Institute of Aeronautical Engineering

(Autonomous)

Dundigal, Hyderabad-500043

Department of Computer Science and Engineering

DISTRIBUTED SYSTEMS

Prepared By:

Mr. RM NOORULLAH	Associate Professor
Mr. N V KRISHNA RAO	Associate Professor
Mr. CH SRIKANTH	Assistant Professor
Mr. M RAKESH	Assistant Professor

UNIT 1
Characterization of Distributed Systems

Chapter 1 Characterization of Distributed Systems

1. Introduction
2. Examples of distributed systems
3. Trends in distributed systems
4. Focus on resource sharing
5. Challenges
6. Case study: The World Wide Web
7. Summary

1.1 Introduction

- Motivation:

  Networks of computers are everywhere!

  – Mobile phone networks
  – Corporate networks
  – Factory networks
  – Campus networks
  – Home networks
  – In-car networks
  – Planetary networks

  Why networked?

  Desire to share resources

- Influence:

  Networked computers impact system designers and implementers

Defining Distributed Systems

- *"A system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing."* [Coulouris]
  - Networked computers could be far apart or in the same room
    - relying on computer networking
    - i.e. cluster and grid

- *"A distributed system is a collection of independent computers that appear to the users of the system as a single computer."* [Tanenbaum]

Consequences of Distributed Systems

- Concurrency
  - Autonomous: Computers carry out tasks independently
  - Cooperative: Computers coordinate actions
- No global clock
  - Hard to synchronize their clocks precisely
  - Coordinate by exchanges messages
- Independent failures
  - Part of network or node faults does stop the running of the whole system

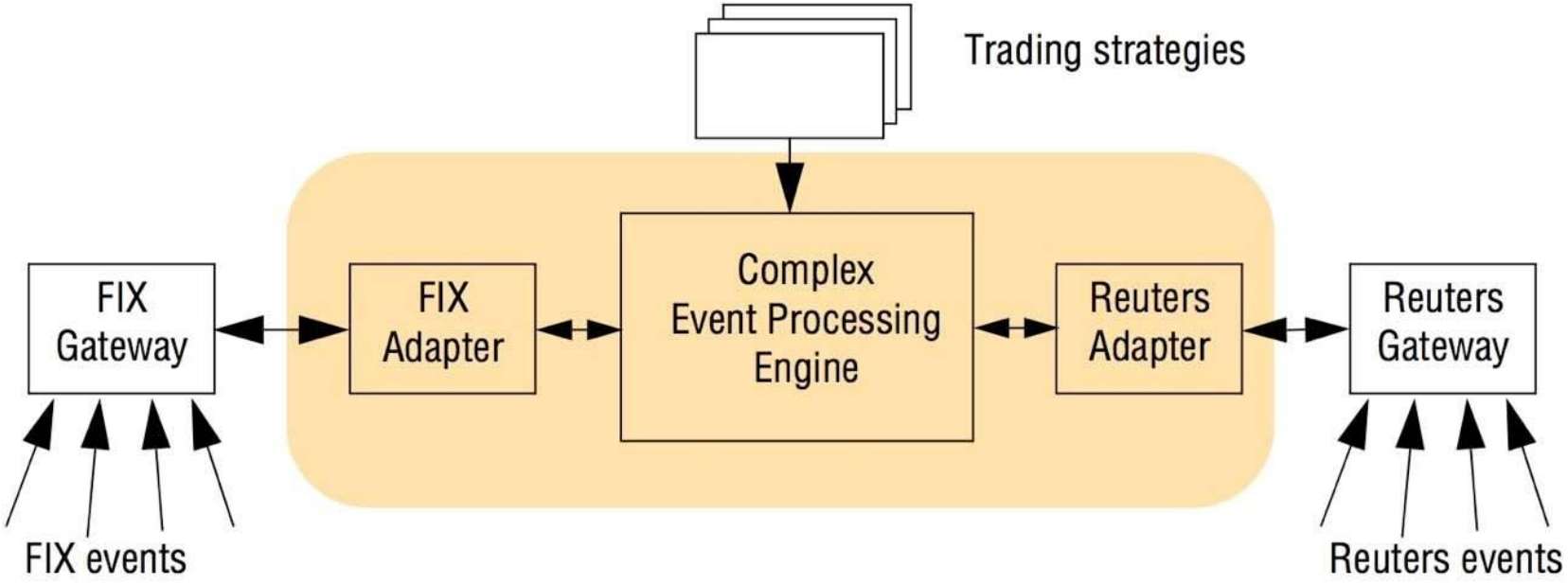
1.2 Examples of Distributed Systems

- Examples
  - 1.2.1 Web search (Google)
  - 1.2.2 Massively multiplayer online games (MMOGs)
  - 1.2.3 Financial trading

Figure 1.1 Selected application domains and associated networked applications

Finance and commerce	eCommerce e.g. Amazon and eBay, PayPal, online banking and trading
The information society	Web information and search engines, ebooks, Wikipedia; social networking: Facebook and MySpace.
Creative industries and entertainment	online gaming, music and film in the home, user-generated content, e.g. YouTube, Flickr
Healthcare	health informatics, on online patient records, monitoring patients
Education	e-learning, virtual learning environments; distance learning
Transport and logistics	GPS in route finding systems, map services: Google Maps, Google Earth
Science	The Grid as an enabling technology for collaboration between scientists
Environmental management	sensor technology to monitor earthquakes, floods or tsunamis

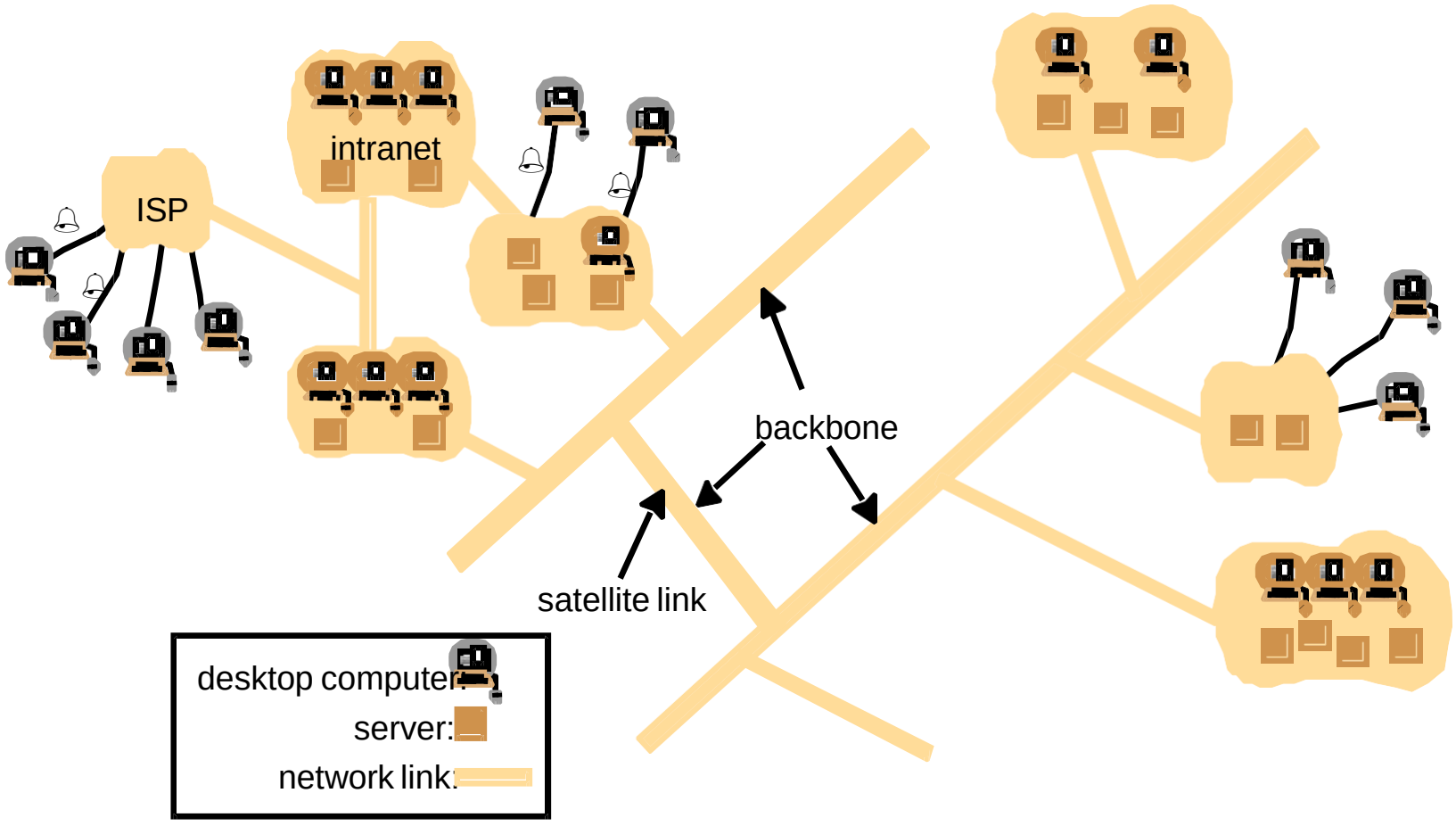
Figure 1.2
An example financial trading system



