select that one process?
- Often, there is no *owner* or *master* that is automatically considered as coordinator
  - E.g., Grapevine, there is no *owner* for a *Registry*
  - By contrast:–DNS has a master for every domain

Solution – an Election

- All nodes currently involved get together to *choose* a coordinator
- If the coordinator crashes or becomes isolated, elect a new coordinator
- If a previously crashed or isolated node, comes on line, a new election *may* have to be held.

Election Algorithms

- Wired systems
  - Bully algorithm
  - Ring algorithm
- Wireless systems
- Very large-scale systems

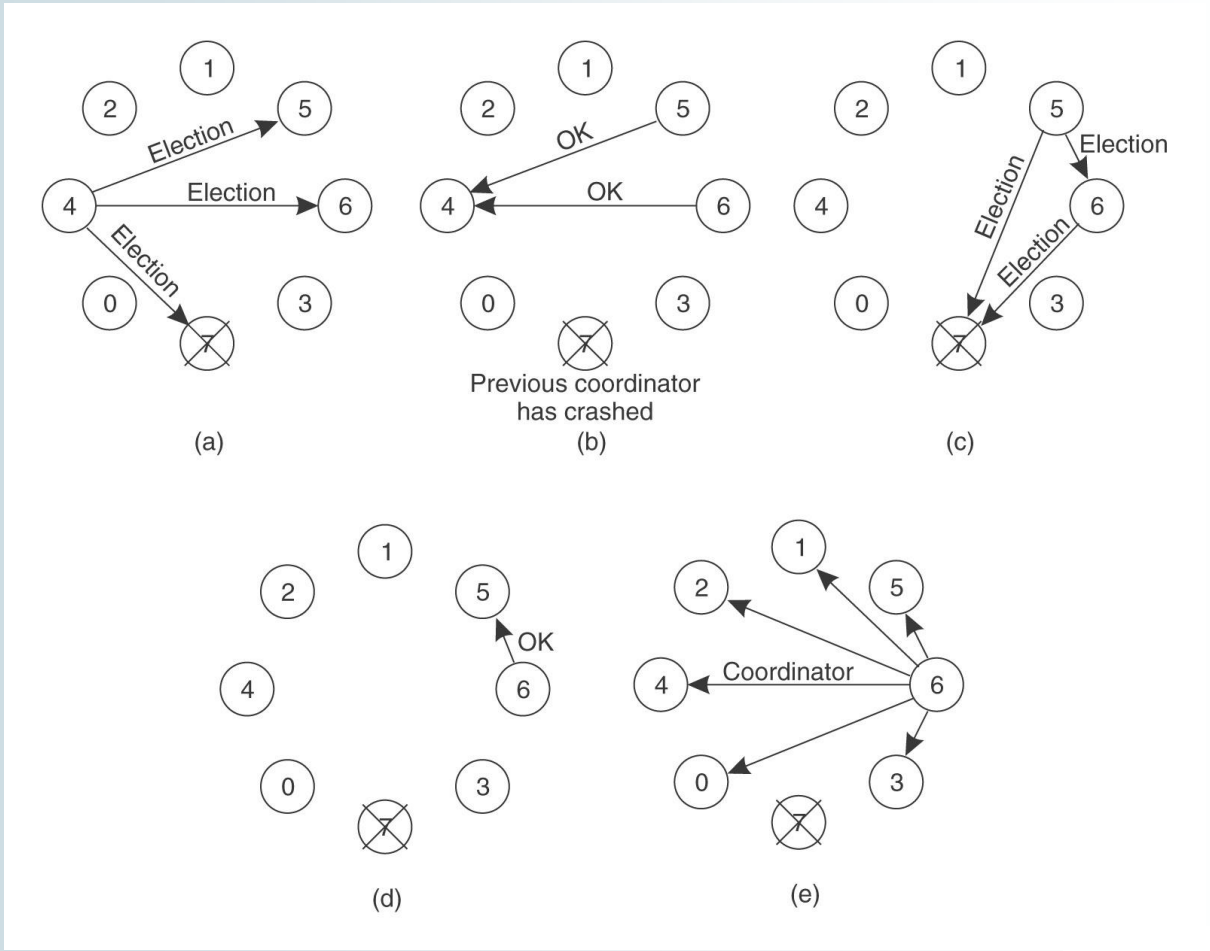
Bully Algorithm

- Assume
  - All processes know about each other
  - Processes numbered uniquely
- Suppose *P* notices no coordinator
  - Sends *election message* to all higher numbered processes
  - If none response, *P* takes over as coordinator
  - If any responds, *P* yields
- …

Bully Algorithm (continued)

- ...
- Suppose *Q* receives *election message*
  - Replies *OK* to sender, saying it will take over
  - Sends a new *election message* to higher numbered processes
- Repeat until only one process left standing
  - Announces victory by sending message saying that it is coordinator

Bully Algorithm (continued)



Bully Algorithm (continued)

- ...
- Suppose *R* comes back on line
  - Sends a new *election message* to higher numbered processes
- Repeat until only one process left standing
  - Announces victory by sending message saying that it is coordinator (if not already coordinator)
- Existing (lower numbered) coordinator yields
  - Hence the term “bully”

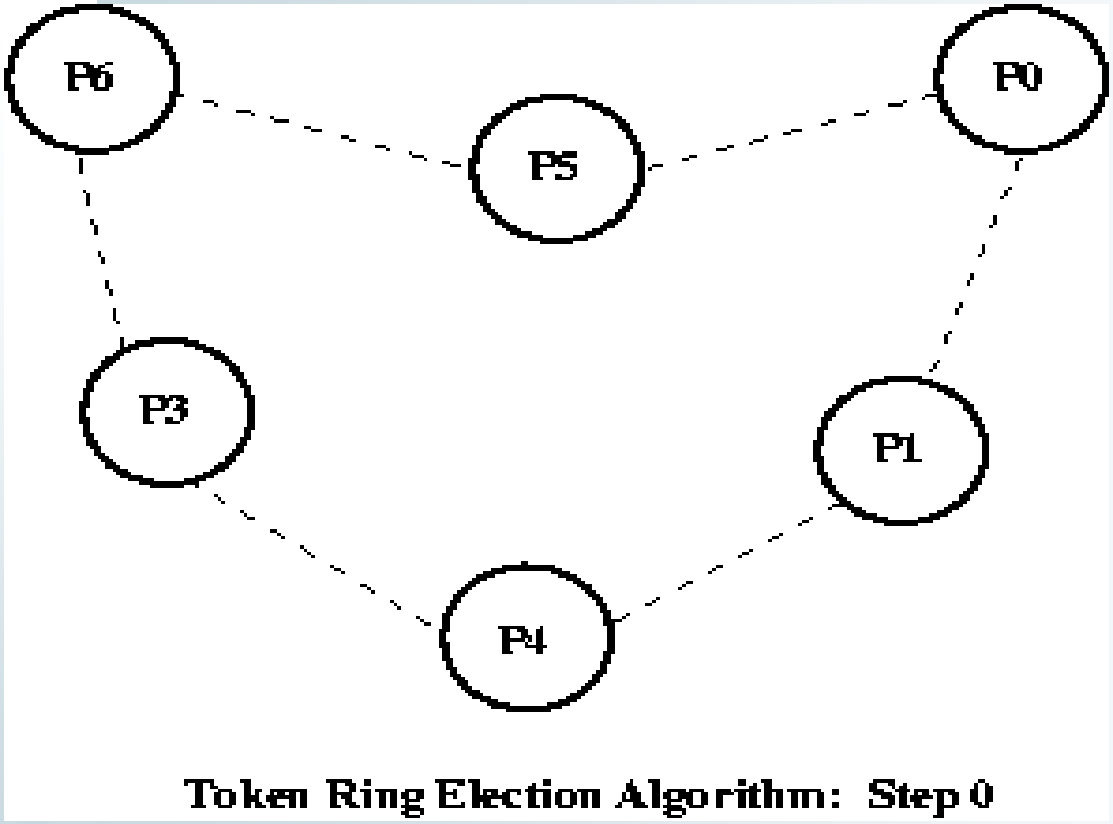
Alternative – Ring Algorithm

- All processed organized in ring
  - Independent of process number
- Suppose *P* notices no coordinator
  - Sends *election message* to successor with own process number in body of message
  - (If successor is down, skip to next process, etc.)
- Suppose *Q* receives an election message
  - Adds own process number to list in message body
- ...

Alternative – Ring Algorithm

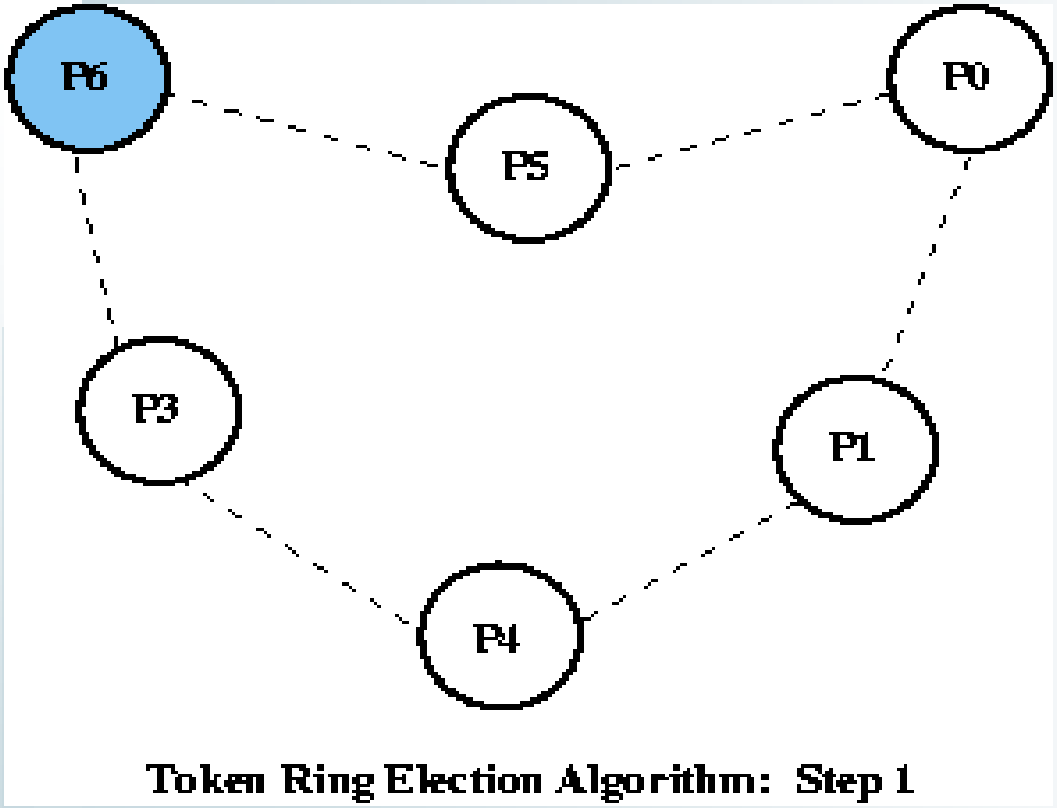
- Suppose *P* receives an election message with its own process number in body
  - Changes message to *coordinator* message, preserving body
  - All processes recognize *highest numbered process* as new coordinator
- If multiple messages circulate …
  - …they will all contain same list of processes (eventually)
- If process comes back on-line
  - Calls new election

Ring Algorithm (continued)



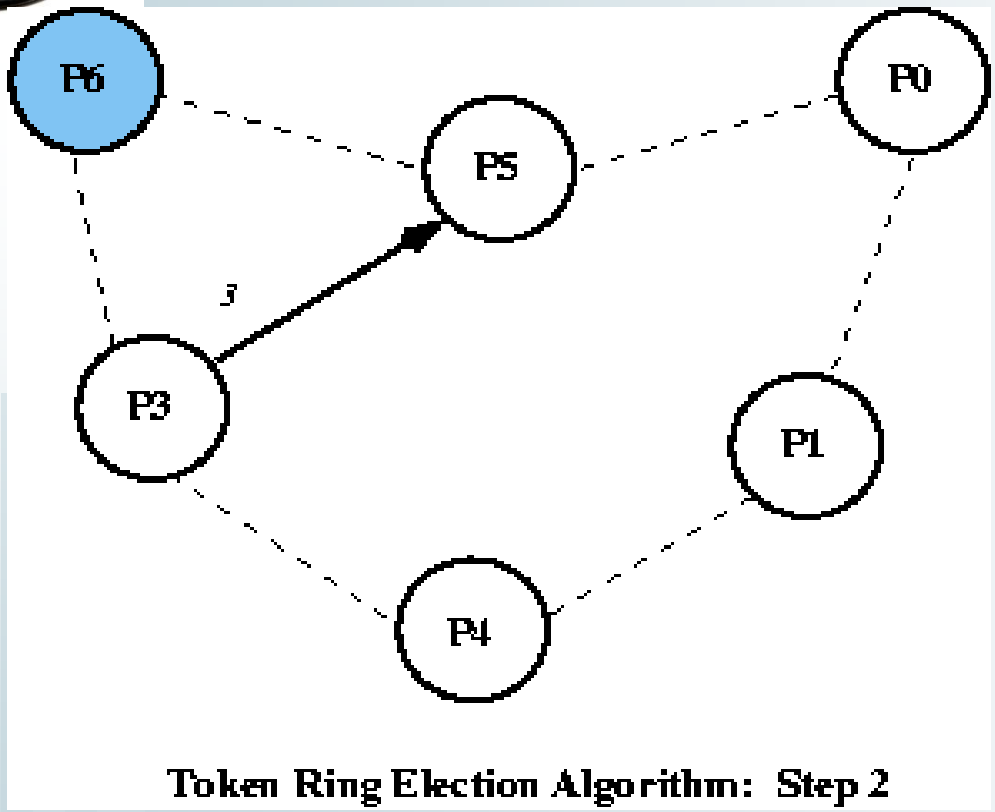
We start with 6 processes,
connected in a logical ring.
Process 6 is the leader,
as it has the highest number.

Ring Algorithm (continued)



Process 6 fails

Ring Algorithm (continued)

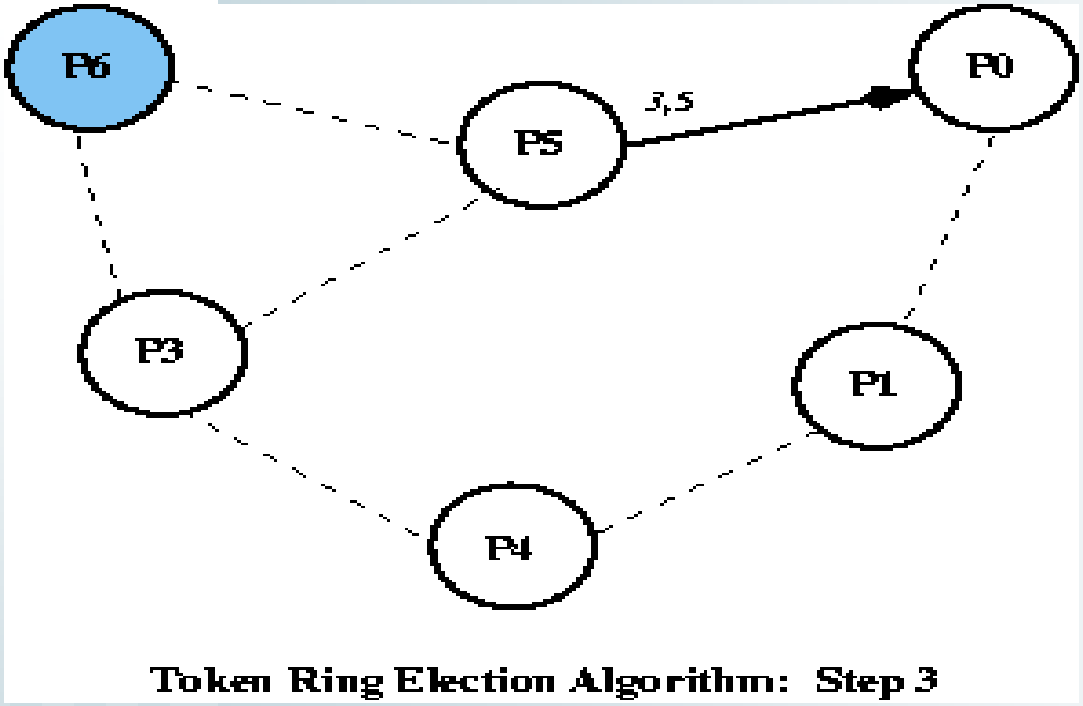


Token Ring Election Algorithm: Step 2

Process 3 notices that Process 6 does not respond

So it starts an election, sending a message containing its id to the next node in the ring.

