Review

殷亚凤

Email: yafeng@nju.edu.cn

Homepage: http://cs.nju.edu.cn/yafeng/

Room 301, Building of Computer Science and Technology

内容安排

- 概述
- 体系结构
- 进程
- **通信**
- **同步化**
- **一致性和复制**
- **容错**
- 基于对象的分布式系统
- 分布式文件系统
- 基于Web的分布式系统

考试

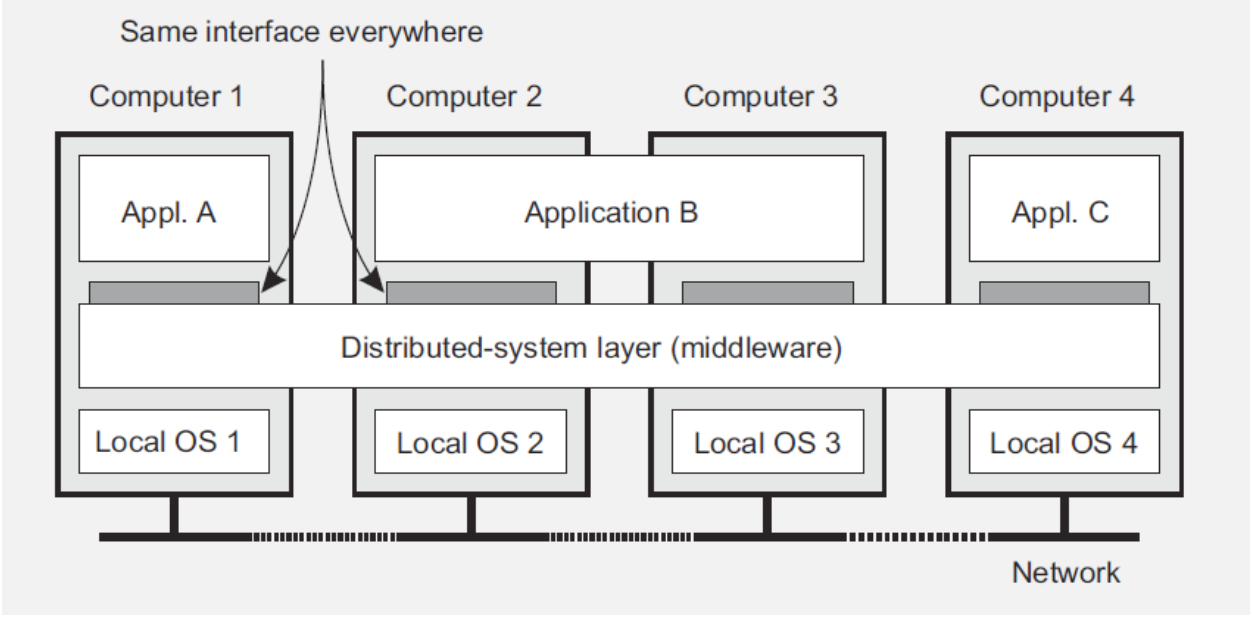
- 没有选择、判断等客观题，题型包括简答、分析、计算题等！

概述 （5分）

- 分布式系统的定义？如何从硬件和软件角度理解分布式系统的内容？
- 分布式系统的设计的4个关键目标？
- 事务处理具有的四个特性？

Distributed Systems: Definition

- A distributed system is a collection of *autonomous computing elements* that appears to its users as a *single coherent system*.



Distributed Information Systems

- The vast amount of distributed systems in use today are forms of traditional information systems, that now integrate legacy systems.
  - Example: Transaction processing systems.
- A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID)
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Goals of Distributed Systems

- Making resources available
- Distribution transparency
- Openness
- Scalability

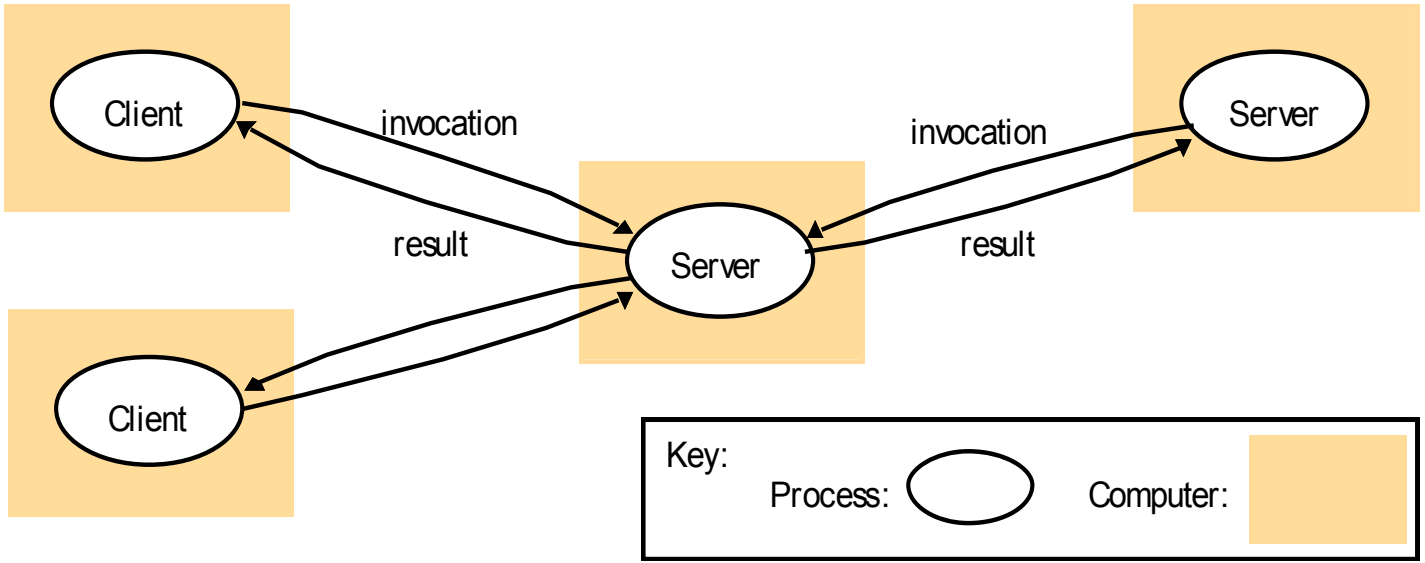
体系结构（5分）

- 分布式系统的体系结构有哪几种？简要解释。

Organization

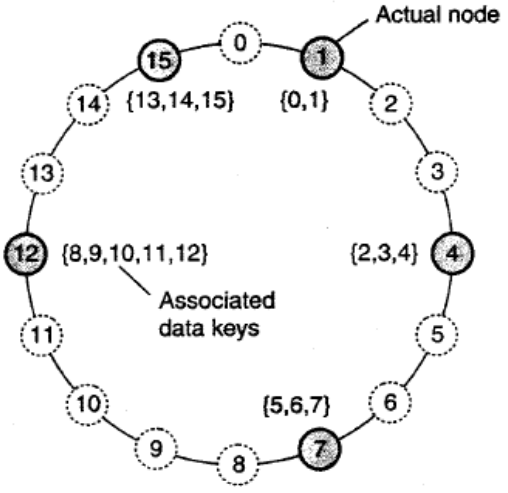
- Centralized
- Decentralized
- Hybrid

Clients Invoke Individual Servers



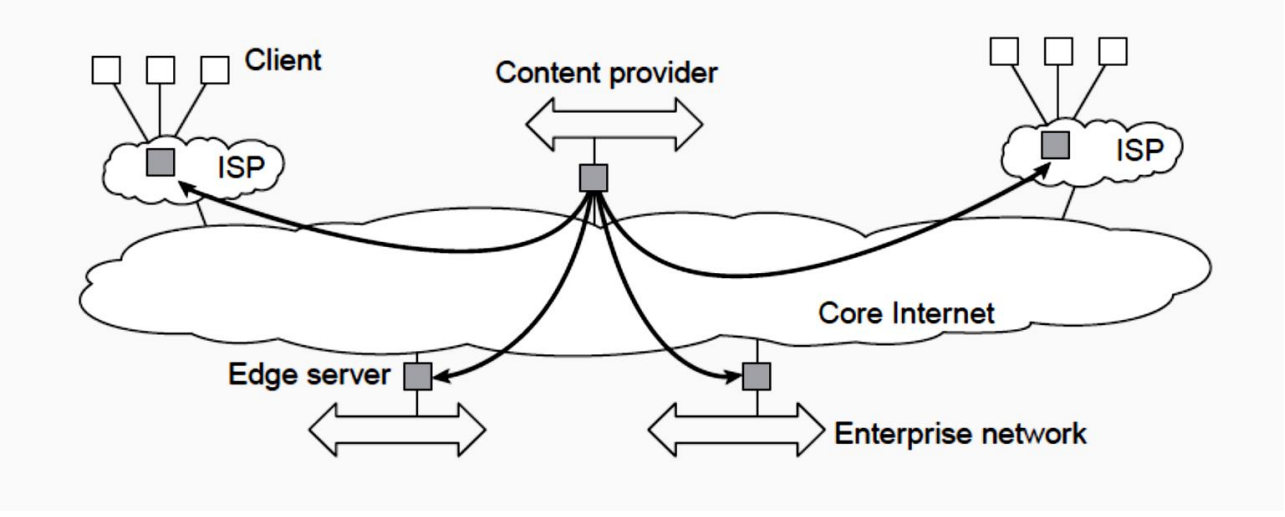
Structured P2P Systems

- Organize the nodes in a structured overlay network such as a logical ring, or a hypercube, and make specific nodes responsible for services based only on their ID.



Hybrid Architectures

- Client-server combined with P2P
- Edge-server architectures, which are often used for Content Delivery Networks.



进程 （10分）

- 什么是进程？进程、程序的区别？
- 为什么要引入线程，分布式系统中进程和线程的联系与区别？
- 用户级线程的主要优势和缺陷？如何解决它的缺陷？
- 有状态服务器、无状态服务器？对于状态无关的服务器使之维持状态的方法？
- 迁移代码时，根据本地资源方式的不同，如何采用不同的做法？

What is a Process?

- Process: An **execution stream** in the context of a **process state**
- **Execution stream**
  - Stream of executing instructions
  - Running piece of code
  - Sequential sequence of instructions
  - "thread of control"
- **Process state**
  - Everything that the running code can affect or be affected by

Processes vs. Programs

- A process is different than a program
  - Program: Static code and static data
  - Process: Dynamic instance of code and data

- No one-to-one mapping between programs and processes
  - **One process** can execute **multiple programs**
  - **One program** can invoke **multiple processes**

Low Performance

- **Resource management**
  - When creating a new process, assign address space, copy data
- **Scheduling**
  - Context switch
    - Process Context: CPU context and storage context
- **Cooperation**
  - IPC, interprocess communication
  - Shared memory

Introduction to Threads

- **Thread**: A minimal software processor in whose context a series of instructions can be executed.

- **Saving** a thread context implies stopping the current execution and saving **all the data needed to** continue the execution at a later stage.

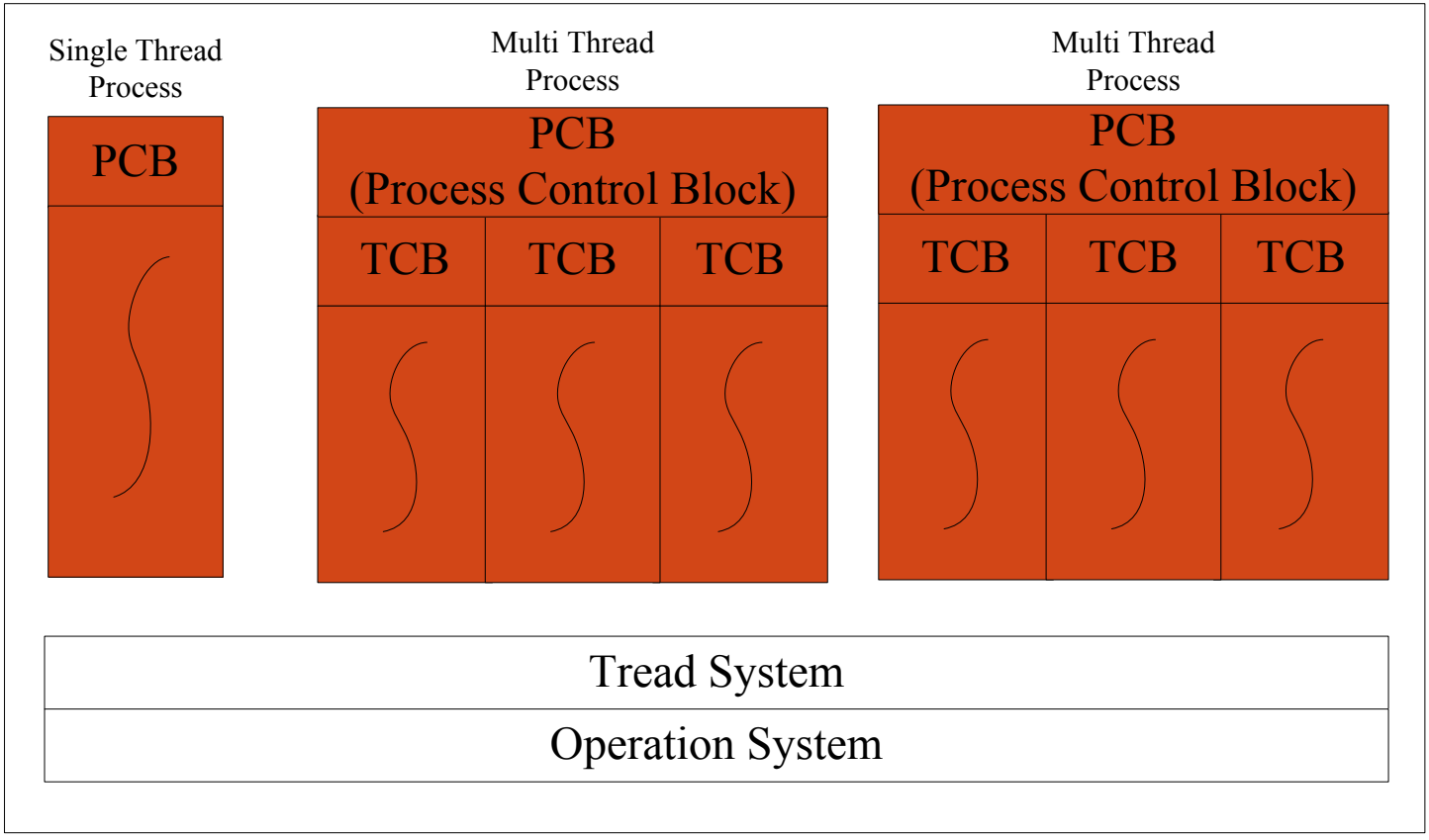
Process and thread : Context Switching

- **Processor context**: The minimal collection of values stored in the **registers of a processor** used for the execution of a series of **instructions** (e.g., stack pointer, addressing registers, program counter).
- **Thread context**: The minimal collection of values stored in **registers and memory**, used for the execution of a series of **instructions** (i.e., processor context, state).
- **Process context**: The minimal collection of values stored in **registers and memory**, used for the execution of a **thread** (i.e., thread context, but now also at least MMU register values).

Context Switching

- **Threads** share the same address space. Thread context switching can be done entirely independent of the operating system.
- **Process** switching is generally more expensive as it involves getting the OS in the loop, i.e., to the kernel.
- **Creating and destroying** threads is much cheaper than doing so for processes.

