

# Support for Distributed Processing

CS 416: Operating Systems Design

Department of Computer Science

Rutgers University

<http://www.cs.rutgers.edu/~vinodg/teaching/416/>

# Motivation

So far, we talked about mechanisms and policies that

- Virtualize the underlying machine

- Support multi-programming of a single machine

With the proliferation of cheap but powerful machines (devices) and ubiquitous network connections

- Computing is occurring on more than one machine

- Examples: email, web searching, NFS, [SETI@home](#), etc

# Why Distributed Systems?

## Distributed system vs. mainframe

Microprocessors offer better price/performance

More scalable => more computing power

Inherent distribution, e.g. computer-supported cooperative work

Reliability, Incremental growth

## Distributed system vs. independent PCs

Some applications require sharing of data, e.g. airline reservations

Sharing of hardware, e.g. expensive devices (color laser printer)

Easier human-to-human communication, e.g. electronic mail

Spread the workload over the machines in the most effective way

**Disadvantages: lack of software, coordination, management, and security are harder**

# Building Distributed Systems

Building distributed systems is HARD!

Why?

Naming

Lack of global knowledge

Synchronization, event ordering

Faults

“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable” – L. Lamport

# Fault Models

## Processor:

**Failstop:** processor fails by halting. The fact that a processor has failed is detectable by other processors.

**Crash:** processor fails by halting. The fact that a processor has failed may not be detectable by other processors.

**Byzantine failures:** processor fails by exhibiting arbitrary behavior.

Fairly easy to see how to extend to components like disks, NICs, etc.

# Fault Models

---

What about networks?

Links can fail altogether – failstop

Messages can be:

- Dropped – failstop (?)

- Delayed for an arbitrarily long time – crash (?)

- Corrupted – byzantine (?)

Networks can be partitioned

# Lack of Global Knowledge

---

Lack of global knowledge also complicates designs.

Two very strong, similar results in distributed systems:

Cannot solve the consensus problem in the face of 1 node failure for asynchronous systems. Straightforward if no failures are possible or the system is synchronous (i.e. communication happens in well-determined rounds)

Cannot solve the coordinated attack problem if messages can be lost. Trivial problem if messages cannot be lost.

# Consensus

## Problem

Every process starts with an initial value in  $\{0, 1\}$

A non-faulty process decides on a value in  $\{0, 1\}$  by entering an appropriate decision state

All non-faulty processes that make a decision are required to choose the same value

Some process must eventually make a decision

This is a very weak condition. In reality, you would want all non-faulty processes to eventually make a decision

## Key assumption

System is completely asynchronous so cannot assume anything about rate of progress. In particular, cannot use timeout. In other words, there is no way of differentiating a crashed node from a slow one.

# Coordinated Attack Problem

## Problem (Gray 1987)

Two divisions of an army are camped on two hilltops overlooking a common valley. In the valley awaits the enemy. It is clear that if both divisions attack the enemy simultaneously, they will win the battle; whereas if only one division attacks, it will be defeated. The divisions do not initially have plans for launching an attack on the enemy, and the commanding general of the first division wishes to coordinate a simultaneous attack. The generals can only communicate by means of a messenger. Normally, it takes the messenger one hour to get from one encampment to another. However, it is possible that he will get lost in the dark or, worse yet, be captured by the enemy. Fortunately, on this particular night, everything goes smoothly. How long will it take them to coordinate an attack?

# Coordinated Attack

---

The answer is NEVER!

Suppose General A sends a message to B saying “let’s attack at 5am,” and the messenger delivers it 1 hour later.

Does this work?

# Coordinated Attack

The answer is NEVER!

Suppose General A sends a message to B saying “let’s attack at 5am,” and the messenger delivers it 1 hour later.

Does this work? No, how does general A find out that general B actually received the message? General B would have to send an ACK. But how does general B find out that general A received the ACK? And so on...

# Impossibility of Coordinated Attack

---

Proof by induction on  $d$ , the number of messages delivered by the time of the attack

Base case:  $d = 0$

Clearly, if no message is delivered, then B will not know of the intended attack and a guaranteed simultaneous attack is impossible

# Impossibility of Coordinated Attack

## Induction

Assume that  $k$  messages are not enough

Show that  $k+1$  is not enough either

Suppose that  $k+1$  is enough. If so, then the sender of the  $k+1$  message attacks without knowing whether his last message arrived.

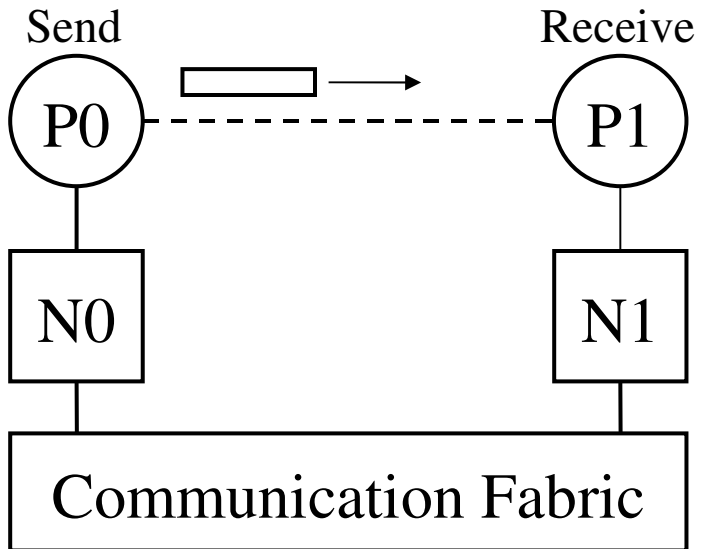
Since whenever 1 general attacks, they both do, the intended receiver of the  $k+1$  message must attack regardless of whether the message was delivered.

In this case, the  $k+1$  message is not necessary, therefore  $k$  message should have been sufficient

# Communication Protocols

---

# Communication Components



# Terminology

## Basic Message Passing:

*Send:* Analogous to mailing a letter

*Receive:* Analogous to picking up a letter from the mailbox

## Network performance:

*Latency:* The time from when a Send is initiated until the first byte is received by a Receive.

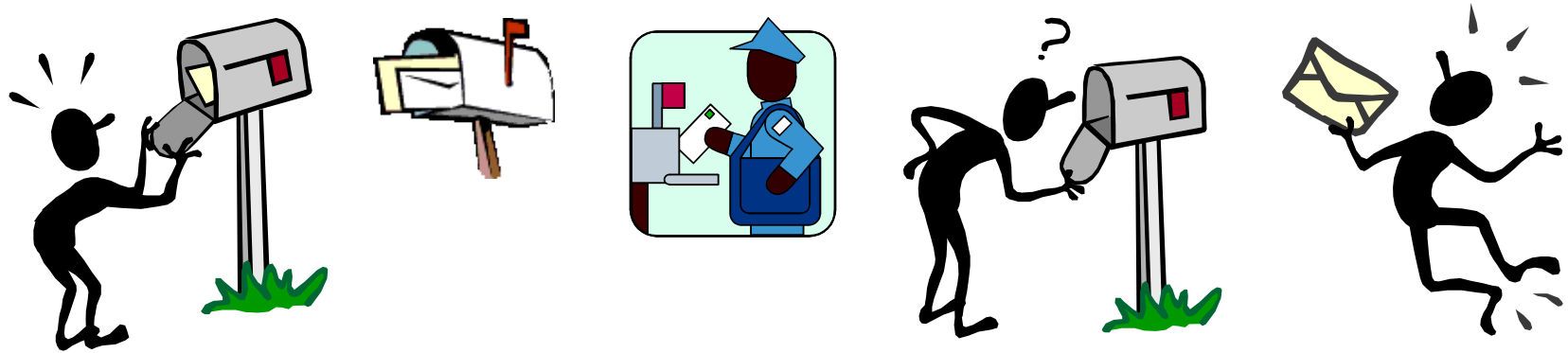
*Bandwidth:* The rate at which a sender is able to send data to a receiver. (e.g., MB/s)

B  $\equiv$  Byte (8 bits), b  $\equiv$  bit

# Basic Message Passing: Easy, Right?

What can be easier than this, right?

Well, think of the post office: to send a letter



# Basic Message Passing: Not So Easy

Why is it so complicated to send a letter if basic message passing is so easy?

Well, it's really not easy! Issues include:

*Naming:* How to specify the receiver?

*Routing:* How to forward the message to the correct receiver through intermediaries?

*Buffering:* What if the out port is not available? What if the receiver is not ready to receive the message?

*Reliability:* What if the message is lost in transit? What if the message is corrupted in transit?

*Blocking:* What if the receiver is ready to receive before the sender is ready to send?