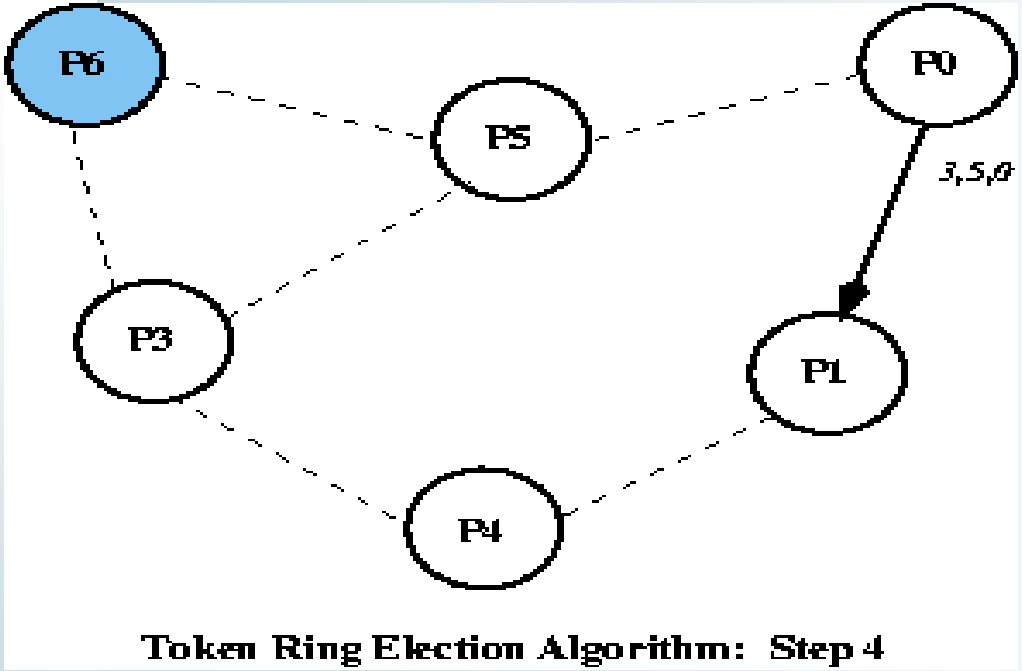
Ring Algorithm (continued)



Token Ring Election Algorithm: Step 3

Process 5 passes the message on, adding its own id to the message.

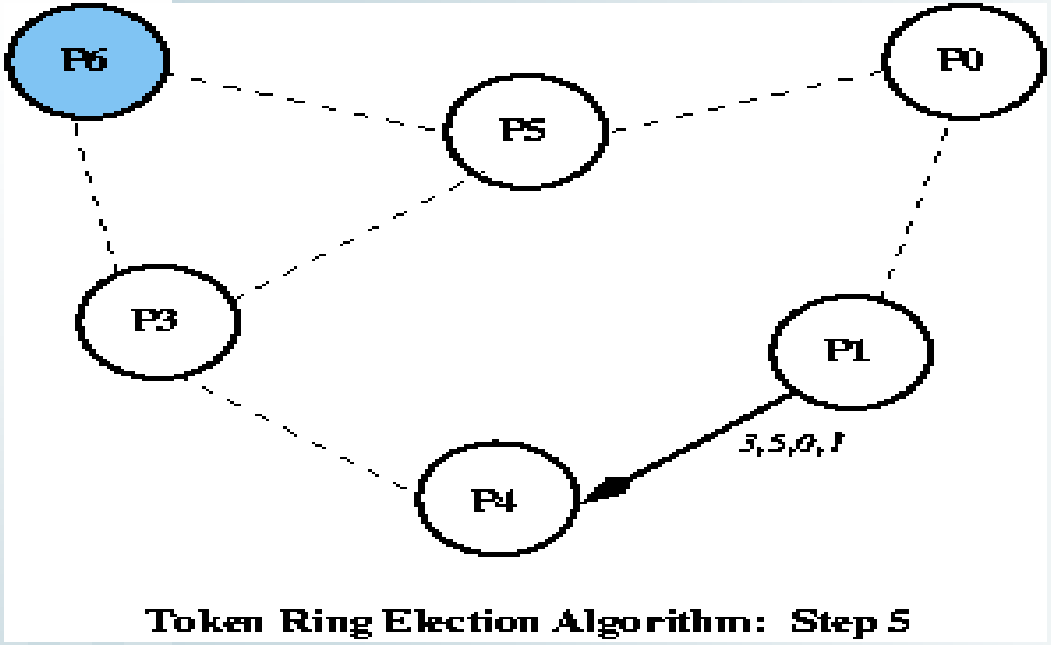
Ring Algorithm (continued)



Token Ring Election Algorithm: Step 4

Process 0 passes the message on,
adding its own id to the message.

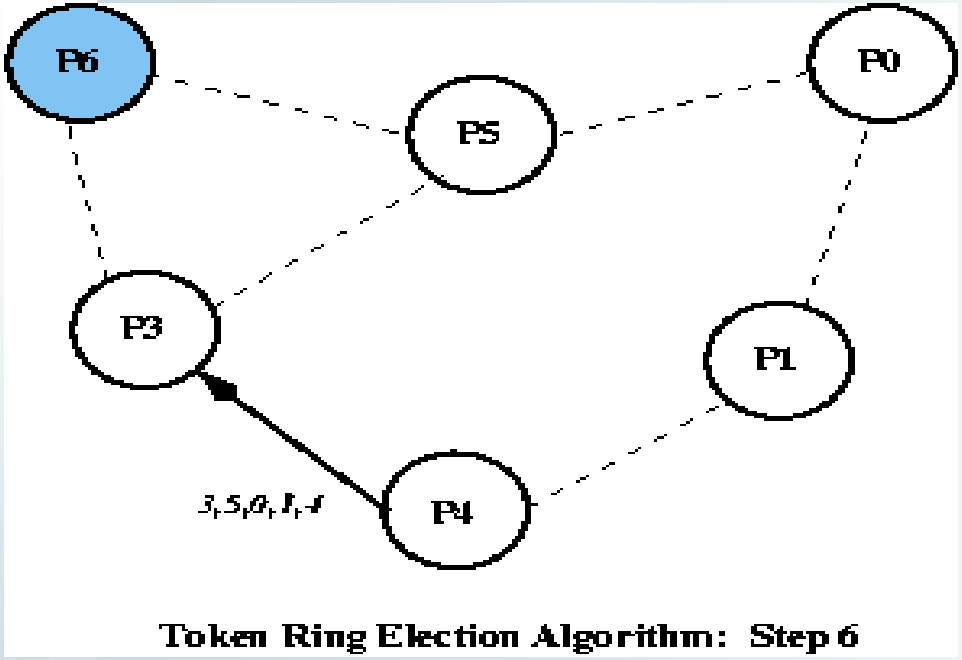
Ring Algorithm (continued)



Token Ring Election Algorithm: Step 5

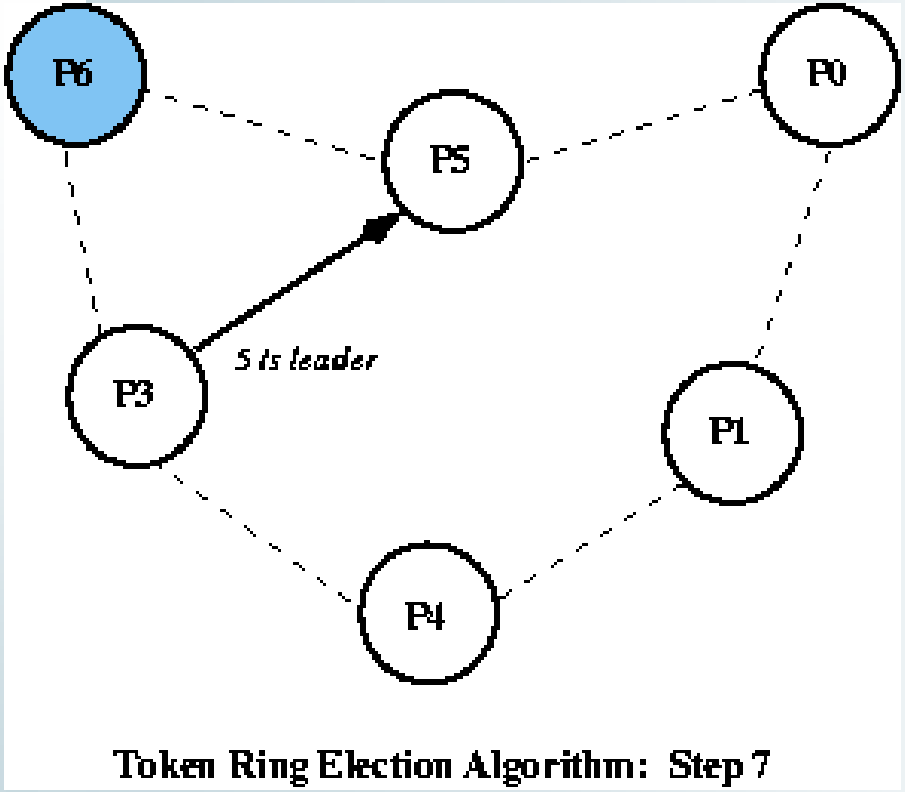
Process 1 passes the message on,
adding its own id to the message.

Ring Algorithm (continued)



Process 4 passes the message on, adding its own id to the message.

Ring Algorithm (continued)



Token Ring Election Algorithm: Step 7

When Process 3 receives the message back,

it knows the message has gone around the ring,

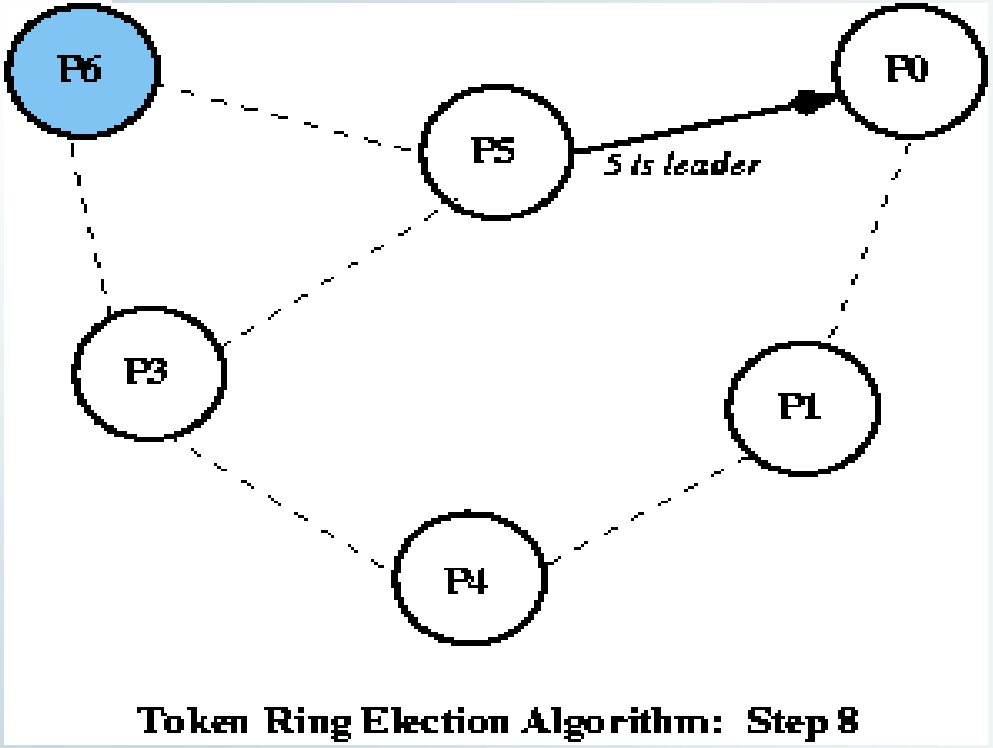
as its own id is in the list.

Picking the highest id in the list,

it starts the coordinator message

"5 is the leader" around the ring.

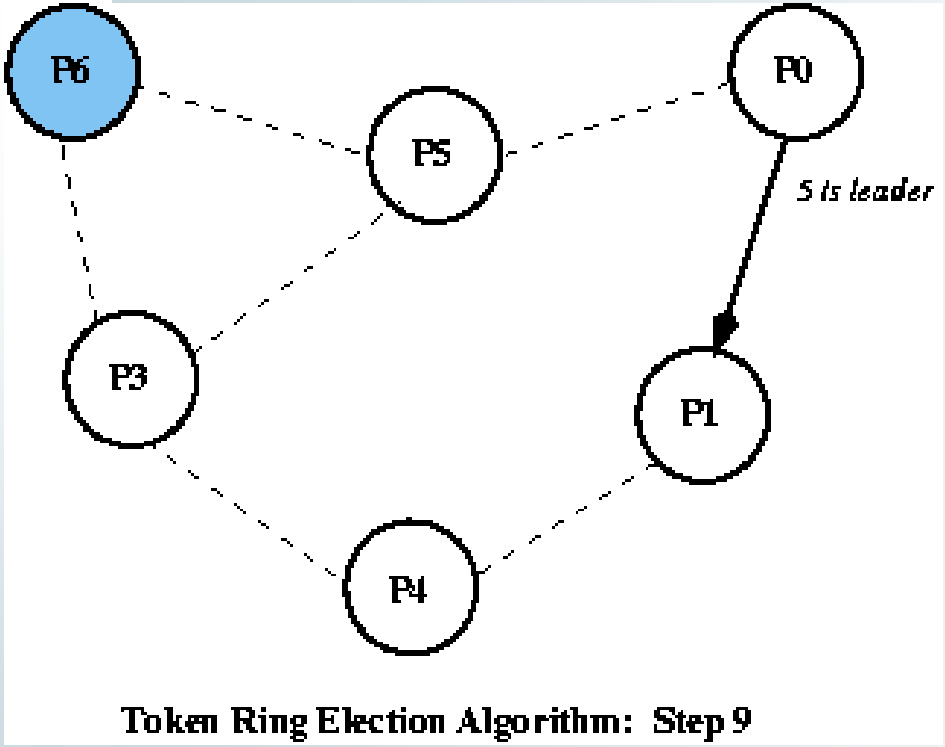
Ring Algorithm (continued)



Token Ring Election Algorithm: Step 8

Process 5 passes on the coordinator message.

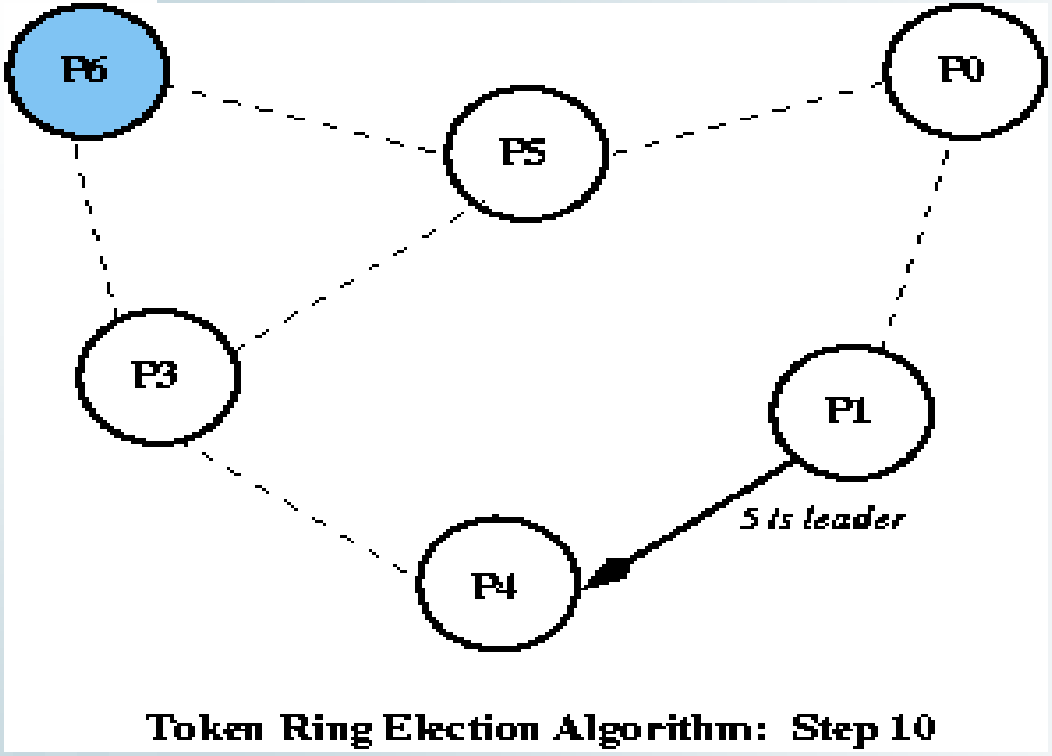
Ring Algorithm (continued)



Token Ring Election Algorithm: Step 9

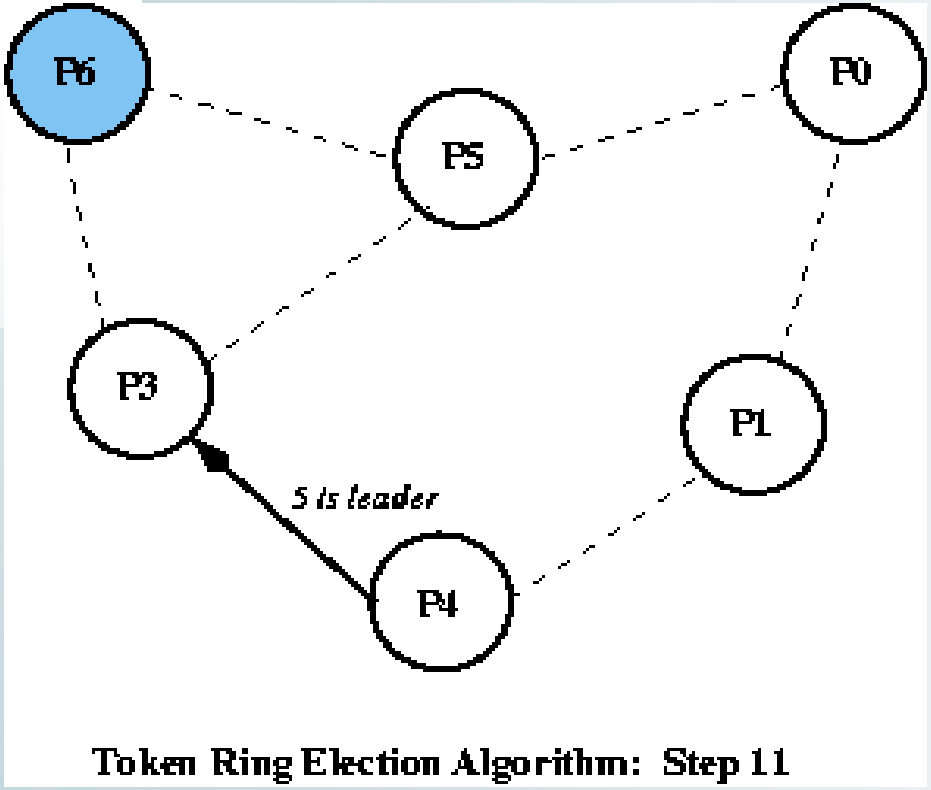
Process 0 passes on the coordinator message.