Thread and Process



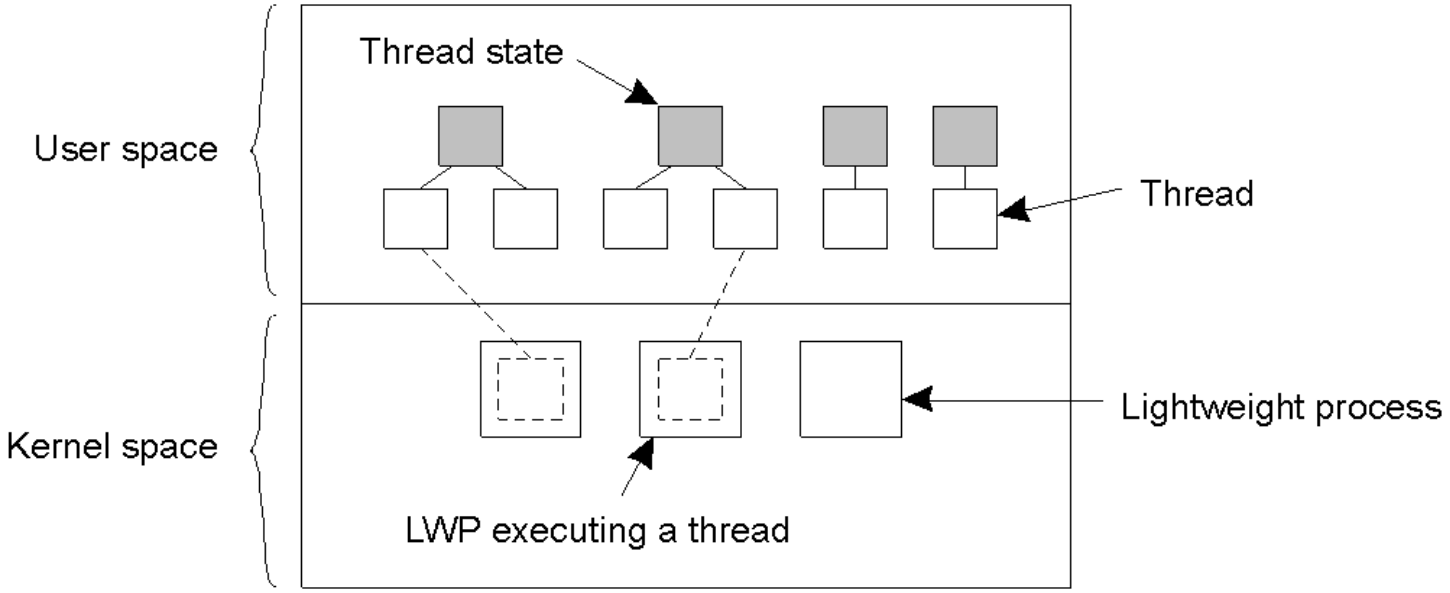
User-Level Thread

- All the threads are created in user processes' address spaces.
- **Advantage**
  - All operations can be completely handled within a single process ⇒ implementations can be extremely efficient.
- **Disadvantage**
  - Difficult to get the support from OS, block
    - All services provided by the kernel are done on behalf of the process in which a thread resides ⇒ if the kernel decides to block a thread, the entire process will be blocked.

Threads and Operating Systems

- Have the **kernel** contain the implementation of a thread package. This means that all operations return as system calls:
  - Operations that block a thread are no longer a problem: the **kernel schedules another available thread** within the same process.
  - Handling **external events** is simple: the kernel (which catches all events) schedules the thread associated with the event.
  - The problem is (or used to be) **the loss of efficiency** due to the fact that each thread operation requires a trap to the kernel.

Thread Implementation



Light weight processes, LWP

Managing local resources

- Object-to-resource binding
  - By identifier: the object requires a specific instance of a resource (e.g. a specific database)
  - By value: the object requires the value of a resource (e.g. the set of cache entries)
  - By type: the object requires that only a type of resource is available (e.g. a color monitor)

Managing local resources

- An object uses local resources that may or may not be available at the target site
- Resource types
  - Fixed: the resource cannot be migrated, such as local hardware
  - Fastened: the resource can, in principle, be migrated but only at high cost
  - Unattached: the resource can easily be moved along with the object (e.g. a cache)

Managing local resources (2/2)

	Unattached	Fastened	Fixed
ID	MV (or GR)	GR (or MV)	GR
Value	CP (or MV, GR)	GR (or CP)	GR
Type	RB (or MV, GR)	RB (or GR, CP)	RB (or GR)

GR = Establish global systemwide reference

MV = Move the resource

CP = Copy the value of the resource

RB = Re-bind to a locally available resource

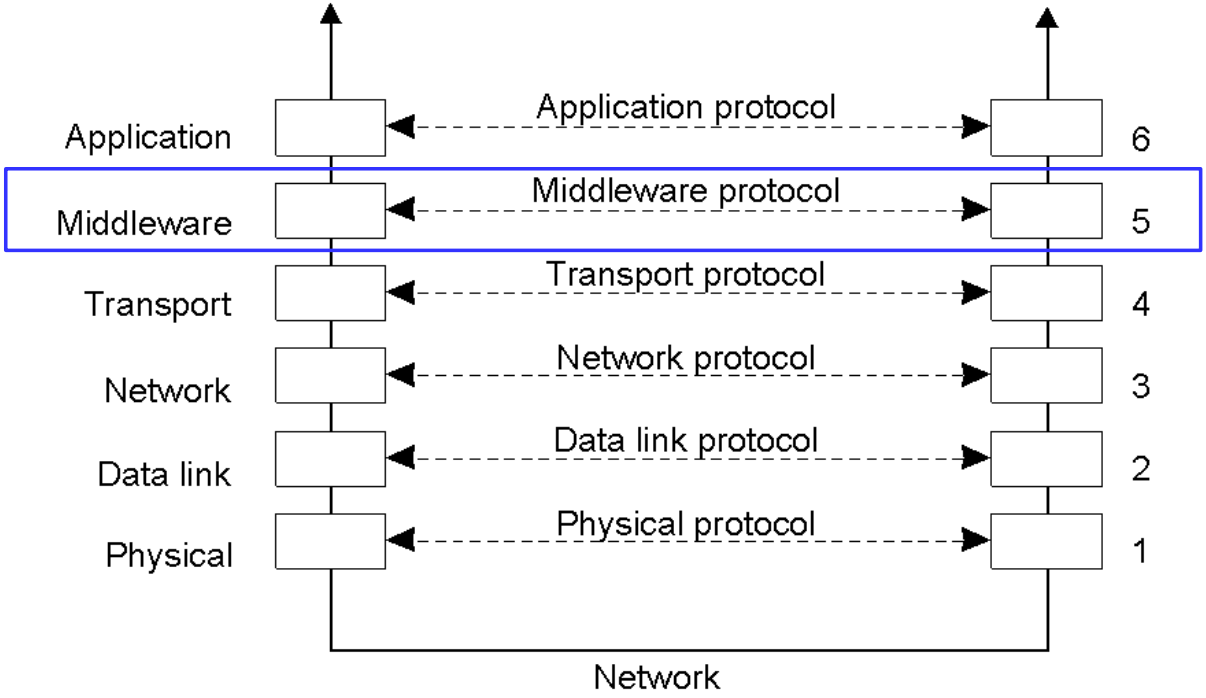
通信 （15分）

- 电子邮件系统采用什么样的通信服务？持久通信、瞬时通信、异步通信、同步通信之间的组合方式，以及每种组合之间的区别？
- 为什么要引入远程过程调用，远程过程调用包含哪些步骤？
- 在RPC中，可能发生的5中失败形式？如何处理？

通信 （15分）

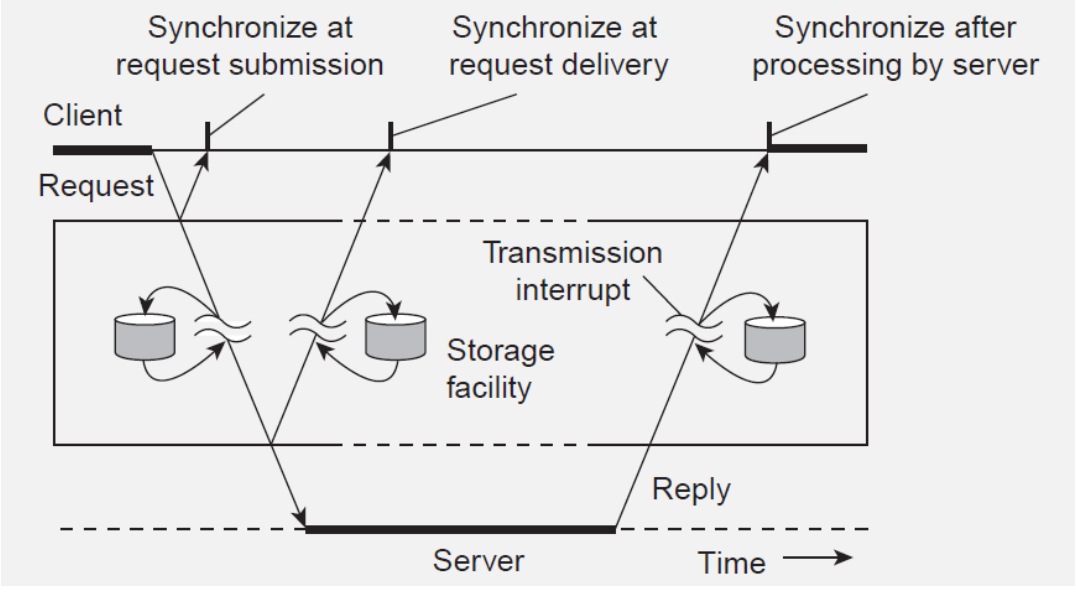
- 什么是消息队列系统？为什么需要消息队列系统？使用队列的松散耦合通信有哪几种组合方式？
- 离散媒体、连续媒体的含义、区别是什么？
- 数据流传输中，如何确保服务质量QoS？交错传输的工作原理？

Middleware Protocols



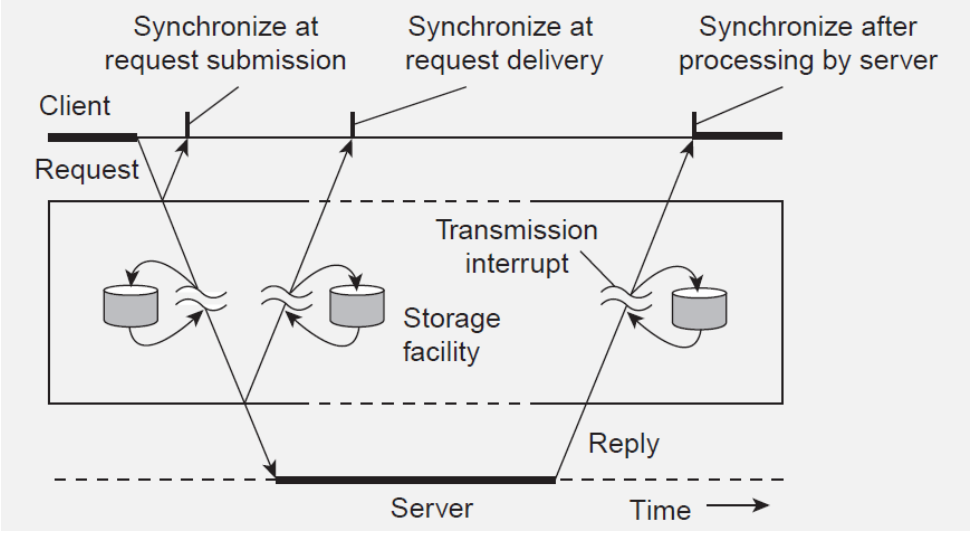
- An adapted reference model for networked communication.

Types of communication



- Persistent communication: A message is stored at a communication server as long as it takes to deliver it.
- Transient communication: Comm. server discards message when it cannot be delivered at the next server, or at the receiver.

Types of communication



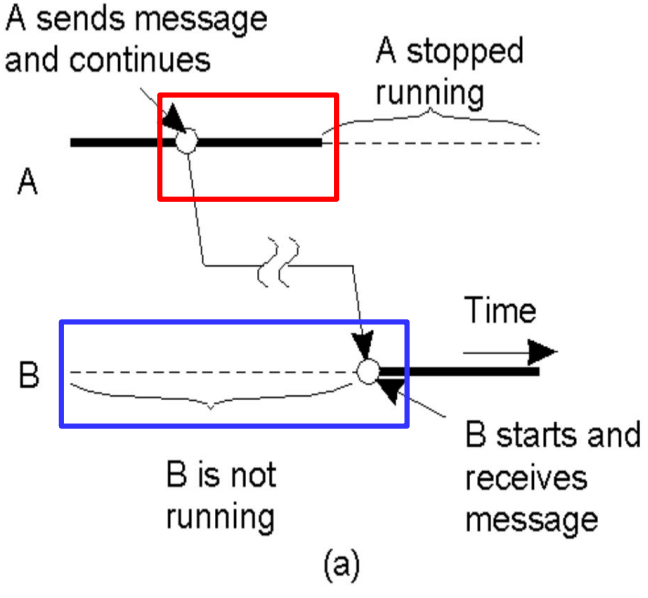
- Asynchronous versus synchronous communication
- Synchronous communication
  - At request submission
  - At request delivery
  - After request processing

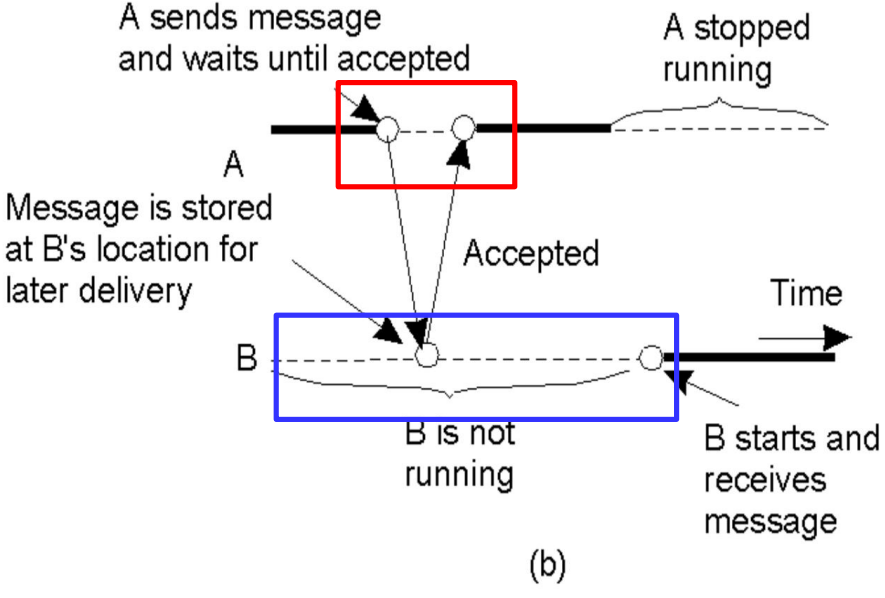
Combination

- Persistent + Asynchronous communication
- Persistent + Synchronous communication

- Transient + Asynchronous communication
- Transient + Synchronous communication

Persistent communication

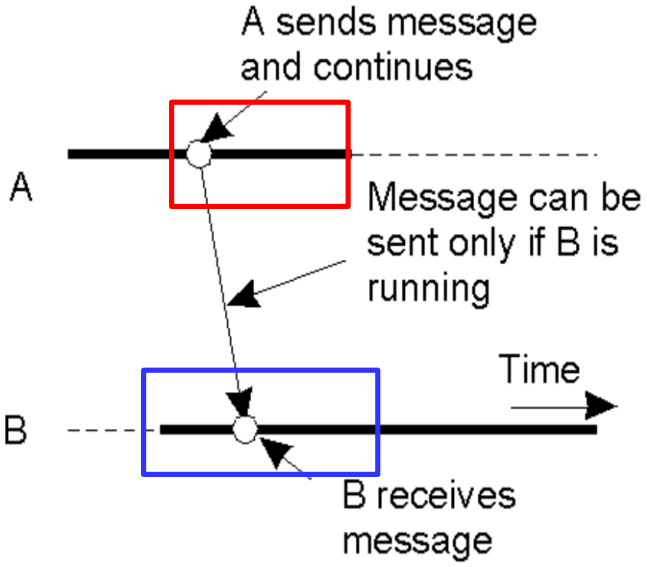




Persistent + Asynchronous communication

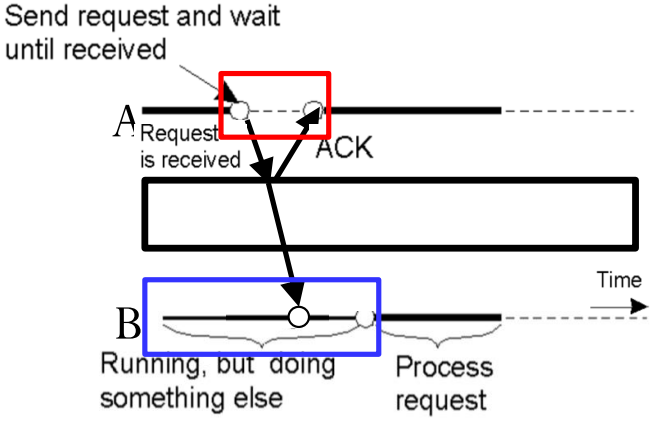
Persistent + Synchronous communication

Transient + Asynchronous communication

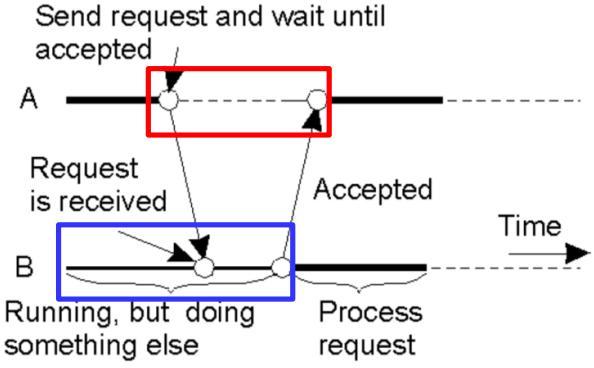


Transient + Asynchronous communication

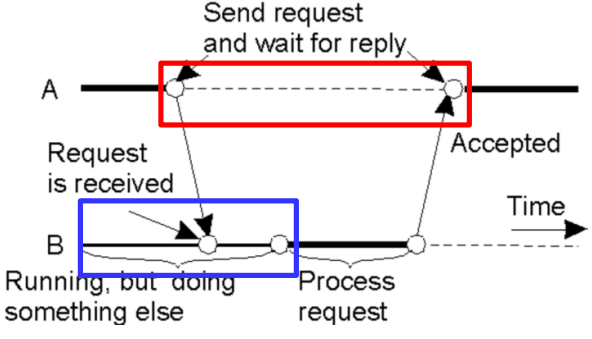
Transient + Synchronous communication



At request submission



At request delivery



After request processing

RPC in Presence of Failures

- Five different classes of failures can occur in RPC systems
  - The client is unable to locate the server
  - The request message from the client to the server is lost
  - The reply message from the server to the client is lost
  - The server crashes after receiving a request
  - The client crashes after sending a request