# Communications Abstractions

---

## Abstractions:

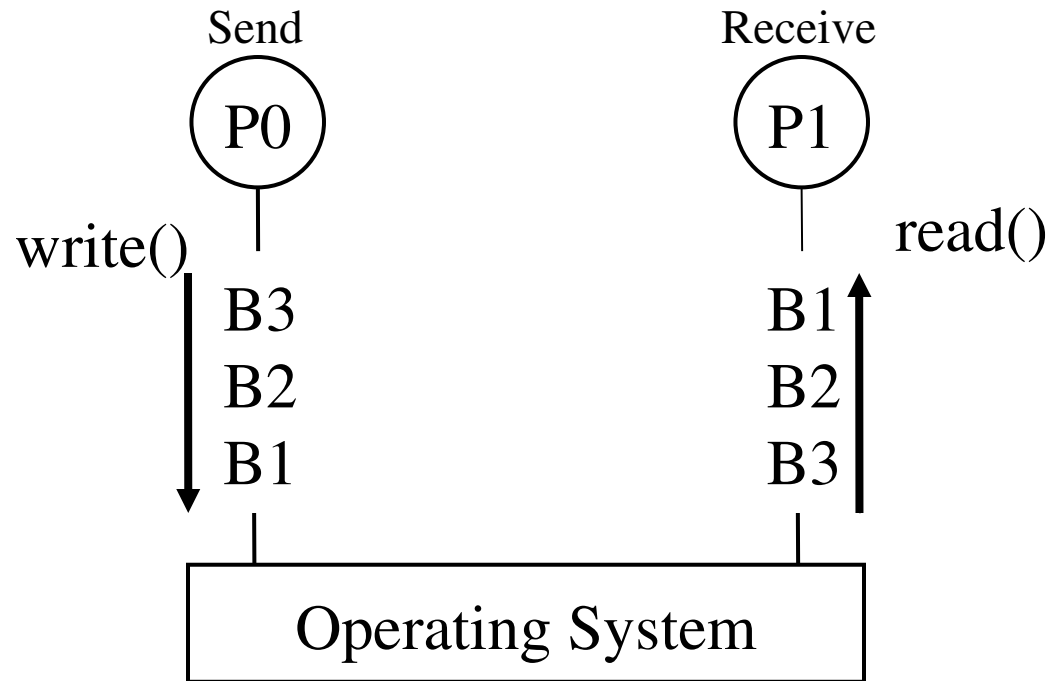
Byte Stream

Datagrams

Shared Memory

Remote Procedure Call

# Byte Streams



# Byte Streams

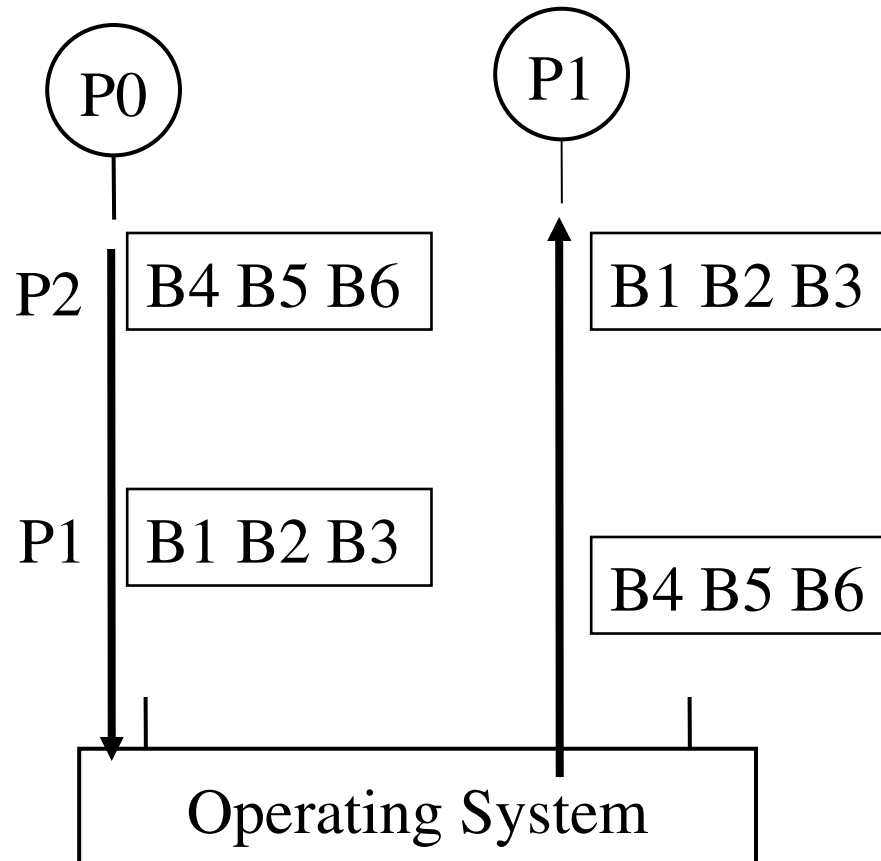
---

No cardinality, i.e. cannot go directly to Nth byte, must read all bytes in FIFO order

OS can “break” groups of bytes into read and writes in whatever grouping is most convenient

E.g. P0 does 1 `write()` call of 1000 bytes. P1 may have to call the `read()` method 10 times to get all 1000 bytes

# Datagrams/Packets



# Datagrams

---

write() system call sends a discrete byte array

The array is called a “datagram” or a “packet”

Discrete: OS cannot “break” a datagram into parts without reassembling it for the user

Either entire datagram/packet is received or none of it is

E.g., a single bad byte means OS cannot deliver the datagram

# Datagram Variants

Every datagram abstraction must pick between these dimensions:

## Ordered vs. Unordered

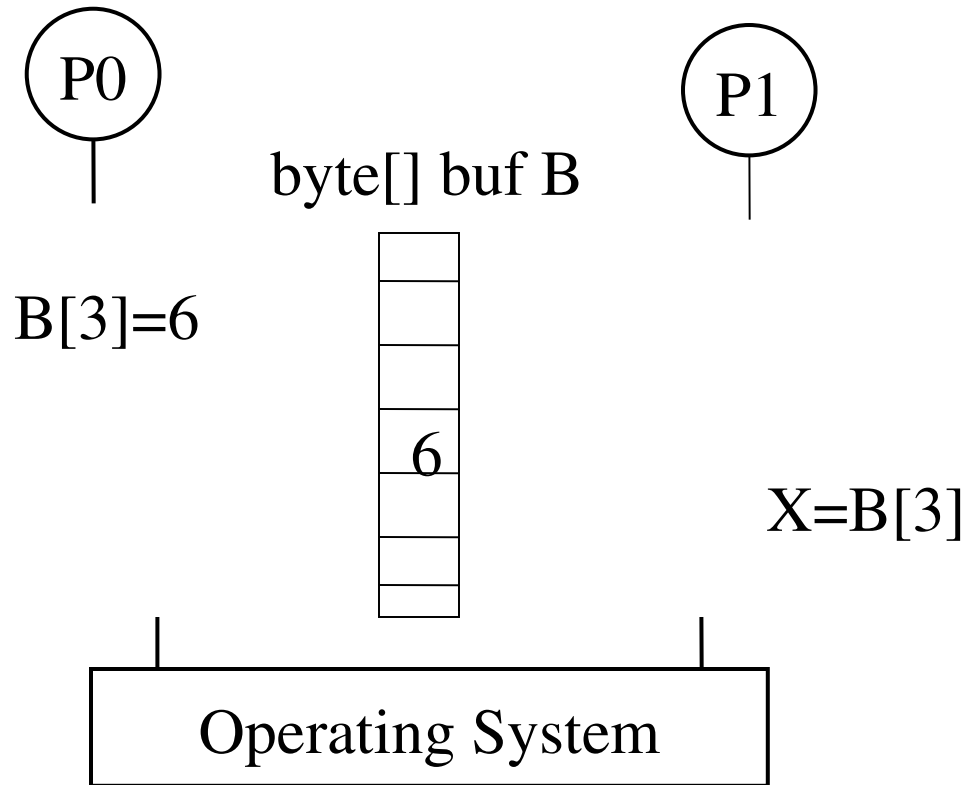
ordered datagrams always received in the order of write() calls, OS free to re-order for unordered.

## Reliable vs. Unreliable

unreliable: OS can “throw away” entire datagram if it gets into trouble, e.g., no more buffers, or the network lost it, or a dog ate it.

reliable: OS will deliver complete datagram under many adverse conditions.

# Shared Memory



# Shared Memory

---

The two, or more, processes share a byte buffer

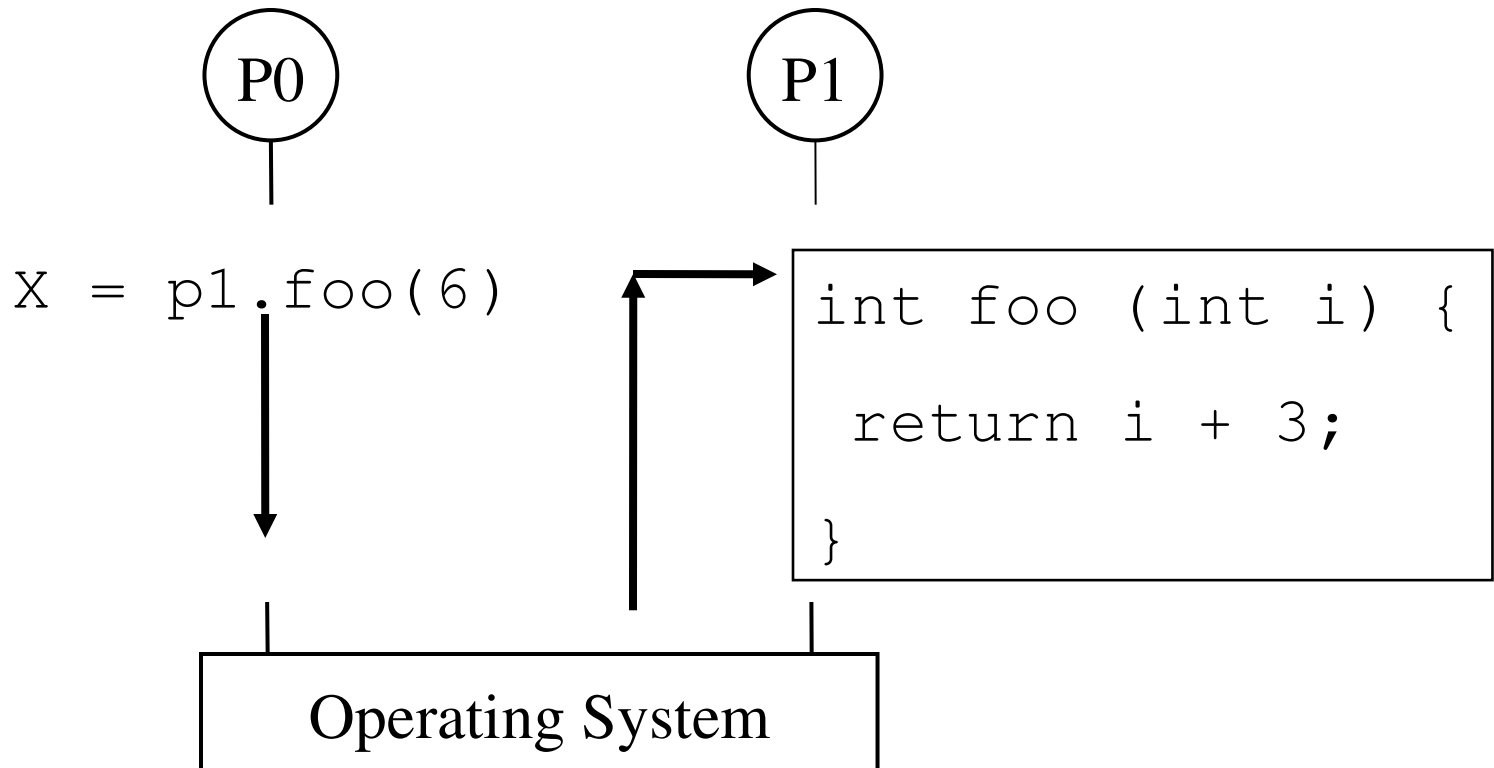
Why not share objects?

Byte buffer is least common denominator

single data-type, no methods, no object hierarchy, no exceptions

Difficult to preserve memory abstraction across a network

# Remote Procedure Call



# RPC

Map communication to a method call

Method invocation on one process (caller) mapped by OS into a call on another process (callee)

Issues:

Parameter passing

What if processes written in different languages?

What if callee crashes or is disconnected during the call?

# Communications Namespaces

---

## File system

## Internet

IP addresses

Domain Name System

TCP and UDP ports

RPC

## System-V (Unix)

Shared memory

Semaphores

Message queues

# Internet Namespaces

IP addresses: Every entity given a 4 byte number

like a phone number

typically written as 4 decimals separated by dots, e.g. 128.6.4.4

Domain Name System (DNS): domains separated by “dot” notation

E.g. remus.rutgers.edu

DNS maps names to IP addresses (names to numbers)

E.g. remus.rutgers.edu -> 128.6.13.3

Use the command “nslookup” to see the mapping

## Internet Namespaces (cont.)

---

TCP: Transmission Control Protocol

UDP: User Datagram Protocol

Communication under these protocols involves an IP address and a “port” at that IP address

The port is a 16-bit integer

TCP and UDP ports are separate namespaces

Use command “netstat” to see which ports are in use.