Ring Algorithm (continued)



Process 1 passes on the coordinator message.

Ring Algorithm (continued)



Token Ring Election Algorithm: Step 11

Process 4 passes on the coordinator message.

Process 3 receives the coordinator message, and stops it.

Ring Algorithm (concluded)

- Suppose *P* receives an election message with its own process number in body
  - Changes message to *coordinator* message, preserving body
  - All processes recognize *highest numbered process* as new coordinator
- If multiple messages circulate …
  - …they will all contain same list of processes (eventually)
- If process comes back on-line
  - Calls new election

Wireless Networks

- Different assumptions
  - Message passing is less reliable
  - Network topology constantly changing
- Expanding ring of broadcast
  - Election messages
  - Decision rules for when to yield
- Not very well developed.
  - Topic of current research

Very Large Scale Networks

- Sometimes more than one node should be selected
- Nodes organized as peers and *super-peers*
  - Elections held within each peer group
  - Super-peers coordinate among themselves

Distributed Deadlock Detection

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding resource(s) is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it upon its task completion.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

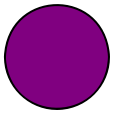
System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Resource Allocation Graph

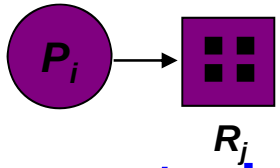
- Process



- Resource Type with 4 instances

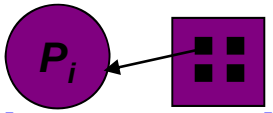
The sequence of Process's recourse utilization

- P_i requests instance of R_j



Request edge

- P_i is holding an instance of R_j

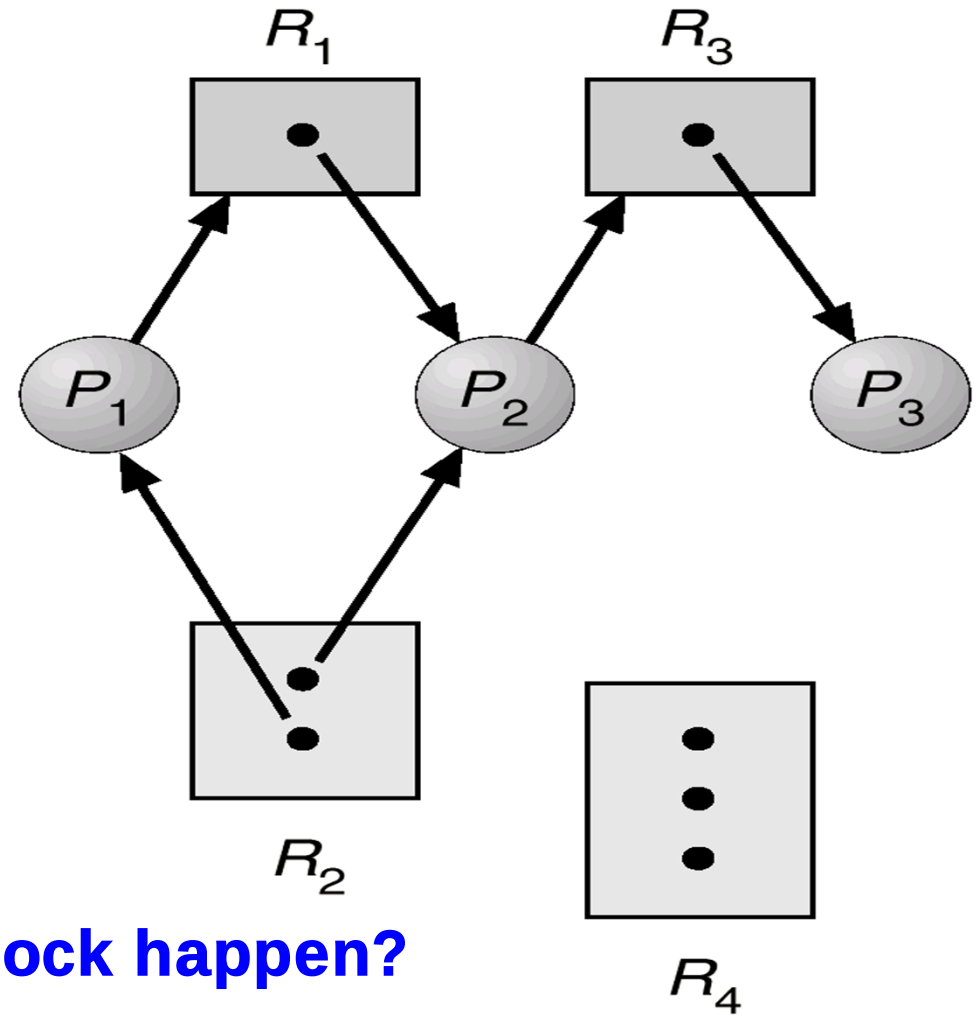


Assignment edge

- P_i releases an instance of R_j

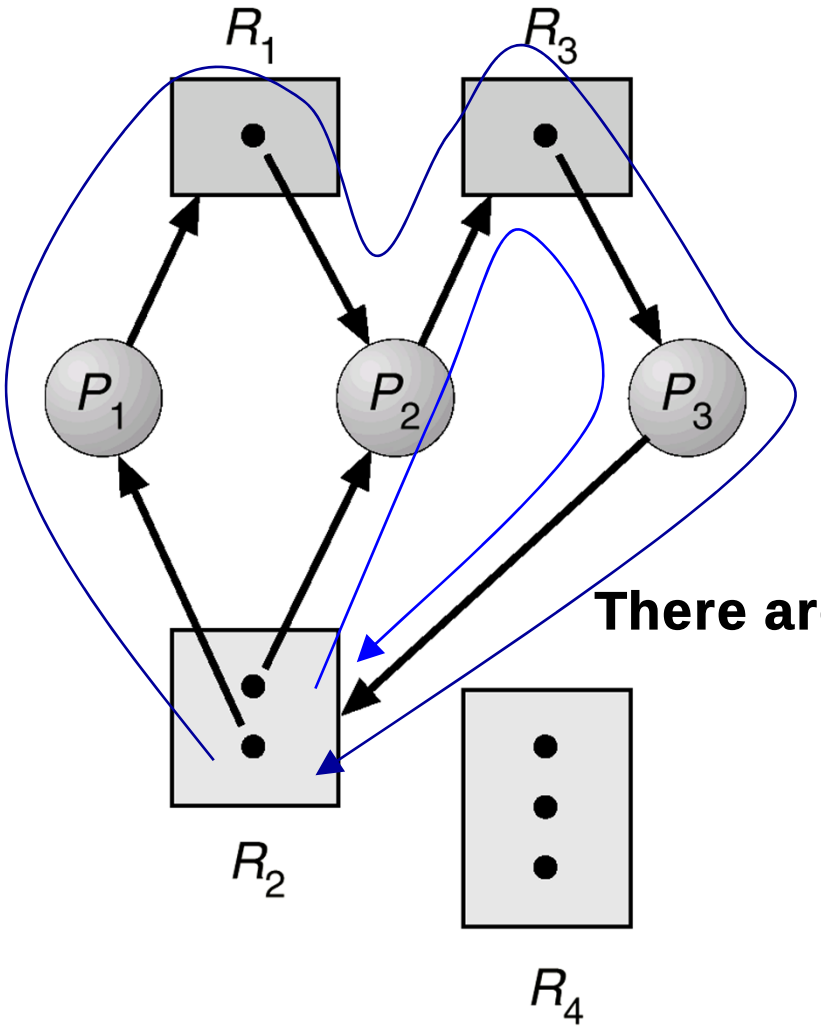
R_j

Resource-allocation graph



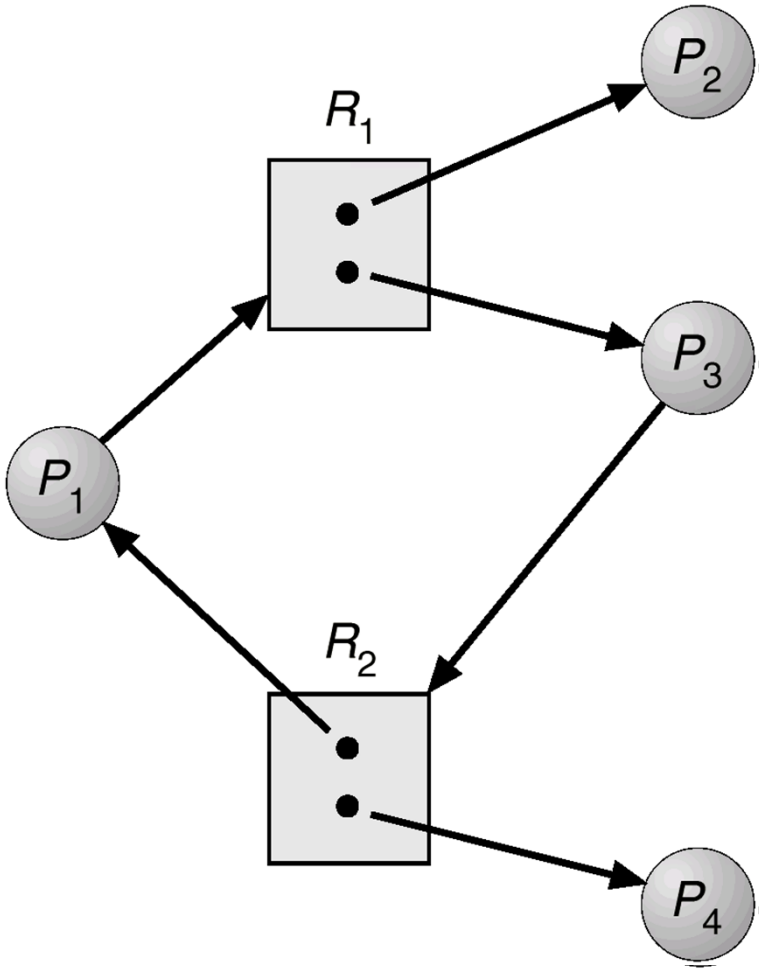
Can a deadlock happen?

Resource Allocation Graph With A Deadlock



There are two cycles found.

Resource Allocation Graph With A Cycle But No Deadlock



- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Two types of deadlocks

- Resource deadlock: uses AND condition.

  AND condition: a process that requires resources for execution can proceed when it has acquired all those resources.

- Communication deadlock: uses OR condition.

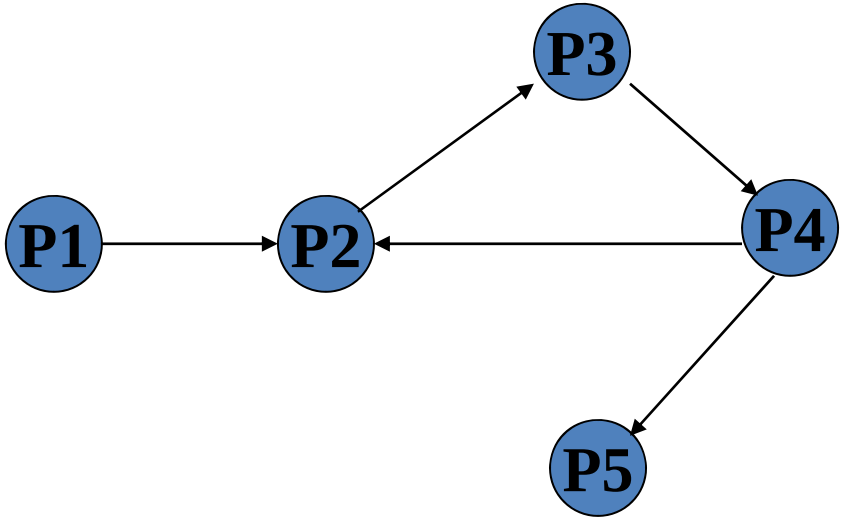
  OR condition: a process that requires resources for execution can proceed when it has acquired at least one of those resources.

Deadlock conditions

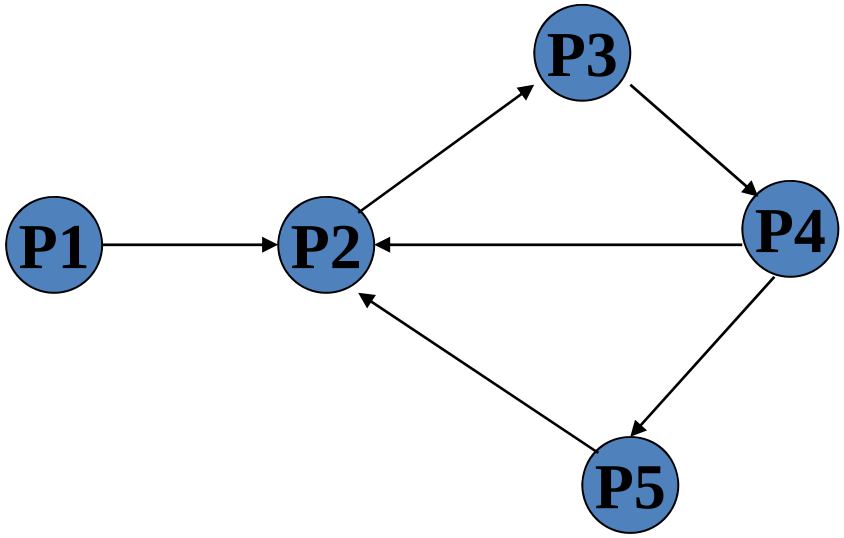
- The condition for deadlock in a system using the AND condition is the existence of a *cycle*.
- The condition for deadlock in a system using the OR condition is the existence of a *knot*.

A knot (K) consists of a set of nodes such that for every node a in K, all nodes in K and only the nodes in K are reachable from node a.

Example: OR condition



No deadlock



Deadlock

DS Deadlock Detection

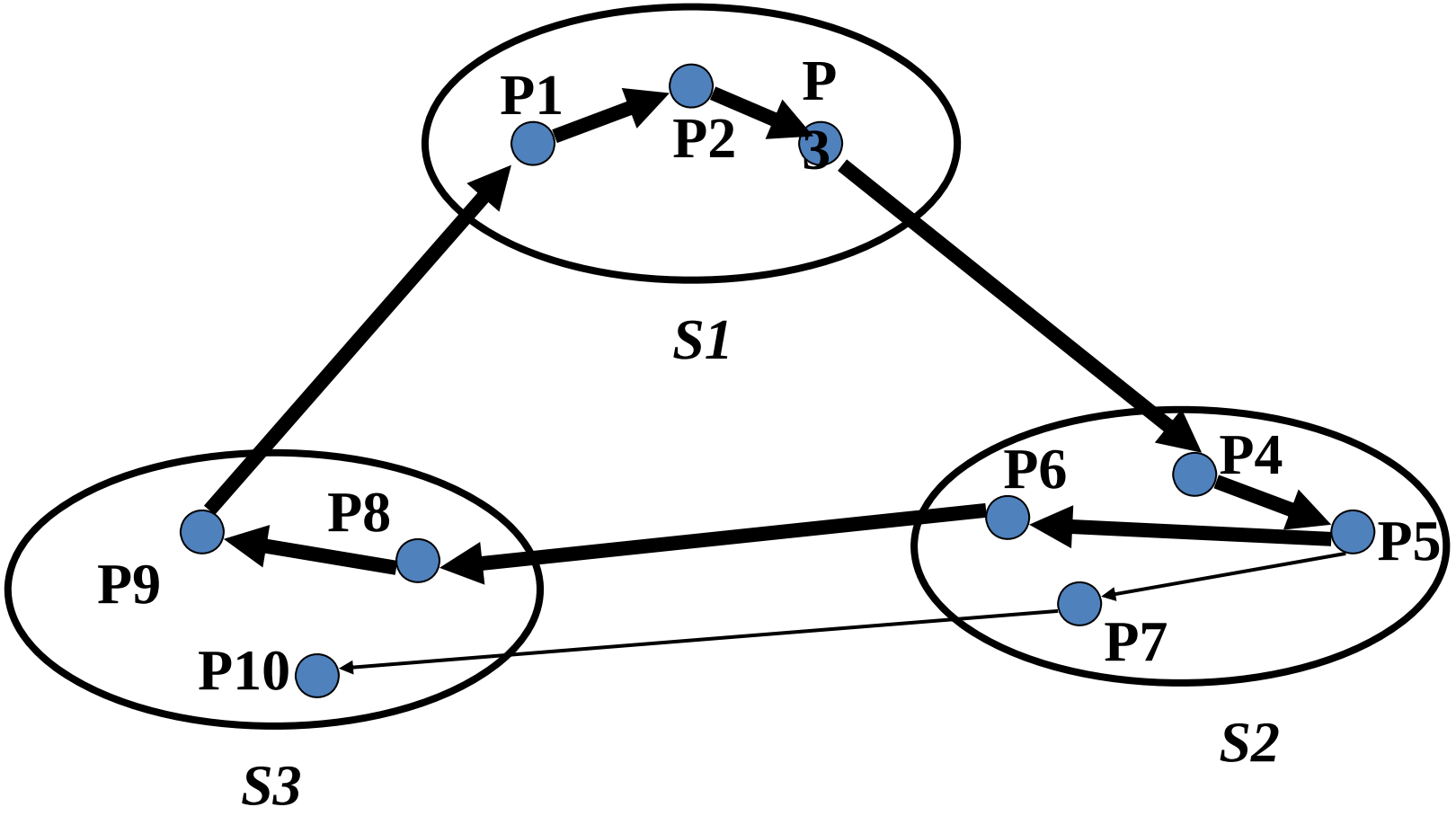
- Bi-partite graph strategy modified
  - Use Wait For Graph (WFG or TWF)
    - All nodes are processes (threads)
    - Resource allocation is done by a process (thread) sending a request message to another process (thread) which manages the resource (client - server communication model, RPC paradigm)
  - A system is deadlocked IFF there is a directed cycle (or knot) in a global WFG