1.3 Trends in distributed systems

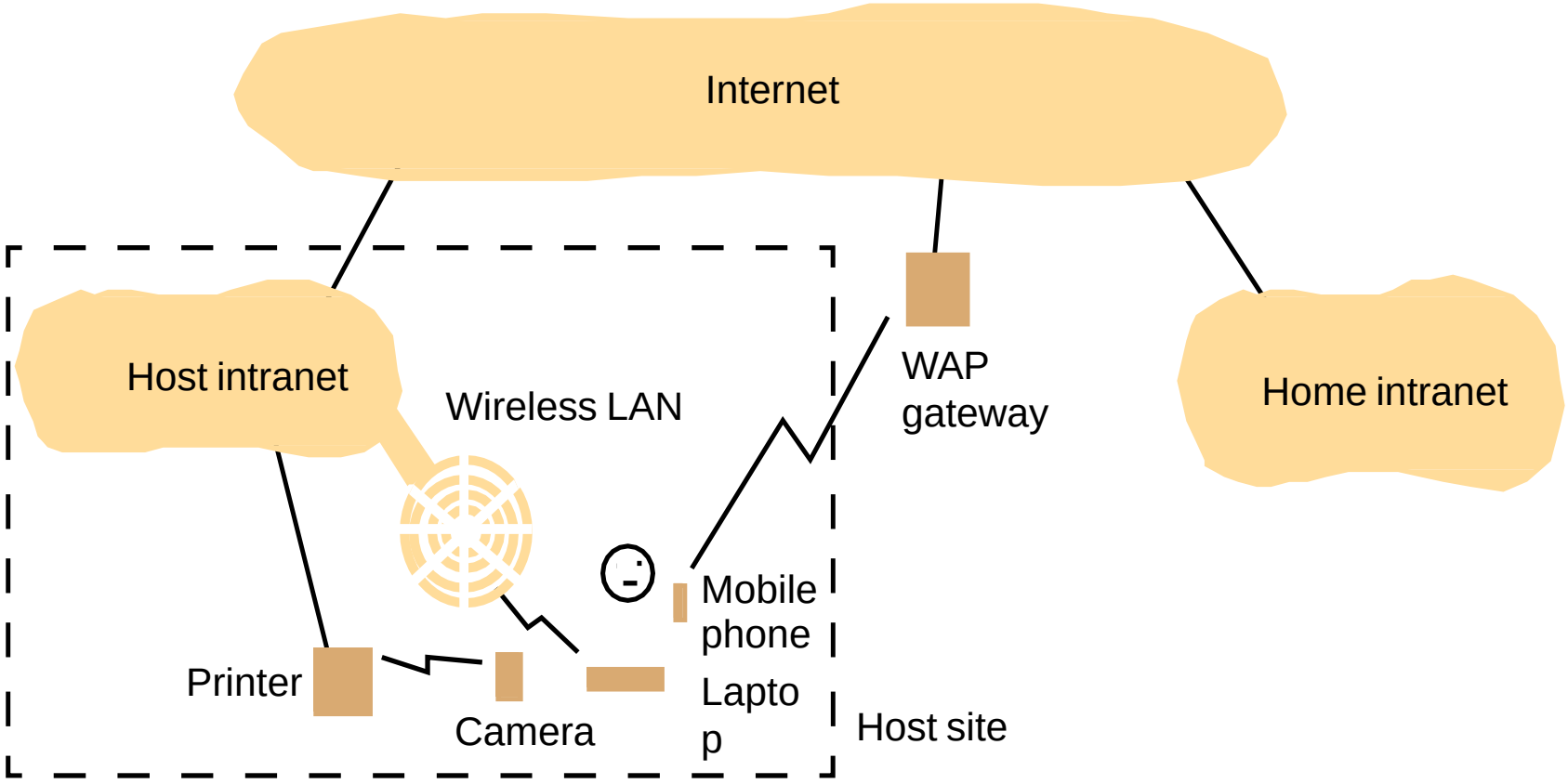
- Influential trends
  - Emergence of pervasive networking technology
  - Emergence of ubiquitous computing coupled with the desire to support mobility
  - Increasing demand of multimedia services
  - View of distributed systems as a utility
- Sections
  1. Pervasive networking and the modern Internet
  2. Mobile and ubiquitous computing
  3. Distributed multimedia systems
  4. Distributed computing as a utility (cloud)

Figure 1.3 Typical Portion of the Internet



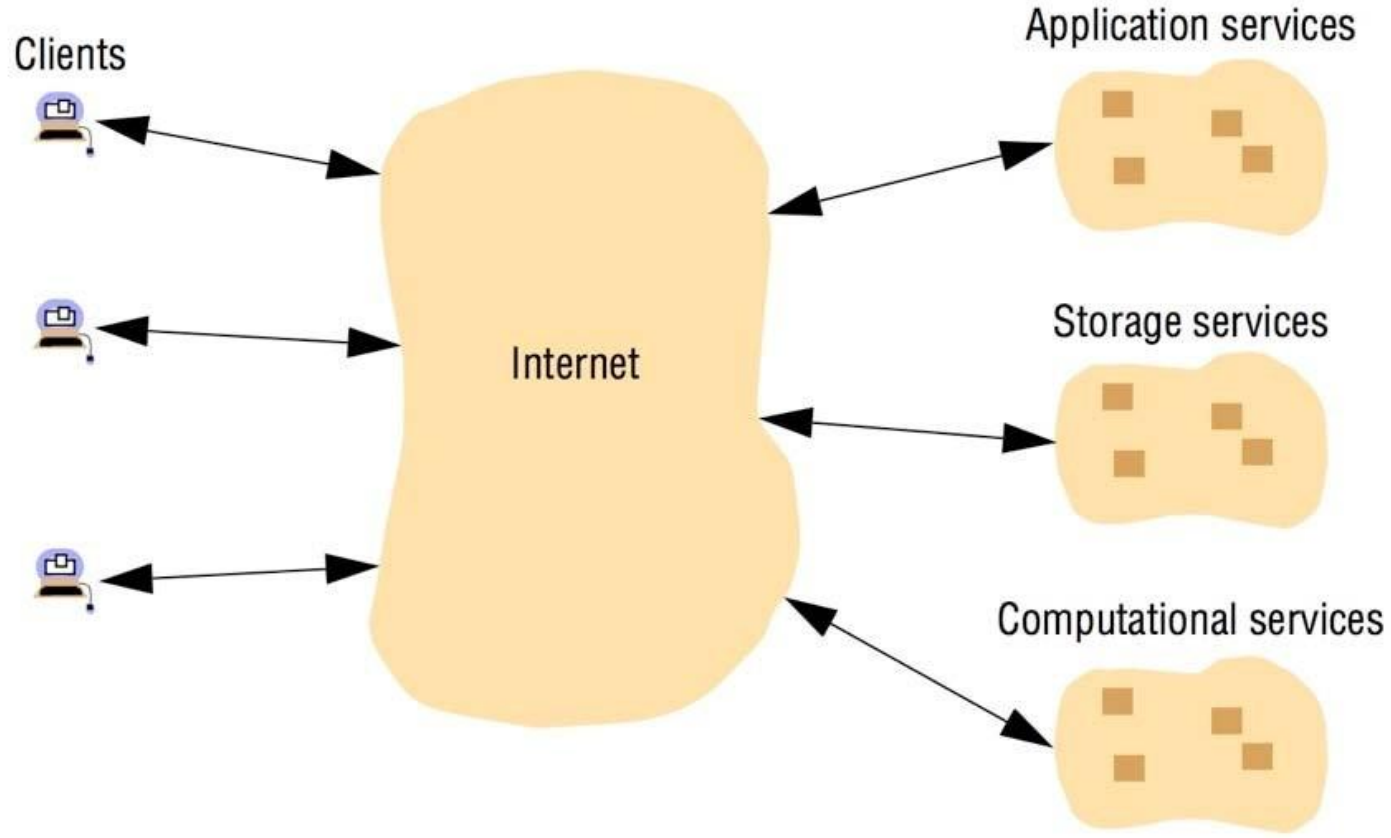
The Internet is a vast collection of computer networks of many different types and hosts supporting various types of services

Figure 1.4 Portable and handheld devices



Support continued access to Home intranet resources via wireless and provision to utilize resources (e.g., printers) that are conveniently located (location-aware computing)

Figure 1.5 Cloud Computing



A ***cloud*** is a set of Internet-based application, storage and computing services sufficient to support most users' needs, thus enabling them to largely or totally dispense with local data storage and application software

1.4 Focus on Resource Sharing

- Users are accustomed to the benefits of resource sharing
  - Share hardware resources such as printers
  - Share data such as files
  - Share specific functionality such as search engines

- Service-oriented: client-server computing
  - remote invocation of an operation to an object

1.5 Challenges

- 1.5.1 Heterogeneity
  - networks, hardware, os, languages...
  - solutions: middleware (i.e. corba), mobile code, virtual machines
- 1.5.2 Openness
  - extended and re-implemented in various ways
  - standard published interfaces, RFC (request for comments)
- 1.5.3 Security
  - confidentiality, integrity, availability
- 1.5.4 Scalability
  - effective with significant increase in resources
  - cost and performance
- 1.5.5 Failure handling
  - detecting
  - masking: hide, less severe (retransmit)
  - tolerating: ignore, timeout
  - recovery: logs, rollback
  - redundancy
- 1.5.6 Concurrency
  - several clients access a shared resource at the same time

Challenges (Cont.)

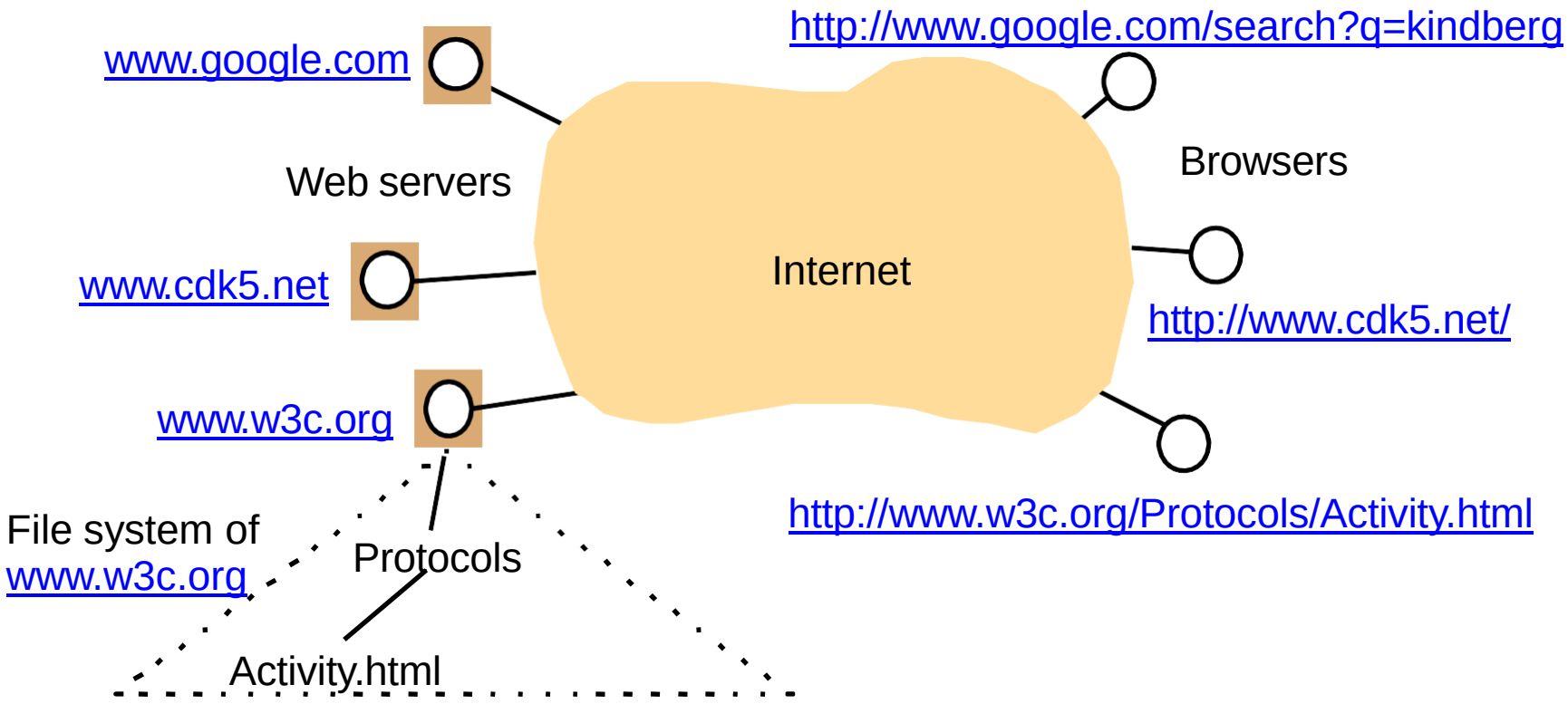
- 1.5.7 Transparency
  - *Access transparency*: enables local and remote resources to be accessed using identical operations
  - *Location transparency*: enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address)
  - *Concurrency transparency*: enables several processes to operate concurrently using shared resources without interference between them
  - *Replication transparency*: enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers
  - *Failure transparency*: enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components
  - *Mobility transparency*: allows the movement of resources and clients within a system without affecting the operation of users or programs.
  - *Performance transparency*: allows the system to be reconfigured to improve performance as loads vary
  - *Scaling transparency*: allows the system and applications to expand in scale without change to the system structure or application algorithms
- 1.5.8 Quality of Service