Client Cannot Locate the Server

- Examples:
  - Server might be down
  - Server evolves (new version of the interface installed and new stubs generated) while the client is compiled with an older version of the client stub
- Possible solutions:
  - Use a special code, such as "-1", as the return value of the procedure to indicate failure. In Unix, add a new error type and assign the corresponding value to the global variable errno.
    - "-1" can be a legal value to be returned, e.g., sum(7, -8)
  - Have the error raise an exception (like in ADA) or a signal (like in C).
    - Not every language has exceptions/signals (e.g., Pascal). Writing an exception/signal handler destroys the transparency

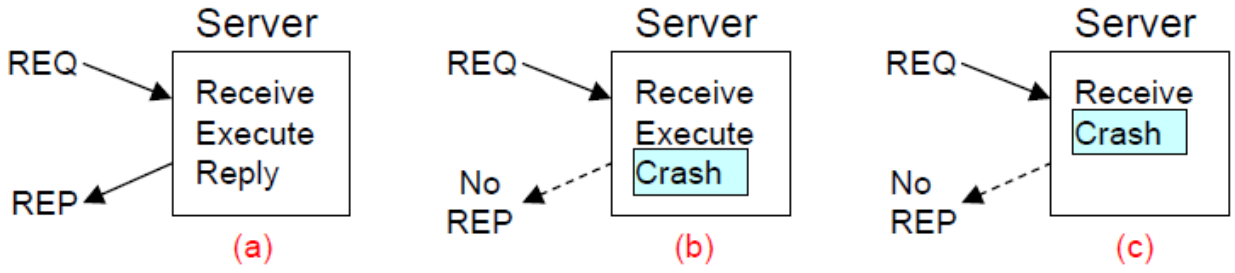
Lost Request Message

- Kernel starts a **timer** when sending the message:
  - If timer expires before reply or ACK comes back: Kernel retransmits
  - If message truly lost: Server will not differentiate between original and retransmission ⇒ everything will work fine
  - If many requests are lost: Kernel gives up and falsely concludes that the server is down ⇒ we are back to "Cannot locate server"

Lost Reply Message

- Kernel starts a **timer** when sending the message:
  - If timer expires before reply comes back: Retransmits the request
    - Problem: **Not sure why no reply** (reply/request lost or server slow) ?
  - If server is just slow: The procedure will be executed several times
    - Problem: What if the request is not idempotent, e.g. money transfer
  - Way out: Client's kernel assigns sequence numbers to requests to allow server's kernel to differentiate retransmissions from original

Server crashes



- Problem: Clients' kernel cannot differentiate between (b) and (c)
- Note: Crash can occur before Receive, but this is the same as (c).

Server crashes

- 3 schools of thought exist on what to do here:
  - Wait until the server reboots and try the operation again. Guarantees that RPC has been executed at least one time (at least once semantic)
  - Give up immediately and report back failure. Guarantees that RPC has been carried out at most one time (at most once semantics)
  - Client gets no help. Guarantees nothing (RPC may have been carried out anywhere from 0 to a large number). Easy to implement.

Client Crashes

- Client sends a request and crashes before the server replies: A computation is active and no parent is waiting for result (orphan)
  - Orphans waste CPU cycles and can lock files or tie up valuable resources
  - Orphans can cause confusion (client reboots and does RPC again, but the reply from the orphan comes back immediately afterwards)

Client Crashes

- Possible solutions
  - Extermination: Before a client stub sends an RPC, it makes a log entry (in safe storage) telling what it is about to do. After a reboot, the log is checked and the orphan explicitly killed off.
    - **Expense of writing** a disk record for every RPC; orphans may do RPCs, thus creating **grandorphans impossible to locate**; **impossibility to kill orphans** if the network is partitioned due to failure.

Client Crashes (2)

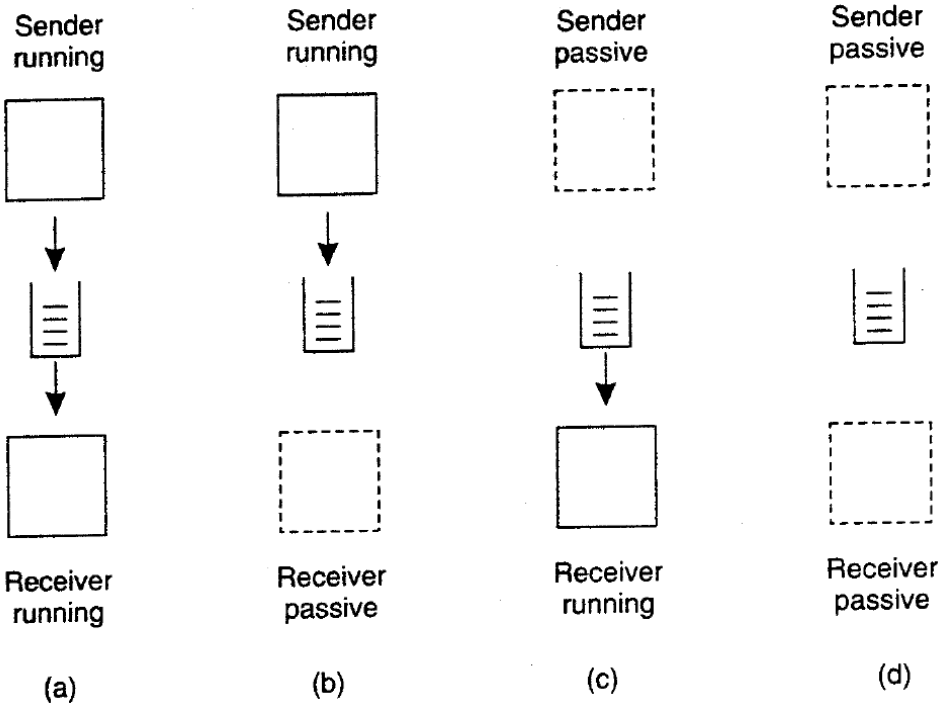
- Possible solutions (cont'd)
  - Reincarnation: Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring the start of a new epoch. When broadcast comes, all remote computations are killed. Solve problem without the need to write disk records
    - If network is partitioned, some orphans may survive. But, when they report back, they are easily detected given their obsolete epoch number
  - Gentle reincarnation: A variant of previous one, but less Draconian
    - When an epoch broadcast comes in, each machine that has remote computations tries to locate their owner. A computation is killed only if the owner cannot be found

Client Crashes (3)

- Possible solutions (cont'd)
  - Expiration: Each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum.
    - Choosing a reasonable value of T in the face of RPCs with wildly differing requirements is difficult

Persistent Messaging

- Usually called message-queuing system, since it involves queues at both ends



Continuous media

- All communication facilities discussed so far are essentially based on a discrete, that is time-independent exchange of information
- Characterized by the fact that values are time dependent:
  - Audio
  - Video
  - Animations
  - Sensor data (temperature, pressure, etc.)

Stream

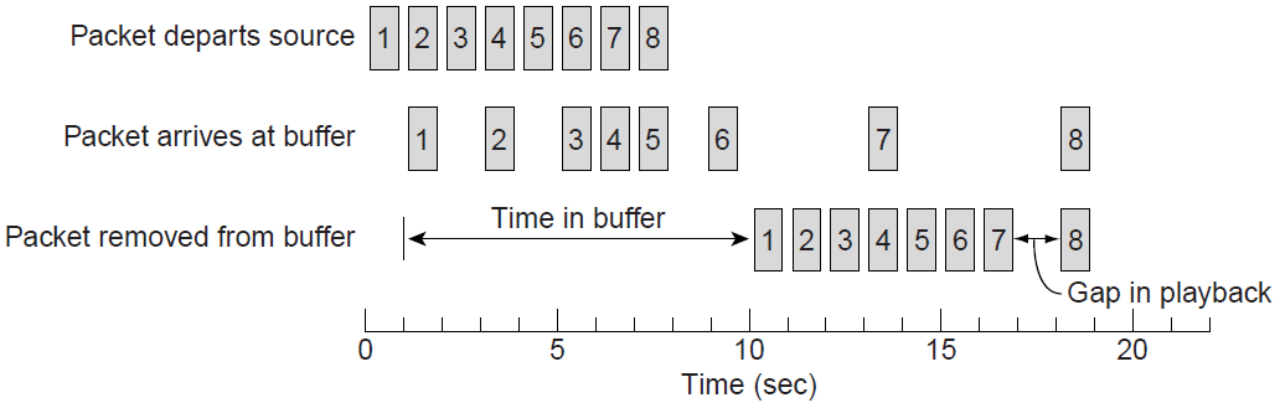
- A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission.
- Some common stream characteristics
  - Streams are unidirectional
  - There is generally a single source, and one or more sinks
  - Often, either the sink and/or source is a wrapper around hardware(e.g., camera, CD device, TV monitor)
  - Simple stream: a single flow of data, e.g., audio or video
  - Complex stream: multiple data flows, e.g., stereo audio or combination audio/video

Streams and QoS

- Streams are all about timely delivery of data. How do you specify this Quality of Service (QoS)? Basics:
  - The required bit rate at which data should be transported.
  - The maximum delay until a session has been set up (i.e., when an application can start sending data).
  - The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
  - The maximum delay variance, or jitter.
  - The maximum round-trip delay.

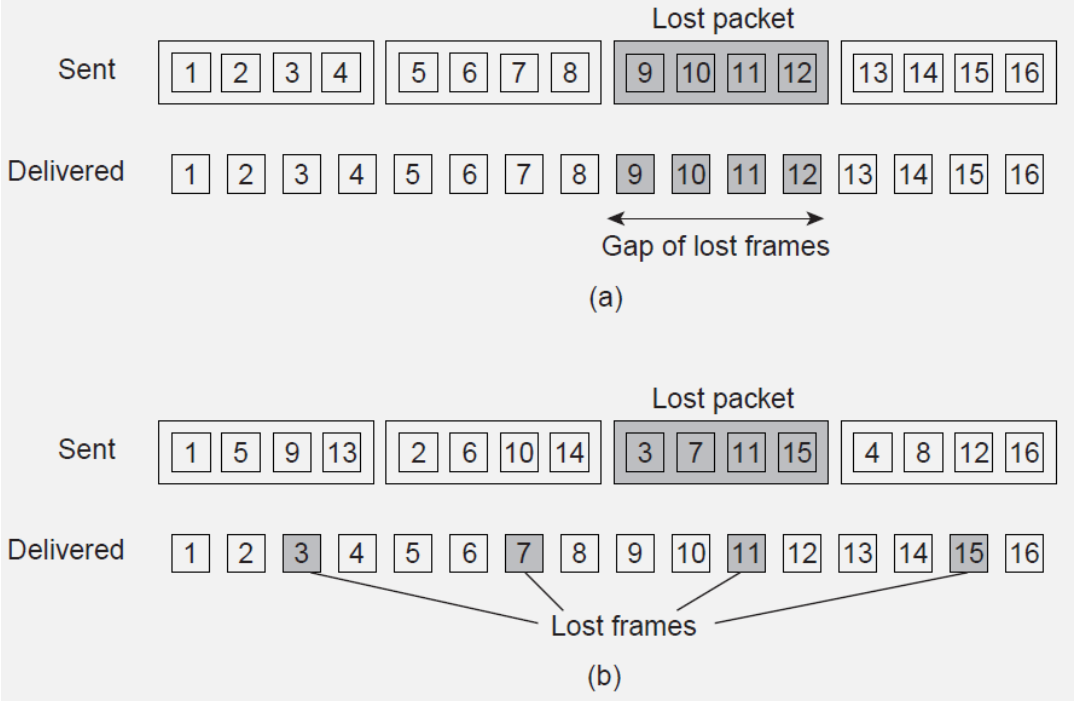
Enforcing QoS

- There are various network-level tools, such as differentiated services by which certain packets can be prioritized.
- Use buffers to reduce jitter:



Enforcing QoS

- How to reduce the effects of packet loss (when multiple samples are in a single packet)?



同步化（25分）

- 分布式系统为什么需要同步？可举例说明。在分布式系统中实现同步，存在哪些困难？可从物理时钟、传输延迟等角度分析。

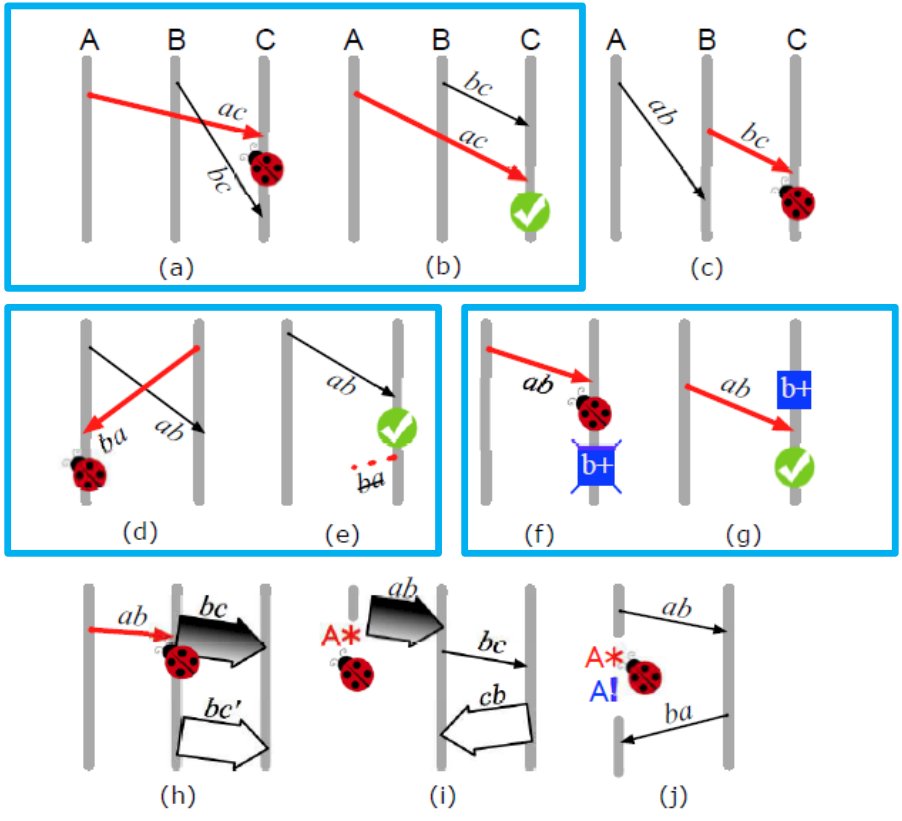
- 有哪些时钟同步算法？Berkeley算法、均值同步算法的具体步骤？

- 逻辑时钟与物理时钟的区别？如何通过Lamport算法校正不同进程的时钟？

同步化（25分）

- Lamport逻辑时钟与向量时钟的联系与区别？向量时钟的工作原理（三个步骤）？
- 如何使用向量时钟，实现因果有序多播？
- 分布式系统中，为了访问共享资源，有哪些互斥算法？请从不通角度分析他们的区别？如进出消息数、进入前延迟、存在问题等？