# System-V Inter-Process Communication (IPC)

---

System-V Unixes (all today) have own namespace for:

- shared memory (segments)

- message queues

- semaphores

These have permissions like the file system, but are not part of the file system

Use `ipcs` command to see the active segments, queues, and semaphores

# Protocol Architecture

To communicate, computers must agree on the syntax and the semantics of communication

E.g., if I were lecturing in Swahili, this lecture would be useless

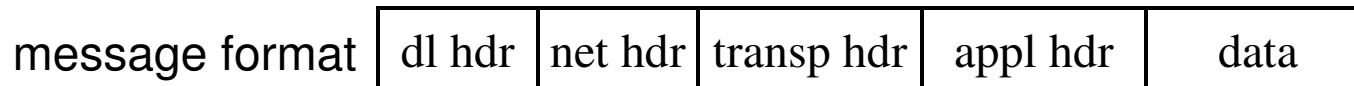
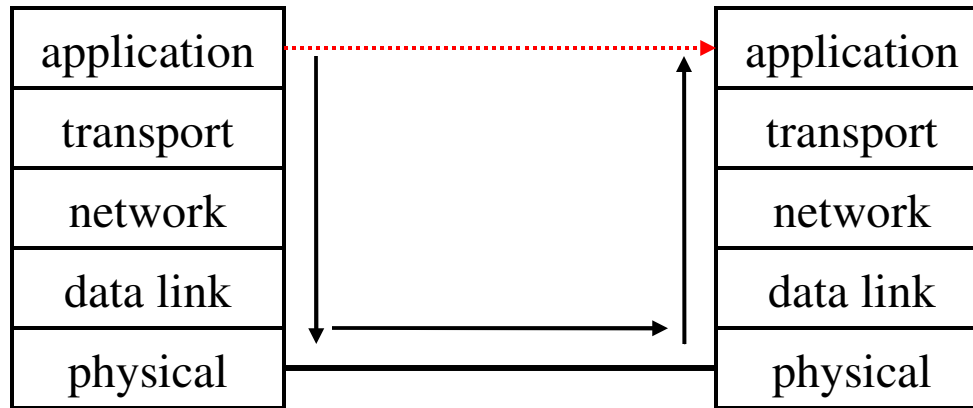
Really hard to implement a reliable communication protocol on top of a network substrate, where packets may be lost or reordered

So why do it? To prevent higher levels from having to implement it.

Common approach: protocol functionality is distributed in multiple layers. Layer  $N$  provides services to layer  $N+1$ , and relies on services of layer  $N-1$

Communication is achieved by having similar layers at both end-points which understand each other

# ISO/OSI Protocol Stack



# Application Layer

---

## Application to application communication

Supports application functionality

## Examples

File transfer protocol (FTP)

Simple mail transfer protocol (SMTP)

Hypertext transfer protocol (HTTP)

Message Passing Interface (MPI)

User can add other protocols, for example a distributed shared memory protocol

# Transport Layer

---

## End-to-end communication

No application semantics – only process-to-process

## Examples

### Transmission control protocol (TCP)

provides reliable byte stream service using  
retransmission

flow control

congestion control

### User datagram protocol (UDP)

provides unreliable unordered datagram service

# Network Layer

---

## Host-to-host

Potentially across multiple networks

## Example: internet protocol (IP)

Understands the host address

Responsible for packet delivery

Provides routing function across the network

But can lose or misorder packets

## So, what did UDP add to IP?

# Network Layer

---

## Host-to-host

Potentially across multiple networks

## Example: internet protocol (IP)

Understands the host address

Responsible for packet delivery

Provides routing function across the network

But can lose or misorder packets

So, what did UDP add to IP? Port addressing, as opposed to simple host addressing

# Data Link/Physical Layer

---

Comes from the underlying network

Physical layer: transmits 0s and 1s over the wire

Data link layer: groups bits into frames and does error control using checksum + retransmission

## Examples

Ethernet

Myrinet

InfiniBand

DSL

Phone network

# Communication Hardware Characteristics: Circuit vs. Packet Switching

## Circuit switching

Example: telephony

Resources are reserved and *dedicated* during the connection

Fixed path between peers for the duration of the connection

## Packet switching

Example: internet

Entering data (variable-length messages) are divided into (fixed-length) packets

Packets in network share resources and may take different paths to the destination

# Network-Level Characteristics: Virtual Circuit vs. Datagram

## Virtual circuits

Cross between circuit and packet switching

Resources are reserved to a **logical** connection, but are **not** dedicated to the connection

Fixed path between peers for the duration of the connection

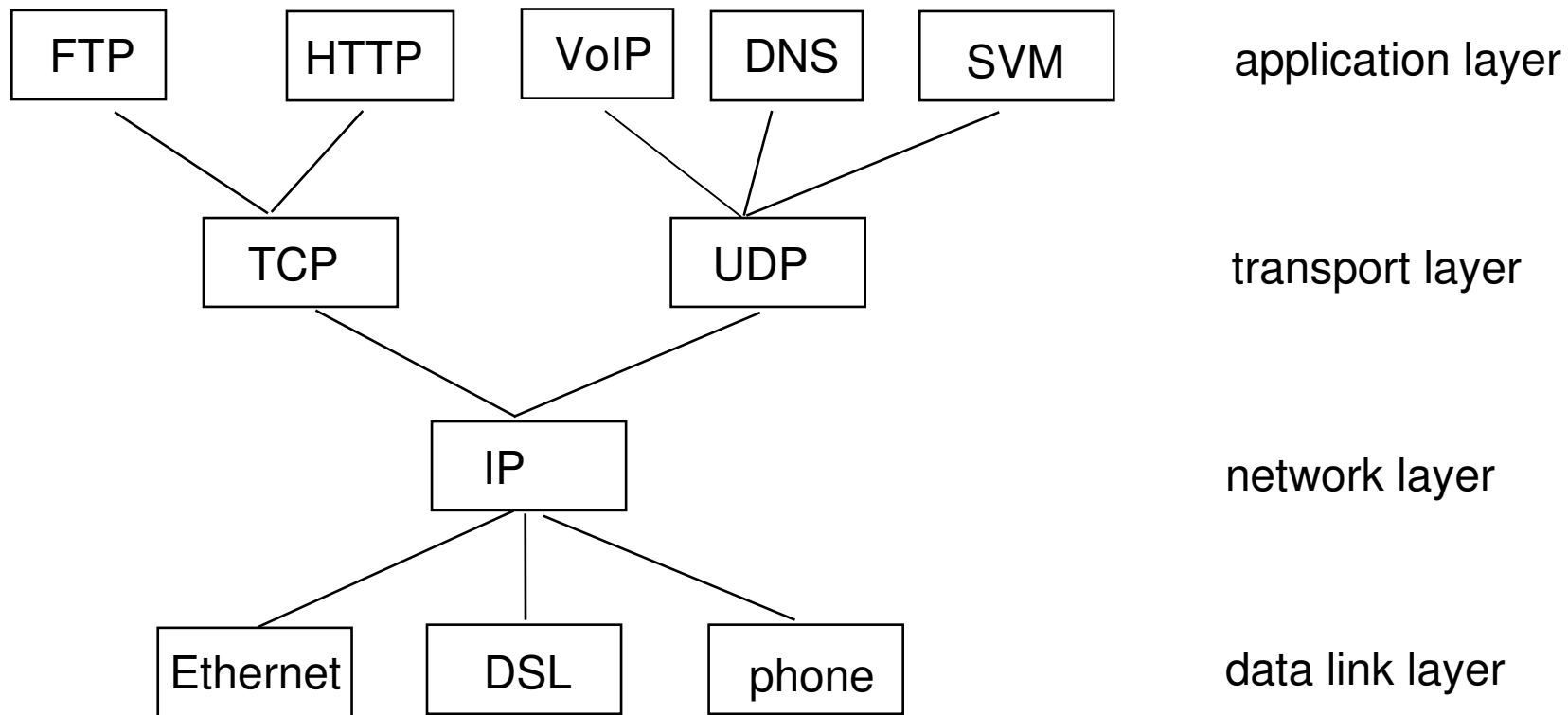
## Datagrams

The path for each message (datagram) is chosen only when the message is sent or received at an intermediate host

Separate messages may take different paths through the network

A datagram is broken into one or more packets for physical transmission

# Internet Hierarchy



# Details of the Network Layer Protocol

---

Addressing: how hosts are named

Service model: how hosts interact with the network, what is the packet format

Routing: how a route from source to destination is chosen

# IP Addressing

## Addresses

unique 32-bit address for each host

dotted-decimal notation: 128.112.102.65 (4 eight-bit numbers)

four address formats: class A (large nets), class B (medium nets), class C (small nets), and class D (multicast).

E.g., a class A address represents “network.local.local.local”, a class C address represents “network.network.network.local”.

## IP to physical address translation

each host only recognizes the physical address of its network interfaces

Address Resolution Protocol (ARP) to obtain the translation

each host caches a list of IP-to-physical translations which expires after a while

# ARP

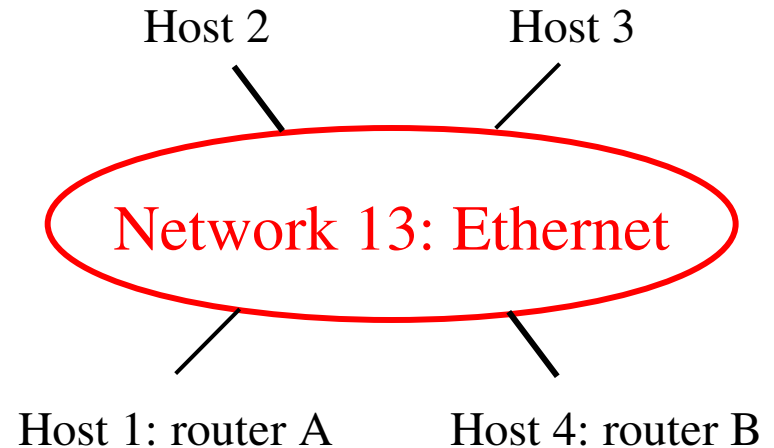
A host broadcasts (on a LAN) a query packet asking for a translation for some IP address

Hosts which know the translation reply

Each host knows its own IP and physical translation

Reverse ARP (RARP) translates physical to IP address and it is used to assign IP addresses dynamically. Has been replaced by the Dynamic Host Configuration Protocol (DHCP)

Router A wants to send an IP packet to router B. It uses ARP to obtain the physical address of router B



# IP Packet

IP transmits data in variable size chunks: datagrams

May drop, reorder, or duplicate datagrams

Each network has a Maximum Transmission Unit (MTU), which is the largest packet it can carry

If packet is bigger than MTU it is broken into fragments which are reassembled at destination

IP packet format:

- source and destination addresses

- time to live: decremented on each hop, packet dropped when TTL=0

- fragment information, checksum, other fields

# IP Routing

Each host has a routing table which tells it where to forward packets for each network, including a default router

How the routing table is maintained:

- two-level approach: intra-domain and inter-domain

- intra-domain: many approaches, ultimately call ARP

- inter-domain: many approaches, e.g. Boundary Gateway Protocol (BGP)

In BGP, each domain designates a “BGP speaker” to represent it

Speakers advertise which domains they can reach

Routing cycles avoided

# Details of the Transport Layer Protocol

---

## User Datagram Protocol (UDP): connectionless

unreliable, unordered datagrams

the main difference from IP: IP sends datagrams between hosts, UDP sends datagrams between processes identified as (host, port) pairs

## Transmission Control Protocol: connection-oriented

reliable; acknowledgment, timeout, and retransmission

byte stream delivered in order (datagrams are hidden)

flow control: slows down sender if receiver overwhelmed

congestion control: slows down sender if network overwhelmed

# TCP: Connection Setup

---

TCP is a connection-oriented protocol

three-way handshake:

client sends a SYN packet: “I want to connect”

server sends back its SYN + ACK: “I accept”

client acks the server’s SYN: “OK”