1.6 Case Study: The World Wide Web

Three major parts

- HTML, Hyper Text Markup Language
- URL, Uniform Resource Locator
  - http://servername[:port] [/pathname] [?arguments]
- HTTP, HyperText Transfer Protocol
  - request-reply protocol (client-server)
  - content types--MIME types, multipurpose internet mail extensions
  - one resource per request
  - simple access control (mostly public)

Figure 1.7 Web Servers and Web Browsers



Other Web Technologies

- web forms
- CGI programs, common gateway interface, run on the server
- applets, run on the client
- RDF, resource description framework, vocabulary for meta-data
- XML, extensible markup language, allow meta-data information to be included

1.7 Summary

- Computer networks and distributed systems are everywhere
- Resource sharing is the main motivating factor for constructing distributed systems
- Challenges: heterogeneity, openness, security, scalability, failure handling, concurrency, transparency

Chapter 2
System Models

Chapter 2 System Models

1. Introduction
2. Physical models
3. Architectural models
4. Fundamental models
5. Summary

2.1 Introduction

- A physical model considers
  - the types of computers and devices that constitute a system
  - their interconnectivity
- An architectural model defines
  - the way in which the components of systems interact with one another and
  - the way in which they are mapped onto an underlying network of computers
- Fundamental models take an abstract perspective
  - in order to describe solutions to individual issues faced by most distributed systems
- Why model?
  - Each model is intended to provide an abstract, simplified but consistent description of a relevant aspect of distributed system design

2.2 Physical Models

- Baseline physical model: one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages
- Three generations
  - Early distributed systems: late 1970s and early 1980s
    - typical 10 and 100 nodes of LAN
  - Internet-scale distributed systems: 1990s
    - network of networks
  - Contemporary distributed systems: 2000s (cloud)
    - mobile devices and wireless networking
- Next-generation: distributed systems of systems?
  - a complex system consisting of a series of subsystems
    - that are systems in their own right, and
    - that come together to perform a particular task or tasks

Figure 2.1
Generations of distributed systems

Distributed systems:	*Early*	*Internet-scale*	*Contemporary*
Scale	Small	Large	Ultra-large
Heterogeneity	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
Openness	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
Quality of service	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

2.3 Architectural Models

- The architecture of a system is its structure in terms of separately specified components.
  - Its goal is to meet present and likely future demands.
  - Major concerns are make the system reliable, manageable, adaptable, and cost-effective.
- Architectural Model
  - Goal: simplifies and abstracts the functions of individual components
  - A three-stage approach
    - **Architectural elements**: looking at the core underlying architectural elements
    - **Architectural patterns**: examining composite architectural patterns that can be used in isolation or in combination in developing more sophisticated distributed systems solutions
    - **Associated middleware solutions**: considering middleware platforms that are available to support the various styles of programming

2.3.1 Architectural Elements

- Four key questions
  1. What are the ***entities*** that are communicating in the distributed system?
  2. How do they communicate or what ***communication*** paradigm is used?
  3. What ***roles and responsibilities*** do they have in the overall architecture?
  4. How are they ***mapped*** on to the physical distributed infrastructure? (What is their placement?)

Communication Entities and Paradigms

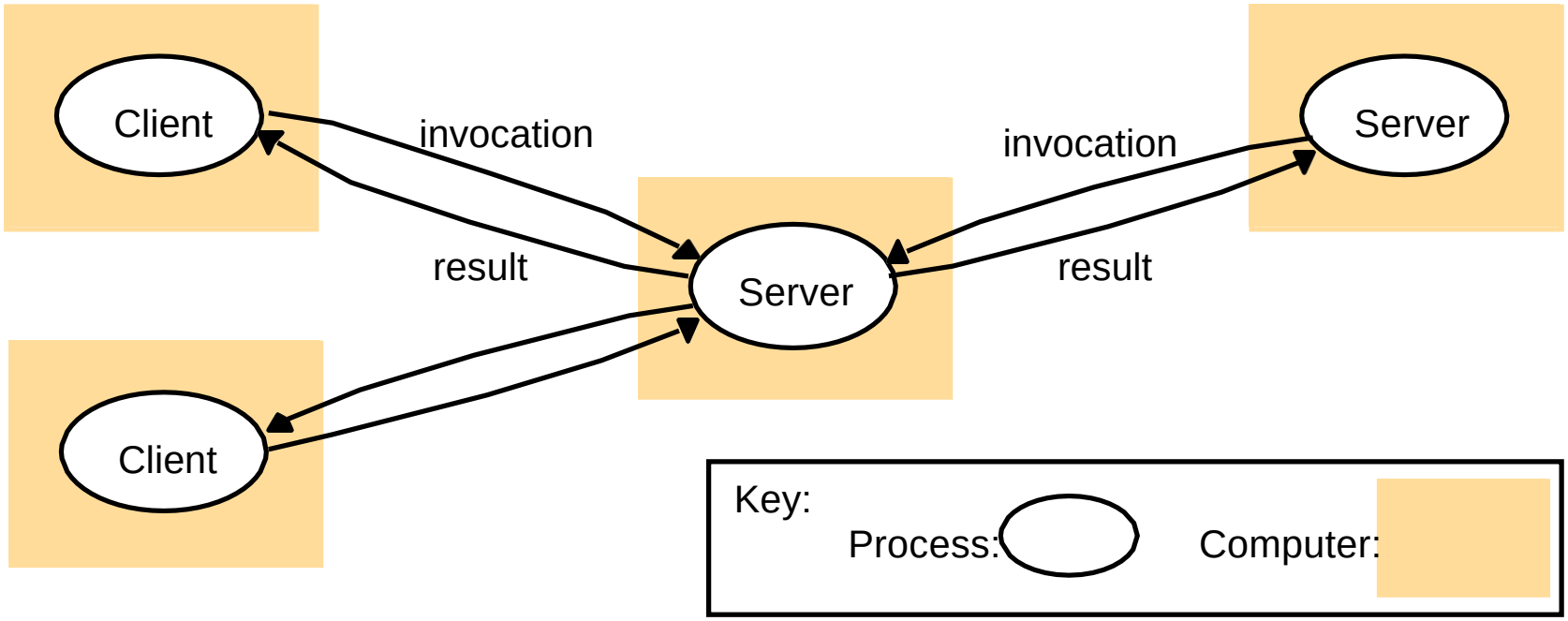
- Communicating entities
  - System-oriented perspective: processes
  - Problem-oriented perspective: objects, components, Web services

- Communication paradigms
  - Interprocess communication: RPC, request-reply protocols
  - Remote invocation: RMI
  - Indirect communication: group communication, publish-subscribe systems, message queues, tuple spaces, distributed shared memory

Figure 2.2 Communicating entities and communication paradigms

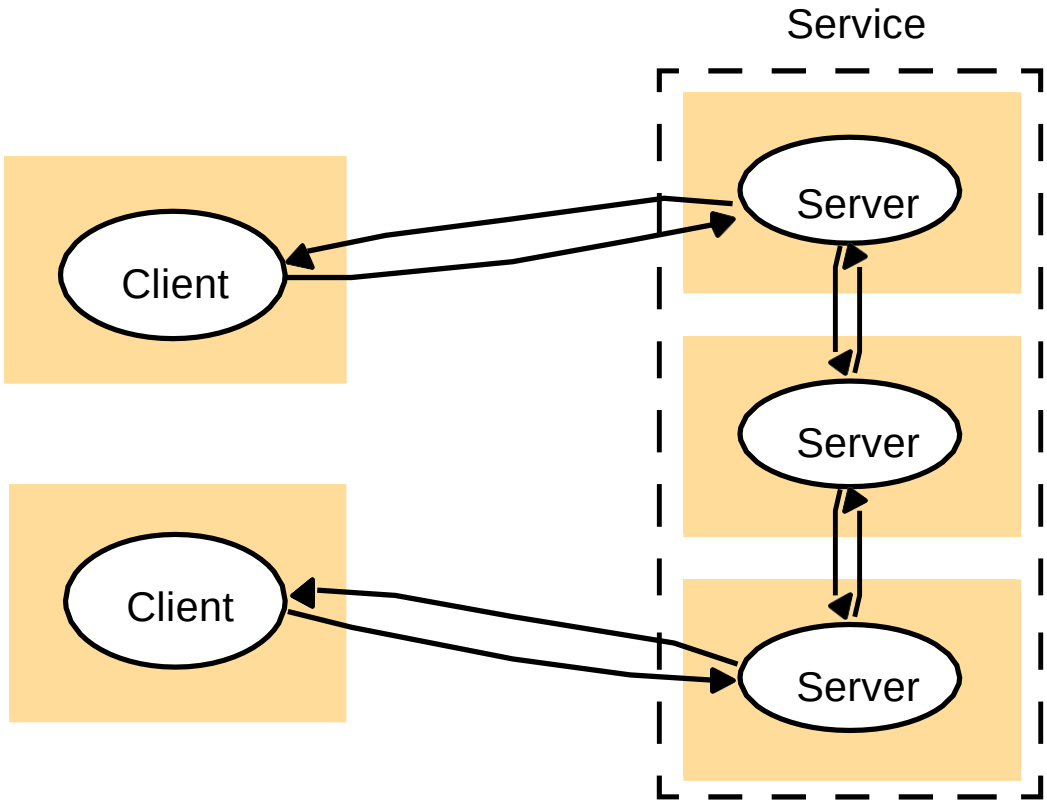
Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Figure 2.3
Clients invoke individual servers



- Servers provide services
- Clients access services

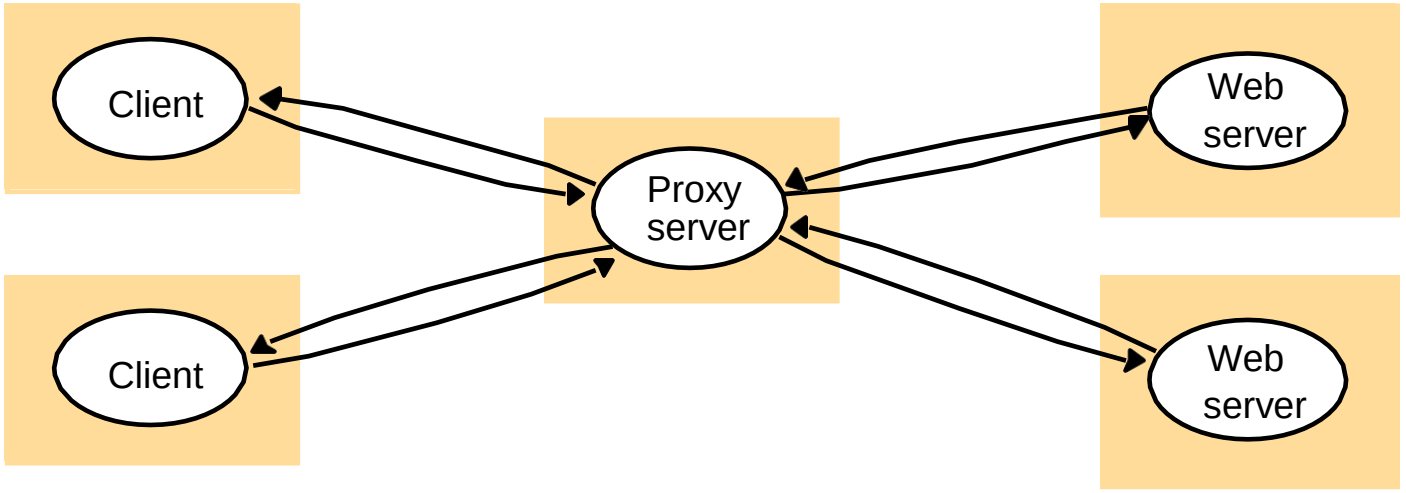
Figure 2.4 A service provided by multiple servers