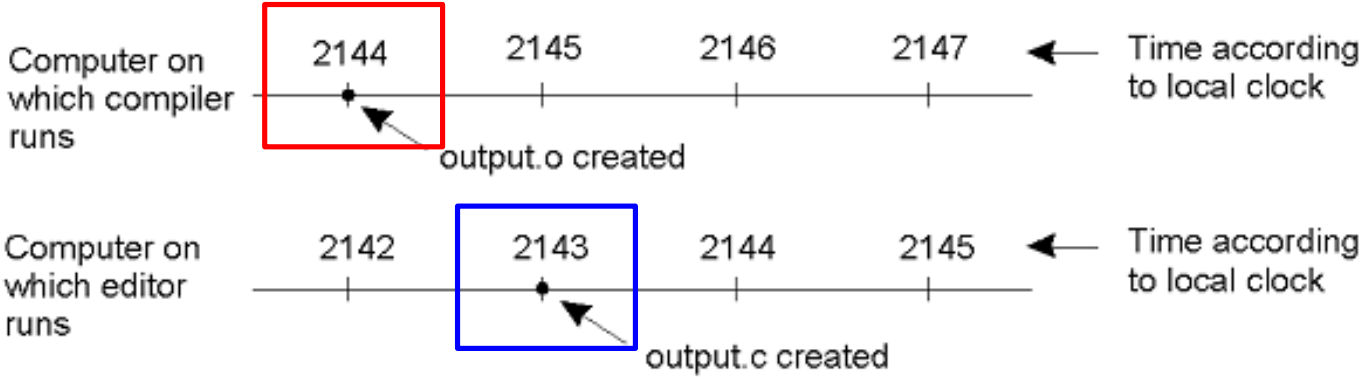
Triggering Patterns



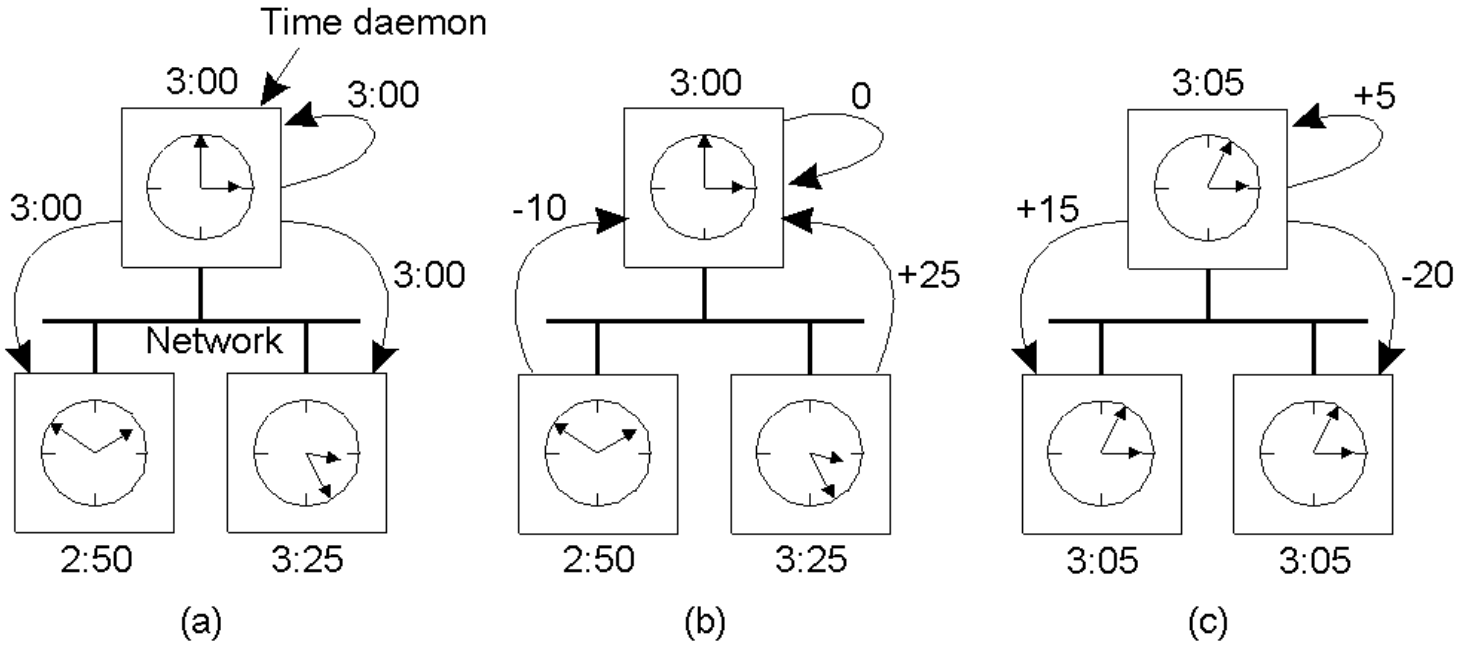
Tanakorn Leesatapornwongsa et al. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems, Asplos 2016.

Clock synchronization is not Trivial

- In a distributed system:
  - Achieving agreement on time is not trivial!



The Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values

b) The machines answer

c) The time daemon tells everyone how to adjust their clock

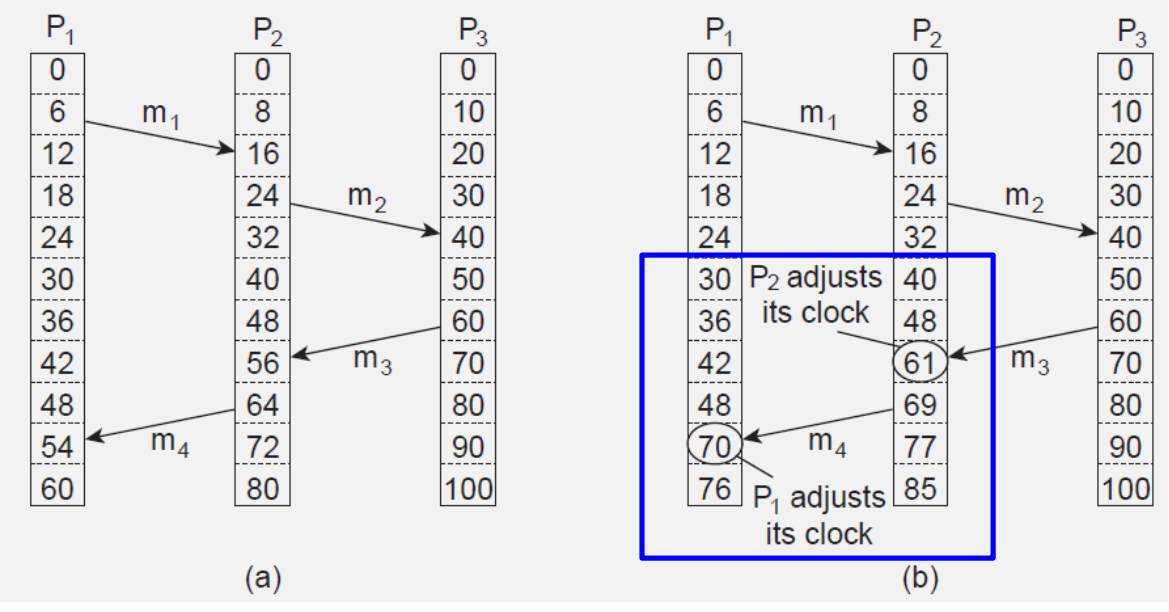
Averaging Algorithm

- Divide the time into fixed length resynchronization intervals
- The i-th interval starts at T_0+iR and runs until $T_0+(i+1)R$, where T_0 is an agreed upon moment in the past, and R is a system parameter
- At the beginning of each interval, each machine broadcasts the current time according to its own clock (broadcasts are not likely to happen simultaneously because the clocks on different machine do not run at exactly the same speed)
- After a machine broadcasts its time, it starts a timer to collect all other broadcasts that arrive during some interval S
- When all broadcasts arrive, an algorithm is run to compute the new time from them
  - Simplest algorithm: Average the values from all other machines
- Variations: discard m highest and m lowest values and average the rest; Add to each message an estimate of the propagation time from source

Logical clocks: The Happened-before relationship

- We first need to introduce a notion of ordering before we can order anything.
- The happened-before relation
  - If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$.
  - If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$.
  - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
- This introduces a partial ordering of events in a system with concurrently operating processes.

Lamport's Algorithm



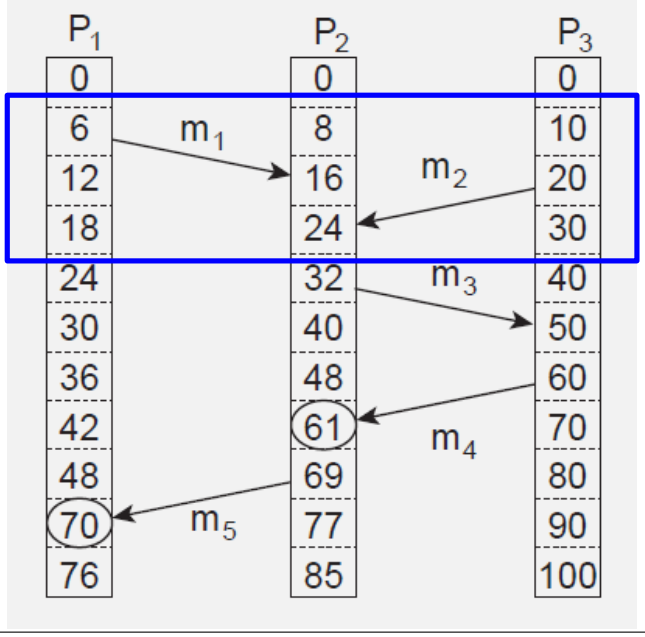
- Three processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clocks.
- Lamport solution:
  - Between every two events, the clock must tick at least once

Lamport's Algorithm

- Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:
  - 1: For any two successive events that take place within P_i , C_i is incremented by 1.
  - 2: Each time a message m is sent by process P_i , the message receives a timestamp $ts(m) = C_i$.
  - 3: Whenever a message m is received by a process P_j , P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.
- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

Vector clocks

- Lamport's clocks:
  - If a causally preceded b then timestamp(a) < timestamp(b)
  - Do not guarantee that if C(a) < C(b) that a causally preceded b
- We cannot conclude that a causally precedes b.



Event a: m_1 is received at T = 16

Event b: m_2 is received at T = 24

Vector clocks

- Solution
  - Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$denotes the number of events that process P_i knows have taken place at process P_j.
  - When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i along with m as vector timestamp $vt(m)$. Result: upon arrival, recipient knows P_i's timestamp.
  - When a process P_j delivers a message m that it received from P_i with vector timestamp $ts(m)$, it

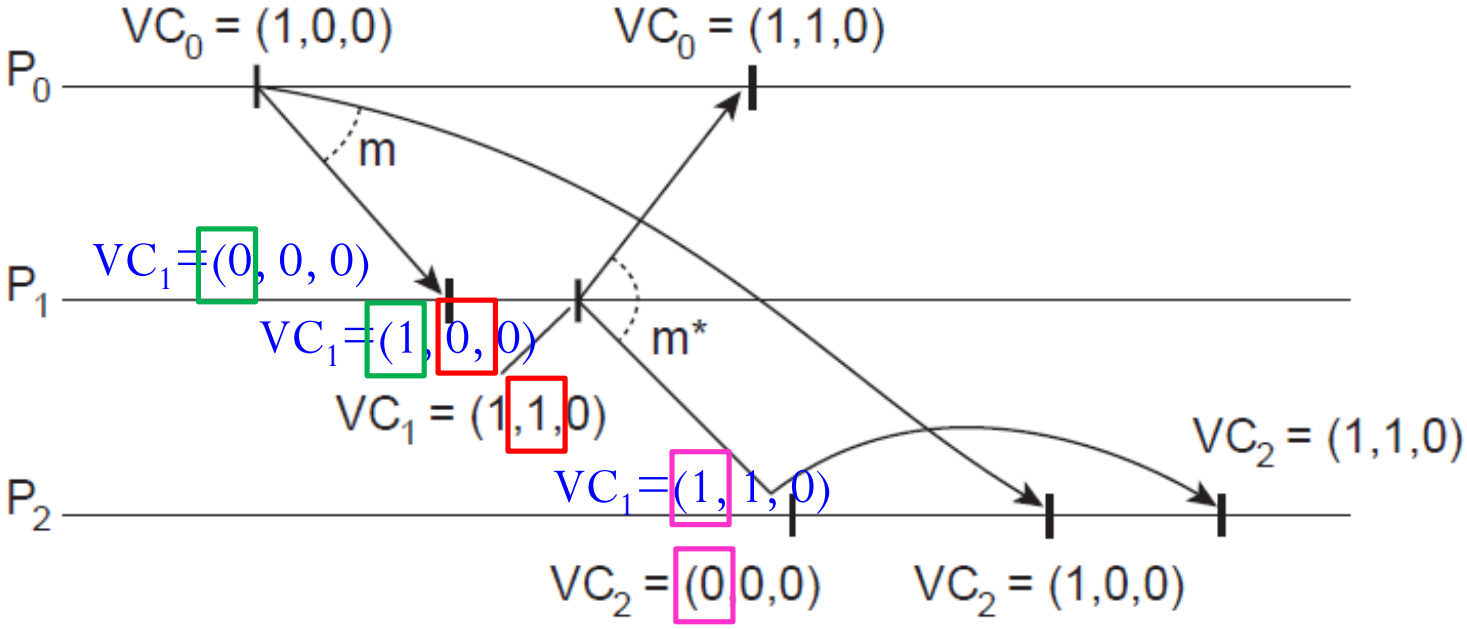
    (1) updates each $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$

    (2) increments $VC_j[j]$ by 1.

Causally ordered multicasting

- We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.
- Adjustment
  - P_i increments $VC_i[i]$ only when sending a message, and P_j "adjusts" VC_j when receiving a message (i.e., effectively does not change $VC_j[j]$).
- P_j postpones delivery of m until:
  - $ts(m)[i] = VC_j[i] + 1$.
  - $ts(m)[k] \leq VC_j[k]\ for\ k \neq i$.

Causally ordered multicasting

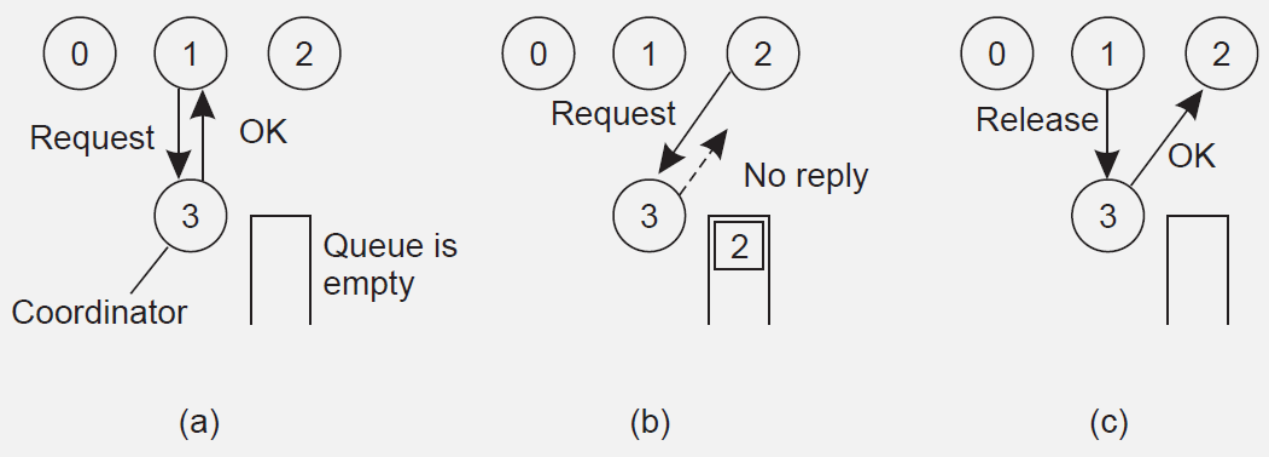


$$ts(m)[i] = VC_j[i] + 1.$$
$$ts(m)[k] \leq VC_j[k] \; for \; k \neq i.$$

Mutual exclusion

- A number of processes in a distributed system want exclusive access to some resource.
- Basic solutions
  - Via a centralized server.
  - Completely decentralized, using a peer-to-peer system.
  - Completely distributed, with no topology imposed.
  - Completely distributed along a (logical) ring.