# TCP: Reliable Communication

Packets can get lost – retransmit when necessary

Each packet carries a sequence number

Sequence number: last byte of data sent before this packet

Receiver acknowledges data after receiving them

Ack up to last byte in contiguous stream received

Optimization: piggyback acks on normal messages

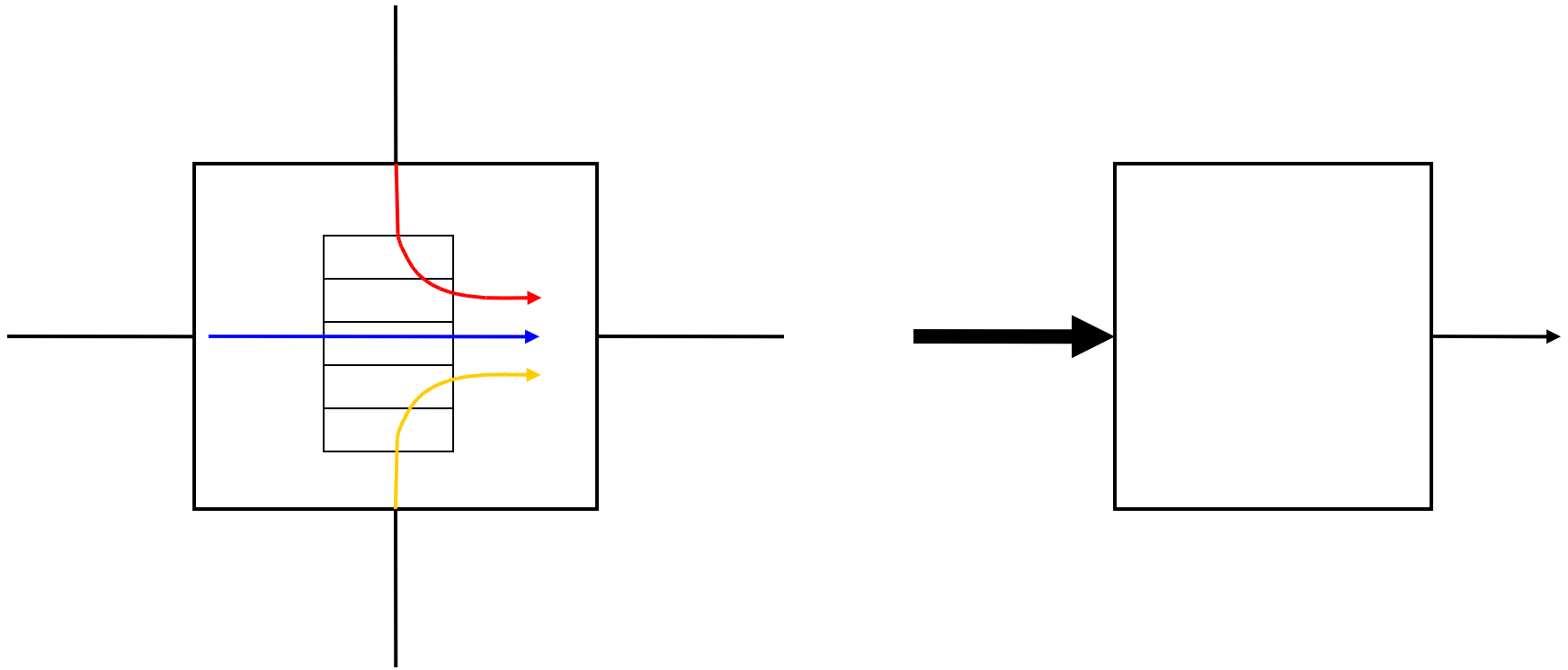
TCP keeps an average round-trip transmission time (RTT)

Timeout if no ack received after twice the estimated RTT and  
resend data starting from the last ack

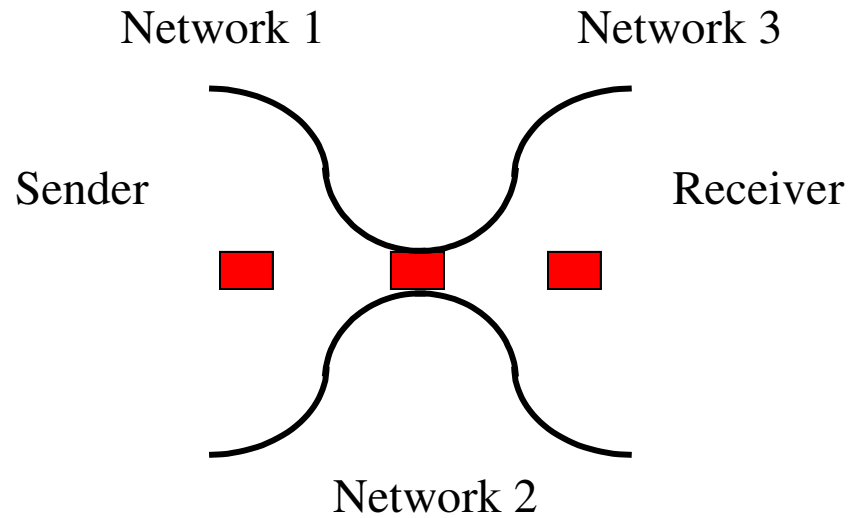
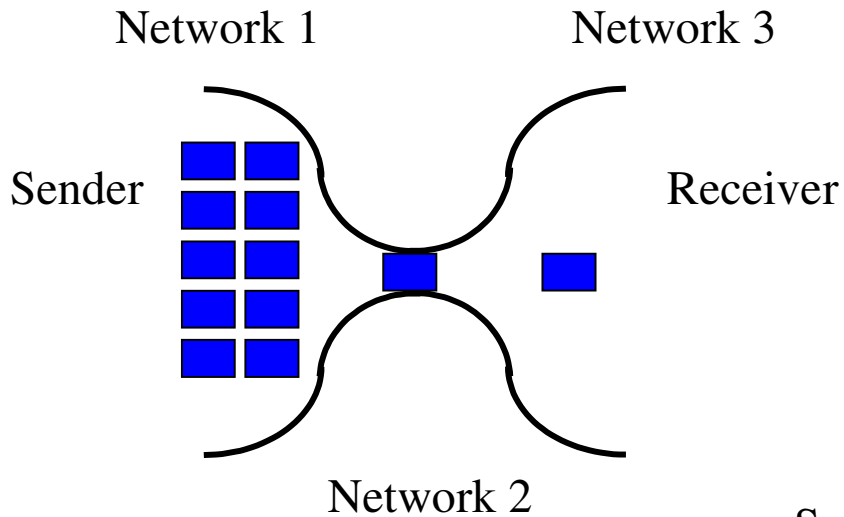
How to retransmit?

Delay sender until get ack? Make copy of data?

# The Need for Congestion Control



# TCP: Congestion Control



# TCP: Congestion Control

Basic idea: only put packets into the network as fast as they are exiting

To maintain high performance, however, have to keep the pipe full

Network capacity is equal to latency-bandwidth product

Really want to send network capacity before receiving an ack

After that, send more whenever get another ack

Keep network full of in-transit data

Only put into the net what is getting out the other end

This is the sliding window protocol

# TCP: Congestion Control

Detect network congestion then reduce amount being sent to alleviate congestion

Detecting congestion: TCP interprets a timeout waiting for an ACK as a symptom of congestion

Is this always right?

**Current approach: slow start + congestion avoidance**

Start by sending 1 packet, increase congestion window multiplicatively with each ACK until timeout. When timeout occurs, restart but make maximum window = current window/2. When window size reaches this threshold, start increasing window additively until timeout.

# Receiver's Window

## An additional complication:

Just because the network has a certain amount of capacity, doesn't mean the receiving host can buffer that amount of data

What if the receiver is not ready to read the incoming data?

## Receiver decides how much memory to dedicate to this connection

Receiver continuously advertises current window size (with ACKS) = allocated memory - unread data

Sender stops sending when the unack-ed data = receiver current window size

Transmission window =  $\min(\text{congestion window, receiver's window})$

# Remote Procedure Call

---

# Remote Procedure Call (RPC)

Transport protocols such as TCP/UDP provide un-interpreted messaging

One option is to simply use this abstraction for parallel/distributed programming

This is what is done in parallel programming because we can assume:

- Homogeneity

- Threads running on different nodes are part of the same computation, so easier to program

- Willing to trade-off some ease-of-programming for performance

Difficult to use this abstraction for distributed computing

- Heterogeneous system

- Different “trust domains”

# RPC (Cont'd)

## Why RPC?

Procedure call is an accepted and well-understood mechanism for control transfer within a program

Presumably, accepted is equivalent to “good” – clean semantics

Providing procedure call semantics for distributed computing makes distributed computing much more like programming on a single machine

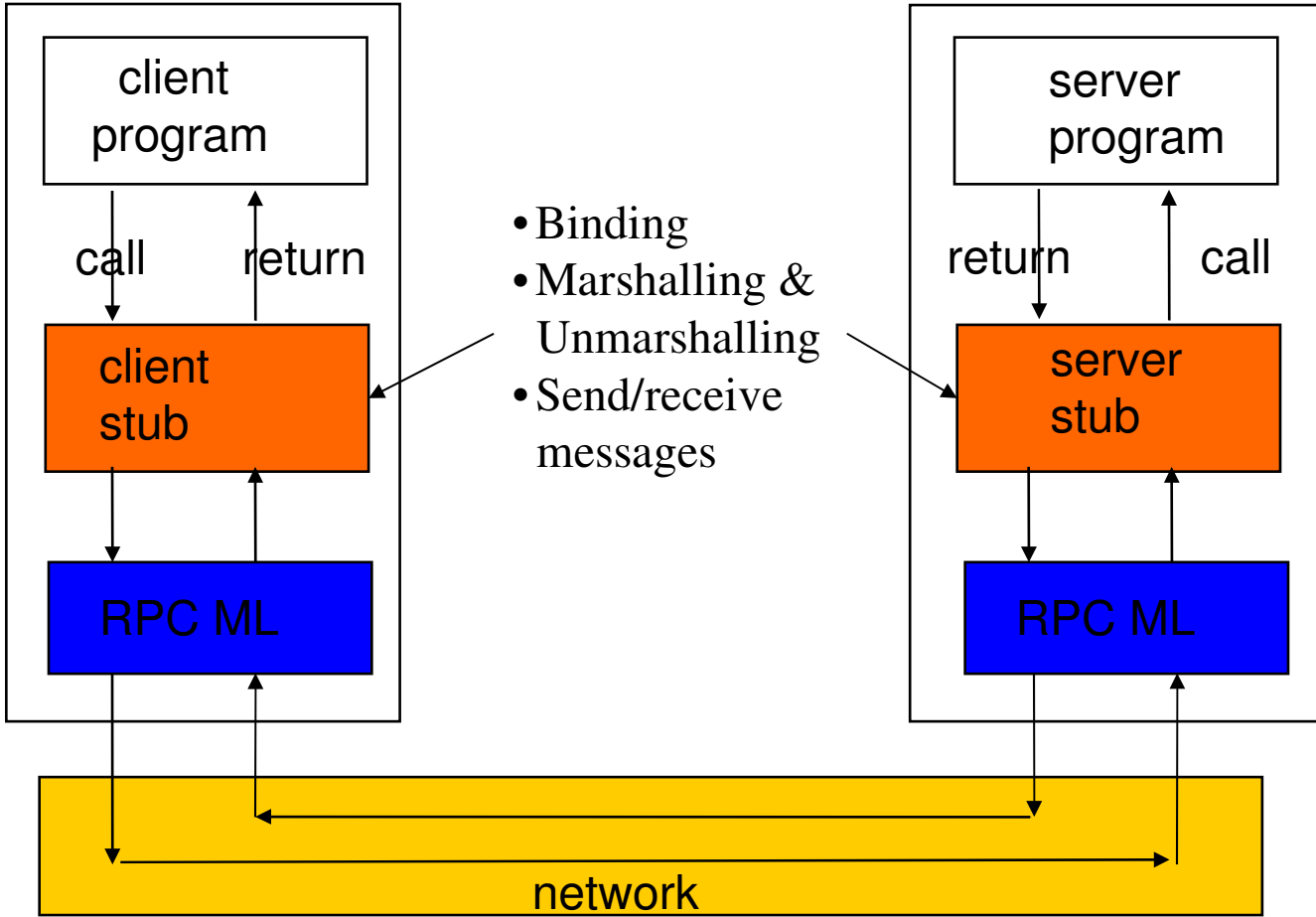
Don't have to worry about remote execution except ...