DS Deadlock Detection, Cycle vs. Knot

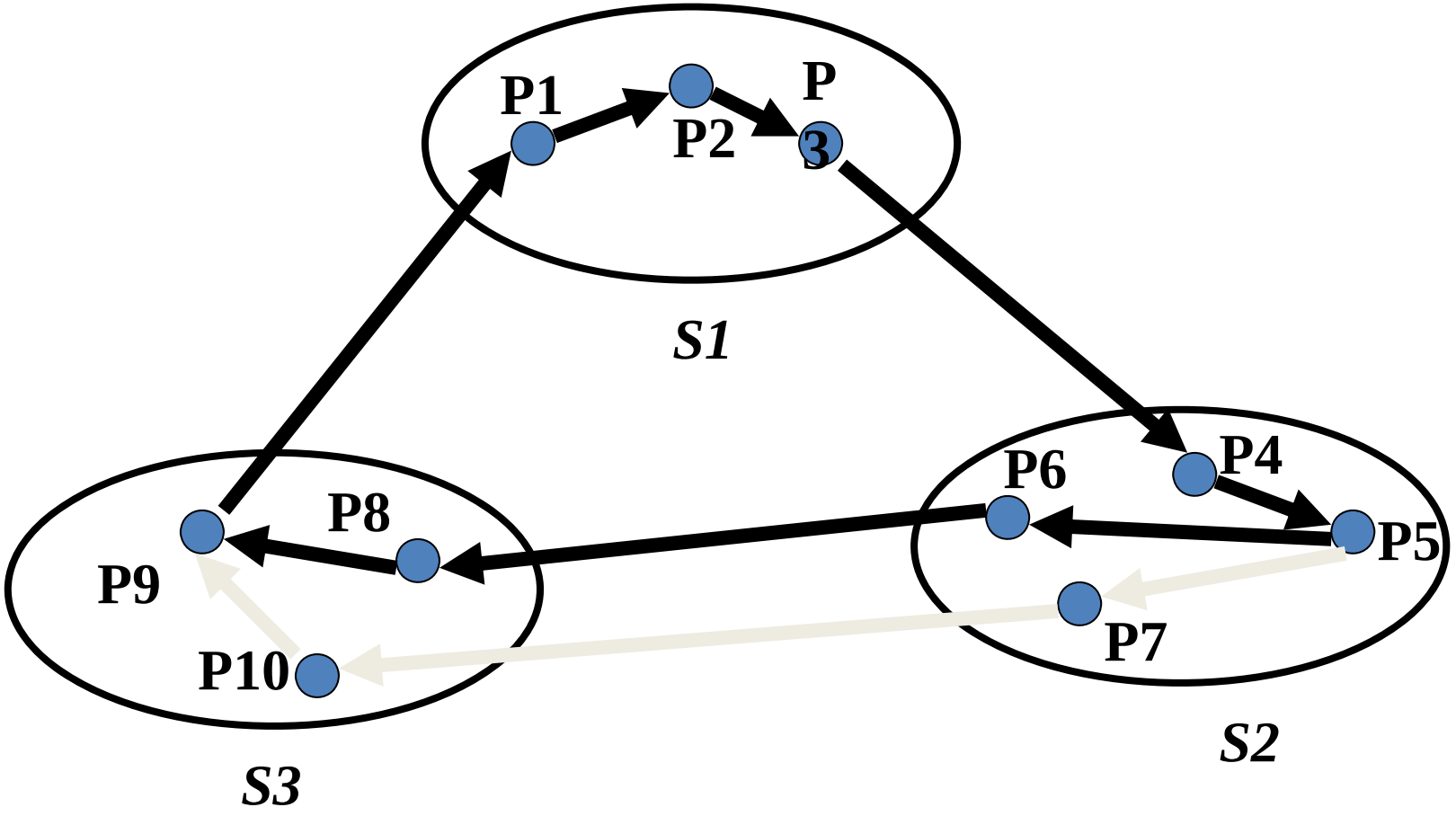
- The AND model of requests requires all resources currently being requested to be granted to un-block a computation
  - A **cycle is sufficient** to declare a deadlock with this model
- The OR model of requests allows a computation making multiple different resource requests to un-block as soon as any are granted
  - A **cycle** is a **necessary** condition
  - A **knot** is a **sufficient** condition

Deadlock in the AND model; there is a cycle but no knot

No Deadlock in the OR model



Deadlock in both the AND model and the OR model; there are cycles and a knot



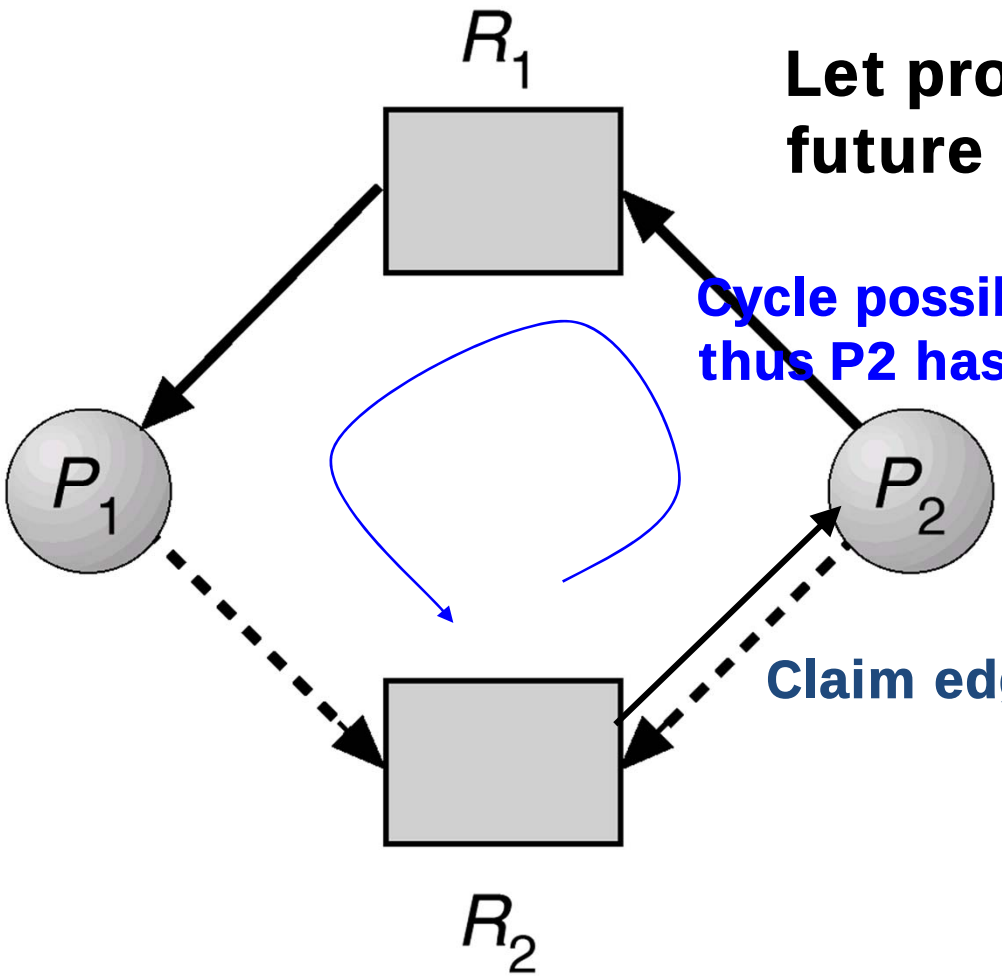
Deadlock Handling Strategies

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

Distributed Deadlock Prevention

- A method that might work is to order the resources and require processes to acquire them in strictly increasing order. This approach means that a process can never hold a high resource and ask for a low one, thus making cycles impossible.
- With global timing and transactions in distributed systems, two other methods are possible -- both based on the idea of assigning each transaction a global timestamp at the moment it starts.
- When one process is about to block waiting for a resource that another process is using, a check is made to see which has a larger timestamp.
- We can then allow the wait only if the waiting process has a lower timestamp.
- The timestamp is always increasing if we follow any chain of waiting processes, so cycles are impossible --- we can used decreasing order if we like.
- It is wiser to give priority to old processes because
  - they have run longer so the system have larger investment on these processes.
  - they are likely to hold more resources.
  - A young process that is killed off will eventually age until it is the oldest one in the system, and that eliminates starvation.

Deadlock Avoidance



Let processes supply OS with future resource requests

Cycle possibly formed (unsafe state), thus P2 has to wait for a safe state

Claim edge (future request)

Control Organization for Deadlock Detection

- Centralized Control
- Distributed Control
- Hierarchical Control

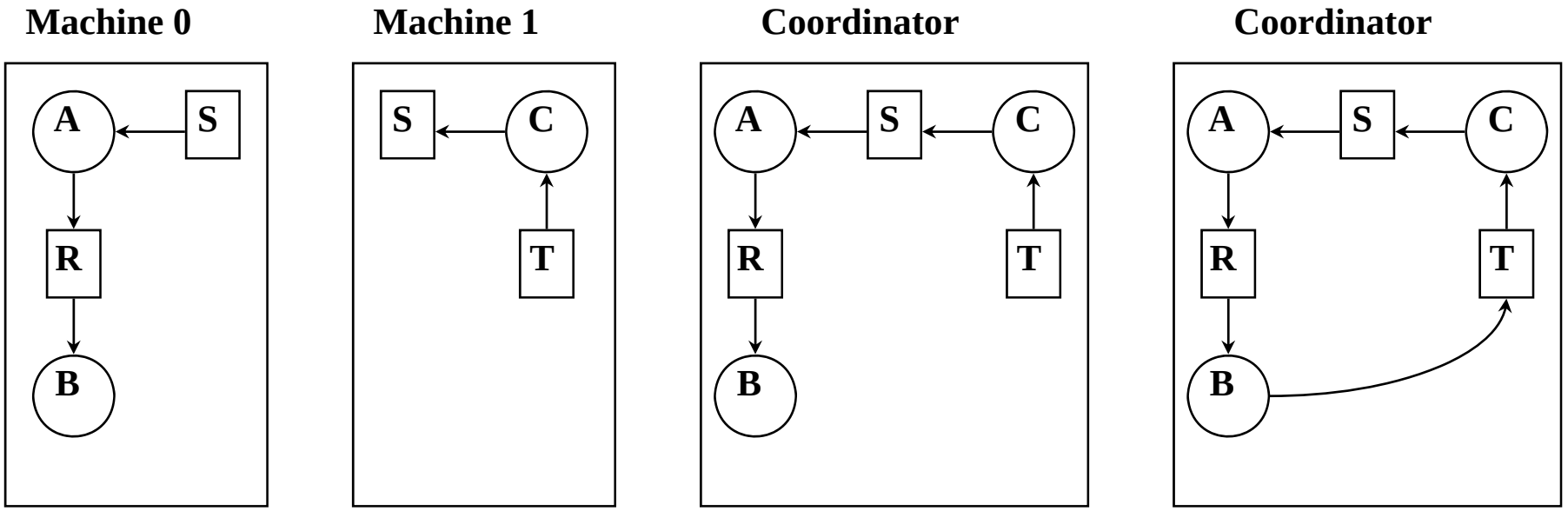
Issues in Deadlock Detection & Resolution

- **Detection**
  - **Progress: No undetected deadlocks**
  - **Safety: No false deadlocks**
- **Resolution**

Centralized Deadlock Detection

- We use a centralized deadlock detection algorithm and try to imitate the non-distributed algorithm.
  - Each machine maintains the resource graph for its own processes and resources.
  - A centralized coordinator maintain the resource graph for the entire system.
  - When the coordinator detect a cycle, it kills off one process to break the deadlock.
  - In updating the coordinator's graph, messages have to be passed.
    - Method 1) Whenever an arc is added or deleted from the resource graph, a message have to be sent to the coordinator.
    - Method 2) Periodically, every process can send a list of arcs added and deleted since previous update.
    - Method 3) Coordinator ask for information when it needs it.

False Deadlocks



- One possible way to prevent false deadlock is to use the Lamport's algorithm to provide global timing for the distributed systems.
- When the coordinator gets a message that leads to a suspect deadlock:
  - It send everybody a message saying "I just received a message with a timestamp T which leads to deadlock. If anyone has a message for me with an earlier timestamp, please send it immediately"
  - When every machine has replied, positively or negatively, the coordinator will see that the deadlock has really occurred or not.

Centralized Deadlock-Detection Algorithms

- The Ho-Ramamoorthy Algorithms
  - The Two-Phase Algorithm
  - The One-phase Algorithm

Centralized Algorithms

- Ho-Ramamoorthy 2-phase Algorithm
  - Each site maintains a status table of all processes initiated at that site: includes all resources locked & all resources being waited on.
  - Controller requests (periodically) the status table from each site.
  - Controller then constructs WFG from these tables, searches for cycle(s).
  - If no cycles, no deadlocks.
  - Otherwise, (cycle exists): Request for state tables again.
  - Construct WFG based *only* on common transactions in the 2 tables.
  - If the same cycle is detected again, system is in deadlock.
  - Later proved: cycles in 2 consecutive reports *need not* result in a deadlock. Hence, this algorithm detects false deadlocks.

Centralized Algorithms...

- Ho-Ramamoorthy 1-phase Algorithm
  - Each site maintains 2 status tables: *resource status* table and *process status* table.
  - Resource table: transactions that have locked or are waiting for resources.
  - Process table: resources locked by or waited on by transactions.
  - Controller periodically collects these tables from each site.
  - Constructs a WFG from transactions common to both the tables.
  - No cycle, no deadlocks.
  - A cycle means a deadlock.

Distributed Deadlock-Detection Algorithms

- A Path-Pushing Algorithm
  - The site waits for deadlock-related information from other sites
  - The site combines the received information with its local TWF graph to build an updated TWF graph
  - For all cycles ‘EX -> T1 -> T2 -> Ex’ which contains the node ‘Ex’, the site transmits them in string form ‘Ex, T1, T2, Ex’ to all other sites where a sub-transaction of T2 is waiting to receive a message from the sub-transaction of T2 at that site

Edge-Chasing Algorithm

- Chandy-Misra-Haas's Algorithm:
  - A probe(i, j, k) is used by a deadlock detection process Pi. This probe is sent by the home site of Pj to Pk.
  - This probe message is circulated via the edges of the graph. Probe returning to Pi implies deadlock detection.
  - Terms used:
    - Pj is *dependent* on Pk, if a sequence of Pj, Pi1,.., Pim, Pk exists.
    - Pj is *locally dependent* on Pk, if above condition + Pj,Pk on same site.
    - Each process maintains an array *dependenti: dependenti(j)* is true if Pi knows that Pj is dependent on it. (initially set to false for all i & j).

Chandy-Misra-Haas's Algorithm

Sending the probe:

if Pi is locally dependent on itself then deadlock.
else for all Pj and Pk such that
(a) Pi is locally dependent upon Pj, and
(b) Pj is waiting on Pk, and
(c) Pj and Pk are on different sites, send probe(i,j,k) to the home site of Pk.

Receiving the probe:

if (d) Pk is blocked, and
(e) ***dependentk(i)*** **is false, and**
(f) Pk has not replied to all requests of Pj,
then begin
dependentk(i) := true;
if k = i then Pi is deadlocked
else ...

Chandy-Misra-Haas's Algorithm

Receiving the probe:

…….

else for all Pm and Pn such that

(a') Pk is locally dependent upon Pm, and

(b') Pm is waiting on Pn, and

(c') Pm and Pn are on different sites, send probe(i,m,n) to the home site of Pn.

end.