- Objects are partitioned/replicated
  - Web: each server manages its objects
  - NIS: replicated login/password info
  - Cluster: closely coupled, scalable (search engines)

Figure 2.5 Web proxy server

- Cache: local copies of remote objects for faster access
- Browser cache
- Proxy server
  - Additional roles: filtering, firewall



Multiple Web proxy servers

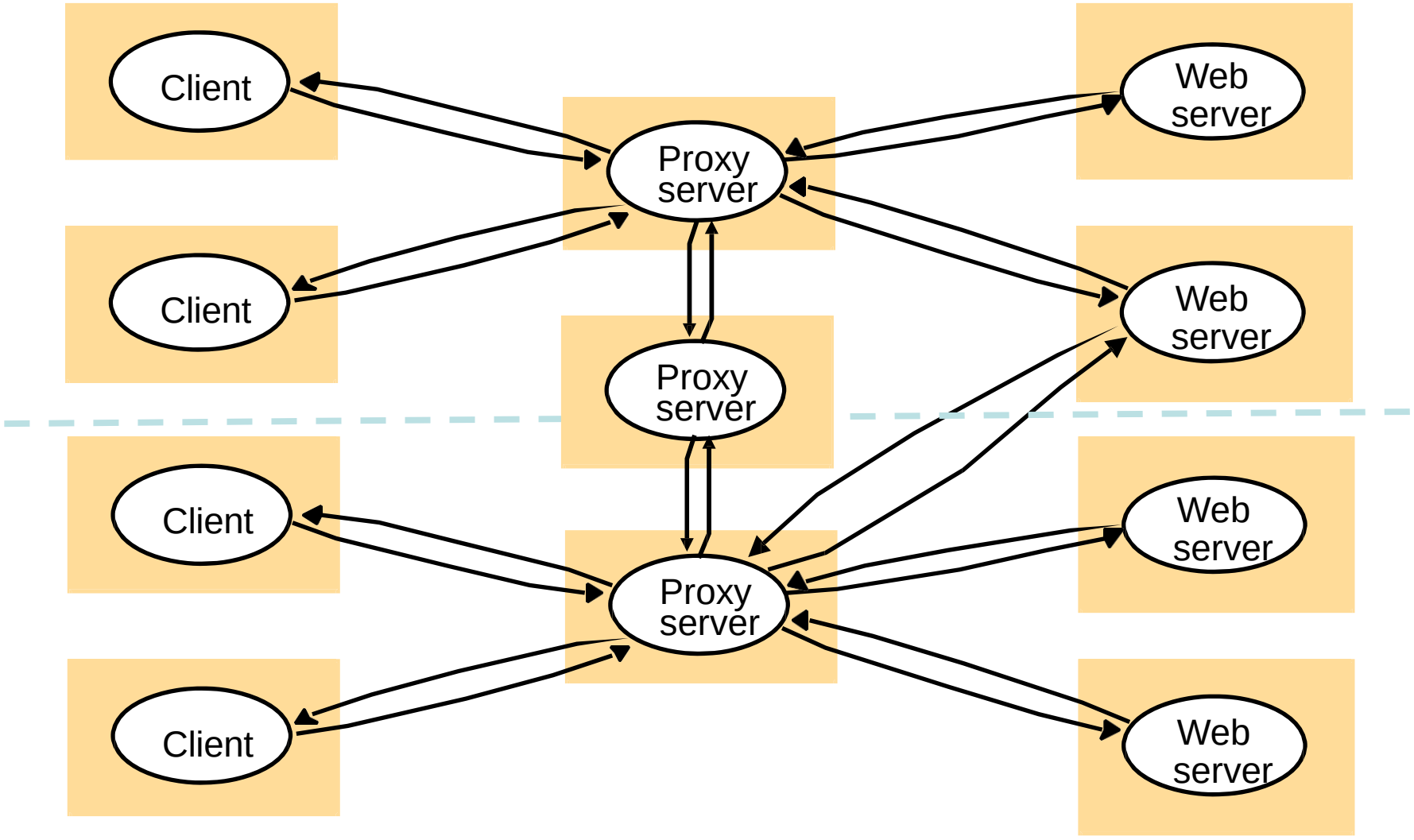
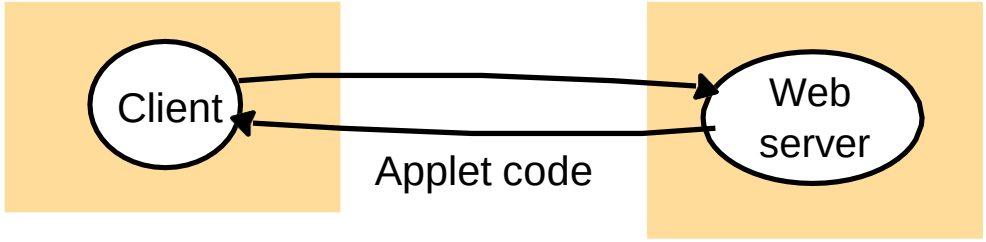


Figure 2.6 Web applets

- Running code locally vs. remotely
  - Network bandwidth
  - What examples have you seen?
  - Security issues?

a) client request results in the downloading of applet code



b) client interacts with the applet

Mobile agents

- Running program that travels from one computer to another
  - Perform tasks on users' behalf
  - Security issues
- How does it compare to one client interacting with multiple servers?
  - What if the program needs to access a lot of remote data?
- How does it compare with applets?
- Agents can provide functionality a remote web site does not provide, but allows access to data.

Peer to Peer

- Peers play similar roles
- No distinction of responsibilities

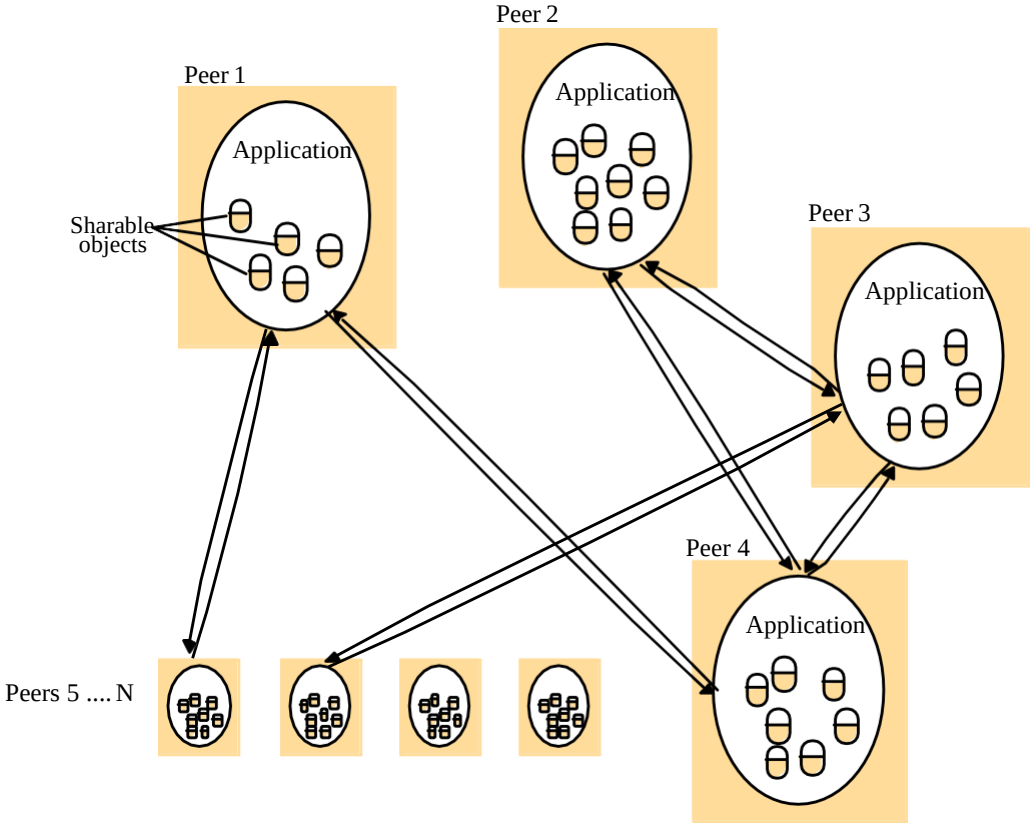


Figure 2.15 Omission and arbitrary failures

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a*send*, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Summary

1. Introduction
2. Physical models
3. Architectural models
4. Fundamental models
5. Summary

UNIT 2
Time and Global States

Chapter 1 Time and Global States

1. Introduction
2. Clocks, events and process states
3. Synchronizing physical clocks
4. Logical time and logical clocks
5. Global states
6. Distributed debugging
7. Summary

1.1 Introduction

- Importance of time in distributed systems
  - A quantity to timestamp events accurately
    - To know what time a particular event occurs
    - i.e. Recording when an e-commerce transaction occurs
  - A synchronization source for several distributed algorithms
    - To maintain consistency of distributed data
    - i.e. Eliminating duplicate updates
  - A timing source for multiple events
    - To provide relative order of two events
    - i.e. Ensuring the order of cause and effect
- Clocks in computers to establish
  - *Time* at which an event occurred
  - *Duration* of an event or interval between two events
  - *Sequence* of a series of events or the order in which events occurred

1.2 Clocks, Events and Process States

- A distributed system consists of a collection *P* of ***N*** processes ***p_i***, *i = 1,2,… N*
  - Each process p_i has a state ***s_i*** consisting of its variables (which it transforms as it executes)
  - Processes communicate only by messages (via a network)
    - **Actions** of processes*: Send, Receive,* change own state
- ***Event****:* the occurrence of a single action that a process carries out as it executes
- Events at a single process p_i, can be placed in a total **ordering** denoted by the relation $\rightarrow_i$ between the events. i.e.
  - ***e*** $\rightarrow_i$ ***e'*** if and only if event *e* occurs before event *e'* at process p_i
- A history of process p_i is a series of events ordered by $\rightarrow_i$
  - ***history*****(p_i)** = $h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

Clocks

To timestamp events, use the computer's clock

- At **real time, *t***, the OS reads the time on the computer's **hardware clock $H_i(t)$**
  - It calculates the time on its **software clock $C_i(t)=$**

  $\alpha H_i(t) + \beta$
  - e.g. a 64 bit value giving nanoseconds since some base time
  - **Clock resolution**: period between updates of the clock value
  - In general, the clock is not completely accurate
  - but if C_i behaves well enough, it can be used to timestamp events at p_i