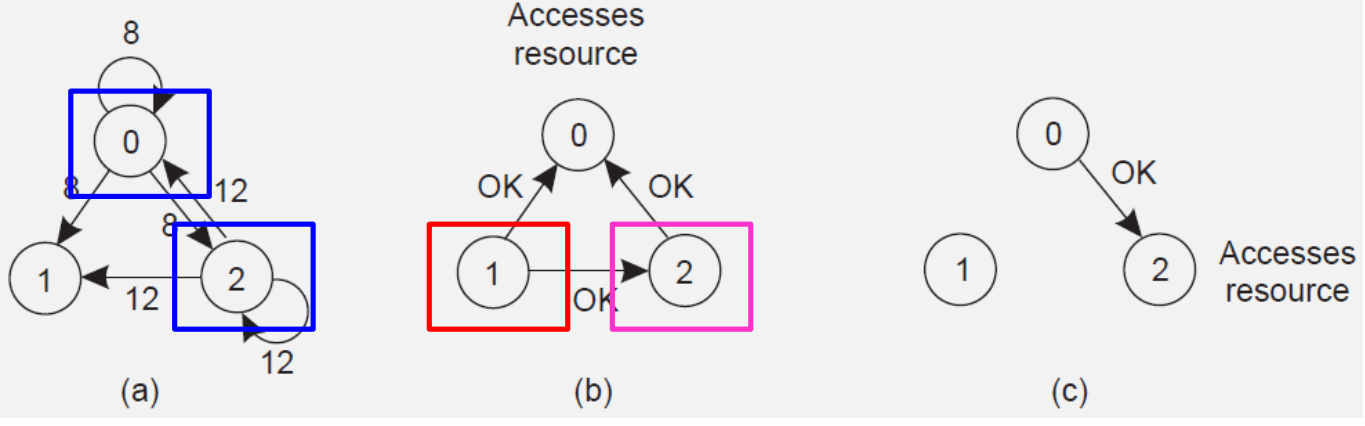
Centralized Mutual Exclusion



- Advantages
  - Obviously it guarantees mutual exclusion
  - It is fair (requests are granted in the order in which they are received)
  - No starvation (no process ever waits forever)
  - Easy to implement (only 3 messages: request, grant and release)
- Shortcomings
  - Coordinator: A single point of failure; A performance bottleneck
  - If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied"

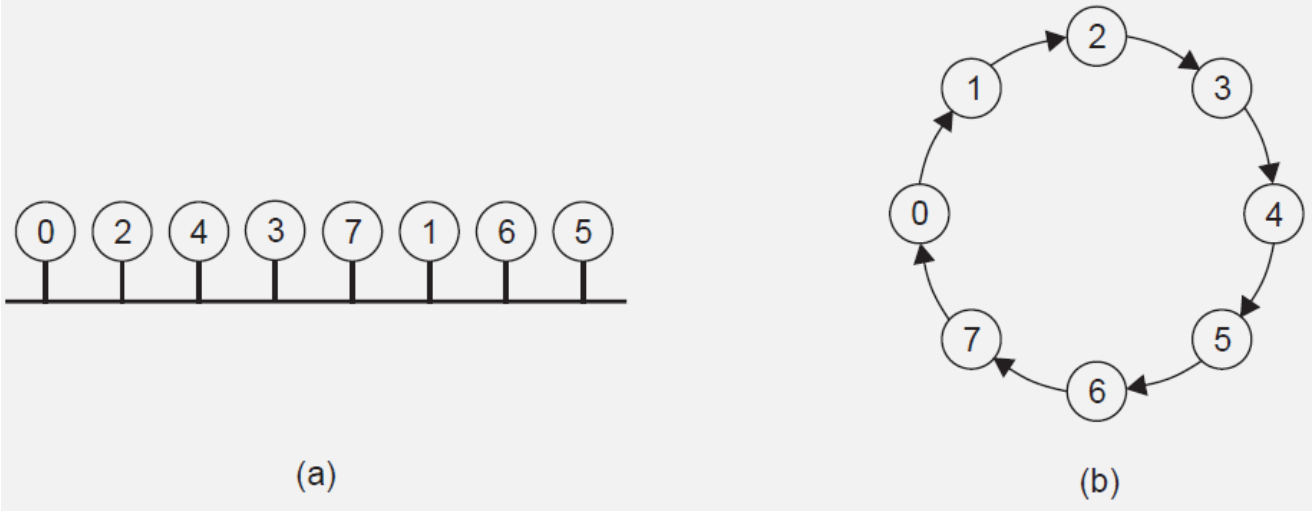
A Distributed Algorithm

- a)Two processes want to enter the same critical region at the same moment.
- b)Process 0 has the lowest timestamp, so it wins.
- c)When process 0 is done, it sends an OK also, so 2 can now enter the critical region.



Mutual exclusion: Token ring algorithm

- Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).



Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

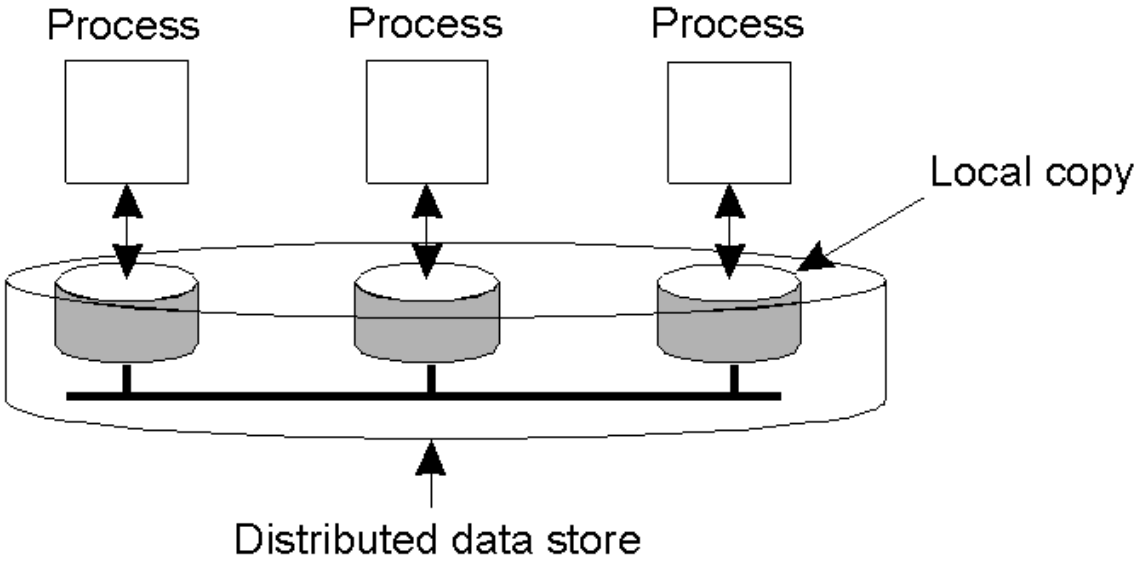
- A comparison of three mutual exclusion algorithms.

一致性和复制 （15分）

- 以数据为中心的一致性模型，有哪些类型？每种一致性模型的工作原理？
- 以客户为中心的一致性模型，有哪些类型？每种一致性模型的工作原理？
- 基于主备份的协议的复制写协议，分别有哪些？每种协议的基本原理？

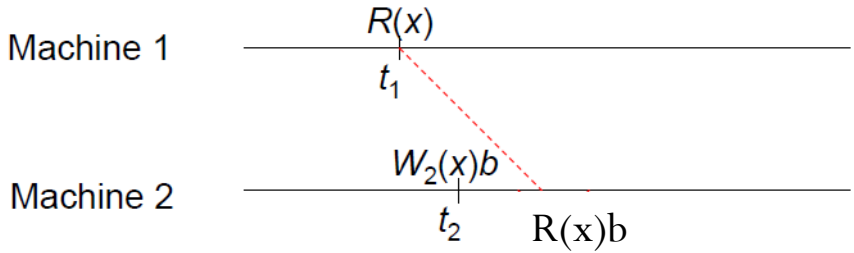
Data-Centric Consistency Models

- The general organization of a logical data store, physically distributed and replicated across multiple processes.



Strict Consistency

- Any *read*(*x*) returns a value corresponding to the result of the most recent *write*(*x*).



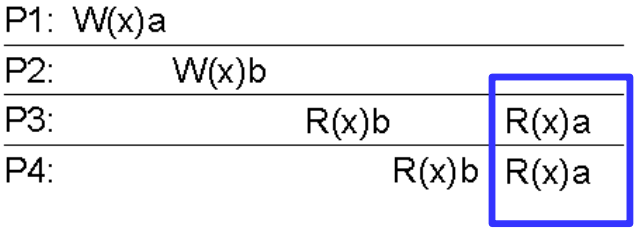
- Relies on absolute global time; all writes are instantaneously visible to all processes and an absolute global time order is maintained.
- Cannot be implemented in a distributed system

Linearizability

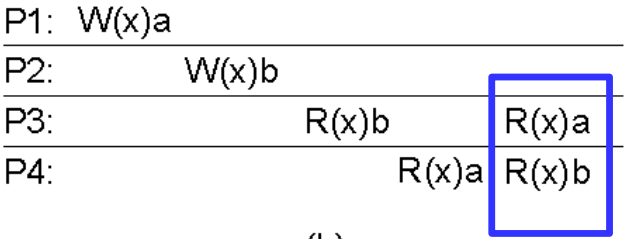
- The result of the execution should satisfy the following criteria:
  - Read and write by all processes were executed in some serial order and each process's operations maintain the order of specified;
  - If $ts_{op1}(x) < ts_{op2}(y)$ then $op1(x)$ should precede $op2(y)$ in this sequence. This specifies that the order of operations in interleaving is consistent with the real times at which the operations occurred in the actual implementation.
- Requires synchronization according to timestamps, which makes it expensive.
- Used only in formal verification of programs.

Sequential Consistency

- Similar to linearizability, but no requirement on timestamp order.
- The result of execution should satisfy the following criteria:
  - Read and write operations by all processes on the data store were executed in some sequential order;
  - Operations of each individual process appear in this sequence in the order specified by its program.
- These mean that all processes see the same interleaving of operations similar to serializability.



(a)



(b)

Sequences for the Processes

Process P1	Process P2	Process P3
x = 1; print (y, z);	y = 1; print (x, z);	z = 1; print (x, y);

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

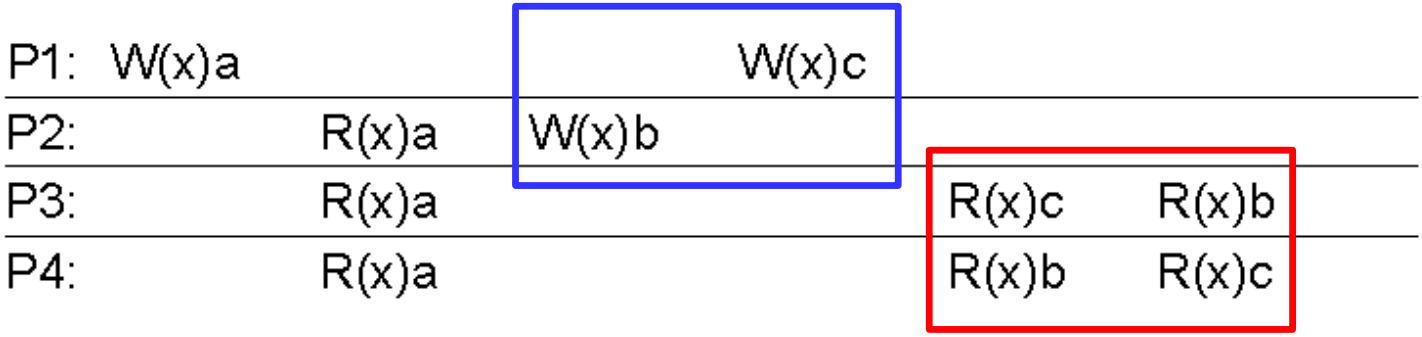
x = 1;
print ((y, z);
y = 1;
print (x, z);
z = 1;
print (x, y);
	x = 1;
	y = 1;
	print (x,z);
	print(y, z);
	z = 1;
	print (x, y);
		y = 1;
		z = 1;
		print (x, y);
		print (x, z);
		x = 1;
		print (y, z);
			y = 1;
			x = 1;
			z = 1;
			print (x, z);
			print (y, z);
			print (x, y);

Prints: 001011 (a)	Prints: 101011 (b)	Prints: 010111 (c)	Prints: 111111 (d)

Casual Consistency (1)

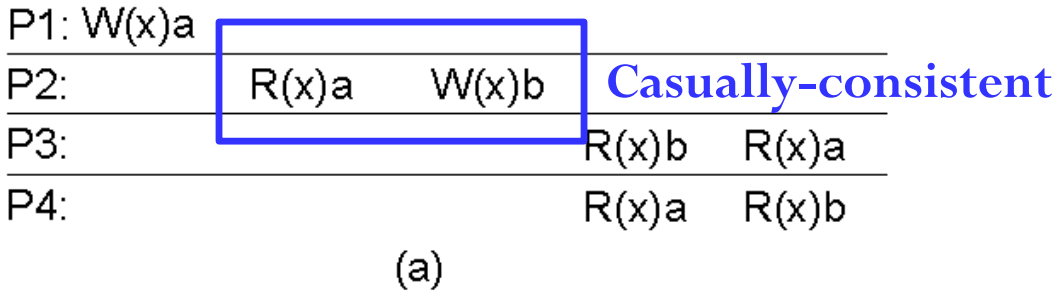
- Necessary condition:
  Writes that are potentially casually related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

Casual Consistency (2)

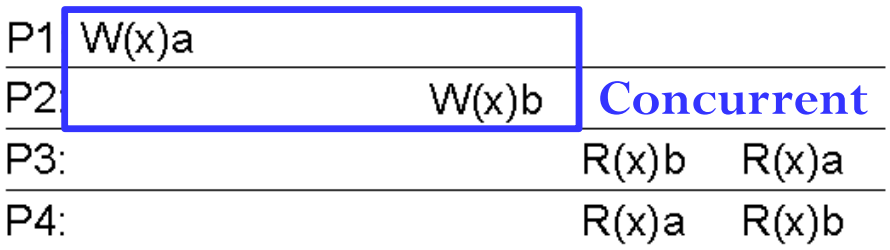


- This sequence is allowed with a casually-consistent store, but not with sequentially or strictly consistent store.

Casual Consistency (3)



(a)



(b)

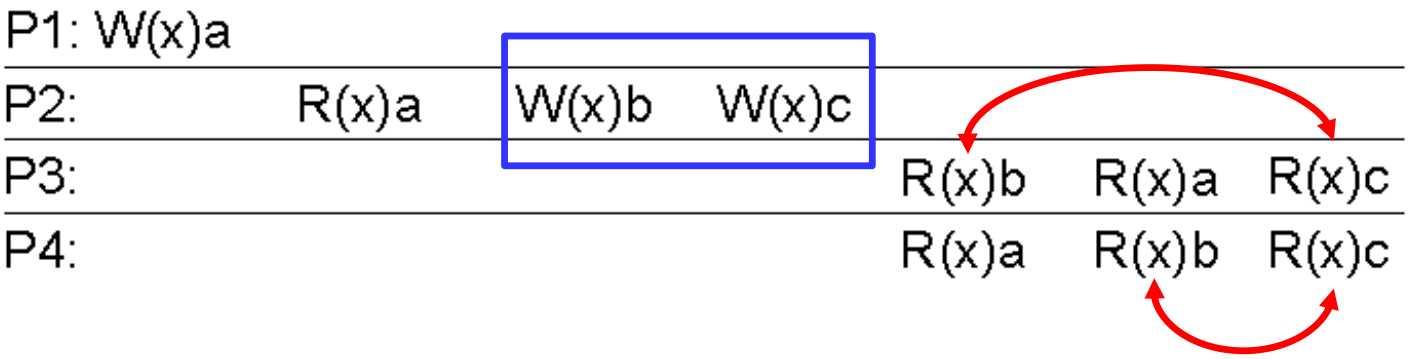
a) A violation of a casually-consistent store.

b) A correct sequence of events in a casually-consistent store.

FIFO Consistency (1)

- Necessary Condition:
  Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

FIFO Consistency (2)



- A valid sequence of events of FIFO consistency

FIFO Consistency (3)

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

- The statements in bold are the ones that generate the output shown.

x = 1;	x = 1;	y = 1;
print (y, z);	y = 1;	print (x, z);
y = 1;	**print(x, z);**	z = 1;
print(x, z);	print (y, z);	**print (x, y);**
z = 1;	z = 1;	x = 1;
print (x, y);	print (x, y);	print (y, z);
Prints: 00	Prints: 10	Prints: 01
(a)	(b)	(c)
(1)	(2)	<(1) ✖

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

a) Consistency models **not** using synchronization operations.