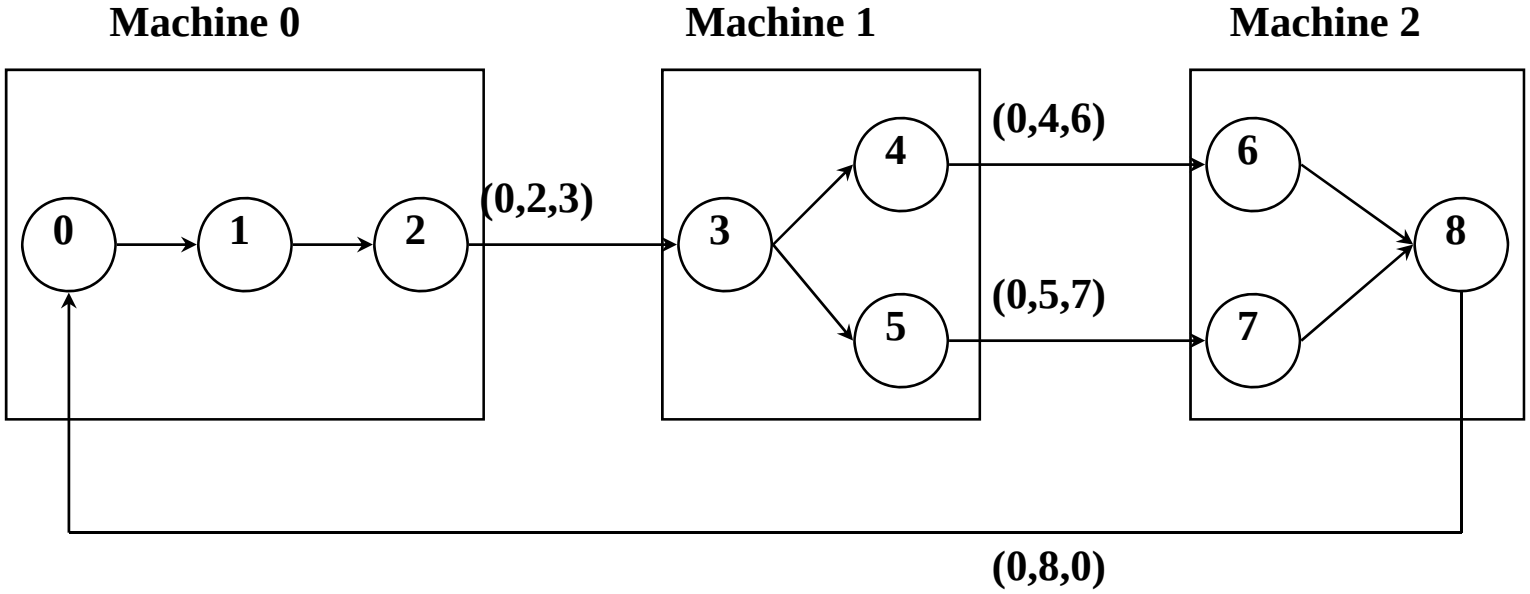
Performance:

For a deadlock that spans m processes over n sites, m(n-1)/2 messages are needed.

Size of the message 3 words.

Delay in deadlock detection O(n).

Chandy-Misra-Haas Algorithm



- There are several ways to break the deadlock:
  - The process that initiates commit suicide -- this is overkilling because several process might initiates a probe and they will all commit suicide in fact only one of them is needed to be killed.
  - Each process append its id onto the probe, when the probe come back, the originator can kill the process which has the highest number by sending him a message. (Even for several probes, they will all choose the same guy)

Other Edge - Chasing Algorithms

- The Mitchell – Merritt Algorithm
- Sinha – Niranjan Algorithm

Chandy et al.'s Diffusion Computation Based Algo

- Initiate a diffusion computation for a blocked process P_i:

  send query (*i, i, j*) to each process P_j in the

  dependent set DS_i of P_i;

  $num_i(i) := |DS_i|$; $wait_i(i) := true$

- When a blocked process P_k receives a query (*i, j, k):*

  if this is the engaging query for process P_k then

  send *query (i, k, m)* to all P_m in its dependent set DS_k;

  $num_k(i) := |DS_k|$; $wait_k(i) := true$

  else if $wait_k(i)$ then send a *reply (i, k, j)* to P_j.

Chandy et al.'s Algo. Contd.

- When a process P_k receives a reply (*i, j, k*):

  if $wait_k(i)$ then begin $num_k(i) := num_k(i) - 1$;

  if $num_k(i) = 0$

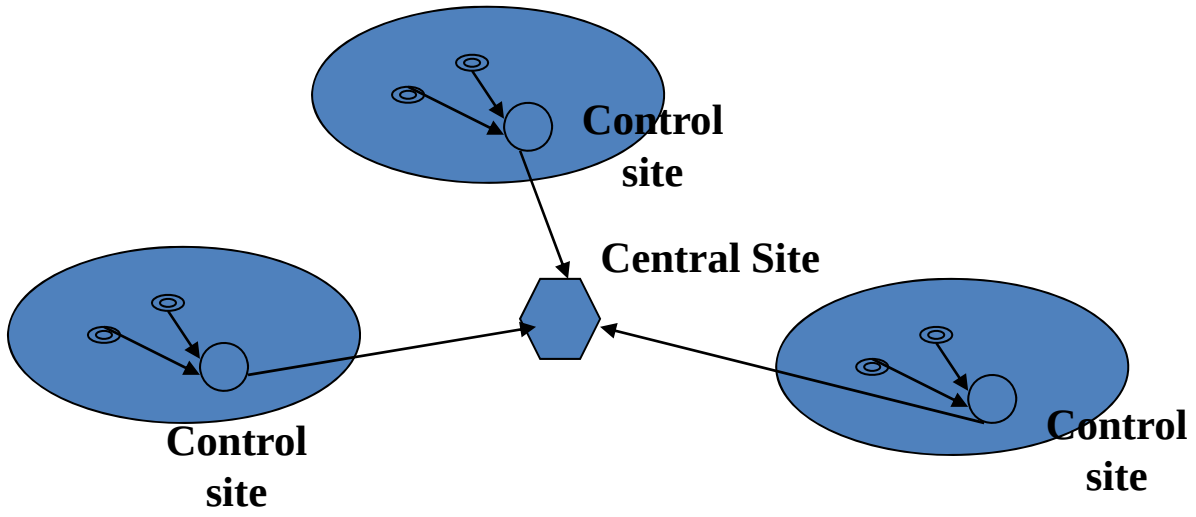
  then if *i = k* then *declare a deadlock*

  else send *reply (i, k, m)* to the process P_m which

  sent the engaging query

Hierarchical Deadlock Detection

- **Follows Ho-Ramamoorthy's 1-phase algorithm. More than 1 control site organized in hierarchical manner.**
- **Each control site applies 1-phase algorithm to detect (intracluster) deadlocks.**
- **Central site collects info from control sites, applies 1-phase algorithm to detect intracluster deadlocks.**



Persistence & Resolution

- Deadlock persistence:
  - Average time a deadlock exists before it is resolved.
- Implication of persistence:
  - Resources unavailable for this period: affects utilization
  - Processes wait for this period unproductively: affects response time.
- Deadlock resolution:
  - Aborting at least one process/request involved in the deadlock.
  - Efficient resolution of deadlock requires knowledge of all processes and resources.
  - If every process detects a deadlock and tries to resolve it independently -> highly inefficient ! Several processes might be aborted.

Deadlock Resolution

- Priorities for processes/transactions can be useful for resolution.
  - Consider priorities introduced in Obermarck's algorithm.
  - Highest priority process initiates and detects deadlock (initiations by lower priority ones are suppressed).
  - When deadlock is detected, lowest priority process(es) can be aborted to resolve the deadlock.
- After identifying the processes/requests to be aborted,
  - All resources held by the victims must be released. State of released resources restored to previous states. Released resources granted to deadlocked processes.
  - All deadlock detection information concerning the victims must be removed at all the sites.

The End / OR is it deadlock?

- We are now entering the *idle state* , waiting for a message from any of the other processes in the room !
- Don’t make us send out probes!

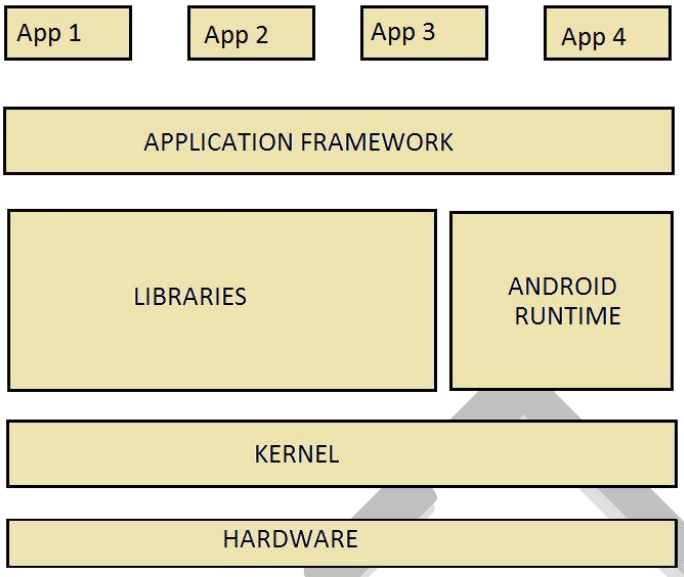
Mobile OS

A mobile operating system (or mobile OS) is an operating system for smartphones, tablets, PDAs, or other mobile devices. While computers such as the typical laptop are mobile, the operating systems usually used on them are not considered mobile ones as they were originally designed for bigger stationary desktop computers that historically did not have or need specific "mobile" features. This distinction is getting blurred in some newer operating systems that are hybrids made for both uses.

Mobile operating systems combine features of a personal computer operating system with other features useful for mobile or handheld use; usually including, and most of the following considered essential in modern mobile systems; a touchscreen, cellular, Bluetooth, Wi-Fi, GPS mobile navigation, camera, video camera, speech recognition, voice recorder, music player, near field communication and infrared blaster.

Mobile devices with mobile communications capabilities (e.g. smartphones) contain two mobile operating systems – the main user-facing software platform is supplemented by a second low-level proprietary real-time operating system which operates the radio and other hardware. Research has shown that these low-level systems may contain a range of security vulnerabilities permitting malicious base stations to gain high levels of control over the mobile device.

A mobile OS is a software platform on top of which other programs called application programs, can run on mobile devices such as PDA, cellular phones, smartphone and etc. A Mobile operating system is a System Software that is specifically designed to run on handheld devices such as Mobile Phones, PDA's. It is a Platform on top of which the application programs run on mobile devices. Each Operating System follows its own Architecture. Mobile devices evolved the way users across the globe leverage services on the go from voice calls to smart devices which enables users to access value added services anytime and anywhere. At present, the mobile devices are able to provide various services to users but still suffers from issues include Performance, security and Privacy, Reliability and Band width costs. In this paper, we pointed out the issues, challenges, Advantages and Disadvantages of various Mobile Operating systems in terms of their Architectures.



Applications

The diagram shows four basic apps (App 1, App 2, App 3 and App 4), just to give the idea that there can be multiple apps sitting on top of Android. These apps are like any user interface you use on Android; for example, when you use a music player, the GUI on which there are buttons to play, pause, seek, etc is an application. Similarly, is an app for making calls, a camera app, and so on. All these apps are not necessarily from Google. Anyone can develop an app and make it available to everyone through Google Play Store. These apps are developed in Java, and are installed directly, without the need to integrate with Android OS.

Application Framework

Scratching further below the applications, we reach the application framework, which application developers can leverage in developing Android applications. The framework offers a huge set of APIs used by developers for various standard purposes, so that they don't have to code every basic task.The framework consists of certain entities; major ones are:

Activity Manager

This manages the activities that govern the application life cycle and has several states. An application may have multiple activities, which have their own life cycles. However, there is one main activity that starts when the application is launched. Generally, each activity in an

application is given a window that has its own layout and user interface. An activity is stopped when another starts, and gets back to the window that initiated it through an activity callback.

- Notification Manager

This manager enables the applications to create customized alerts

- Views

Views are used to create layouts, including components such as grids, lists, buttons, etc.

- Resource Managers

Applications do require external resources, such as graphics, external strings, etc. All these resources are managed by the resource manager, which makes them available in a standardized way.

- Content Provider

Applications also share data. From time to time, one application may need some data from another application. For example, an international calling application will need to access the user's address book. This access to another application's data is enabled by the content providers.

Libraries

This layer holds the Android native libraries. These libraries are written in C/C++ and offer capabilities similar to the above layer, while sitting on top of the kernel. A few of the major native libraries include

- Surface Manager: Manages the display and compositing window-ing manager. - Media framework: Supports various audio and video formats and codecs including their playback and recording.
- System C Libraries: Standard C library like libc targeted for ARM or embedded devices.
- OpenGL ES Libraries : These are the graphics libraries for rendering 2D and 3D graphics.
- SQLite : A database engine for Android.

Kernel

The Android OS is derived from Linux Kernel 2.6 and is actually created from Linux source, compiled for mobile devices. The memory management, process management etc. are mostly similar. The kernel acts as a Hardware Abstraction Layer between hardware and the Android software stack.

Mobile OS Special Constraints:

Smaller screen size	Stay focused on the user's immediate task. Display only the information that users need at any given moment. For example, a customer relationship management system can provide a massive amount of information, but users only require a small amount of that information at one time. Design the UI so that users can perform tasks easily and access information quickly.
One screen appears at a time	Use a single screen if possible. If your application requires multiple screens to be open at the same time, use a split screen or rethink the flow of your application.
Shorter battery life	Try to handle data transmission efficiently. The less often the device needs to transmit data, the longer the battery lasts.
Wireless network connections	Try to simplify how your application creates network connections. Compared with standard LANs, longer latency periods that are inherent in some wireless network connections can influence how quickly users receive information that is sent over the network.
Slower processor speeds	Avoid processor-intensive tasks where possible. Slower processor speeds can affect how users perceive the responsiveness of an application.
Less available memory	Free up as much memory as possible. For example, while an application is not being used, try to keep it from using memory.

Special Service Requirements

Support for specific communication protocol
Support for a variety of input mechanisms
Compliance with open standards
Extensive library support
Support for Integrated Development Environment

Commercial Mobile OS

Smartphones are now participating nearly in each and every sphere of life like business, education, workplace and healthcare. The Worldwide Mobile Communications Device Open Operating System Sales (WMCDOOS) provides total market of 104,898 to End Users by OS. There are over 1.3 million active applications in Google Play App Store. Android is the first open source, Linux-based and modern mobile handset platform. Google developed it for handset manufacturers like T-Mobile, Sprint Nextel, Google, Intel, Samsung, etc.. It offers to consumers a richer, less expensive, better mobile experience and various features like 3D, SQLite, Connectivity, WebKit, Dalvik and FreeType etc. Since android provides open source operating system; users by Microsoft for smartphones and Pocket PCs.

Its origins dated back to Windows CE in 1996, though Windows Mobile itself first appeared in 2000 as *PocketPC 2000*. It was renamed "Windows Mobile" in 2003, at which point it came in several versions (similar to the desktop versions of Windows) and was aimed at business and enterprise consumers. By 2007, it was the most popular smartphone software in the U.S., but this popularity faded in the following years. In February 2010, facing competition from rival OSs including iOSand Android, Microsoft announced Windows Phone to supersede Windows Mobile. As a result, Windows Mobile has been deprecated. Windows Phone is incompatible with Windows Mobile devices and software. The last version of Windows Mobile, released after the announcement of Windows Phone, was 6.5.5. After this, Microsoft ceased development on Windows Mobile, in order to concentrate on Windows Phone.

Most versions of Windows Mobile have a standard set of features, such as multitasking and the ability to navigate a file system similar to that of Windows 9x andWindows NT, including support for many of the same file types. Similarly to its desktop counterpart, it comes bundled with a set of applications that perform basic tasks. Internet Explorer Mobile is the default web browser, and Windows Media Player is the default media player used for playing digital media. The mobile version of Microsoft Office, is the default office suite.

Internet Connection Sharing, supported on compatible devices, allows the phone to share its Internet connection with computers via USB and Bluetooth. Windows Mobile supports virtual private networking over PPTP protocol. Most devices with mobile connectivity also have a Radio Interface Layer. The Radio Interface Layer provides the system interface between the Cell Core layer within the Windows Mobile OS and the radio protocol stack used by the wireless modem hardware. This allows OEMs to integrate a variety of modems into their equipment.

The user interface changed dramatically between versions, only retaining similar functionality. The *Today Screen*, later called the *Home Screen*, shows the current date, owner information, upcoming appointments, e-mails, and tasks. The taskbar display the current time as well as the volume level. Devices with a cellular radio also show the signal strength on said taskbar.

Palm OS (also known as **Garnet OS**) is a mobile operating system initially developed by Palm, Inc., for personal digital assistants (PDAs) in 1996. Palm OS was designed for ease of use with a touchscreen-based graphical user interface. It is provided with a suite of basic applications for personal information management. Later versions of the OS have been extended to support smartphones. Several other licensees have manufactured devices powered by Palm OS.

Following Palm's purchase of the Palm trademark, the currently licensed version from ACCESS was renamed *Garnet OS*. In 2007, ACCESS introduced the successor to Garnet OS, called Access Linux Platform and in 2009, the main licensee of Palm OS, Palm, Inc., switched from Palm OS to webOS for their forthcoming devices.

Palm OS was originally developed under the direction of Jeff Hawkins at Palm Computing, Inc. Palm was later acquired by U.S. Robotics Corp., which in turn was later bought by 3Com, which made the Palm subsidiary an independent publicly traded company on March 2, 2000.

In January 2002, Palm set up a wholly owned subsidiary to develop and license Palm OS, which was named PalmSource. PalmSource was then spun off from Palm as an independent company on October 28, 2003. Palm (then called palmOne) became a regular licensee of Palm OS, no longer in control of the operating system.

In September 2005, PalmSource announced that it was being acquired by ACCESS.

In December 2006, Palm gained perpetual rights to the Palm OS source code from ACCESS.[9] With this Palm can modify the licensed operating system as needed without paying further royalties to ACCESS. Together with the May 2005 acquisition of full rights to the *Palm* brand name, only Palm can publish releases of the operating system under the name 'Palm OS'.

As a consequence, on January 25, 2007, ACCESS announced a name change to their current Palm OS operating system, now titled *Garnet OS.*

Palm OS is a proprietary mobile operating system. Designed in 1996 for Palm Computing, Inc.'s new Pilot PDA, it has been implemented on a wide array of mobile devices, including smartphones, wrist watches,handheld gaming consoles, barcode readers and GPS devices.

Palm OS versions earlier than 5.0 run on Motorola/Freescale DragonBall processors. From version 5.0 onwards, Palm OS runs on ARM architecture-based processors.