Skew between computer clocks in a distributed system

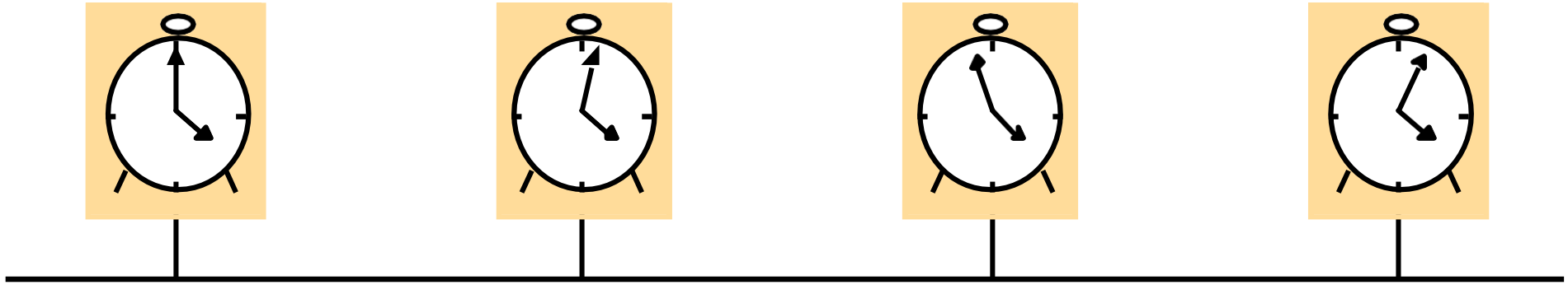


Figure 1.1

Computer clocks are not generally in perfect agreement

- ***Clock skew***: the difference between the times on two clocks (at any instant)
- Computer clocks use crystal-based clocks that are subject to physical variations
  - ***Clock drift***: they count time at different rates and so diverge (frequencies of oscillation differ)
  - ***Clock drift rate***: the difference per unit of time from some ideal reference clock
  - Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
  - High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Coordinated Universal Time (UTC)

- UTC is an international standard for time keeping
  - It is based on atomic time, but occasionally adjusted to astronomical time
  - International Atomic Time is based on very accurate physical clocks (drift rate 10^{-13})
- It is broadcast from radio stations on land and satellite (e.g. GPS)
- Computerswith receivers can synchronize their clocks with these timing signals (by requesting time from GPS/UTC source)
  - Signals from land-based stations are accurate to about 0.1-10 millisecond
  - Signals from GPS are accurate to about 1 microsecond

1.3 Synchronizing physical clocks

Two models of synchronization

- External synchronization: a computer's clock C_i is synchronized with an external authoritative time source S, so that:
  - $|S(t) - C_i(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, I of real time
  - The clocks C_i are **accurate** to within the bound D.
- Internal synchronization: the clocks of a pair of computers are synchronized with one another so that:
  - $| C_i(t) - C_j(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, I of real time
  - The clocks C_i and C_j **agree** within the bound D.

Internally synchronized clocks are not necessarily externally synchronized, as they may drift collectively

- if the set of processes P is synchronized externally within a bound D, it is also internally synchronized within bound $2D$ *(worst case polarity)*

Clock correctness

- ***Correct clock***: a hardware clock H is said to be correct if its drift rate is within a bound $\rho > 0$ (e.g. 10^{-6} secs/ sec)

This means that the error in measuring the interval between real times t and t' is bounded:

– $(1 - \rho)$ $(t' - t) \leq H(t') - H(t) \leq (1 + \rho)$ $(t' - t)$ (where $t'>t$)

  – Which forbids jumps in time readings of hardware clocks

- ***Clock monotonicity***: weaker condition of correctness

– $t' > t \Rightarrow$ $C(t') > C(t)$

  – e.g. required by Unix *make*

  – A hardware clock that runs fast can achieve monotonicity by adjusting the values of α and β such that $C_i(t) = \alpha H_i(t) + \beta$

- ***Faulty clock***: a clock not keeping its correctness condition

  – *crash failure* - a clock stops ticking

  – *arbitrary* failure - any other failure

    - e.g. jumps in time; Y2K bug

1.3.1 Synchronization in a synchronous system

A synchronous distributed system is one in which the following bounds are defined :

–the time to execute each step of a process has known lower and upper bounds

–each message transmitted over a channel is received within a known bounded time (min and max)

–each process has a local clock whose drift rate from real time has a known bound

- Internal synchronization in a synchronous system
  - One process p_1 sends its local time t to process p_2 in a message m
  - p_2 could set its clock to $t + T_{trans}$ where T_{trans} is the time to transmit m
  - T_{trans} is unknown but $min \leq T_{trans} \leq max$
  - uncertainty $u = max\text{-}min$. Set clock to $t + (max - min)/2$ then skew $\leq u/2$

1.3.2 Cristian's method for an asynchronous system

- A time server *S* receives signals from a UTC source
  - Process *p* requests time in m_r and receives *t* in m_t from *S*
  - *p* sets its clock to $t + T_{round}/2$
  - Accuracy $\pm (T_{round}/2 - min)$:
    - because the earliest time *S* puts *t* in message m_t is *min* after *p* sent m_r
    - the latest time S puts t in message m_t was *min* before m_t arrived at *p*
    - the time by *S*'s clock when m_t arrives is in the range $[t+min, t + T_{round} - min]$
    - the width of the range is $T_{round} - 2min$

T_{round} is the round trip time recorded by *p*
min is the minimum transmission time

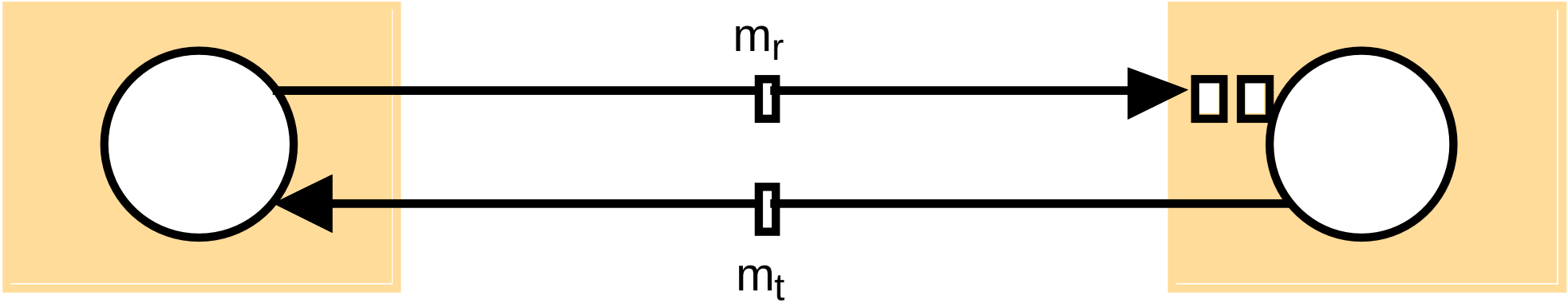


Figure 1.2

1.3.3 The Berkeley algorithm

- Problem with Cristian's algorithm
  - a single time server might fail, so they suggest the use of a group of synchronized servers
  - it does not deal with faulty servers
- Berkeley algorithm (also 1989)
  - An algorithm for internal synchronization of a group of computers
  - A *master* polls to collect clock values from the others (*slaves*)
  - The master uses round trip times to estimate the slaves' clock values
  - It takes an average (eliminating any above some average round trip time or with faulty clocks)
  - It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
  - Measurements
    - 15 computers, clock synchronization 20-25 millisecs drift rate < 2×10^{-5}
    - If master fails, can elect a new master to take over (not in bounded time)

1.3.4 Network Time Protocol (NTP)

- A time service for the Internet - synchronizes clients to UTC
  - Reliability from redundant paths, scalable, authenticates time sources
- Architecture
  - Primary servers are connected to UTC sources
  - Secondary servers are synchronized to primary servers
  - Synchronization subnet - lowest level servers in users' computers
    - strata: the hierarchy level

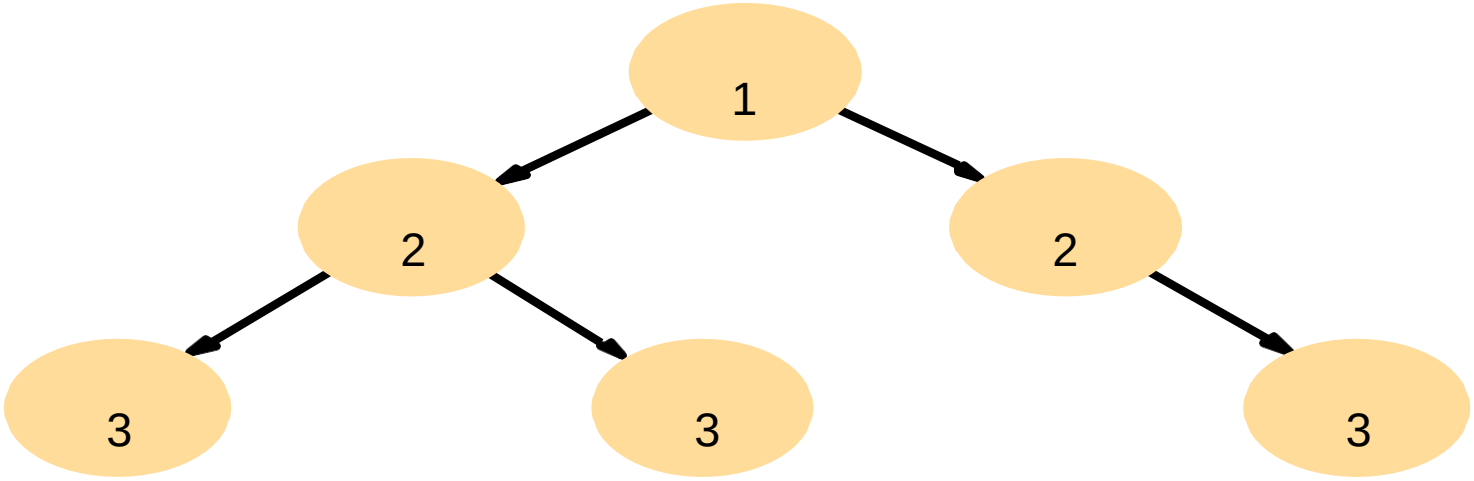


Figure 1.3 An example synchronization subnet in an NTP implementation

NTP - synchronization of servers

- The synchronization subnet can reconfigure if failures occur
  - a primary that loses its UTC source can become a secondary
  - a secondary that loses its primary can use another primary
- Modes of synchronization for NTP servers:
  - Multicast
    - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
  - Procedure call
    - A server accepts requests from other computers (like Cristian's algorithm)
    - Higher accuracy. Useful if no hardware multicast.
  - Symmetric
    - Pairs of servers exchange messages containing time information
    - Used where very high accuracies are needed (e.g. for higher levels)

Messages exchanged between a pair of NTP peers

- All modes use UDP
- Each message bears timestamps of recent events:
  - Local times of *Send* and *Receive* of previous message
  - Local times of *Send* of current message
- Recipient notes the time of receipt T_i (we have T_{i-3}, T_{i-2}, T_{i-1}, T_i)
- In symmetric mode there can be a non-negligible delay between messages