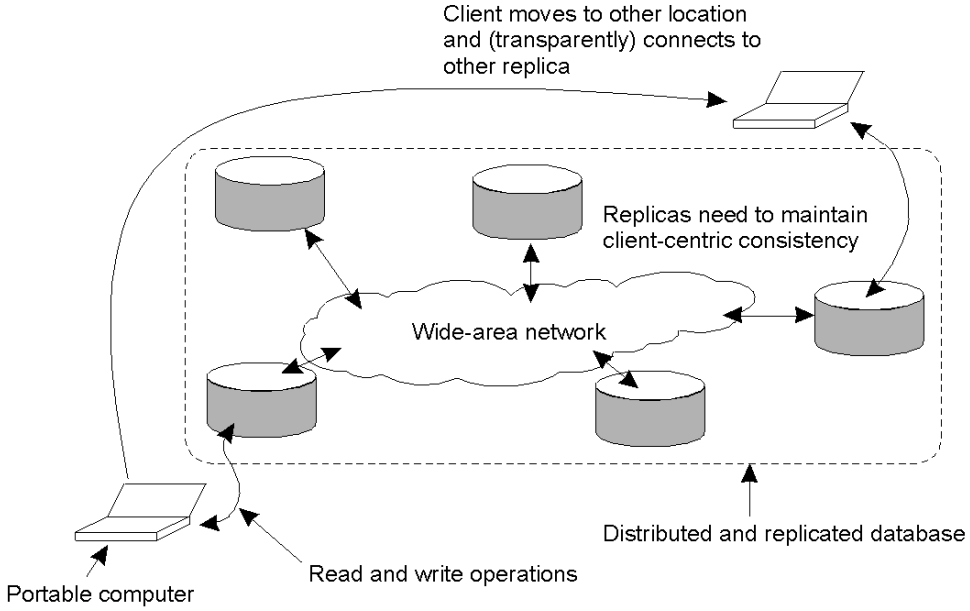
Client-Centric Consistency

- More relaxed form of consistency → only concerned with replicas being eventually consistent (**eventual consistency**).
- In the absence of any further updates, all replicas converge to identical copies of each other → only requires guarantees that updates will be propagated.
- Easy if a user always accesses the same replica; problematic if the user accesses different replicas.
  - Client-centric consistency: guarantees for a single client the consistency of access to a data store.

Client-Centric Consistency (2)

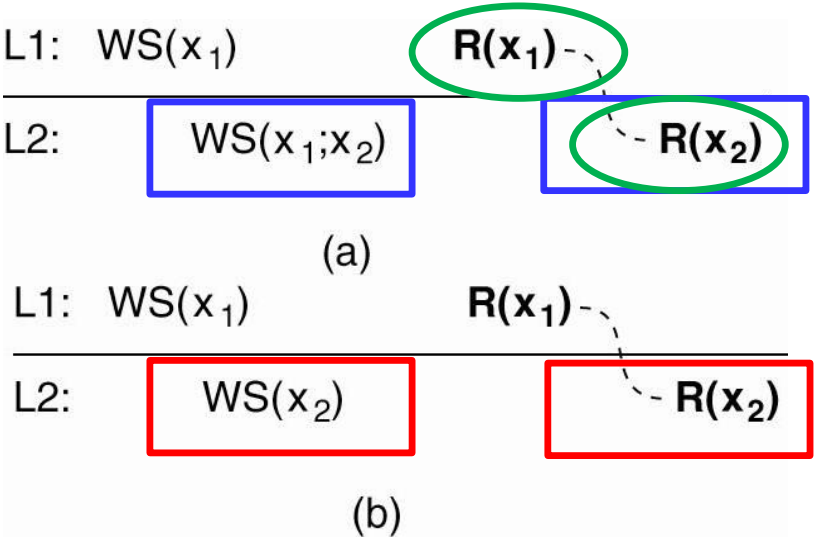
- Monotonic reads
  - If a process reads the value of a data item *x*, any successive read operation on *x* by that process will always return that same value or a more recent value.
- Monotonic writes
  - A write operation by a process on a data item *x* is completed before any successive write operation on *x* by the same process.
- Read your writes
  - The effect of a write operation by a process on data item *x* will always be seen by a successive read operation on *x* by the same process.
- Writes follow reads
  - A write operation by a process on a data item *x* following a previous read operation on *x* by the same process is guaranteed to take place on the same or more recent value of *x* that was read.

Eventual Consistency



- The principle of a mobile user accessing different replicas of a distributed database.

Monotonic Reads

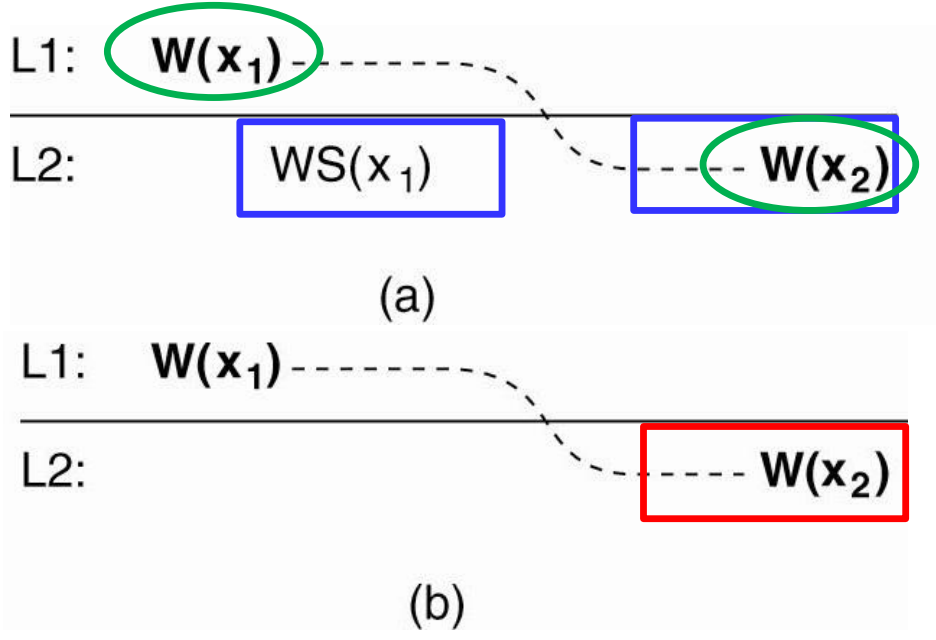


- The read operations performed by a single process *P* at two different local copies of the same data store.

a) A monotonic-read consistent data store

b) A data store that does not provide monotonic reads.

Monotonic Writes

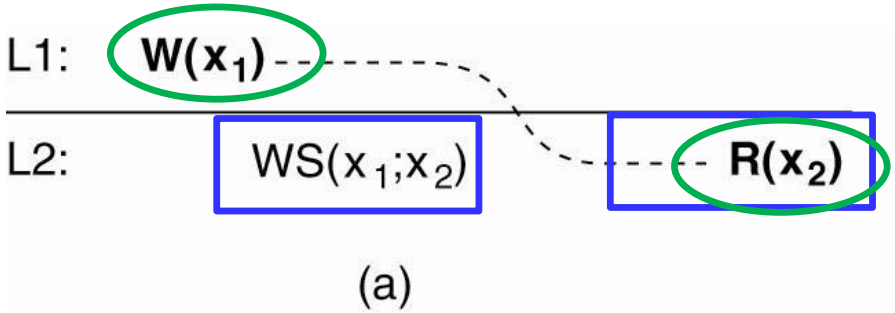


- The write operations performed by a single process *P* at two different local copies of the same data store

a) A monotonic-write consistent data store.

b) A data store that does not provide monotonic-write consistency.

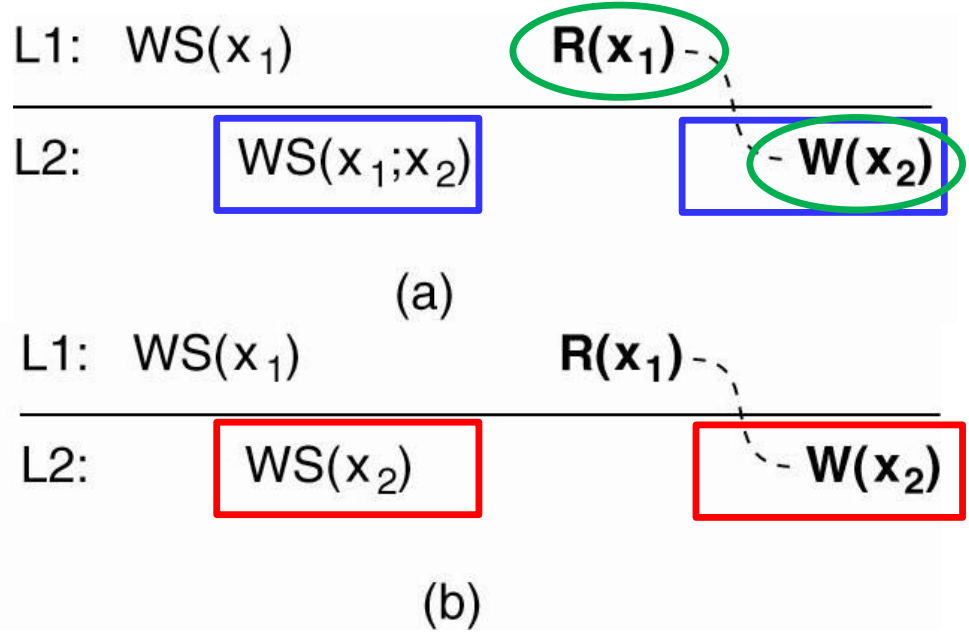
Read Your Writes



a) A data store that provides read-your-writes consistency.

b) A data store that does not.

Writes Follow Reads



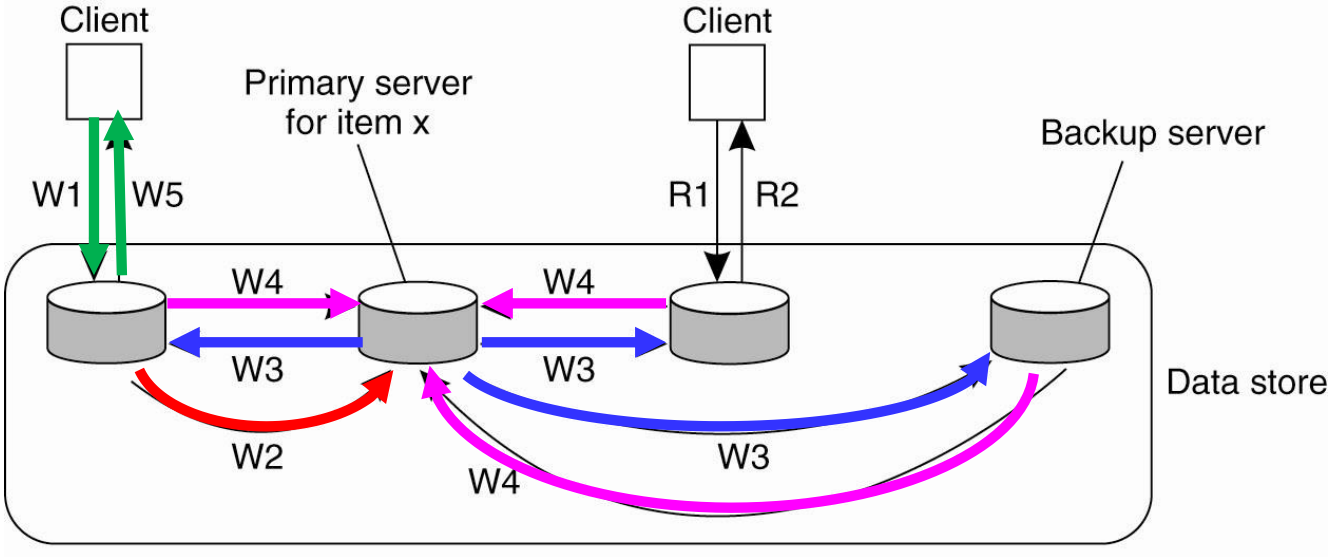
a) A writes-follow-reads consistent data store

b) A data store that does not provide writes-follow-reads consistency

Replication Protocols

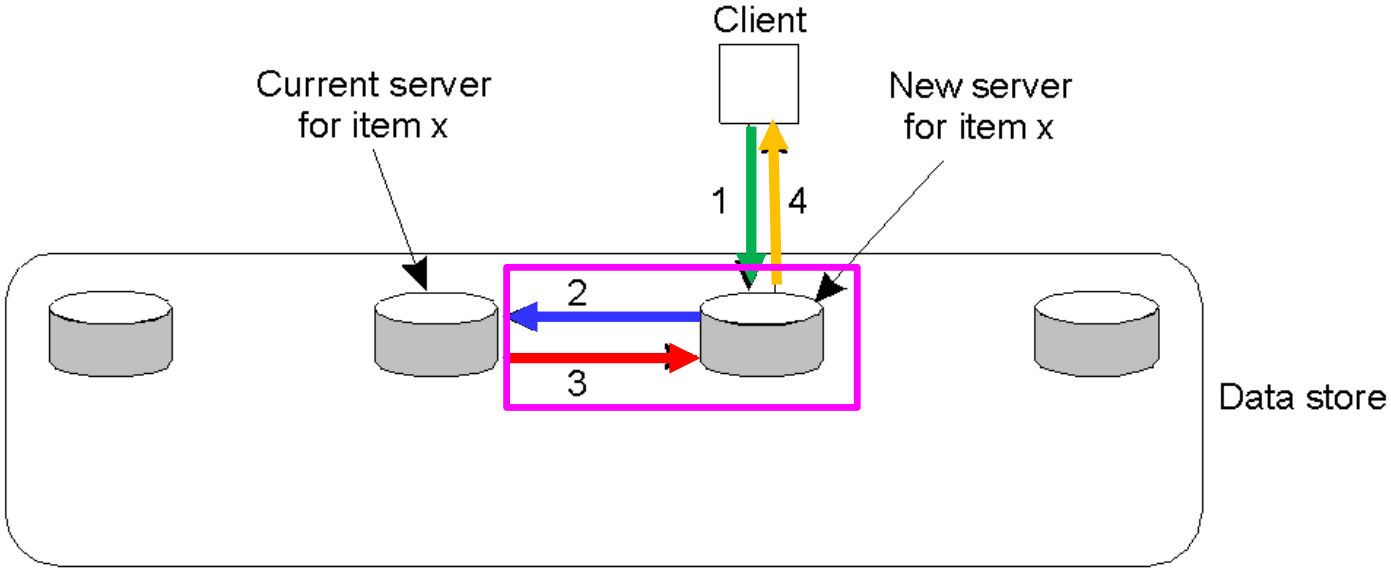
- Sequential consistency
  - Primary-based protocols
    - Remote-Write protocols
    - Local-Write protocols
  - Replicated Write protocols
    - Active replication
    - Quorum-based protocols

Remote-Write Protocols



- Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded.

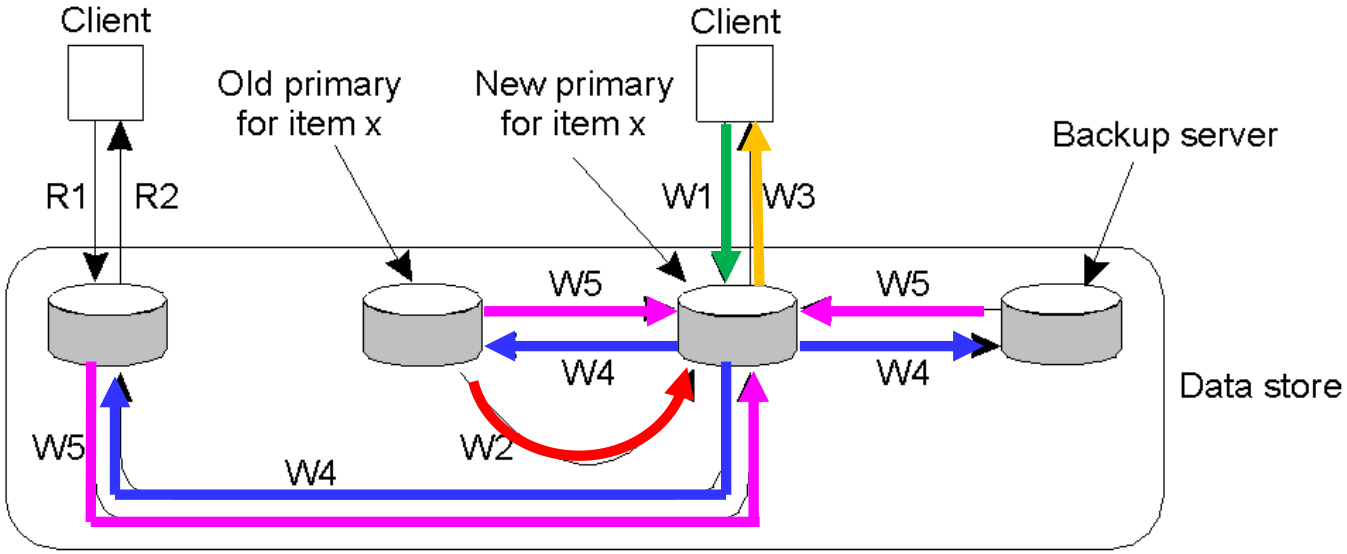
Local-Write Protocols (1)



1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

- Primary-based local-write protocol in which a single copy is migrated between processes.