The key features of the current Palm OS Garnet are:

Simple, single-tasking environment to allow launching of full screen applications with a basic, common GUI set

Monochrome or color screens with resolutions up to 480x320 pixel

Handwriting recognition input system called Graffiti 2

HotSync technology for data synchronization with desktop computers

Sound playback and record capabilities

Simple security model: Device can be locked by password, arbitrary application records can be made private

TCP/IP network access

Serial port/USB, infrared, Bluetooth and Wi-Fi connections

Expansion memory card support

Defined standard data format for personal information management applications to store calendar, address, task and note entries, accessible by third-party applications.

Included with the OS is also a set of standard applications, with the most relevant ones for the four mentioned PIM operations.

Symbian was a closed-source mobile operating system (OS) and computing platform designed for smartphones.[6] Symbian was originally developed by Symbian Ltd., as a descendant of Psion's EPOC and runs exclusively on ARM processors, although an unreleased x86 port existed.

Symbian was previously an open-source platform developed by the now defunct Symbian Foundation in 2009, as the successor of the original **Symbian OS** before being transitioned to a non-open license in 2011. Symbian was used by many major mobile phone brands, like Samsung, Motorola, Sony Ericsson, and above all by Nokia. It was briefly the most popular smartphone OS on a worldwide average until the end of 2010 – at a time when smartphones were
in limited use, when it was overtaken byAndroid, as Google and its partners achieved wide adoption.

Symbian rose to fame from its use with the S60 platform built by Nokia, first released in 2002 and powering most Nokia smartphones. UIQ, another Symbian platform, ran in parallel, but these two platforms were not compatible with each other. Symbian^3 was officially released in Q4 2010 as the successor of S60 and UIQ, first used in theNokia N8, to use a single platform for the OS. In May 2011 an update, Symbian Anna, was officially announced, followed by Nokia Belle (previously Symbian Belle) in August 2011.

iOS (originally **iPhone OS**) is a mobile operating system created and developed by Apple Inc. and distributed exclusively for Apple hardware. It is the operating system that presently powers many of the company's mobile devices, including the iPhone, iPad, and iPod touch. It is the second most popular mobile operating system in the world by sales, after Android. iPad tablets are also the second most popular, by sales, against Android since 2013, when Android tablet sales increased by 127%.[7]

Originally unveiled in 2007, for the iPhone, it has been extended to support other Apple devices such as the iPod Touch (September 2007), iPad (January 2010), iPad Mini (November 2012) and second-generation Apple TV onward (September 2010). As of January 2015, Apple's App Store contained more than 1.4 million iOS applications, 725,000 of which are native for iPads.[8] These mobile apps have collectively been downloaded more than 100 billion times.[9]

The iOS user interface is based on the concept of direct manipulation, using multi-touch gestures. Interface control elements consist of sliders, switches, and buttons. Interaction with the OS includes gestures such as *swipe*, *tap*, *pinch*, and *reverse pinch*, all of which have specific definitions within the context of the iOS operating system and its multi-touch interface. Internal accelerometers are used by some applications to respond to shaking the device (one common result is the undo command) or rotating it in three dimensions (one common result is switching from portrait to landscape mode).

iOS shares with OS X some frameworks such as Core Foundation and Foundation Kit; however, its UI toolkit is Cocoa Touch rather than OS X's Cocoa, so that it provides the UIKit framework rather than the AppKit framework. It is therefore not compatible with OS X for applications. Also while iOS also shares the Darwinfoundation with OS X, Unix-like shell access is not available for users and restricted for apps, making iOS not fully Unix-compatible either.

Major versions of iOS are released annually. The current release, iOS 9.3, was released on March 21, 2016. In iOS, there are four abstraction layers: the Core OS layer, the Core Services layer, the Media layer, and the Cocoa Touch layer. The current version of the operating system (iOS 9), dedicates around 1.3 GB of the device's flash memory for iOS itself.[10] It runs on the iPhone 4S and later, iPad 2 and later, iPad Pro, all models of the iPad Mini, and the 5th-generation iPod Touch and later.

Android is a mobile operating system (OS) currently developed by Google, based on the Linux kernel and designed primarily for touchscreen mobile devices such as smartphones and tablets. Android's user interface is mainly based on direct manipulation, using touch gestures that loosely correspond to real-world actions, such as swiping, tapping and pinching, to manipulate on-screen objects, along with a virtual keyboard for text input. In addition to touchscreen devices, Google has further developed Android TV for televisions, Android Auto for cars, and Android Wear for wrist watches, each with a specialized user interface. Variants of Android are also used on notebooks, game consoles, digital cameras, and other electronics.

Android has the largest installed base of all operating systems of any kind. Android has been the best selling OS on tablets since 2013, and on smartphones it is dominant by any metric.

Initially developed by Android, Inc., which Google bought in 2005, Android was unveiled in 2007, along with the founding of the Open Handset Alliance – a consortium of hardware, software, and telecommunication companies devoted to advancing open standards for mobile devices. As of July 2013, the Google Play store has had over one million Android applications ("apps") published, and over 50 billion applications downloaded.[18] An April–May 2013 survey of mobile application developers found that 71% of developers create applications for Android,[19] and a 2015 survey found that 40% of full-time professional developers see Android as their priority target platform, which is comparable to Apple's iOS on 37% with both platforms far above others.[20] At Google I/O 2014, the company revealed that there were over one billion active monthly Android users, up from 538 million in June 2013.

Android's source code is released by Google under open source licenses, although most Android devices ultimately ship with a combination of open source and proprietary software, including proprietary software required for accessing Google services. Android is popular with technology companies that require a ready-made, low-cost and customizable operating system for high-tech devices. Its open nature has encouraged a large community of developers and enthusiasts to use the open-source code as a foundation for community-driven projects, which add new features for advanced users[23] or bring Android to devices originally shipped with other operating systems. At the same time, as Android has no centralised update system most Android devices fail to receive security updates: research in 2015 concluded that almost 90% of Android phones in use had known but unpatched security vulnerabilities due to lack of updates and support. The success of Android has made it a target for patent litigation as part of the so-called "smartphone wars" between technology companies

Software Development Kit

The iOS SDK (Software Development Kit) (formerly iPhone SDK) is a software development kit developed by Apple Inc. and released in February 2008 to developnative applications for iOS.

On October 17, 2007, in an open letter posted to Apple's "Hot News" weblog, Steve Jobs announced that a software development kit (SDK) would be made available to third-party developers in February 2008.[1] The SDK was released on March 6, 2008, and allows developers to make applications for the iPhone and iPod Touch, as well as test them in an "iPhone simulator". However, loading an application onto the devices is only possible after paying an iOS Developer Program fee, which is $99.00 USD per year.[2] Since the release of Xcode 3.1, Xcode is the development environment for the iOS SDK. iPhone applications, like OS Xapplications, are written in Swift and Objective-C,[3] with some elements of an application able to be written in C or C++.

Developers are able to set any price above a set minimum for their applications to be distributed through the App Store, of which they will receive a 70% share. Alternately, they may opt to release

the application for free and need not pay any costs to release or distribute the application except for the membership fee.[25]

Since its release, there has been some controversy regarding the refund policy in the fine print of the Developer Agreement with Apple. According to the agreement that developers must agree to, if someone purchases an app from the app store, 30% of the price goes to Apple, and 70% to the developer. If a refund is granted to the customer (at Apple's discretion), the 30% is returned to the customer from Apple, and 70% from the developer; however, Apple can then take another 30% of the cost from the developer to make up for Apple's loss

Android software development is the process by which new applications are created for the Android operating system. Applications are usually developed in Javaprogramming language using the Android software development kit (SDK), but other development environments are also available.

The Android software development kit (SDK) includes a comprehensive set of development tools. These include a debugger, libraries, a handset emulator based onQEMU, documentation, sample code, and tutorials. Currently supported development platforms include computers running Linux (any modern desktop Linux distribution),Mac OS X 10.5.8 or later, and Windows XP or later. As of March 2015, the SDK is not available on Android itself, but the software development is possible by using specialized Android applications.

Until around the end of 2014, the officially supported integrated development environment (IDE) was Eclipse using the Android Development Tools (ADT) Plugin, thoughIntelliJ IDEA IDE (all editions) fully supports Android development out of the box,[7] and NetBeans IDE also supports Android development via a plugin. As of 2015,Android Studio, made by Google and powered by IntelliJ, is the official IDE; however, developers are free to use others. Additionally, developers may use any text editor to edit Java and XML files, then use command line tools (Java Development Kit and Apache Ant are required) to create, build and debug Android applications as well as control attached Android devices (e.g., triggering a reboot, installing software package(s) remotely).

Enhancements to Android's SDK go hand in hand with the overall Android platform development. The SDK also supports older versions of the Android platform in case developers wish to target their applications at older devices. Development tools are downloadable components, so after one has downloaded the latest version and platform, older platforms and tools can also be downloaded for compatibility testing.

Android applications are packaged in .apk format and stored under /data/app folder on the Android OS (the folder is accessible only to the root user for security reasons). APK package contains .dex files (compiled byte code files called Dalvik executables), resource files, etc.

BlackBerry OS

BlackBerry OS is a proprietary mobile operating system developed by BlackBerry Ltd for its BlackBerry line of smartphone handheld devices. The operating system provides multitasking and supports specialized input devices that have been adopted by BlackBerry Ltd. for use in its handhelds, particularly the trackwheel, trackball, and most recently, the trackpad and touchscreen.

The BlackBerry platform is perhaps best known for its native support for corporate email, through MIDP 1.0 and, more recently, a subset of MIDP 2.0, which allows complete wireless activation and synchronization with Microsoft Exchange, Lotus Domino, or Novell GroupWise email, calendar, tasks, notes, and contacts, when used with BlackBerry Enterprise Server. The operating system also supports WAP 1.2.Updates to the operating system may be automatically available from wireless carriers that support the BlackBerry over the air software loading (OTASL) service.

Third-party developers can write software using the available BlackBerry API classes, although applications that make use of certain functionality must be digitally signed.Research from June 2011 indicated that approximately 45% of mobile developers were using the platform at the time of publication. BlackBerry OS was discontinued after the release of BlackBerry 10 but BlackBerry will continue support for the BlackBerry OS.

The Windows Software Development Kit (SDK) for Windows 8 contains headers, libraries, and a selection of tools that you can use when you create apps that run on Windows operating systems. You can use the Windows SDK, along with your chosen development environment, to write Windows Store apps (only on Windows 8) using web technologies (such as HTML5, CSS3, and JavaScript), native (C++), and managed (C#, Visual Basic) code; desktop applications that use the native (Win32/COM) programming model; or desktop applications that use the managed (.NET Framework) programming model.

The Windows SDK also includes the Windows App Certification Kit (ACK) 2.2 to test your app for the Windows 8 Certification Program and the Windows 7 Logo Program. If you also want to test your app on Windows RT, use the Windows App Certification Kit for Windows RT .

The Windows SDK no longer ships with a complete command-line build environment. You must install a compiler and build environment separately. If you require a complete development environment that includes compilers and a build environment, you can download Visual Studio Express , which includes the appropriate components of the Windows SDK. To download the SDK and install it on another computer, click the download link and run the setup. Then in the Specify Location dialog box, click

Mobile commerce

The phrase mobile commerce was originally coined in 1997 by Kevin Duffey at the launch of the Global Mobile Commerce Forum, to mean "the delivery of electronic commerce capabilities directly into the consumer's hand, anywhere, via wireless technology."[1] Many choose to think of Mobile Commerce as meaning "a retail outlet in your customer's pocket

The Global Mobile Commerce Forum, which came to include over 100 organisations, had its fully minuted launch in London on 10 November 1997. Kevin Duffey was elected as the Executive Chairman at the first meeting in November 1997. The meeting was opened by Dr Mike Short, former chairman of the GSM Association, with the very first forecasts for mobile commerce from Kevin Duffey (Group Telecoms Director ofLogica) and Tom Alexander (later CEO of Virgin Mobile and then of Orange). Over 100 companies joined the Forum within a year, many forming mobile commerce teams of their own, e.g. MasterCard and Motorola. Of these one hundred companies, the first two were Logica and Cellnet (which later became O2). Member organisations such as Nokia, Apple, Alcatel, and Vodafone began a series of trials and collaborations.

Mobile commerce services were first delivered in 1997, when the first two mobile-phone enabled Coca Cola vending machines were installed in the Helsinki area in Finland. The machines accepted payment via SMStext messages. This work evolved to several new mobile applications such as the first mobile phone-based banking service was launched in 1997 by Merita Bank of Finland, also using SMS. Finnair mobile check-in was also a major milestone, first introduced in 2001

M-COMMERCE APPLICATIONS The general m-commerce applications are:

1. Mobile ticketing

Tickets can be sent to mobile phones using a variety of technologies. Users are then able to use their tickets immediately by presenting their phones at the venue.Tickets can be booked and cancelled on the mobile with the help of simple applicationdownloads or by accessing WAP portals of various Travel agents or direct service providers. Mobile ticketing for airports,

ballparks, and train stations, for example, will not only streamline unexpected metropolitan traffic surges, but also help users remotely secure parking spots (even while in their vehicles) and greatly facilitate mass surveillance at transport hubs.

2. Mobile vouchers, coupons and loyalty cards

Mobile ticketing technology can also be used for the distribution of vouchers, coupons and loyalty cards. The voucher, coupon, or loyalty card is represented by a virtual token that is sent to the mobile phone. Presenting a mobile phone with one of these tokens at the point of sale allows the

customer to receive the same benefits as another customer who has a loyalty card or other paper coupon/voucher. Mobile delivery enables:

economy of scale
quicker and easier delivery
effective target marketing
privacy-friendly data mining on consumer behaviour
environment-friendly and resources-saving efficacy

Content purchase and delivery

Currently, mobile content purchase and delivery mainly consists of the sale of ring-tones, wallpapers, and games for mobile phones. The convergence of mobile phones, mp3 players and video players into a single device will result in an increase in the purchase and delivery of full-length music tracks and video. Download speeds, if increased to 4G levels, will make it possible to buy a movie on a mobile device in a couple of seconds, while on the go.

4. Location-based services

Unlike a home PC, the location of the mobile phone user is an important piece of information used during mobile commerce transactions. Knowing the location of the user allows for location based services such as:

- local maps
- local offers
- local weather

people tracking and monitoring

HW 2 Due Tuesday 10/18

Lecture 6: Semaphores and Monitors

CSE 120: Principles of Operating Systems

Alex C. Snoeren

UCSDCSE
Computer Science and Engineering

Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks work, but they have some drawbacks when critical sections are long
  - Spinlocks – inefficient
  - Disabling interrupts – can miss or delay important events
- Instead, we want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section
- Look at two common high-level mechanisms
  - Semaphores: binary (mutex) and counting
  - Monitors: mutexes and condition variables
- Use them to solve common synchronization problems

Semaphores

- Semaphores are another data structure that provides mutual exclusion to critical sections
  - Block waiters, interrupts enabled within CS
  - Described by Dijkstra in THE system in 1968
- Semaphores can also be used as atomic counters
  - More later
- Semaphores support two operations:
  - wait(semaphore): decrement, block until semaphore is open
    - Also P(), after the Dutch word for test, or down()
  - signal(semaphore): increment, allow another thread to enter
    - Also V() after the Dutch word for increment, or up()

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    - » In other words, signal() has "history" (c.f. condition vars later)
    - » This "history" is a counter

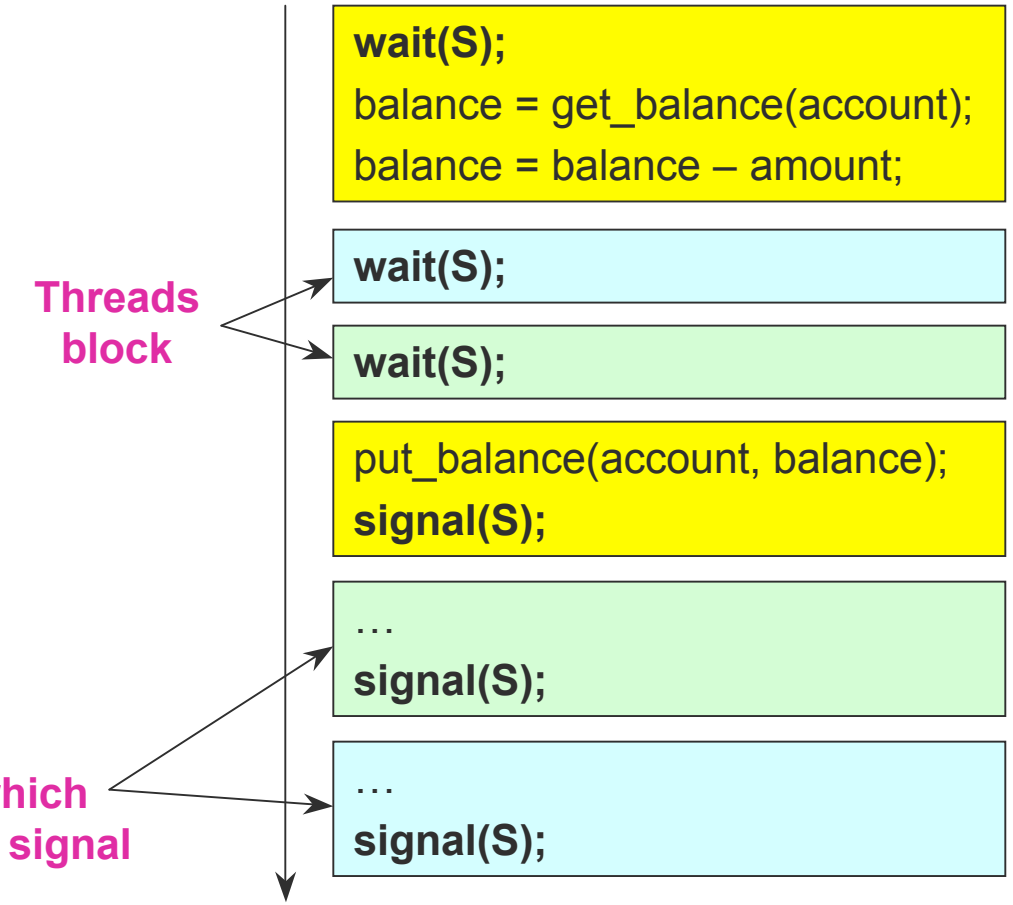
Semaphore Types

- Semaphores come in two types
- Mutex semaphore
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
  - Number of threads determined by the semaphore "count"
    - » mutex has count = 1, counting has count = N