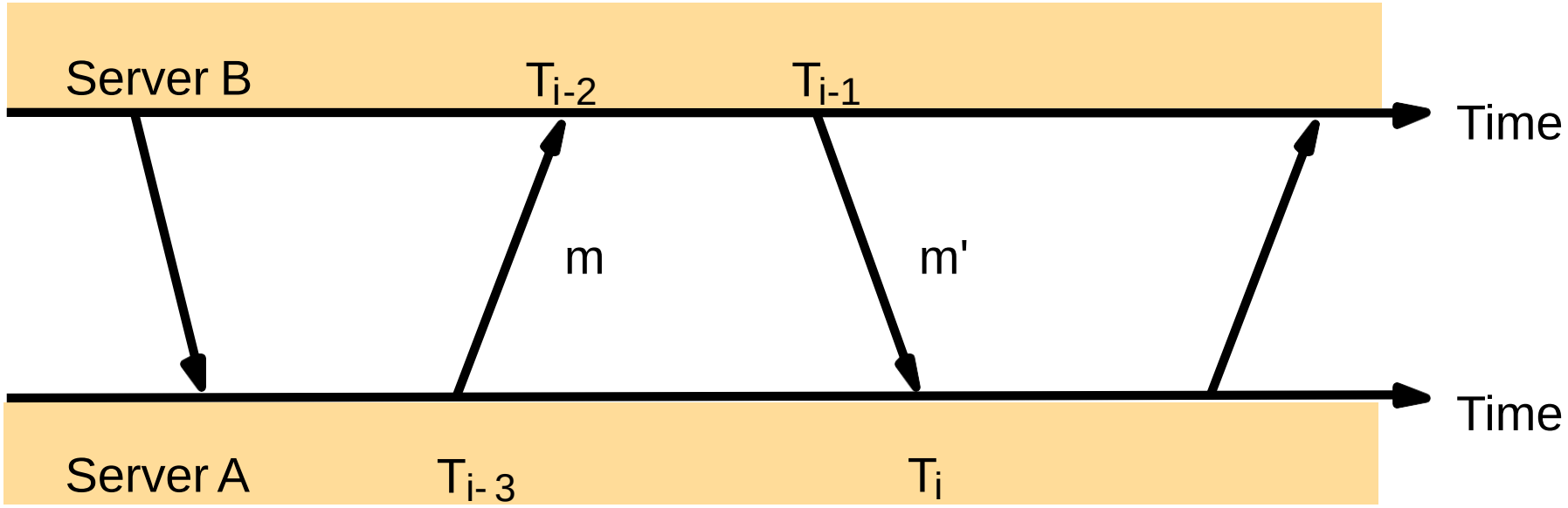


Figure 1.4

Accuracy of NTP

- Estimations of clock offset and message delay
  - For each pair of messages between two servers, NTP estimates an offset o_i (between the two clocks) and a delay d_i (total time for the two messages, which take t and t')

$T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$

  - This gives us (by adding the equations) :

    $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$

  - Also (by subtracting the equations)

$o = o_i + (t' - t)/2$ where $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$

  - Using the fact that t, $t'>0$ it can be shown that

$o_i - d_i/2 \leq o \leq o_i + d_i/2$.

    - Thus o_i is an estimate of the offset and d_i is a measure of the accuracy
  - Data filtering
    - NTP servers filter pairs $<o_i, d_i>$, estimating reliability from variation (dispersions), allowing them to select peers; and synchronization based on the lowest dispersion or min d_i ok
    - A relatively high filter dispersion represents relatively unreliable data
  - Accuracy of tens of milliseconds over Internet paths (1 ms on LANs)

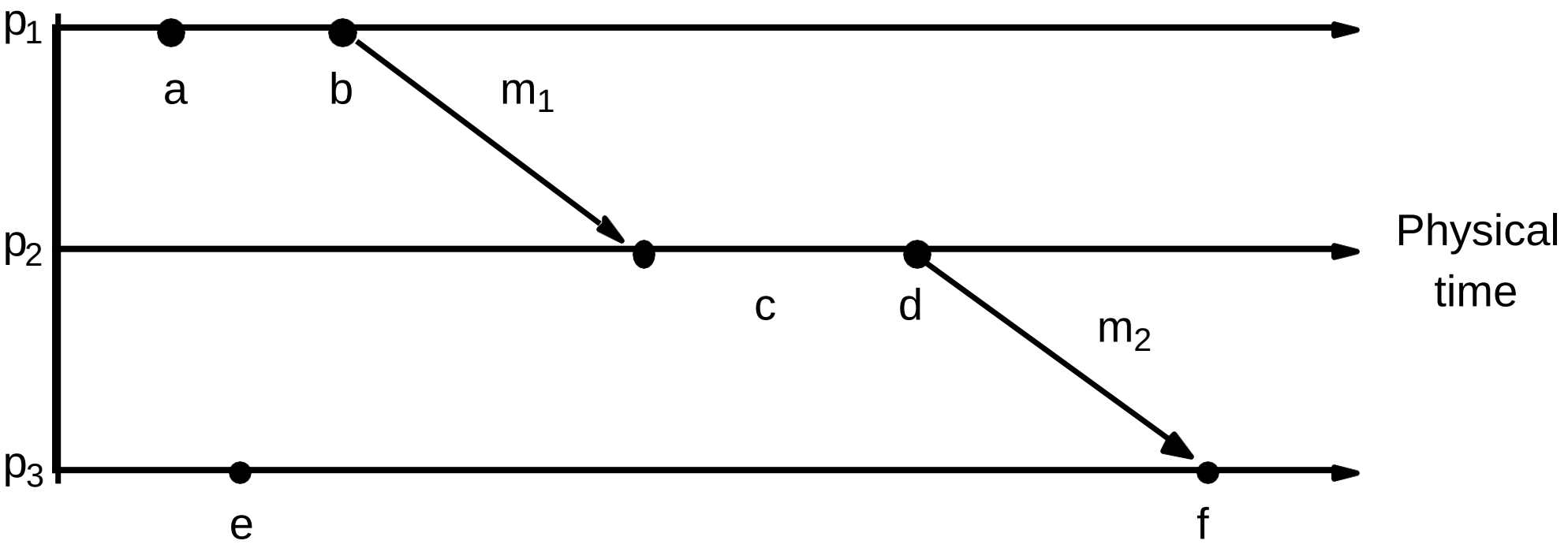
1.4 Logical time and logical clocks

- Instead of synchronizing clocks, event ordering can be used
  1. If two events occurred at the same process *pi* (*i* = 1, 2, … *N*) then they occurred in the order observed by *pi*, that is order □ $\rightarrow_i$
  2. when a message, *m* is sent between two processes, *send(m)* happened before *receive(m)*
- Lamport[1978] generalized these two relationships into the

happened-before relation: $e \rightarrow_i e'$

- HB1: if $e \rightarrow_i e'$ in process p_i, then $e \rightarrow e'$
- HB2: for any message *m*, send(*m*) $\rightarrow$ receive(*m*)
- HB3: if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Figure 1.5 Events occurring at three processes



- HB1: a→b, c→d, e→f
- HB2: b→c, d→f
- HB3: a→b→c→d→f
- **a||e**: a and e are concurrent (neither a→e nor e→a)

Lamport's logical clocks

- Each process p_i has a logical clock L_i
  - a monotonically increasing software counter
  - not related to a physical clock
- Apply Lamport timestamps to events with happened-before relation
  - LC1: L_i is incremented by 1 before each event at process p_i
  - LC2:

    (a)when process p_i sends message m, it piggybacks $t =$ L_i

    (b)when p_j receives (m,t), it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive* (m)

$e \rightarrow e'$ implies $L(e)<L(e')$, but $L(e)<L(e')$ does not imply $e \rightarrow e'$

$L(b)>L(e)$ but $b||e$

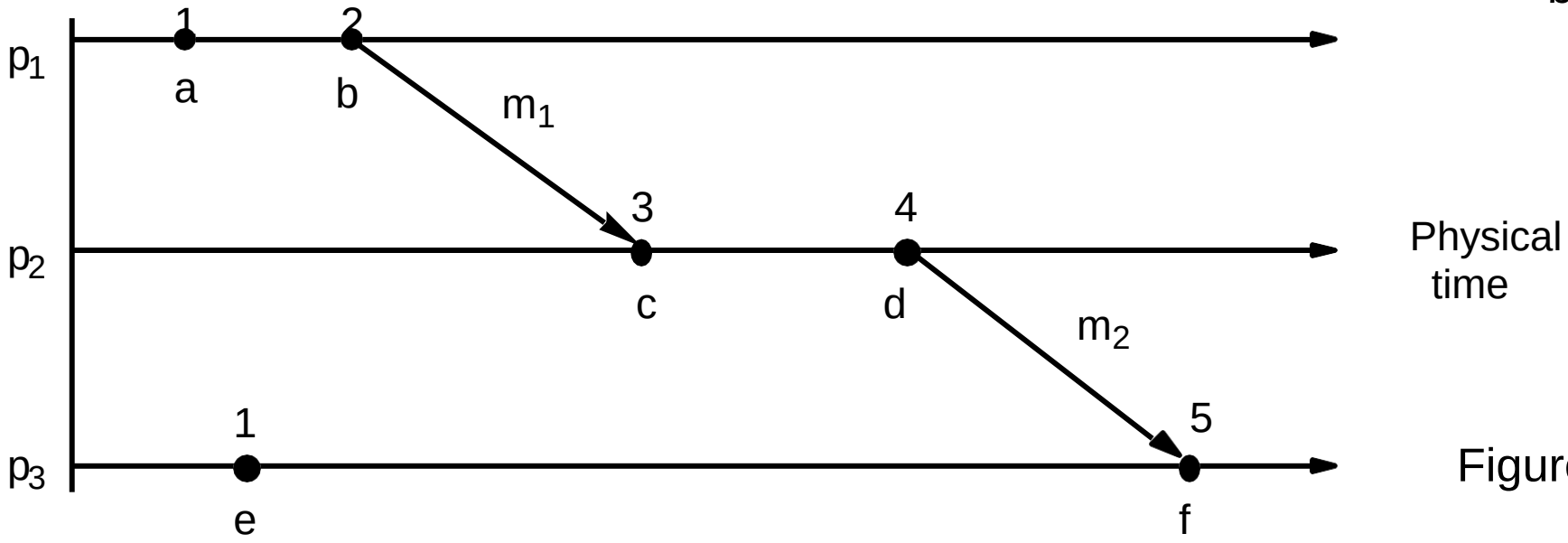


Figure 1.6

Totally ordered logical clocks

- Some pairs of distinct events, generated by different processes, may have numerically identical Lamport timestamps
  - Different processes may have same Lamport time
- Totally ordered logical clocks
  - If e is an event occurring at p_i with local timestamp T_i, and if e' is an event occurring at p_j with local timestamp T_j
  - Define global logical timestamps for the events to be (T_i, i) and (T_j, j)
  - Define $(T_i, i) < (T_j, j)$ iff
    - $T_i < T_j$ or
    - $T_i = T_j$ and $i < j$
  - No general physical significance since process identifiers are arbitrary

Vector clocks

- Shortcoming of Lamport clocks:

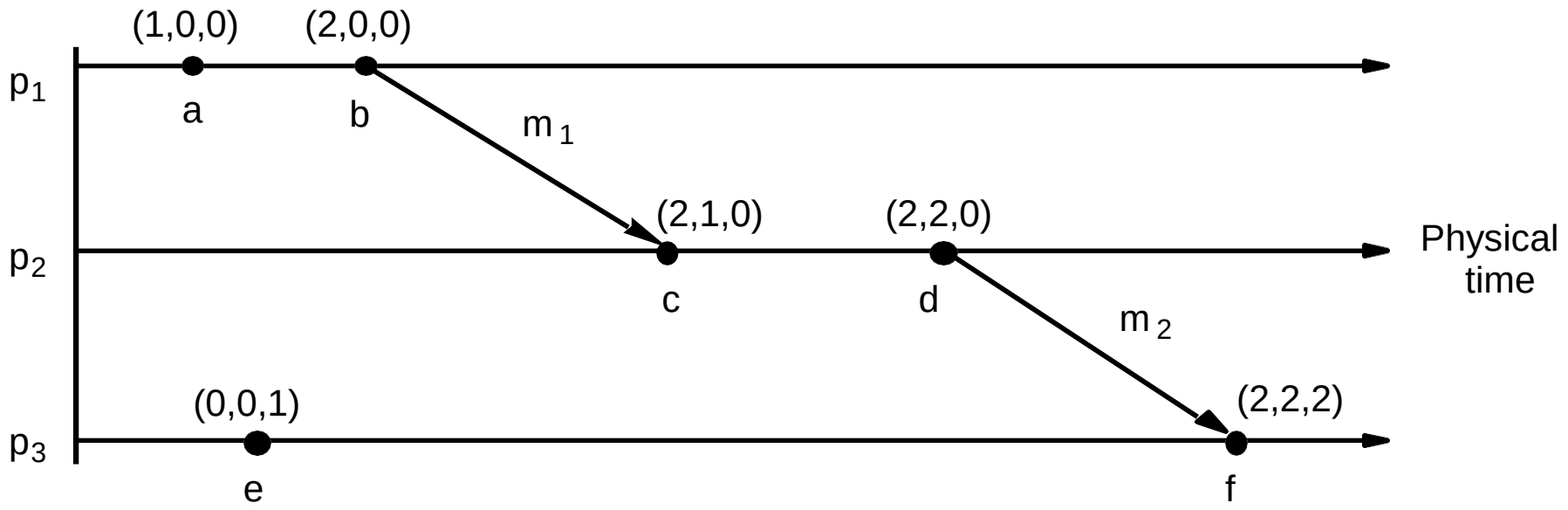
$L(e) < L(e')$ doesn't imply $e \rightarrow e'$

- Vector clock: an array of N integers for a system of N processes
  - Each process keeps its own vector clock V_i to timestamp local events
  - Piggyback vector timestamps on messages
- Rules for updating vector clocks:
  - $Vi[i]$ is the number of events that pi has timestamped
  - $Vi[i]$ ($j \neq i$) is the number of events at p_j that p_i has been affected by VC1: Initially, $V_i[\ j\ ] := 0$ for p_i, j=1.. N (N processes)

VC2: before p_i timestamps an event, $V_i[\ i\ ] := V_i[\ i\ ]+1$ VC3: p_i piggybacks $t = V_i$ on every message it sends

VC4: when p_i receives a timestamp t, it sets $V_i[\ j\ ] := \max(V_i[\ j\ ] , t[\ j\ ])$ for j=1..N (merge operation)

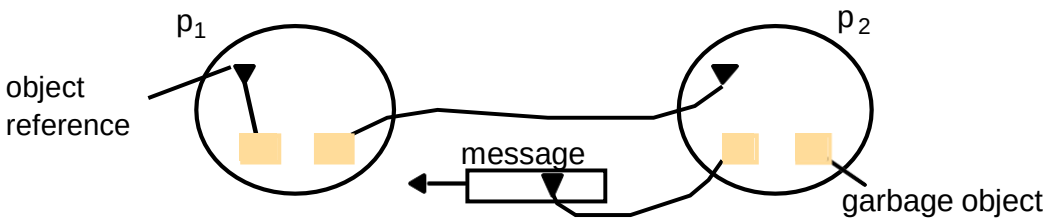
Figure 1.7 Vector timestamps for events shown in Figure 1.5



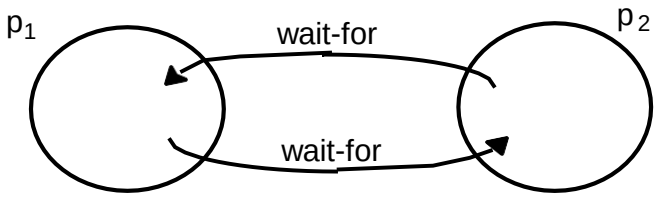
- Compare vector timestamps
  - V=V' iff V[j] = V'[j] for j=1..N
  - V<=V' iff V[j] <= V'[j] for j=1..N
  - V<V' iff V<= V' ^ V!=V'
- Figure 2.7 shows
  - *a*→*f* since V(a) < V(f)
  - c || e since neither V(c) <= V(e) nor V(e) <= V(c)

1.5 Global states

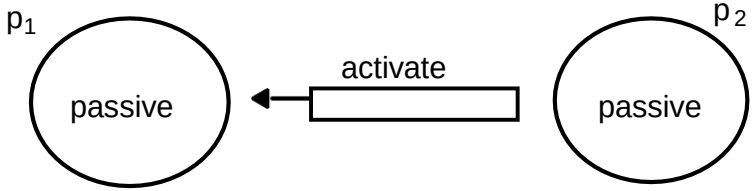
- How do we find out if a particular property is true in a distributed system? For examples, we will look at:
  - Distributed Garbage Collection
  - Deadlock Detection
  - Termination Detection
  - Debugging



a. Garbage collection



b. Deadlock



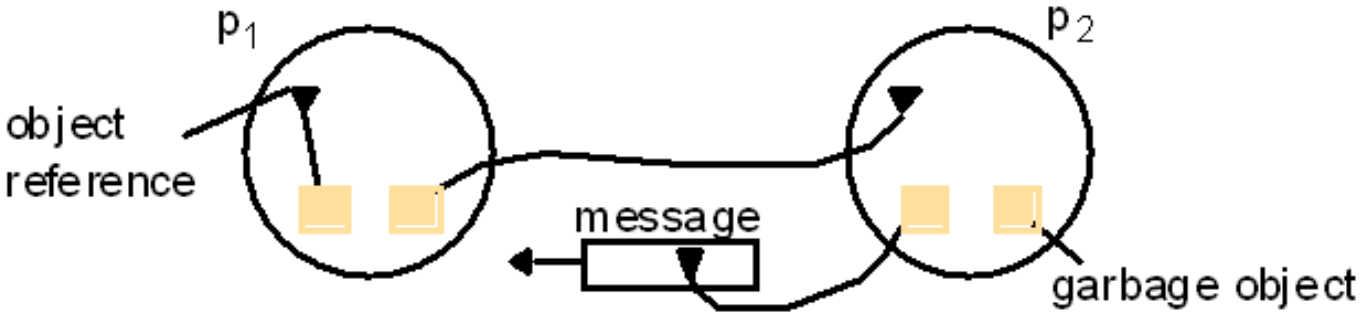
c. Termination

Figure 1.8
Detecting
global
properties

Distributed Garbage Collection

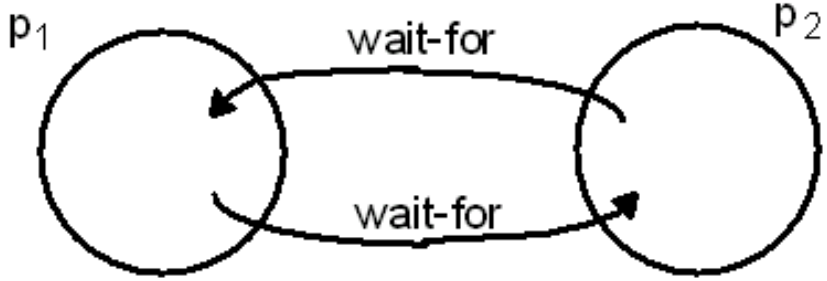
- Objects are identified as ***garbage*** when there are no longer any references to them in the system
- Garbage collection reclaims memory used by those objects
- In figure 1.8a, process p_2 has two objects that do not have any references to other objects, but one object does have a reference to a message in transit. It is not garbage, but the other p_2 object is
- Thus we must consider communication channels as well as

object references to determine unreferenced objects



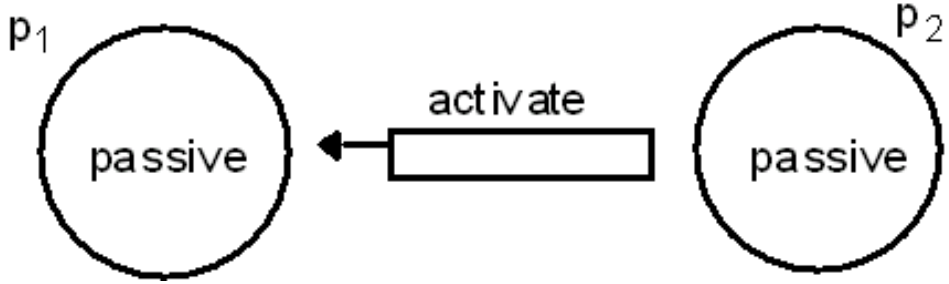
Deadlock Detection

- A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and there is a cycle in the graph of the ***waits-for*** relationship
- In figure 1.8b, both p_1 and p_2 wait for a message from the other, so both are blocked and the system cannot continue



Termination Detection

- It is difficult to tell whether a distributed algorithm has terminated. It is not enough to detect whether each process has halted
  - In figure 1.8c, both processes are in passive mode, but there is an activation request in the network
  - Termination detection examines multiple states like deadlock detection, except that a deadlock may affect only a portion of the processes involved, while *termination detection must ensure that all of the processes have completed*



Distributed Debugging

- Distributed processes are complex to debug. One of many possible problems is that consistency restraints must be evaluated for simultaneous attribute values in multiple processes at different instants of time (Section 1.6)