Local-Write Protocols (2)



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

- Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

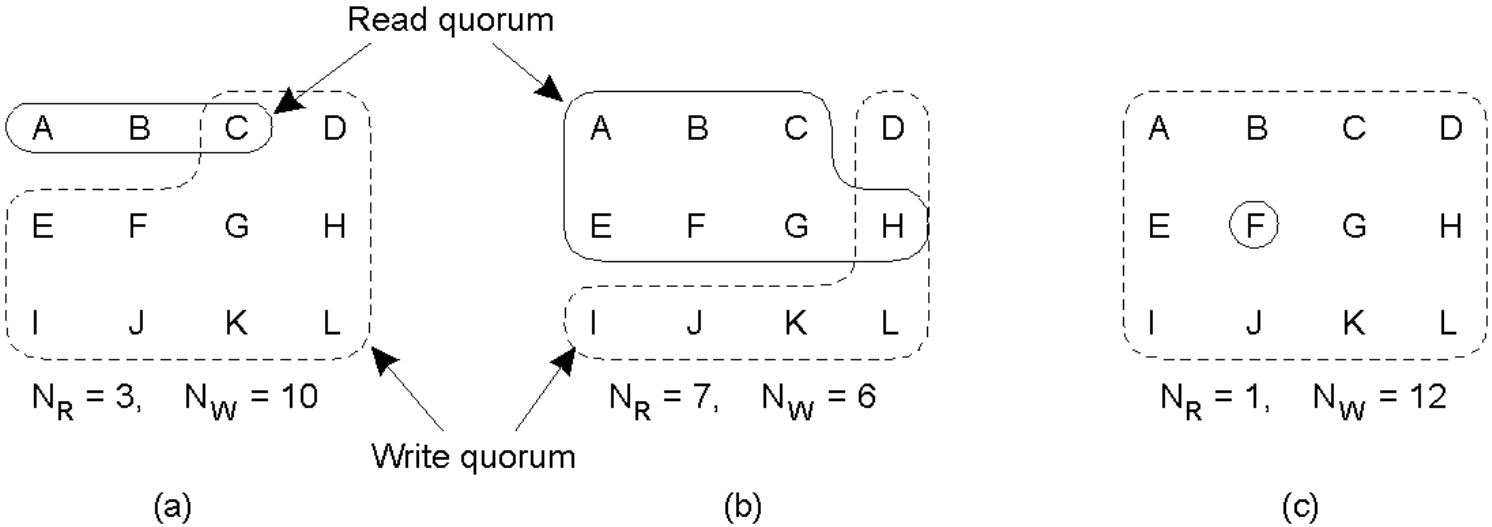
Replicated Write protocols: Active Replication

- Requires a process, for each replica, that can perform the update on it
- How to enforce the update order?
  - Totally-ordered multicast mechanism needed
  - Can be implemented by Lamport timestamps
  - Can be implemented by sequencer
- Problem of replicated invocations
  - If an object *A* invokes another object *B*, all replicas of *A* will invoke *B* (multiple invocations)

Quorum-Based Protocol

- Assign a vote to each *copy* of a replicated object (say *Vi*) such that $\sum_i V_i = V$
- Each operation has to obtain a *read quorum* (V_r) to read and a *write quorum* (V_w) to write an object
- Then the following rules have to be obeyed in determining the quorums:
  - $V_r + V_w > V$ — an object is not read and written by two transactions concurrently
  - $V_w > V/2$ — two write operations from two transactions cannot occur concurrently on the same object

Quorum-Based Protocols



- Three examples of the voting algorithm:

a) A correct choice of read and write set

b) A choice that may lead to write-write conflicts

c) A correct choice, known as ROWA (read one, write all)

容错性（10分）

- 分布式系统中的故障、错误、失败三种的定义、联系与区别？
- 拜占庭协定问题/拜占庭将军问题的定义、原理、过程、结论？
- 为什么需要两阶段提交，两阶段提交的原理和过程？

Fundamental Definitions

- Failure
  - A system is said to fail when it cannot meet its promises.
- Error
  - The part of the a system's state that may lead to a failure.
- Fault
  - The cause of an error is called a fault.

Faults to Failures

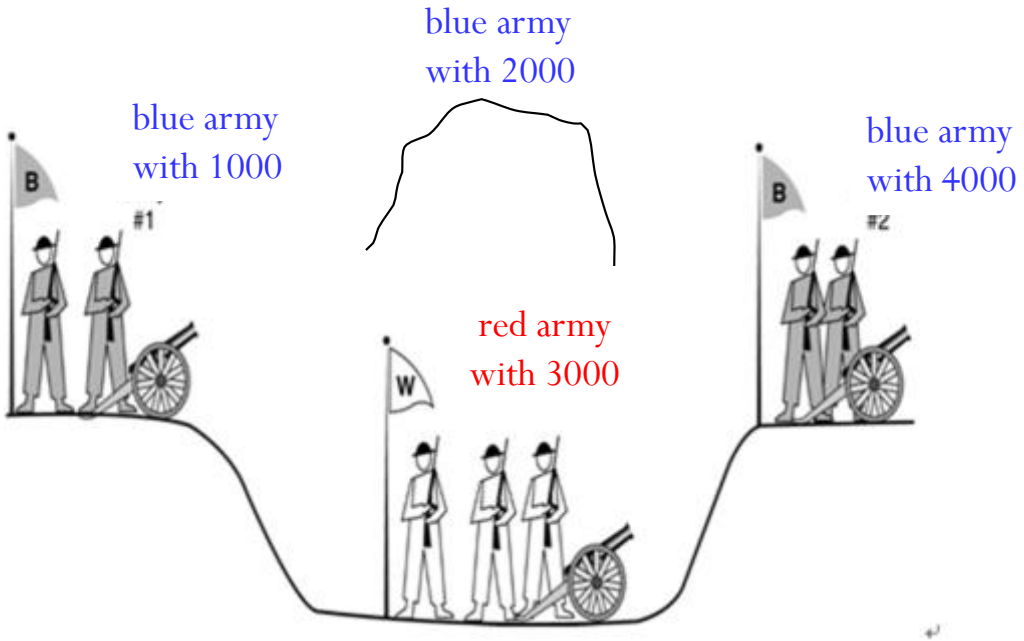
Byzantine agreement problem

- Communication is perfect but the processors are not (in Byzantine faults) .

- Problem:

  n blue generals want to coordinate their attacks on the red army. But *m* of them are traitors.

Byzantine agreement problem



Question:

Whether the loyal generals can still reach agreement?

Byzantine agreement problem

Generals exchange troop strengths, at the end, each general has a vector of length n corresponding to all the armies.

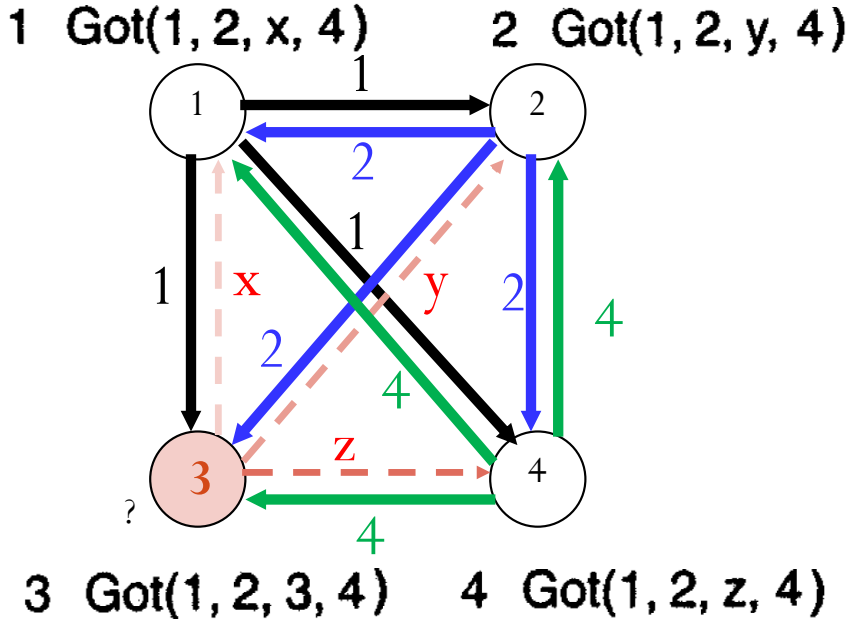
Let n=4, m=1

general 1 has 1K troop

general 2 has 2K troop

general 3 is traitor

general 4 has 4K troop



1 Got
(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

2 Got
(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

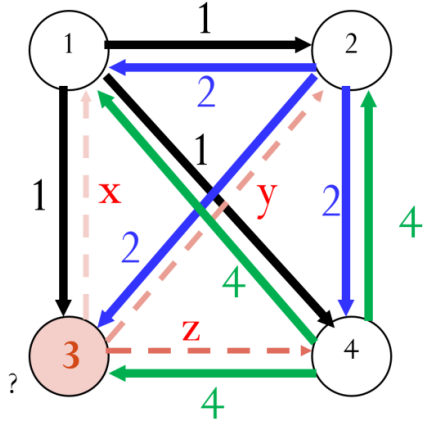
4 Got
(1, 2, x, 4)
(1, 2, y, 4)
(i, j, k, l)

1 is not sure if 2 sends him true message

1 is not sure if 3 sends him true message

1 is not sure if 4 sends him true message

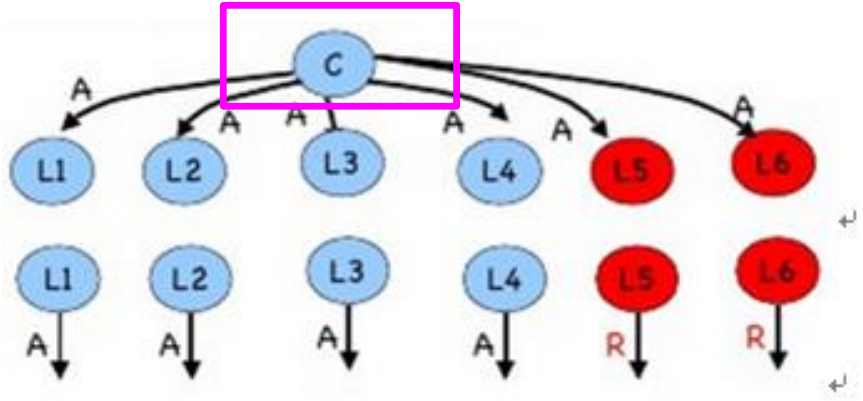
	P1	P2	P3	P4
step 2:	(1, 2, x, 4)	(1, 2, y, 4)	(1, 2, 3, 4)	(1, 2, z, 4)
step 3:	(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)	(1, 2, x, 4)
	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)	(1, 2, y, 4)
	(1, 2, z, 4)	(1, 2, z, 4)	(1, 2, z, 4)	(i, j , k, l)
step 4:	(1, 2, u, 4)	(1, 2, u, 4)	~~(1, 2, u, 4)~~	(1, 2, u, 4)



Byzantine Generals Problem

A commanding general must send an order to his n - 1 lieutenant generals such that

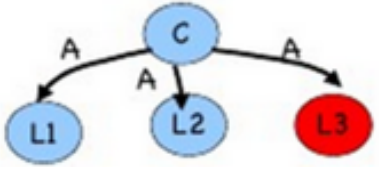
- IC1. All loyal lieutenants obey the same order.
- IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.



Oral Message Algorithm

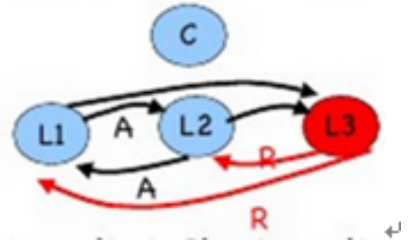
Algorithm OM(0):

1. Commander sends his value to every lieutenant
2. Each lieutenant uses the value received or "retreat" if no value received



Algorithm OM(m), m > 0:

1. Commander sends his value to every lieutenant
2. For each *i*, let v_i be the value that lieutenant *i* receives from the commander or "retreat". Lieutenant *i* acts as the commander in OM(m-1) to send the value v_i to each of the other n-2 other lieutenants
3. For each *i*, and each $j <> i$, let v_j be the value lieutenant *i* received from lieutenant *j* in step 2. Lieutenant *i* uses the value *majority* ($v_1, \ldots, v_{n-1}$)



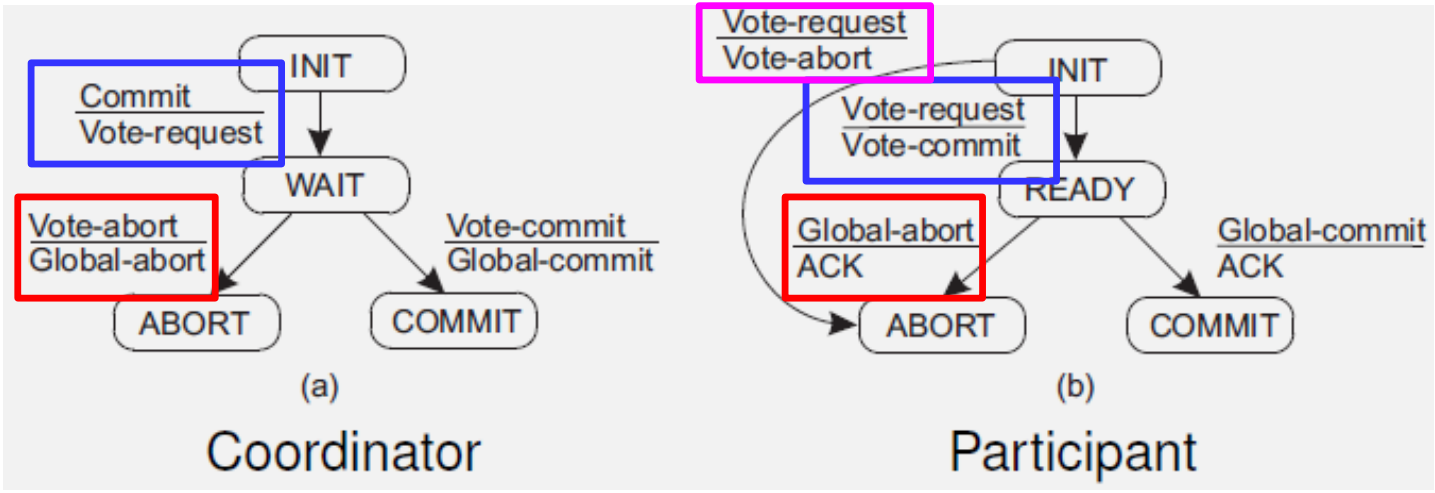
Two-phase commit

- The client who initiated the computation acts as coordinator; processes required to commit are the participants

  - Phase 1a: Coordinator sends vote-request to participants (also called a pre-write)
  - Phase 1b: When participant receives vote-request it returns either vote-commit or vote-abort to coordinator. If it sends vote-abort, it aborts its local computation

  - Phase 2a: Coordinator collects all votes; if all are vote-commit, it sends global-commit to all participants, otherwise it sends global-abort
  - Phase 2b: Each participant waits for global-commit or global-abort and handles accordingly.

Two-phase commit



2PC – Failing participant

- Participant crashes in state S, and recovers to S
  - Initial state: No problem: participant was unaware of protocol
  - Ready state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make ⇒ log the coordinator's decision
  - Abort state: Merely make entry into abort state idempotent, e.g., removing the workspace of results
  - Commit state: Also make entry into commit state idempotent, e.g., copying workspace to storage.
- When distributed commit is required, having participants use **temporary workspaces** to keep their results allows for simple recovery in the presence of failures.

2PC – Failing participant

- When a recovery is needed to READY state, check state of other participants ⇒ no need to log coordinator's decision.
- Recovering participant P contacts another participant Q

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

- If all participants are in the READY state, the protocol blocks. Apparently, the coordinator is failing. Note: The protocol prescribes that we need the decision from the coordinator.