Using Semaphores

- Use is similar to our locks, but semantics are different

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```



It is **undefined** which thread runs after a signal

Semaphores in Nachos

```
wait (S) {
  Disable interrupts;
  while (S->value == 0) {
    enqueue(S->q, current_thread);
    thread_sleep(current_thread);
  }
  S->value = S->value – 1;
  Enable interrupts;
}
```

```
signal (S) {
  Disable interrupts;
  thread = dequeue(S->q);
  thread_start(thread);
  S->value = S->value + 1;
  Enable interrupts;
}
```

- thread_sleep() assumes interrupts are disabled
  - Note that interrupts are disabled only to enter/leave critical section
  - How can it sleep with interrupts disabled?
- Need to be able to reference current thread

Using Semaphores

- We've looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account
- Now we're going to use semaphores to look at more interesting examples
  - Readers/Writers
  - Bounded Buffers

Readers/Writers Problem

- Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers
  - But only one writer
- How can we use semaphores to control access to the object to implement this protocol?
- Use three variables
  - int readcount – number of threads reading object
  - Semaphore mutex – control access to readcount
  - Semaphore w_or_r – exclusive writing or reading

Readers/Writers

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}
```

```
reader {
    wait(mutex);       // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);   // unlock readcount
    Read;
    wait(mutex);      // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);   // unlock readcount}
}
```

Readers/Writers Notes

- If there is a writer
  - First reader blocks on w_or_r
  - All other readers block on mutex
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
- The last reader to exit signals a waiting writer
  - If no writer, then readers can continue
- If readers and writers are waiting on w_or_r, and a writer exits, who goes first?
- Why doesn't a writer need to use mutex?

Bounded Buffer

- Problem: There is a set of resource buffers shared by producer and consumer threads
- Producer inserts resources into the buffer set
  - Output, disk blocks, memory pages, processes, etc.
- Consumer removes resources from the buffer set
  - Whatever is generated by the producer
- Producer and consumer execute at different rates
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff

Bounded Buffer (2)

- Use three semaphores:
  - mutex – mutual exclusion to shared set of buffers
    - » Binary semaphore
  - empty – count of empty buffers
    - » Counting semaphore
  - full – count of full buffers
    - » Counting semaphore

Bounded Buffer (3)

```
Semaphore mutex = 1;   // mutual exclusion to shared set of buffers
Semaphore empty = N;   // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)
```

```
producer {
  while (1) {
    Produce new resource;
    wait(empty); // wait for empty buffer
    wait(mutex); // lock buffer list
    Add resource to an empty buffer;
    signal(mutex); // unlock buffer list
    signal(full);      // note a full buffer
  }
}
```

```
consumer {
  while (1) {
    wait(full);        // wait for a full buffer
    wait(mutex);    // lock buffer list
    Remove resource from a full buffer;
    signal(mutex); // unlock buffer list
    signal(empty); // note an empty buffer
    Consume resource;
  }
}
```

Bounded Buffer (4)

- Why need the mutex at all?
- Where are the critical sections?
- What happens if operations on mutex and full/empty are switched around?
  - The pattern of signal/wait on full/empty is a common construct often called an interlock
- Producer-Consumer and Bounded Buffer are classic examples of synchronization problems
  - The Mating Whale problem in Project 1 is another
  - You can use semaphores to solve the problem
  - Use readers/writers and bounded buffer as examples for hw

Semaphore Summary

- Semaphores can be used to solve any of the traditional synchronization problems
- However, they have some drawbacks
  - They are essentially shared global variables
    - » Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
  - No control or guarantee of proper usage
- Sometimes hard to use and prone to bugs
  - Another approach: Use programming language support

Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent procedure invocations
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways

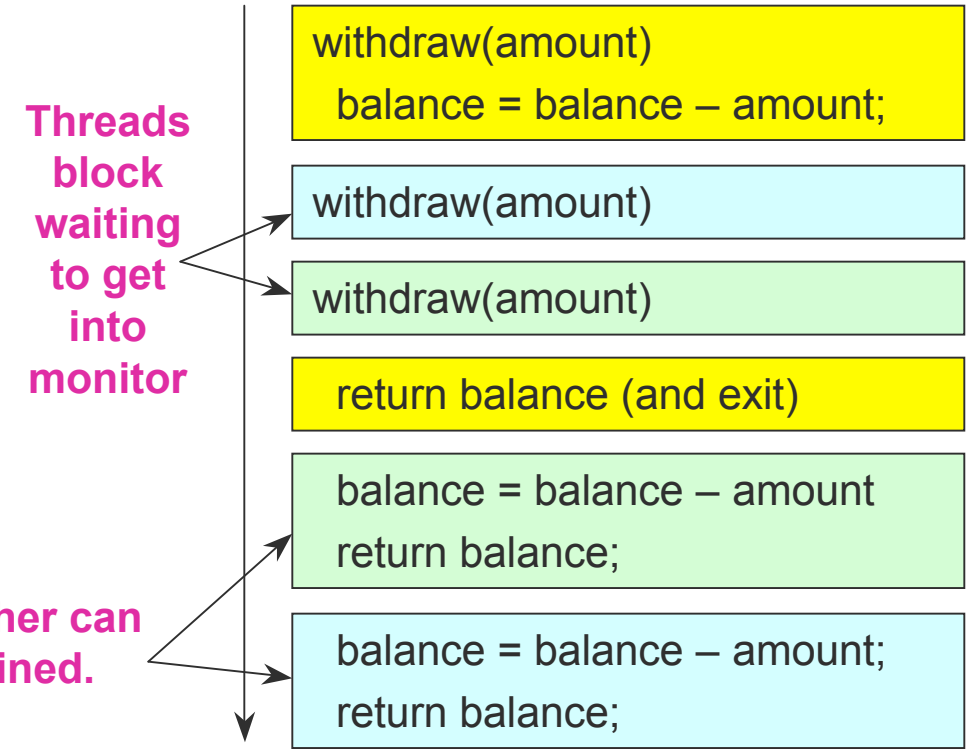
Monitor Semantics

- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    - » So the monitor has to have a wait queue…
  - If a thread within a monitor blocks, another one can enter
- What are the implications in terms of parallelism in monitor?

Account Example

```
Monitor account {
  double balance;

  double withdraw(amount) {
    balance = balance – amount;
    return balance;
  }
}
```



When first thread exits, another can enter. Which one is undefined.

- Hey, that was easy
- But what if a thread wants to wait inside the monitor?
  - » Such as “mutex(empty)” by reader in bounded buffer?

Condition Variables

- Condition variables provide a mechanism to wait for events (a “rendezvous point”)
  - Resource available, no more writers, etc.
- Condition variables support three operations:
  - Wait – release monitor lock, wait for C/V to be signaled
    - » So condition variables have wait queues, too
  - Signal – wakeup one waiting thread
  - Broadcast – wakeup all waiting threads
- Note: Condition variables are not boolean objects
  - “if (condition_variable) then” … does not make sense
  - “if (num_resources == 0) then wait(resources_available)” does
  - An example will make this more clear

Monitor Bounded Buffer

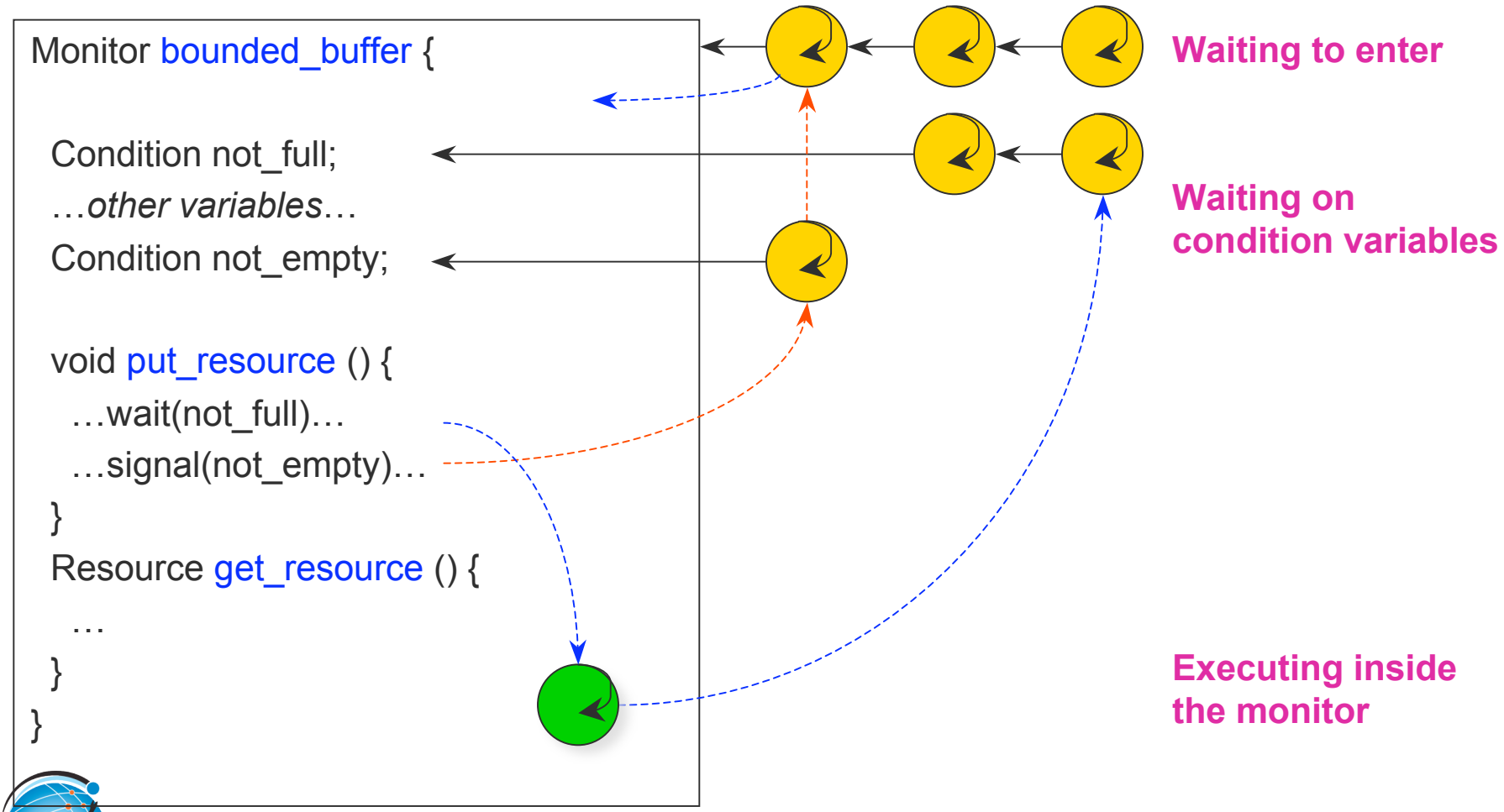
```
Monitor bounded_buffer {
  Resource buffer[N];
  // Variables for indexing buffer
  Condition not_full, not_empty;

  void put_resource (Resource R) {
    while (buffer array is full)
        wait(not_full);
    Add R to buffer array;
    signal(not_empty);
  }
```

```
  Resource get_resource() {
    while (buffer array is empty)
        wait(not_empty);
    Get resource R from buffer array;
    signal(not_full);
    return R;
  }
} // end monitor
```

- What happens if no threads are waiting when signal is called?

Monitor Queues



Condition Vars != Semaphores

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics (such is life, worse yet to come)
  - However, they each can be used to implement the other
- Access to the monitor is controlled by a lock
  - wait() blocks the calling thread, and gives up the lock
    - » To call wait, the thread has to be in the monitor (hence has lock)
    - » Semaphore::wait just blocks the thread on the queue
  - signal() causes a waiting thread to wake up
    - » If there is no waiting thread, the signal is lost
    - » Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
    - » Condition variables have no history

Signal Semantics

- There are two flavors of monitors that differ in the scheduling semantics of signal()
  - Hoare monitors (original)
    - » signal() immediately switches from the caller to a waiting thread
    - » The condition that the waiter was anticipating is guaranteed to hold when waiter executes
    - » Signaler must restore monitor invariants before signaling
  - Mesa monitors (Mesa, Java)
    - » signal() places a waiter on the ready queue, but signaler continues inside monitor
    - » Condition is not necessarily true when waiter runs again
      - Returning from wait() is only a hint that something changed
      - Must recheck conditional case

Hoare vs. Mesa Monitors

- Hoare

```
if (empty)
        wait(condition);
```

- Mesa

```
while (empty)
        wait(condition);
```

- Tradeoffs
  - Mesa monitors easier to use, more efficient
    - » Fewer context switches, easy to support broadcast
  - Hoare monitors leave less to chance
    - » Easier to reason about the program

Condition Vars & Locks

- Condition variables are also used without monitors in conjunction with blocking locks
  - This is what you are implementing in Project 1
- A monitor is "just like" a module whose state includes a condition variable and a lock
  - Difference is syntactic; with monitors, compiler adds the code
- It is "just as if" each procedure in the module calls acquire() on entry and release() on exit
  - But can be done anywhere in procedure, at finer granularity
- With condition variables, the module methods may wait and signal on independent conditions

Using Cond Vars & Locks

- Alternation of two threads (ping-pong)
- Each executes the following:

```
Lock lock;
Condition cond;