Abstraction helps to hide:

The possibly heterogeneous nature of the hardware platform

The fact that the distributed machines do not share memory

# RPC Structure



# RPC Structure (Cont'd)

Stubs make RPCs look “just” like normal procedure calls

## Binding

Naming

Location

## Marshalling & Unmarshalling

Translate internal data  $\leftrightarrow$  message representation

How to transmit pointer-based data structure (e.g. graph)?

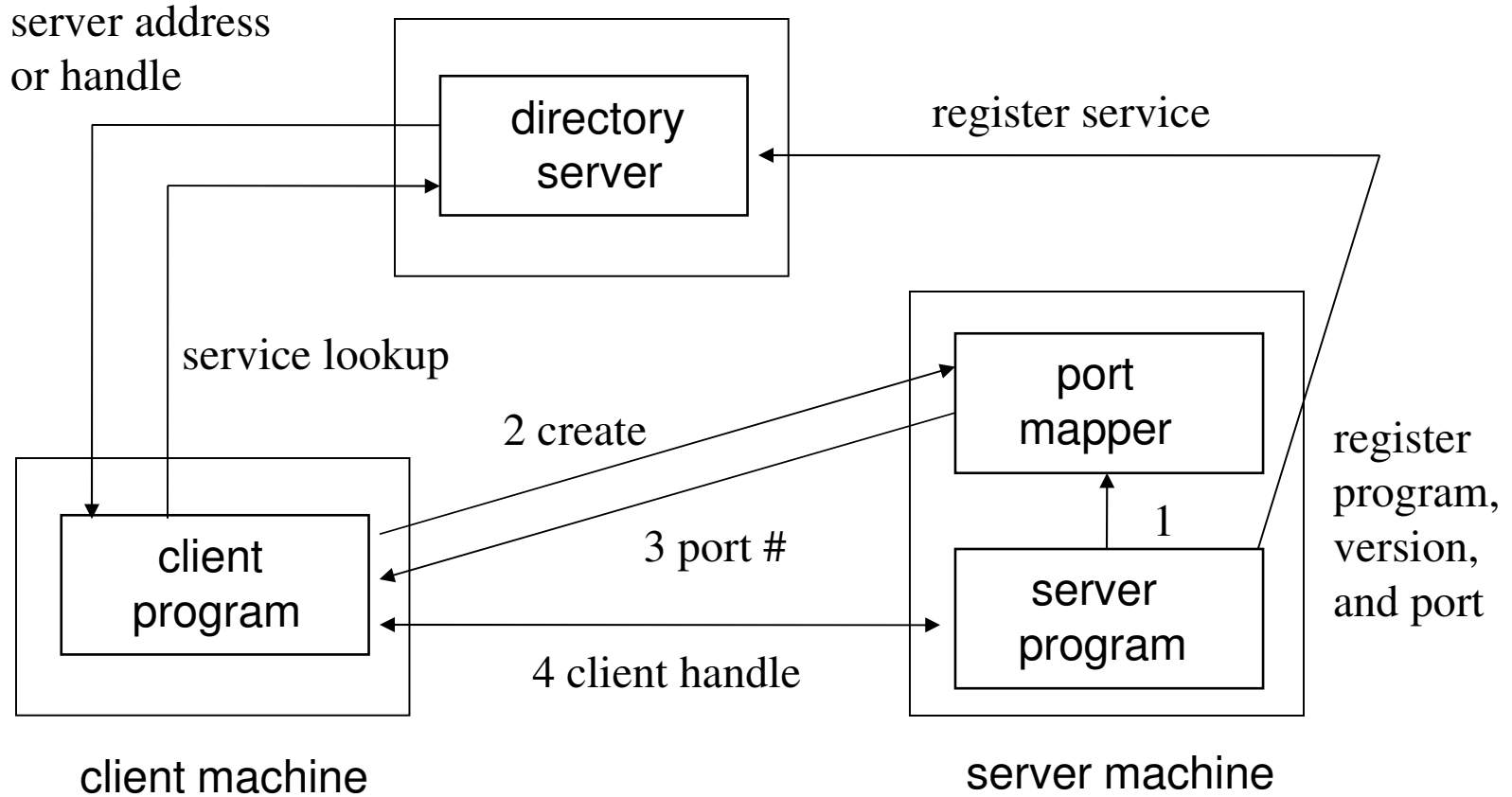
Serialization

How to transmit data between heterogeneous machines?

Virtual data types

## Send/receive messages

# RPC Binding



# Client Stub Example

---

```
void remote_add(Server s, int *x, int *y, int *sum) {  
    s.sendInt(AddProcedure);  
    s.sendInt(*x);  
    s.sendInt(*y);  
    s.flush()  
    status = s.receiveInt();  
    /* if no errors */  
    *sum = s.receiveInt();  
}
```

# Server Stub Example

```
void serverLoop(Client c) {
    while (1) {
        int Procedure = c_receiveInt();
        switch (Procedure) {

            case AddProcedure:
                int x = c.receiveInt();
                int y = c.receiveInt();
                int sum;
                add(x, y, &sum);
                c.sendInt(StatusOK);
                c.sendInt(sum);
                break;

        }
    }
}
```

# RPC Semantics

While goal is to make RPC look like local procedure call as much as possible, there are some differences in the semantics that cannot/should not be hidden

Global variables are not accessible inside the RPC

Call-by-copy/restore for reference-style params; call-by-value for others

Communication errors that may leave client uncertain about whether the call really happened

various semantics possible: at-least-once, at-most-once, exactly-once

difference is visible unless the call is idempotent, i.e. multiple executions of the call have the same effect (no side effects).  
E.g. read the first 1K bytes of a file.

# RPC Semantics

At-least-once: in case of timeouts, keep trying RPC until actually completes

At-most-once: try once and report failure after timeout period

Exactly-once: ideal but difficult to guarantee; one approach is to use at-least-once semantics and have a cache of previously completed operations on the server side; the cache has to be logged into stable storage

# Transactions

---

# Transactions

Next layer up in communication abstraction

A unit of computation that has the ACID properties

**Atomic:** each transaction either occurs completely or not at all – no partial results.

**Consistent:** when executed alone and to completion, a transaction preserves whatever invariants have been defined for the system state.

**Isolated:** any set of transactions is serializable, i.e. concurrent transactions do not interfere with each other.

**Durable:** effects of committed transactions should survive subsequent failures.

Can you see why this is a useful mechanism to support the building of distributed systems? Think of banking system

# Transactions

---

Transaction is a mechanism for both synchronization and tolerating failures

Isolation → synchronization

Atomicity, durability → failures

Isolation: two-phase locking

Atomicity: two-phase commit

Durability: stable storage and recovery

# Two-Phase Locking

For isolation, we need concurrency control by using locking, or more specifically, two-phase locking

Read/write locks to protect concurrent data

Mapping locks to data is the responsibility of the programmer

What happens if the programmer gets it wrong?

Acquire/release locks in two phases

Phase 1 (growing phase): acquire locks as needed

Phase 2 (shrinking phase): once release any lock, cannot acquire any more locks. Can only release locks from now on

# Two-Phase Locking

Usually, locks are acquired when needed (not at the beginning of the transaction, to increase concurrency), but held until transaction either commits or aborts – strict two-phase locking

Why? A transaction always reads a value written by a committed transaction

What about deadlock?

If process refrains from updating permanent state until the shrinking phase, failure to acquire a lock can be dealt with by releasing all acquired locks, waiting a while, and trying again (may cause livelock)

Other approaches: Order locks; Avoid deadlock; Detect & recover

If all transactions use two-phase locking, it can be proven that **all schedules formed by interleaving them are serializable (I in ACID)**

# Atomicity and Recovery

---

## 3 levels of storage

Volatile: memory

Nonvolatile: disk

Stable storage: mirrored disks or RAID

## 4 classes of failures

Transaction abort

System crash

Media failure (stable storage is the solution)

Catastrophe (no solution for this)

# Transaction Abort Recovery

---

Atomic property of transactions stipulates the undo of any modifications made by a transaction before it aborts

Two approaches

Update-in-place

Deferred-update

How can we implement these two approaches?

# Transaction Abort Recovery

Atomic property of transactions stipulates the undo of any modifications made by a transaction before it aborts

Two approaches

Update-in-place

Deferred-update

How can we implement these two approaches?

Update-in-place: write-ahead log and rollback if aborted

Deferred-update: private workspace

# System Crash Recovery

---

Maintain a log of initiated transaction records, aborts, and commits on nonvolatile (better yet, stable) storage

Whenever commits a transaction, force description of the transaction to nonvolatile (better yet, stable) storage

What happens after a crash?

# System Crash Recovery

---

Maintain a log of initiated transaction records, aborts, and commits on nonvolatile (better yet, stable) storage

Whenever commits a transaction, force description of the transaction to nonvolatile (better yet, stable) storage

What happens after a crash? State can be recovered by reading and undoing the non-committed transactions in the log (from end to beginning)

# Distributed Recovery

---

All processes (possibly running on different machines) involved in a transaction must reach a consistent decision on whether to commit or abort

Isn't this the consensus problem? How is this doable?

# Two-Phase Commit

Well, not quite the consensus problem – can unilaterally decide to abort. That is, system is not totally asynchronous

Two-phase commit protocol used to guarantee atomicity

Process attempting to perform transaction becomes “coordinator”

Protocol executes in two phases.

# Phase 1: Obtaining a Decision

$C_i$  adds  $\langle \text{prepare } T \rangle$  record to the log

$C_i$  sends  $\langle \text{prepare } T \rangle$  message to all sites

When a site receives a  $\langle \text{prepare } T \rangle$  message, the transaction manager determines if it can commit the transaction

If no: add  $\langle \text{no } T \rangle$  record to the log and respond to  $C_i$  with  $\langle \text{abort } T \rangle$

If yes:

add  $\langle \text{ready } T \rangle$  record to the log

force *all log records* for  $T$  onto stable storage

send  $\langle \text{ready } T \rangle$  message to  $C_i$

## Phase 1 (Cont)

---

### Coordinator collects responses

All respond “ready”,  
decision is *commit*

At least one response is “abort”,  
decision is *abort*

At least one participant fails to respond within time out  
period,  
decision is *abort*

## Phase 2: Recording Decision in the Database

---

Coordinator adds a decision record

$\langle \text{abort } T \rangle$  or  $\langle \text{commit } T \rangle$

to its log and forces record onto stable storage

Once that record reaches stable storage it is irrevocable  
(even if failures occur)

Coordinator sends a message to each participant  
informing it of the decision (commit or abort)

Participants take appropriate action locally

## Failure Handling in 2PC – Site Failure

The log contains a  $\langle \text{commit } T \rangle$  record

In this case, the site executes **redo**( $T$ )

The log contains an  $\langle \text{abort } T \rangle$  record

In this case, the site executes **undo**( $T$ )

The contains a  $\langle \text{ready } T \rangle$  record; consult  $C_i$

If  $C_i$  is down, site sends **query-status**  $T$  message to the other sites

The log contains no control records concerning  $T$

In this case, the site executes **undo**( $T$ )

## Failure Handling in 2PC – Coordinator $C_i$ Failure

If an active site contains a  $\langle \text{commit } T \rangle$  record in its log, the  $T$  must be committed

If an active site contains an  $\langle \text{abort } T \rangle$  record in its log, then  $T$  must be aborted

If some active site does *not* contain the record  $\langle \text{ready } T \rangle$  in its log then the failed coordinator  $C_i$  cannot have decided to commit  $T$

Rather than wait for  $C_i$  to recover, it is preferable to abort  $T$

All active sites have a  $\langle \text{ready } T \rangle$  record in their logs, but no additional control records

In this case we must wait for the coordinator to recover

Blocking problem –  $T$  is blocked pending the recovery of site  $S_i$

# Transactions – What's the Problem?

---

Transaction seems like a very useful mechanism for distributed computing

Why is it not used everywhere?