2PC – Failing participant

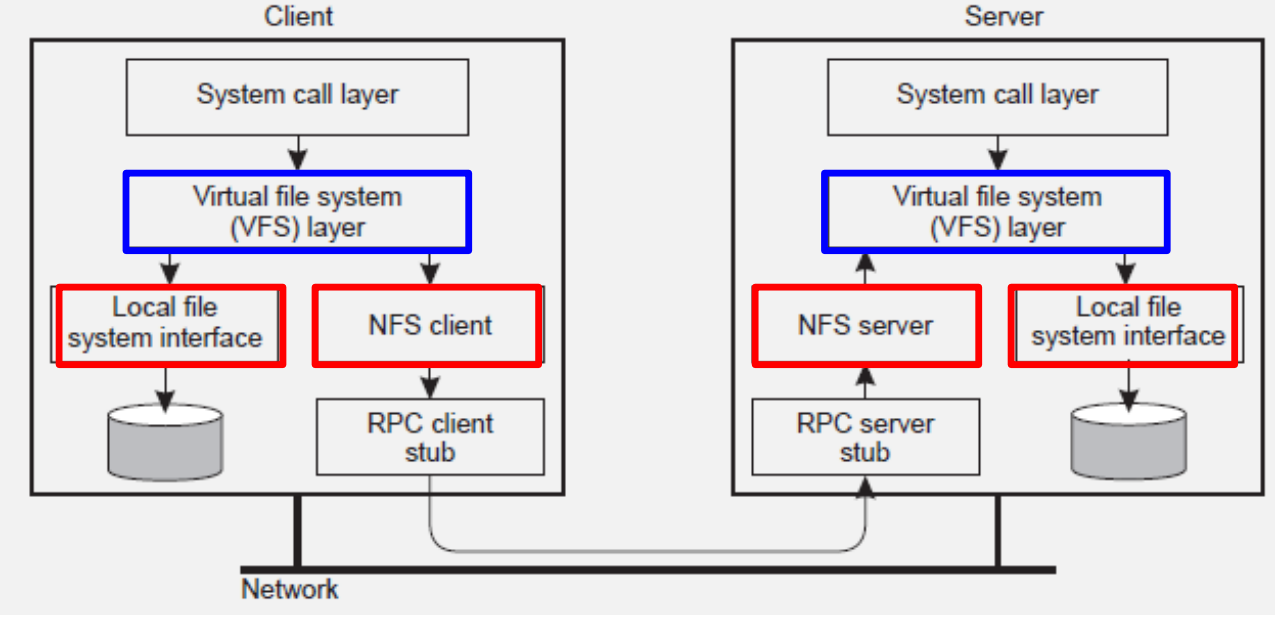
- The real problem lies in the fact that the coordinator's final decision may not be available for some time (or actually lost).
- Let a participant P in the READY state timeout when it hasn't received the coordinator's decision; P tries to find out what other participants know (as discussed).
- Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes

其他（5分）

- 网络文件系统的基本思想，UNIX系统中的基本NFS体系结构和工作过程？有哪些类型的分布式文件系统？
- Coda文件系统中的客户端缓存、服务器端复制？
- 介绍一个典型的基于Web的分布式系统，如WWW？

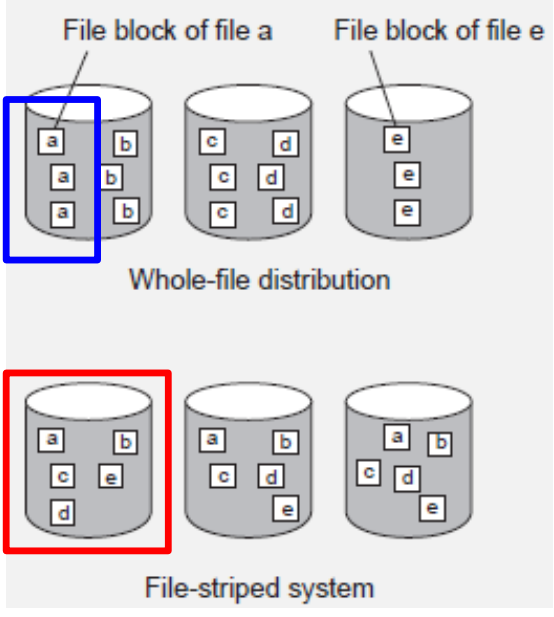
Example: NFS Architecture

- NFS is implemented using the Virtual File System abstraction, which is now used for lots of different operating systems.



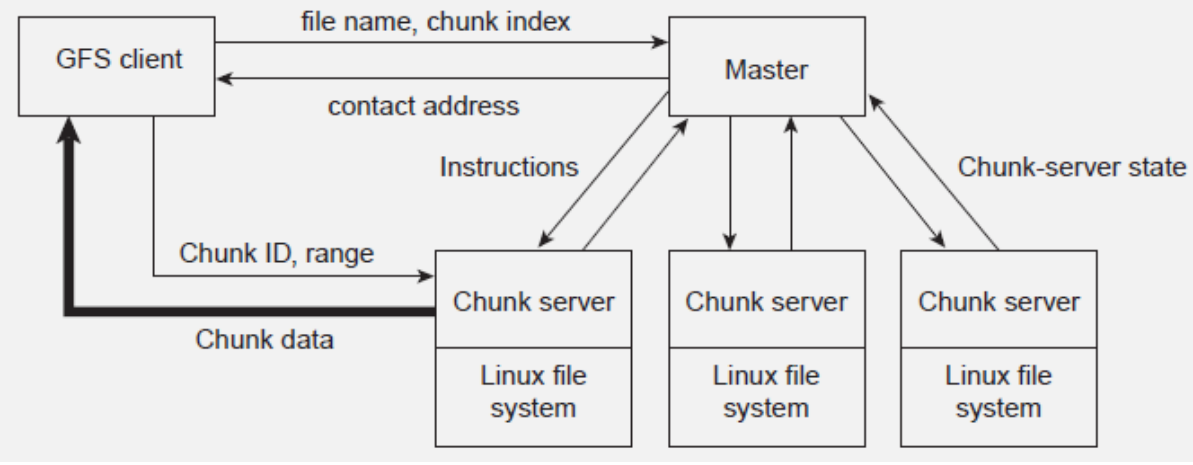
Cluster-Based File Systems

- With very large data collections, following a simple client-server approach is not going to work ⇒ for speeding up file accesses, apply striping techniques by which files can be fetched in parallel.



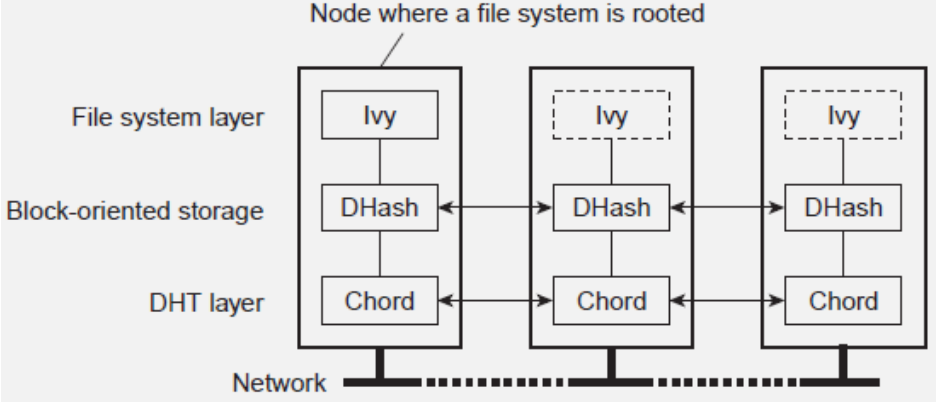
Example: Google File System

- Divide files in large 64 MB chunks, and distribute/replicate chunks across many servers:
  - The master maintains only a (file name, chunk server) table in main memory ⇒ minimal I/O
  - Files are replicated using a primary-backup scheme; the master is kept out of the loop



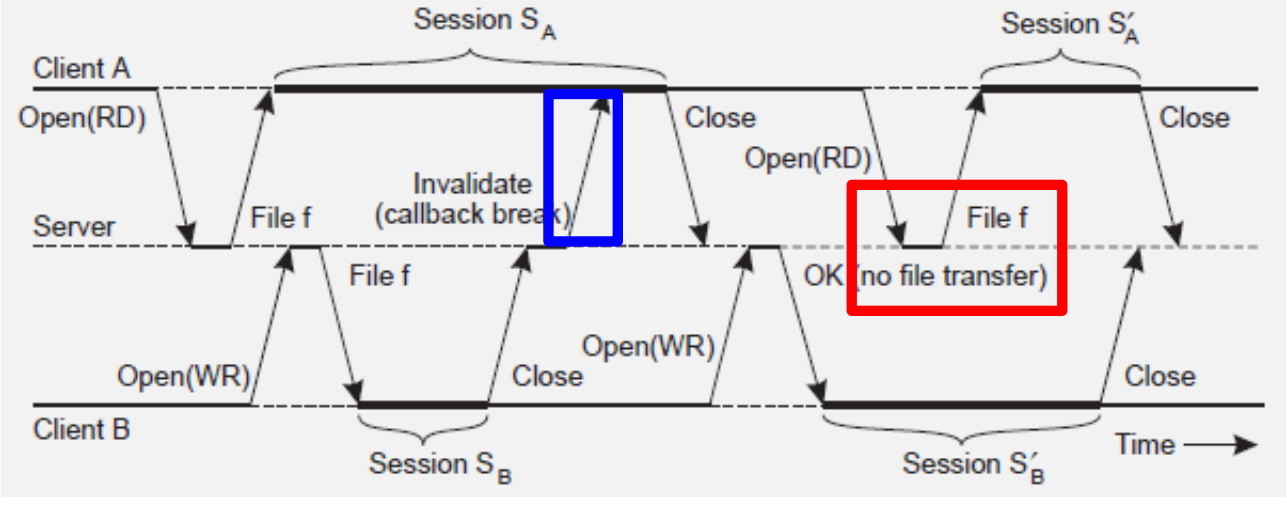
P2P-based File Systems

- Store data blocks in the underlying P2P system:
  - Every data block with content D is stored on a node with hash $h(D)$. Allows for integrity check.
  - Public-key blocks are signed with associated private key and looked up with public key.
  - A local log of file operations to keep track of $\langle blockID, h(D) \rangle$ pairs.



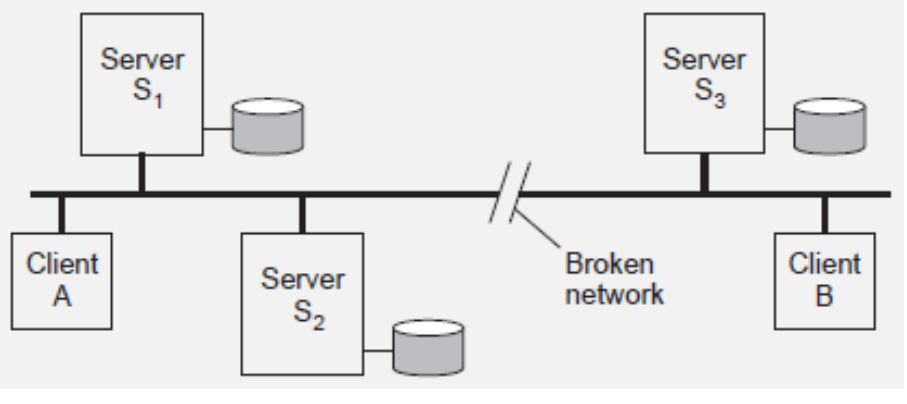
Example: Client-side caching in Coda

- By making use of transactional semantics, it becomes possible to further improve performance.



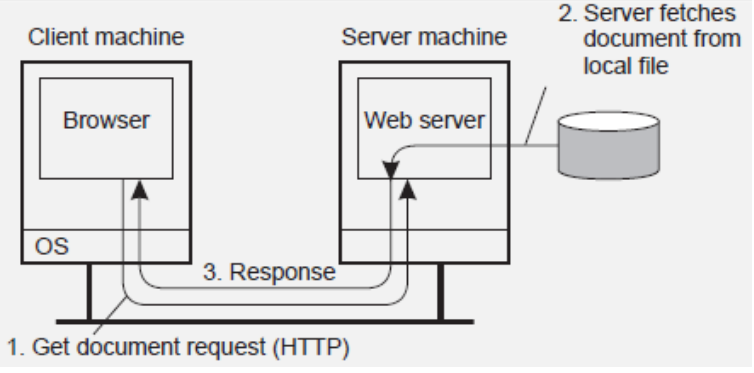
Example: Server-side replication in Coda

- Ensure that concurrent updates are detected:
  - Each client has an Accessible Volume Storage Group (AVSG): is a subset of the actual VSG.
  - Version vector $CVV_i(f)[j] = k \Rightarrow S_i$ knows that S_j has seen version k of f .
  - Example: A updates $f \Rightarrow S_1 = S_2 = [+1, +1, +0]$; B updates $f \Rightarrow S_3 = [+0, +0, +1]$.



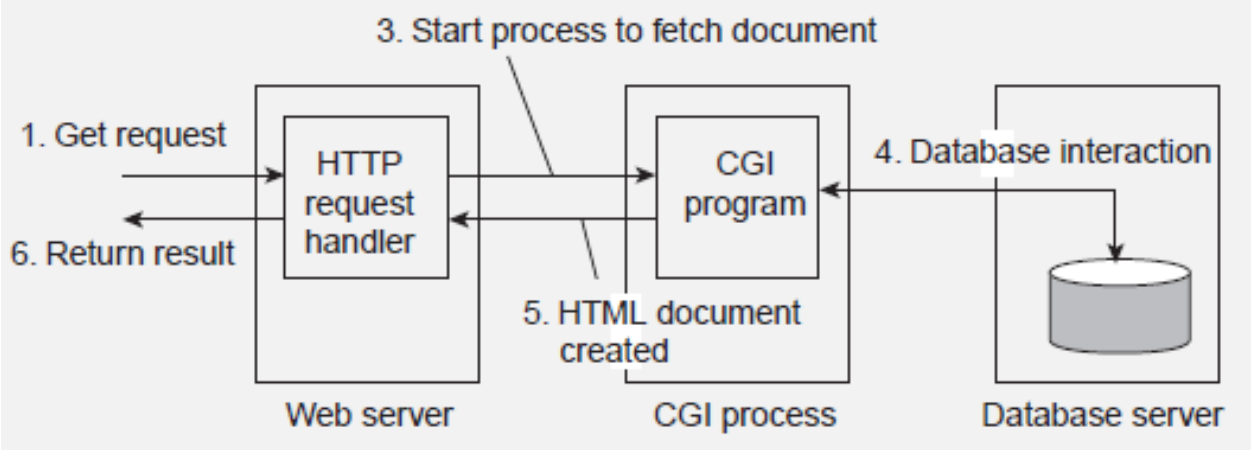
Distributed Web-based systems

- The WWW is a huge client-server system with millions of servers; each server hosting thousands of hyperlinked documents.
  - Documents are often represented in text (plain text, HTML, XML)
  - Alternative types: images, audio, video, applications (PDF, PS)
  - Documents may contain scripts, executed by client-side software



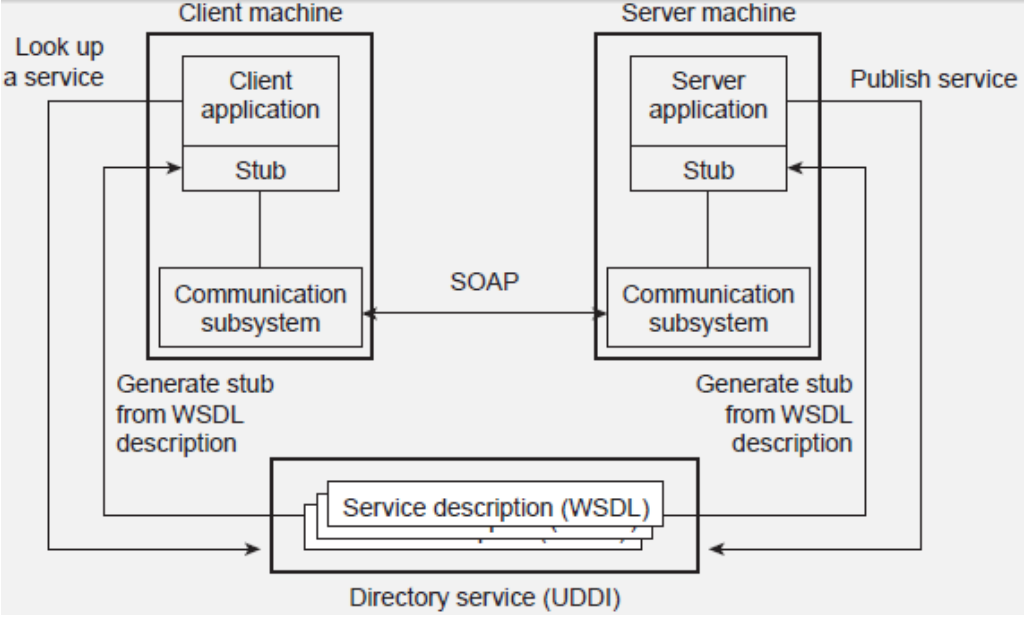
Multi-tiered architectures

- Already very soon, Web sites were organized into three tiers.



Web services

- At a certain point, people started recognizing that it is was more than just user ↔ site interaction: sites could offer services to other sites ⇒ standardization is then badly needed.



专题报告/课程Project（10分）

- 大数据处理系统简介

- Raft