void ping_pong () {
  acquire(lock);
  while (1) {
      printf("ping or pong\n");
      signal(cond, lock);
      wait(cond, lock);
  }
  release(lock);
```

Must acquire lock before you can wait (similar to needing interrupts disabled to call Sleep in Nachos)

Wait atomically releases lock and blocks until signal()

Wait atomically acquires lock before it returns

Monitors and Java

- A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object's monitor
  - A thread enters an object's monitor by
    - » Executing a method declared "synchronized"
      - Can mix synchronized/unsynchronized methods in same class
    - » Executing the body of a "synchronized" statement
      - Supports finer-grained locking than an entire procedure
      - Identical to the Modula-2 "LOCK (m) DO" construct
- Every object can be treated as a condition variable
  - Object::notify() has similar semantics as Condition::signal()

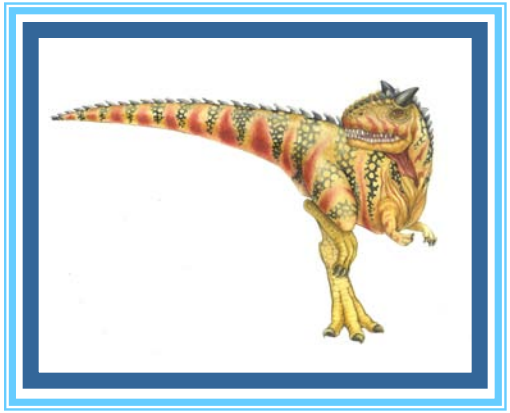
Summary

- Semaphores
  - wait()/signal() implement blocking mutual exclusion
  - Also used as atomic counters (counting semaphores)
  - Can be inconvenient to use
- Monitors
  - Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
    - » Only one thread can execute within a monitor at a time
  - Relies upon high-level language support
- Condition variables
  - Used by threads as a synchronization point to wait for events
  - Inside monitors, or outside with locks

Next time…

- Read Chapters 5 and 7

Chapter 7: Deadlocks



Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Chapter Objectives

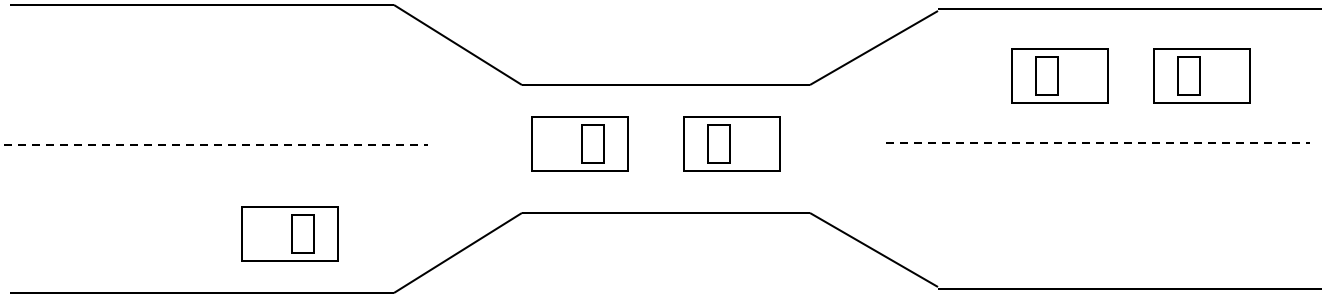
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example
  - semaphores *A* and *B*, initialized to 1 P_0 P_1

    wait (A); wait(B) wait (B); wait(A)

Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.

- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

A set of vertices V and a set of edges E.

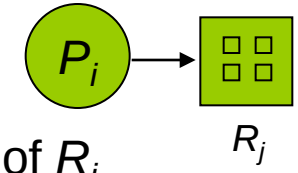
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

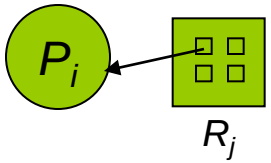
- Process

- Resource Type with 4 instances

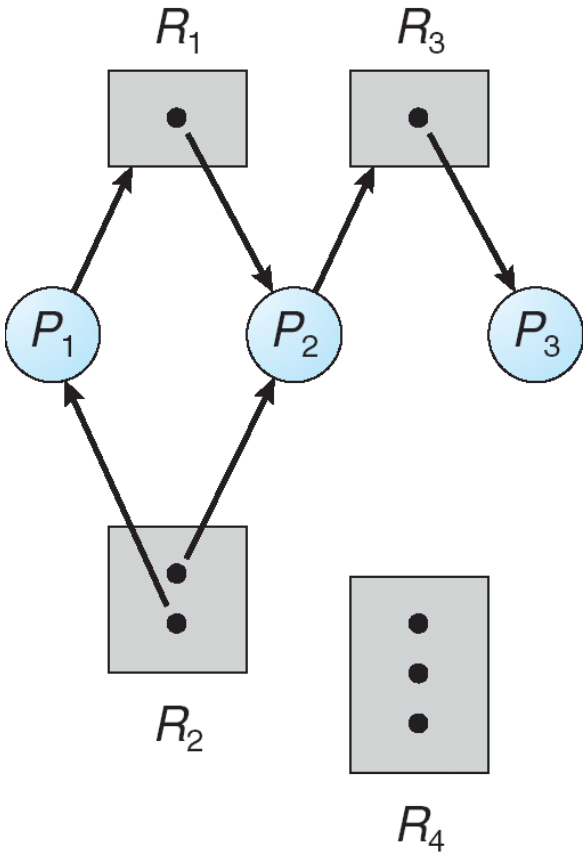
- P_i requests instance of R_j



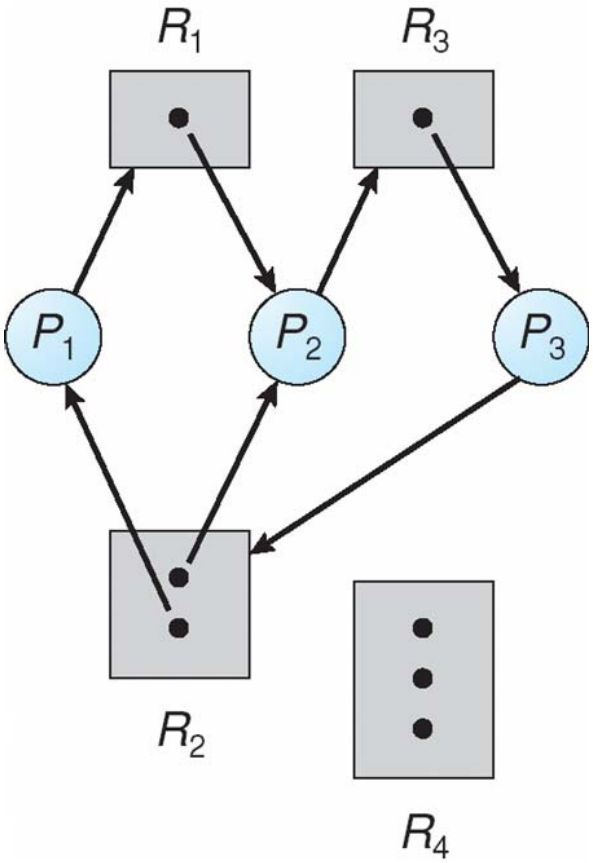
- P_i is holding an instance of R_j



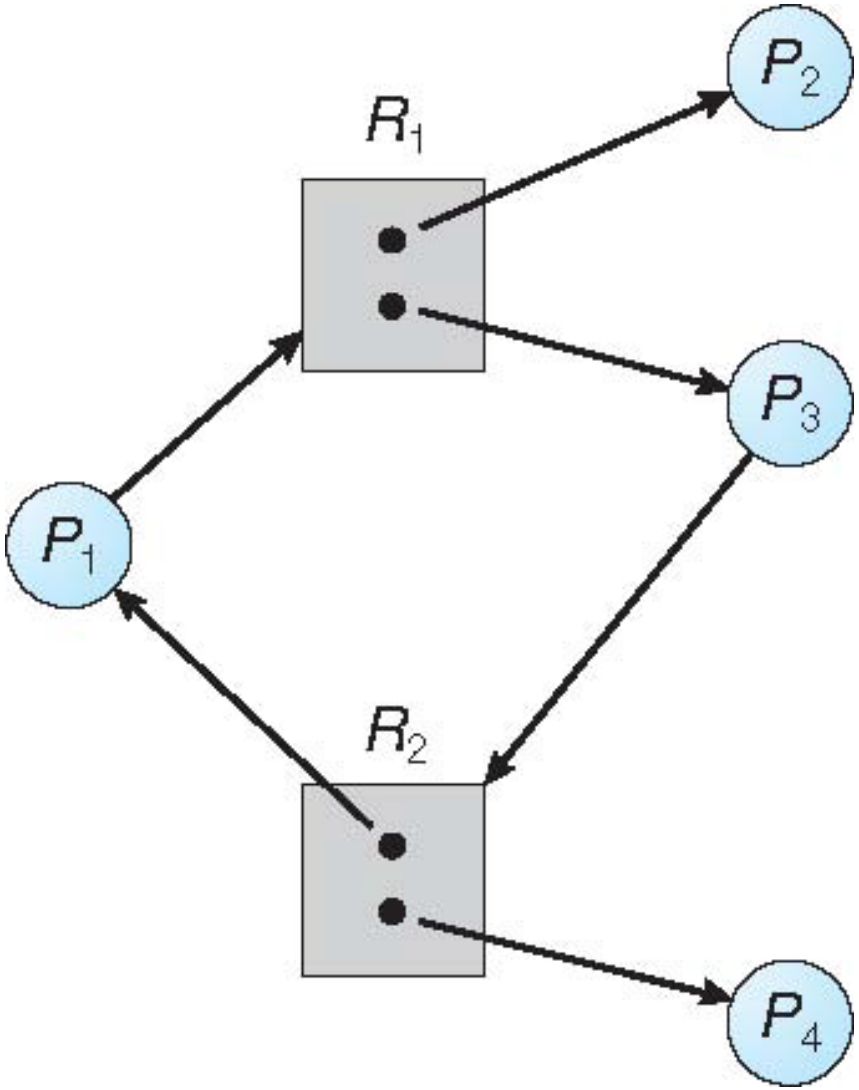
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

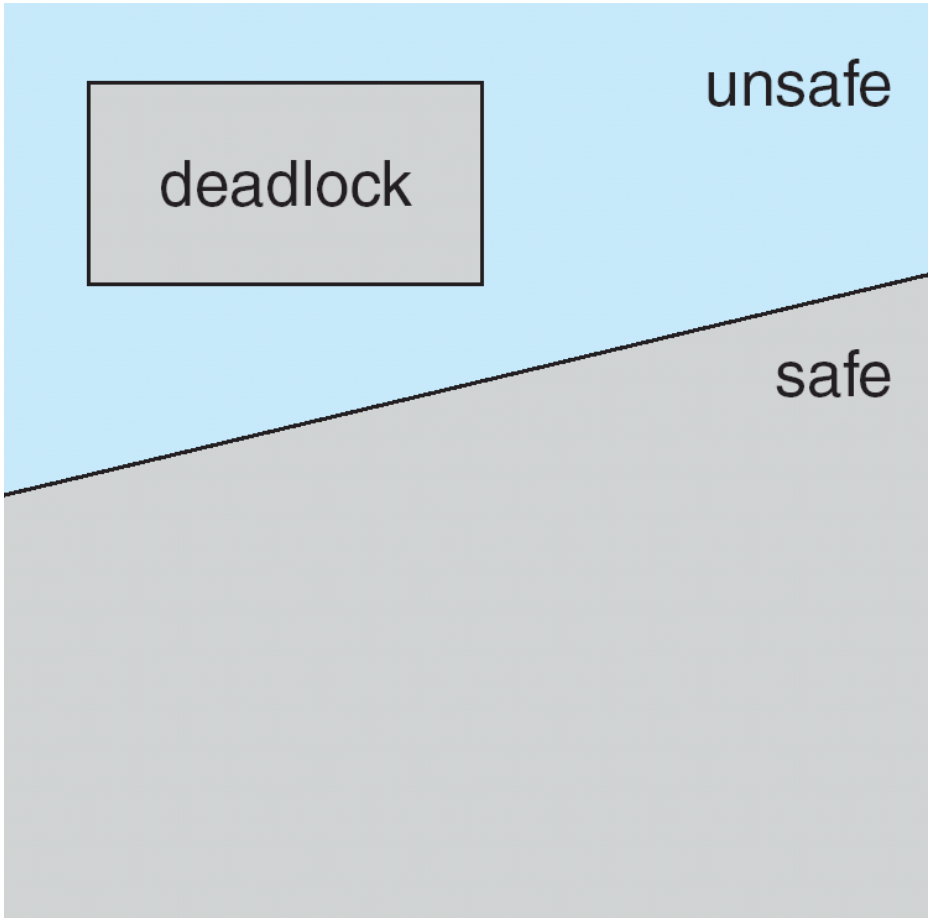
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



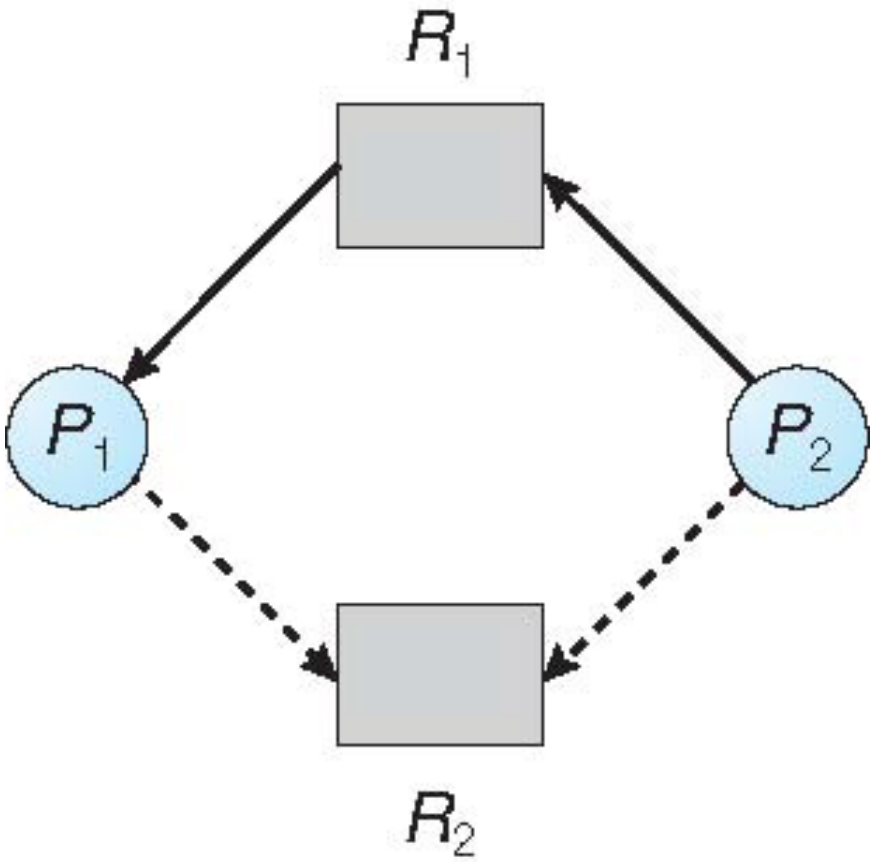
Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

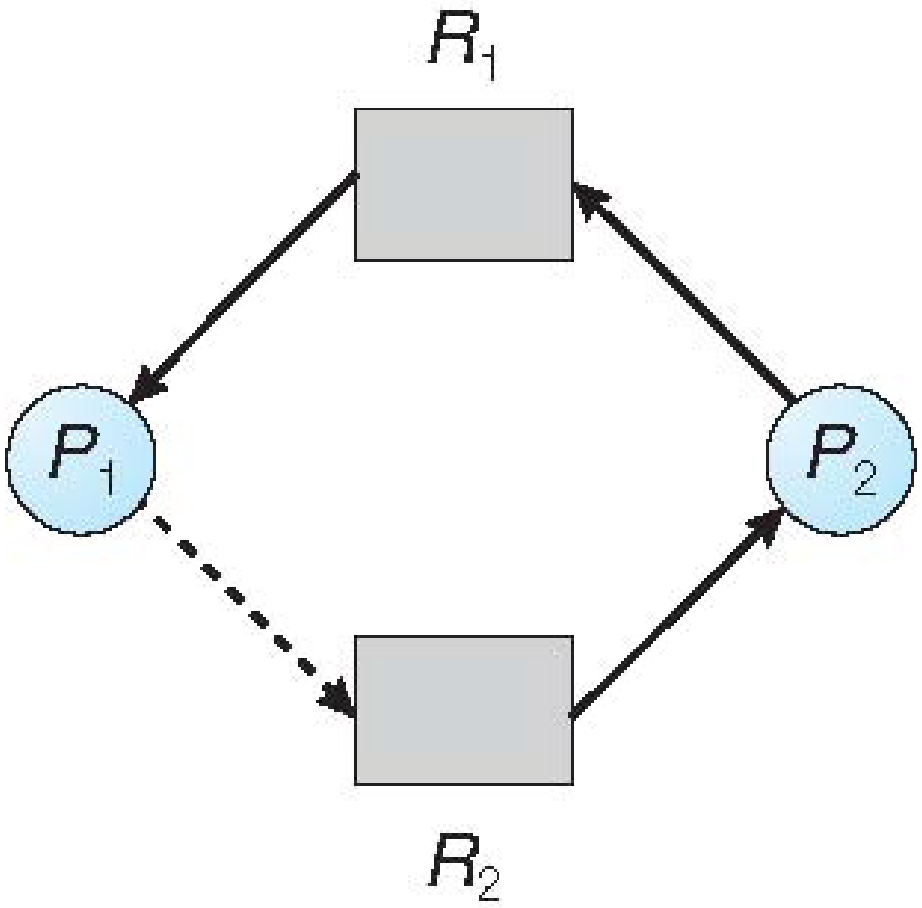
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max**: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need$[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1

2. Find an *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *If safe $\Rightarrow$ the resources are allocated to Pi*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

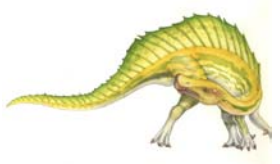
- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances)

Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	



Example (Cont.)

- The content of the matrix *Need* is defined to be *Max* – *Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

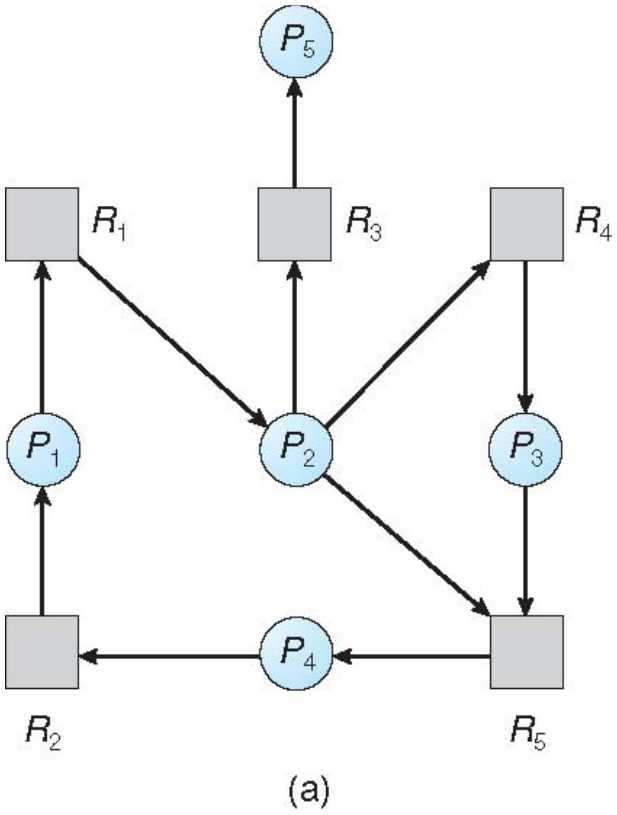
Deadlock Detection

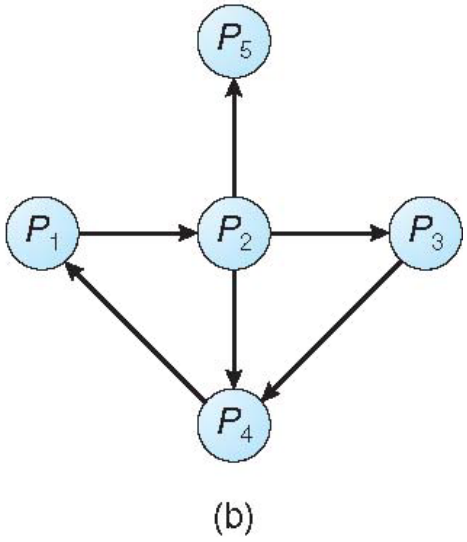
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- **Available**: A vector of length *m* indicates the number of available resources of each type.
- **Allocation**: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i*][*j*] = *k*, then process P_i is requesting *k* more instances of resource type.R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*

2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4

Detection Algorithm (Cont.)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(m x $n^{2)}$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

Example (Cont.)

- P_2 requests an additional instance of type *C*

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

End of Chapter 7