# Transactions – What's the Problem?

---

Transaction seems like a very useful mechanism for distributed computing

Why is it not used everywhere? ACID properties are not always required. Weaker semantics can improve performance. Examples: when all operations in the distributed system are idempotent/read only (BitTorrent-style systems) or non-critical (search engine results)

# Distributed Algorithms

---

Have already talked about consensus and coordinated attack problems

Now:

- Happened-before relation

- Distributed mutual exclusion

- Distributed elections

- Distributed deadlock prevention and avoidance

- Distributed deadlock detection

# Happened-Before Relation

It is sometimes important to determine an ordering of events in a distributed system. Example: resources can only be used after they are granted.

The happened-before relation ( $\rightarrow$ ) provides a partial ordering of events

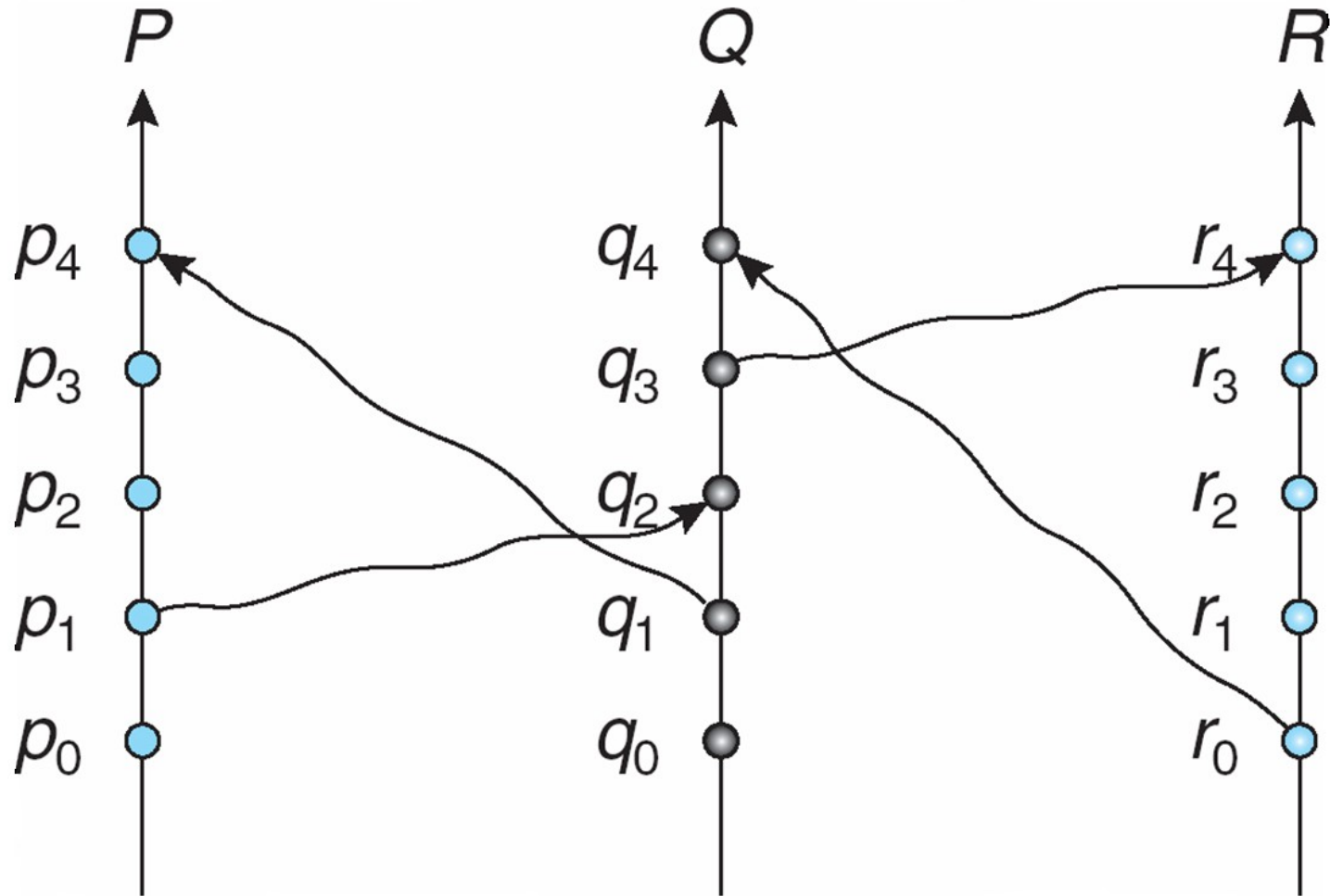
If A and B are events in the same process, and A was executed before B, then  $A \rightarrow B$

If A is the event of sending a msg by one process and B is the event of receiving the msg by another, then  $A \rightarrow B$

If  $A \rightarrow B$  and  $B \rightarrow C$ , then  $A \rightarrow C$

If events A and B are not related by the  $\rightarrow$  relation, they executed “concurrently”

## Relative Time for Three Concurrent Processes



# Achieving Global Ordering

Common or synchronized clock not available, so use “timestamps” to achieve global ordering

Global ordering requirement: If  $A \rightarrow B$ , then the timestamp of A is less than the timestamp of B

The timestamp can take the value of a logical clock, i.e. a simple counter that is incremented between any two successive events executed within a process

If event A was executed before B in a process, then  $LC(A) < LC(B)$

If A is the event of receiving a msg with timestamp t and  $LC(A) < t$ , then  $LC(A) = t + 1$

If  $LC(A)$  in one process i is the same as  $LC(B)$  in another process j, then use process ids to break ties and create a total ordering

# Distributed Mutual Exclusion

Centralized approach: one process chosen as coordinator. Each process that wants to enter the CS sends a request msg to the coordinator. When process receives a reply msg, it can enter the CS. After exiting the CS, the process sends a release msg to the coordinator. The coordinator queues requests that arrive while some process is in the CS.

Properties? Ensures mutual exclusion? Performance?  
Starvation? Fairness? Reliability?

If coordinator dies, an election has to take place (will talk about this soon)

# Distributed Mutual Exclusion

Fully distributed approach: when a process wants to enter the CS, it generates a new timestamp TS and sends the message *request(Pi,TS)* to all other processes, including itself. When the process receives all replies, it can enter the CS, queuing incoming requests and deferring them. Upon exit of the CS, the process can reply to all its deferred requests.

Three rules when deciding whether a process should reply immediately to a request:

If process in CS, then it defers its reply

If process does not want to enter CS, then it replies immediately

If process does want to enter CS, then it compares its own request timestamp with the timestamp of the incoming request. If its own request timestamp is larger, then it replies immediately. Otherwise, it defers the reply

Properties? Ensures mutual exclusion? Performance? Starvation? Fairness?  
Reliability?

## DME: Fully Distributed Approach (Cont)

The decision whether process  $P_j$  replies immediately to a  $request(P_i, TS)$  message or defers its reply is based on three factors:

If  $P_j$  is in its critical section, then it defers its reply to  $P_i$

If  $P_j$  does *not* want to enter its critical section, then it sends a *reply* immediately to  $P_i$

If  $P_j$  wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp  $TS$

If its own request timestamp is greater than  $TS$ , then it sends a *reply* immediately to  $P_i$  ( $P_i$  asked first)

Otherwise, the reply is deferred

## Desirable Behavior of Fully Distributed Approach

Freedom from Deadlock is ensured

Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering

The timestamp ordering ensures that processes are served in a first-come, first served order

The number of messages per critical-section entry is

$$2 \times (n - 1)$$

# Three Undesirable Consequences

---

The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex

If one of the processes fails, then the entire scheme collapses

This can be dealt with by continuously monitoring the state of all the processes in the system

Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section

This protocol is therefore suited for small, stable sets of cooperating processes

# Distributed Mutual Exclusion

Token passing approach: idea is to circulate a token (a special message) around the system. Possession of the token entitles the holder to enter the CS. Processes logically organized in a ring structure.

Properties? Ensures mutual exclusion? Performance?  
Starvation? Fairness? Reliability?

If token is lost, then election is necessary to generate a new token.

If a process fails, a new ring structure has to be established.

# Distributed Deadlock Avoidance

The deadlock prevention and avoidance algorithms we talked about before can also be used in distributed systems.

Prevention: resource ordering of all resources in the system.  
Simple and little overhead.

Avoidance: Banker's algorithm. High overhead (too many msgs, centralized banker) and excessively conservative.

New deadlock avoidance algorithms: wait-die and wound-wait.  
Idea is to avoid circular wait. Both use timestamps assigned to processes at creation time.

# Wait-Die Scheme

Based on a nonpreemptive technique

If  $P_i$  requests a resource currently held by  $P_j$ ,  $P_i$  is allowed to wait only if it has a smaller timestamp than does  $P_j$  ( $P_i$  is older than  $P_j$ )

Otherwise,  $P_i$  is rolled back (dies)

Example: Suppose that processes  $P_1$ ,  $P_2$ , and  $P_3$  have timestamps 5, 10, and 15 respectively

if  $P_1$  request a resource held by  $P_2$ , then  $P_1$  will wait

If  $P_3$  requests a resource held by  $P_2$ , then  $P_3$  will be rolled

# Wound-Wait Scheme

Based on a preemptive technique; counterpart to the wait-die system

If  $P_i$  requests a resource currently held by  $P_j$ ,  $P_i$  is allowed to wait only if it has a larger timestamp than does  $P_j$  ( $P_i$  is younger than  $P_j$ ).

Otherwise  $P_j$  is rolled back ( $P_j$  is wounded by  $P_i$ )

Example: Suppose that processes  $P_1$ ,  $P_2$ , and  $P_3$  have timestamps 5, 10, and 15 respectively

If  $P_1$  requests a resource held by  $P_2$ , then the resource will be preempted from  $P_2$  and  $P_2$  will be rolled back

If  $P_3$  requests a resource held by  $P_2$ , then  $P_3$  will wait

# Distributed Deadlock Detection

---

Problem: Above schemes may result in too many rollbacks

Deadlock detection eliminates this problem.

Deadlock detection is based on a wait-for graph describing the resource allocation state. Assuming a single resource of each type, a cycle in the graph represents a deadlock.

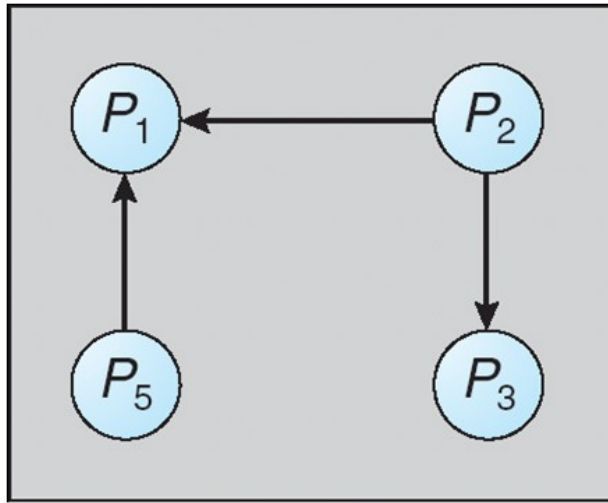
Problem is how to maintain the wait-for graph.

## Wait-for graphs

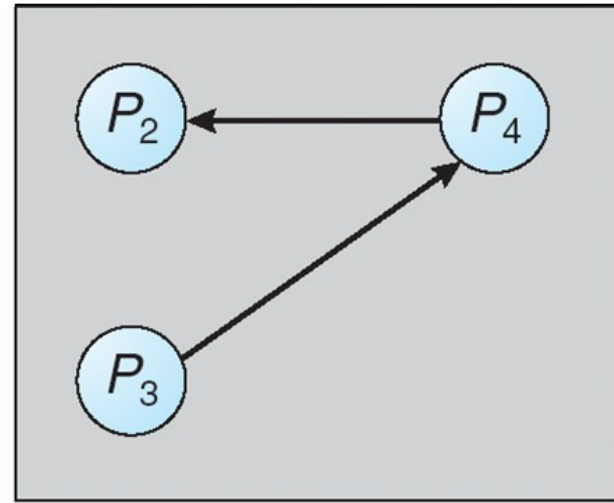
Local wait-for graphs at each local site. The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site

May also use a global wait-for graph. This graph is the union of all local wait-for graphs.

# Two Local Wait-For Graphs

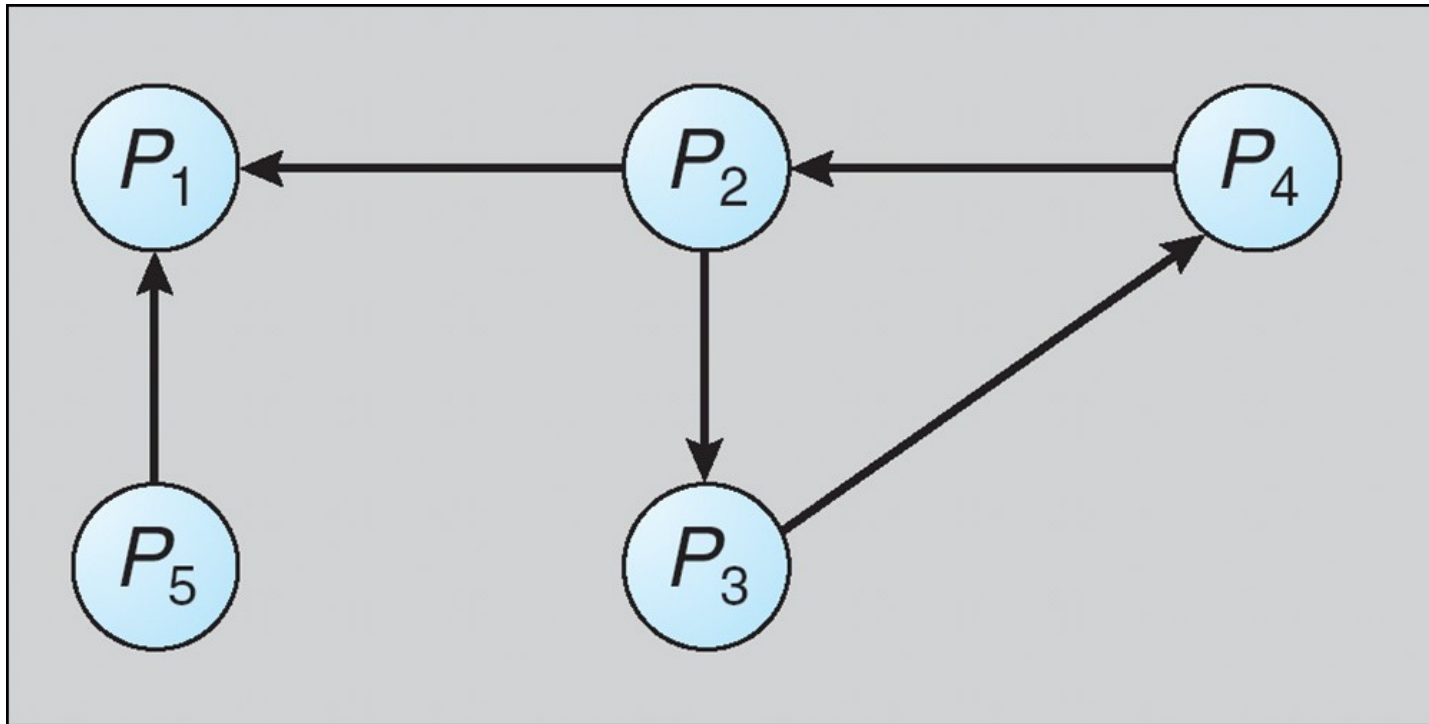


site  $S_1$



site  $S_2$

# Global Wait-For Graph



# Deadlock Detection – Centralized Approach

Each site keeps a local wait-for graph

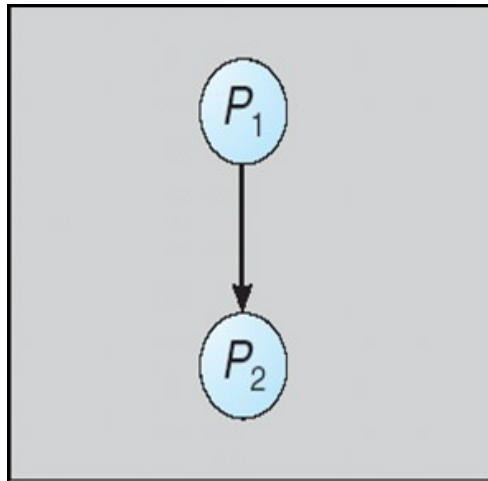
A global wait-for graph is maintained in a single coordination process

There are three different options (points in time) when the wait-for graph may be constructed:

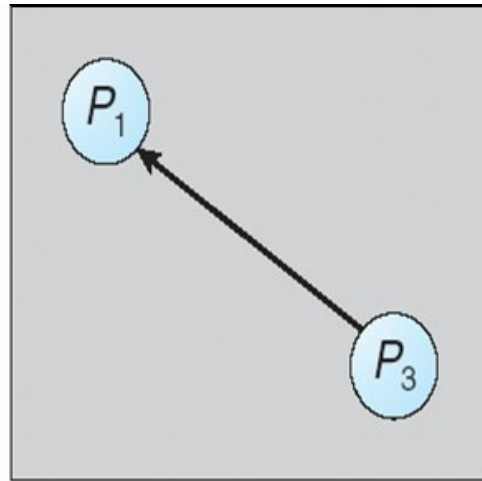
1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
2. Periodically, when a number of changes have occurred in a wait-for graph
3. Whenever the coordinator needs to invoke the cycle-detection algorithm

Unnecessary rollbacks may occur as a result of false cycles

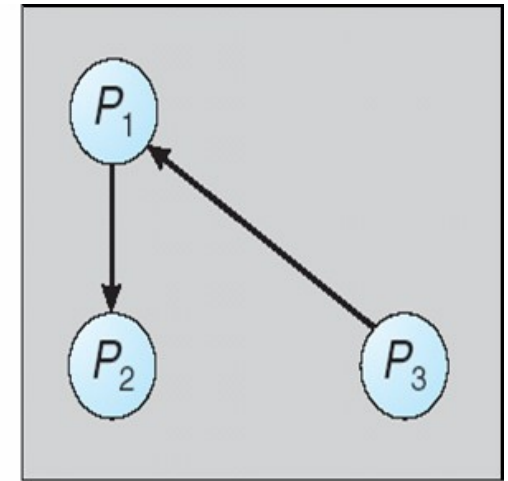
# False cycles example



site  $S_1$



site  $S_2$



coordinator

Suppose we have delete  $P_1 \rightarrow P_2$  and insert  $P_2 \rightarrow P_3$ ,  
but messages arrive in opposite order?

## Detection Algorithm Based on Option 3

Append unique identifiers (timestamps) to requests from different sites

When process  $P_i$ , at site  $A$ , requests a resource from process  $P_j$ , at site  $B$ , a request message with timestamp  $TS$  is sent

The edge  $P_i \rightarrow P_j$  with the label  $TS$  is inserted in the local wait-for of  $A$ . The edge is inserted in the local wait-for graph of  $B$  only if  $B$  has received the request message and cannot immediately grant the requested resource

# The Algorithm

1. The controller sends an initiating message to each site in the system
2. On receiving this message, a site sends its local wait-for graph to the coordinator
3. When the controller has received a reply from each site, it constructs a graph as follows:
  - (a) The constructed graph contains a vertex for every process in the system
  - (b) The graph has an edge  $P_i \rightarrow P_j$  if and only if  
there is an edge  $P_i \rightarrow P_j$  in one of the wait-for graphs,  
or  
an edge  $P_i \rightarrow P_j$  with some label  $TS$  appears in more than one wait-for graph

# Fully Distributed Approach

All controllers share equally the responsibility for detecting deadlock

Every site constructs a wait-for graph that represents a part of the total graph

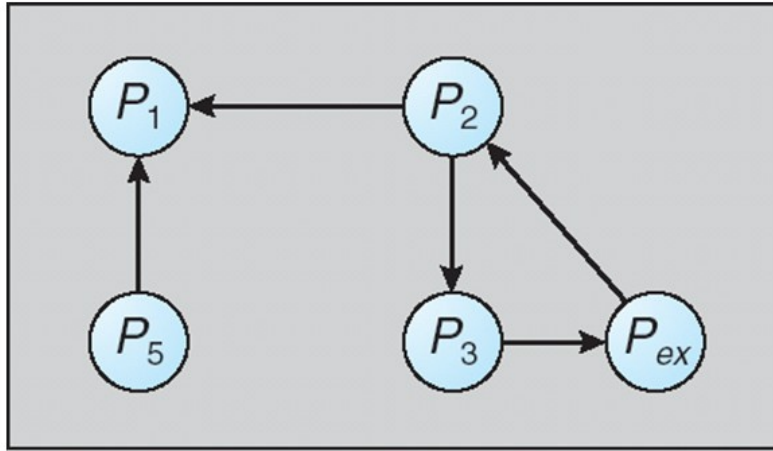
We add one additional node  $P_{ex}$  to each local wait-for graph

If a local wait-for graph contains a cycle that does not involve node  $P_{ex}$ , then the system is in a deadlock state

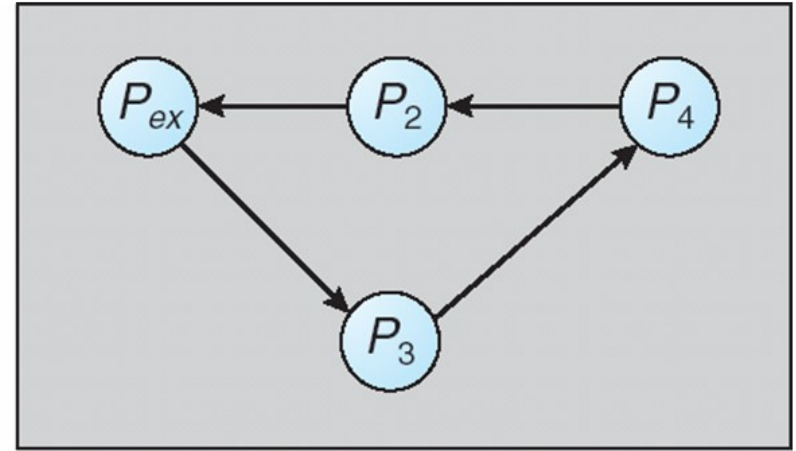
A cycle involving  $P_{ex}$  implies the possibility of a deadlock

To ascertain whether a deadlock does exist, a distributed deadlock-detection algorithm must be invoked

# Augmented Local Wait-For Graphs

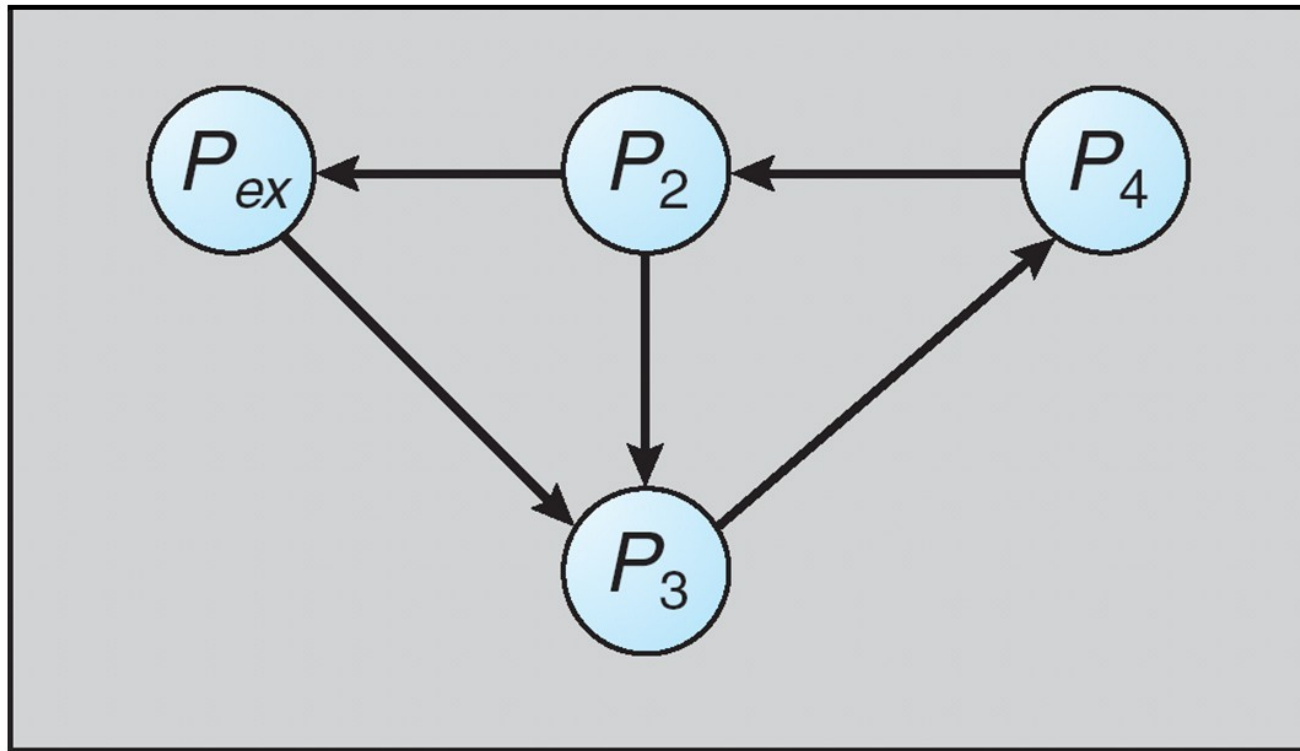


site  $S_1$



site  $S_2$

## Augmented Local Wait-For Graph in Site $S_2$



site  $S_2$

# Distributed Elections

Several algorithms that we saw depended on a coordinator process.

Election algorithms assume that a unique priority number is associated with each process (the process id to simplify matters). The algorithms elect the active process with the largest priority number as the coordinator. This number must be sent to each active process in the system. The algorithms provide a mechanism for a recovered process to identify the current coordinator.

# Distributed Elections

The Bully algorithm: suppose that a process sends a request that is not answered by the coordinator within an interval  $T$ . In this situation, the coordinator is assumed to have failed and the process tries to elect itself as the new coordinator.

Process  $P_i$  sends an election msg to every process  $P_j$  with a higher priority number ( $j > i$ ). Process  $P_i$  waits for a time  $T$  for an answer from any of those processes.

If no response is received, all processes  $P_j$  are assumed to have failed and  $P_i$  elects itself as the new coordinator.  $P_i$  starts a copy of the coordinator and sends a msg to all processes with priority less than  $i$  informing them that it is the new coordinator.

If a response is received,  $P_i$  begins a time interval  $T'$ , waiting to receive a msg informing it that a process with a higher priority number has been elected. If no such msg is received, the process with higher priority is assumed to have failed, and  $P_i$  re-starts the algorithm.

# Distributed Elections

---

The process that completes its algorithm has the highest number and is elected the coordinator. It has sent its number to all other active processes. After a process recovers, it immediately starts the algorithm. If there are no active processes with higher numbers, the recovering process becomes the coordinator (even if the current coordinator is still active).

# Distributed File Systems

---

# File Service

Implemented by a user/kernel process called file server

A system may have one or several file servers running at the same time

## Two models for file services

upload/download: files move between server and clients, few operations (read file & write file), simple, requires storage at client, good if whole file is accessed

remote access: files stay at server, rich interface with many operations, less space at client, efficient for small accesses

# Directory Service

Provides naming usually within a hierarchical file system

Clients can have the same view (global root directory) or different views of the file system (remote mounting)

Location transparency: location of the file doesn't appear in the name of the file

ex: `/server1/dir1/file` specifies the server but not where the file is located  
→ server can move the file in the network without changing the path

Location independence: a single name space that looks the same on all machines, files can be moved between servers without changing their names -> difficult

# Two-Level Naming

Symbolic name (external), e.g. prog.c; binary name (internal), e.g. local i-node number as in Unix

Directories provide the translation from symbolic to binary names

## Binary name format

i-node: no cross references among servers

(server, i-node): a directory in one server can refer to a file on a different server

Capability specifying address of server, number of file, access permissions, etc

{binary\_name+}: binary names refer to the original file and all of its backups

# File Sharing Semantics

## UNIX semantics: total ordering of R/W events

easy to achieve in a non-distributed system

in a distributed system with one server and multiple clients with no caching at client, total ordering is also easily achieved since R and W are immediately performed at server

## Session semantics: writes are guaranteed to become visible only when the file is closed

allow caching at client with lazy updating → better performance

if two or more clients simultaneously write: one file (last one or non-deterministically) replaces the other

# File Sharing Semantics (cont'd)

Immutable files: create and read file operations (no write)

writing a file means to create a new one and enter it into the directory replacing the previous one with the same name:  
atomic operations

collision in writing: last copy or non-deterministically  
what happens if the old copy is being read?

Transaction semantics: mutual exclusion on file accesses; either all file operations are completed or none is. Good for banking systems

# Server System Structure

---

File + directory service: combined or not

Cache directory hints at client to accelerate the path name look up – directory and hints must be kept coherent

State information about clients at the server

stateless server: no client information is kept between requests

stateful server: servers maintain state information about clients between requests

# Stateless vs. Stateful Servers

---

## Stateless Server

- requests are self-contained
- better fault tolerance
- open/close at client (fewer msgs)
- no space reserved for tables
- thus, no limit of open files

## Stateful Server

- shorter messages
- better performance (info in memory until close)
- open/close at server
- file locking possible
- read ahead possible

# Caching

Three possible places: server's memory, client's disk, client's memory

Caching in server's memory: avoids disk access but still have network access

Caching at client's disk (if available): tradeoff between disk access and remote memory access

Caching at client in main memory

- inside each process address space: no sharing at client

- in the kernel: kernel involvement on hits

- in a separate user-level cache manager: flexible and efficient if paging/sharing can be controlled from user-level

Server-side caching eliminates coherence problem. Client-side cache coherence? Next...

# Client Cache Coherence in DFS

How to maintain coherence (according to a model, e.g. session semantics) of copies of the same file at various clients

Write-through: writes sent to the server as soon as they are performed at the client → high traffic, requires cache managers to check (modification time) with server before can provide cached content to any client

Delayed write: coalesces multiple writes; better performance but ambiguous semantics

Write-on-close: implements session semantics

Central control: file server keeps a directory of open/cached files at clients and sends invalidations → Unix semantics, but problems with robustness and scalability

# File Replication

Multiple copies are maintained, each copy on a separate file server - multiple reasons:

Increase availability: file accessible even if a server is down

Increase reliability: file accessible even if a server loses its copy

Improve scalability: reduce the contention by splitting the workload over multiple servers

## Replication transparency

explicit file replication: programmer controls replication

lazy file replication: copies made by the server in background

use group communication: all copies made at the same time in the foreground

How replicas should be modified? Next...

# Modifying Replicas: Voting Protocol

Updating all replicas using a coordinator works but is not robust (if coordinator is down, no updates can be performed) => Voting: updates (and reads) can be performed if some specified # of servers agree.

## Voting Protocol:

A version # (incremented at write) is associated with each file

To perform a read, a client has to assemble a read quorum of  $N_r$  servers; similarly, a write quorum of  $N_w$  servers for a write

If  $N_r + N_w > N$ , then any read quorum will contain at least one most recently updated file version

For reading, client contacts  $N_r$  active servers and chooses the file with largest version #

For writing, client contacts  $N_w$  active servers asking them to write. Succeeds if they all say yes.

# Modifying Replicas: Voting Protocol

$N_r$  is usually small (reads are frequent), but  $N_w$  is usually close to  $N$  (want to make sure all replicas are updated). Problem with achieving a write quorum in the face of server failures

Voting with ghosts: allows to establish a write quorum when several servers are down by temporarily creating dummy (ghost) servers (at least one must be real)

Ghost servers are not permitted in a read quorum (they don't have any files)

When server comes back, it must restore its copy first by obtaining a read quorum. Only after the copy is restored, can it return to operation

A membership service keeps track of all servers, creates ghosts for crashed servers, and eliminates ghosts when servers reboot

# Network File System (NFSv3)

A stateless DFS from Sun; only state is map of handles to files

An NFS server exports directories

Clients access exported directories by mounting them

Because NFS is stateless, OPEN and CLOSE RPCs are not provided by the server (implemented at the client); clients need to block on close until all dirty data are stored on disk at the server

NFS provides file locking (separate network lock manager protocol) but UNIX semantics is not achieved due to client caching

- dirty cache blocks are sent to server in chunks, every 30 sec or at close

- a timer is associated with each cache block at the client (3 sec for data blocks, 30 sec for directory blocks). When the timer expires, the entry is discarded (if clean, of course)

- when a file is opened, last modification time at the server is checked

# Network File System (NFSv4)

NFSv4 is stateful; implements OPEN and CLOSE

Locking protocol has been integrated into the protocol

“Open Delegation” allows client itself to handle OPEN, CLOSE, and locking operations. Eliminates traffic for OPEN, CLOSE, locking, and cache consistency checks

Delegations have leases just like locks. Delegations may be revoked (when another client accesses the same file for writing) using a “callback” from the server to the client

Compound RPC allows client to send many operations as a single request to server. Server replies for the operations are also grouped into one reply to client

# The End

---