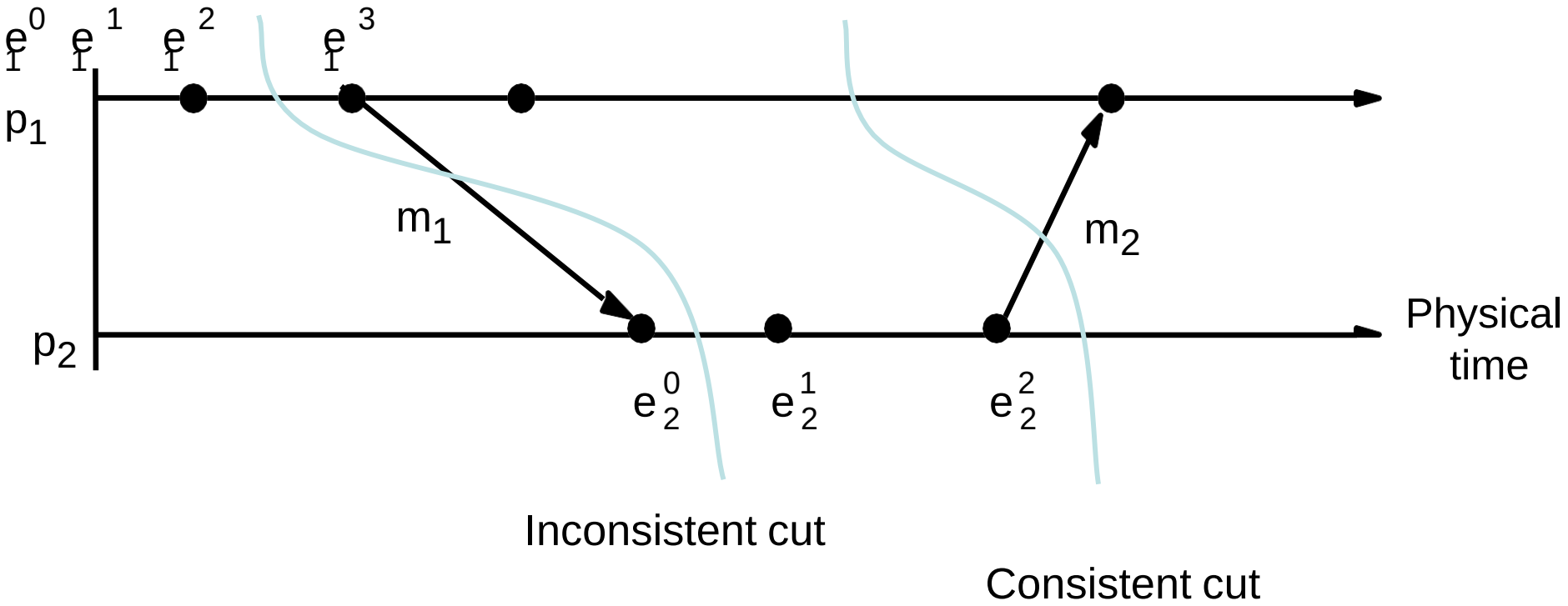
- All four of the distributed problems discussed in this section have particular solutions, but all of them also illustrate the need to observe global states. We will now look at a general approach to observing global states

1.5.1 Global states and consistent cuts

- Without global time identified by perfectly synchronized clocks, the ability to identify successive states in an individual process does not translate into the ability to identify successive states in distributed processes
- We can assemble meaningful global states from local states recorded at different local times in many circumstances, but must do so carefully and recognize limits to our capabilities
- A general system P of N processes p_i (i=1..N)
  - p_i's history: history(p_i)=h_i=<e_i^0 , e_i^1 , e_i^2 ,...>
  - finite prefix of p_i's history: h_i^k= <e_i^0, e_i^1, e_i^2 , …, e_i^k >
  - state of p_i immediately before the kth event occurs: s_i^k
  - global history H=h_1 U h_2 U…U h_N
  - A cut of the system's execution is a subset of its global history that is a union of prefix of process histories C=h_1^{c1} U h_2^{c2} U…U h_N^{cN}
  - A cut C is consist if, for each event it contains, it also contains all the events that happened-before that event: For all events $e \in C$, $f \rightarrow e \Rightarrow f \in C$
  - A consistent global state is one that corresponds to a consistent cut

Figure 1.9 Cuts

- Figure 2.9 gives an example of an inconsistent cut_{ic} and a consistent cut_{cc}. The distinguishing characteristic is that
  - cut_{ic} includes the receipt of message m_1 but *not* the sending of it, while
  - cut_{cc} includes the sending *and* receiving of m_1 *and* cuts between the sending and receipt of the message m_2.
- A consistent cut cannot violate temporal causality by implying that a result occurred before its cause, as in message m_1 being received before the cut and being sent after the cut.



1.5.2 Global state predicates

- A Global State Predicate is a function that maps from the set of global process states to True or False.
- Detecting a condition like deadlock or termination requires evaluating a Global State Predicate.
- A Global State Predicate is stable: once a system enters a state where it is true, such as deadlock or termination, it remains true in all future states reachable from that state. However, when we monitor or debug an application, we are interested in non stable predicates.

1.5.3 The Snapshot Algorithm

- Chandy and Lamport defined a snapshot algorithm to determine global states of distributed systems
- The goal of a snapshot is to record a set of process and channel states (a snapshot) for a set of processes so that, even if the combination of recorded states may not have occurred at the same time, the recorded global state is consistent
  - The algorithm records states locally; it does not gather global states at one site.
- The snapshot algorithm has some assumptions
  - Neither channels nor processes fail
  - Reliable communications ensure every message sent is received exactly once
  - Channels are unidirectional
  - Messages are received in FIFO order
  - There is a path between any two processes
  - Any process may initiate a global snapshot at any time
  - Processes may continue to function normally during a snapshot

Snapshot Algorithm

- For each process, incoming channels are those which other processes can use to send it messages. Outgoing channels are those it uses to send messages. Each process records its state and for each incoming channel a set of messages sent to it. The process records for each channel, any messages sent after it recorded its state and before the sender recorded its own state. This approach can differentiate between states in terms of messages transmitted but not yet received
- The algorithm uses special marker messages, separate from other messages, which prompt the receiver to save its own state if it has not done so and which can be used to determine which messages to include in the channel state.

  •The algorithm is determined by two rules
  - The marker receiving rule
  - The marker sending rule
- Figure 14.10 shows the algorithm

Figure 1.10 Chandy and Lamport's 'snapshot' algorithm

Marker receiving rule for process p_i

On p_i's receipt of a *marker* message over channel *c*: *if* (p_i has not yet recorded its state) it

records its process state now;

records the state of *c* as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of *c* as the set of messages it has received over *c* since it saved its state.

end if

Marker sending rule for process p_i

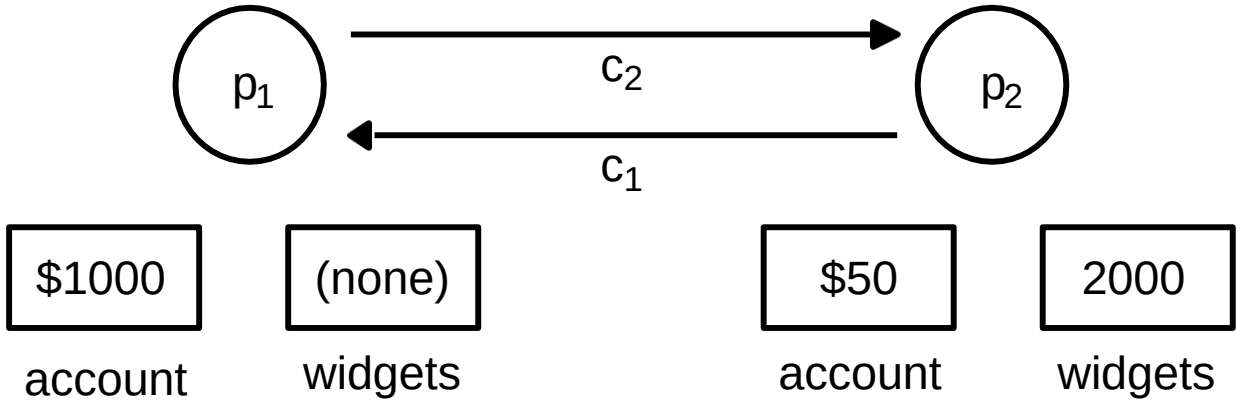
After p_i has recorded its state, for each outgoing channel *c*: p_i sends one marker message over *c*

(before it sends any other message over *c*).

Example

- Figure 1.11 shows an initial state for two processes.
- Figure 1.12 shows four successive states reached and identified after state transitions by the two processes.
- Termination: it is assumed that all processes will have recorded their states and channel states a finite time after some process initially records its state.

Figure 1.11 Two processes and their initial states



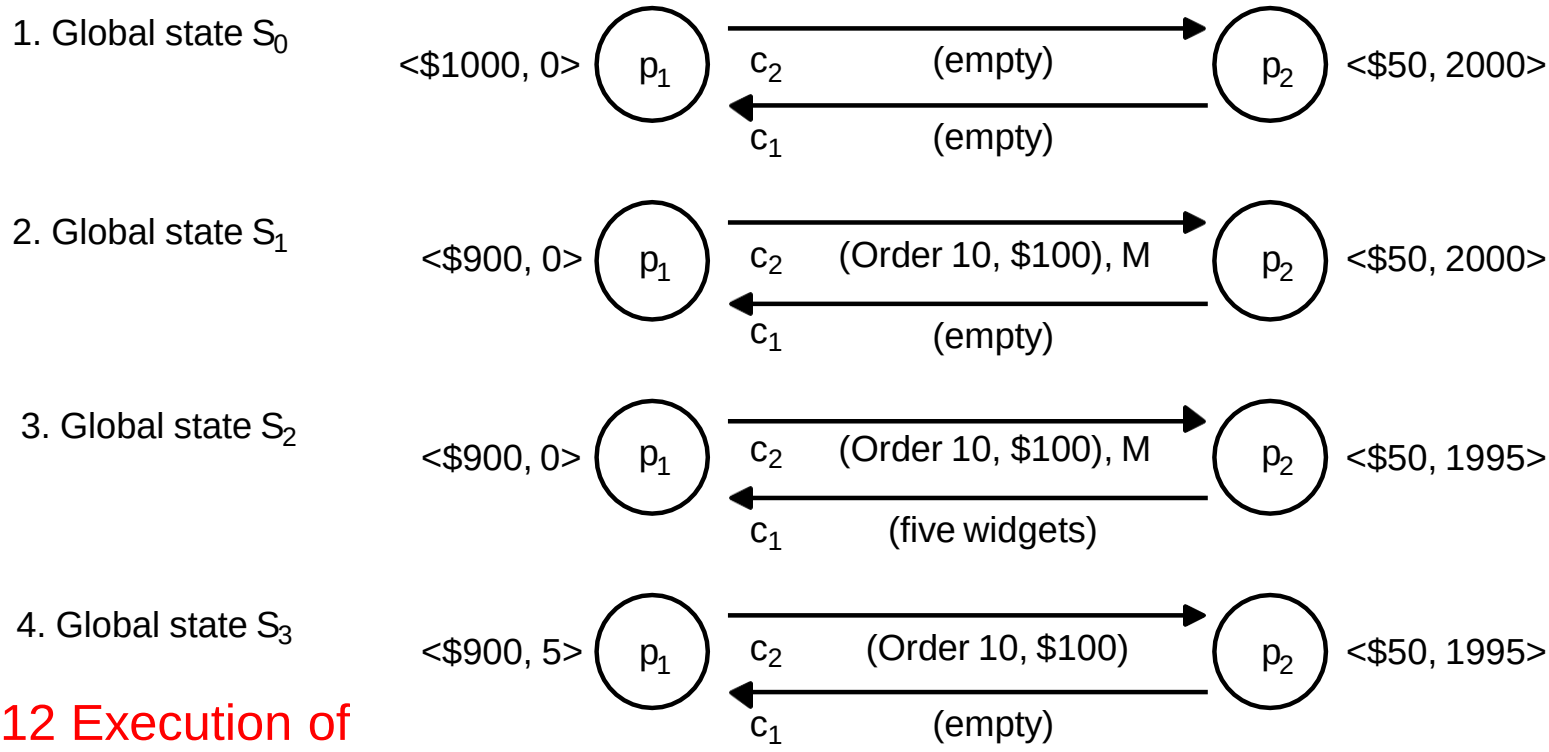


Figure 1.12 Execution of processes in Figure 2.11

(M = marker message)

Characterizing a state

- A snapshot selects a consistent cut from the history of the execution. Therefore the state recorded is consistent. This can be used in an ordering to include or exclude states that have or have not recorded their state before the cut. This allows us to distinguish events as pre-snap or post-snap events.
- The reachability of a state (figure 1.13) can be used to determine stable predicates.

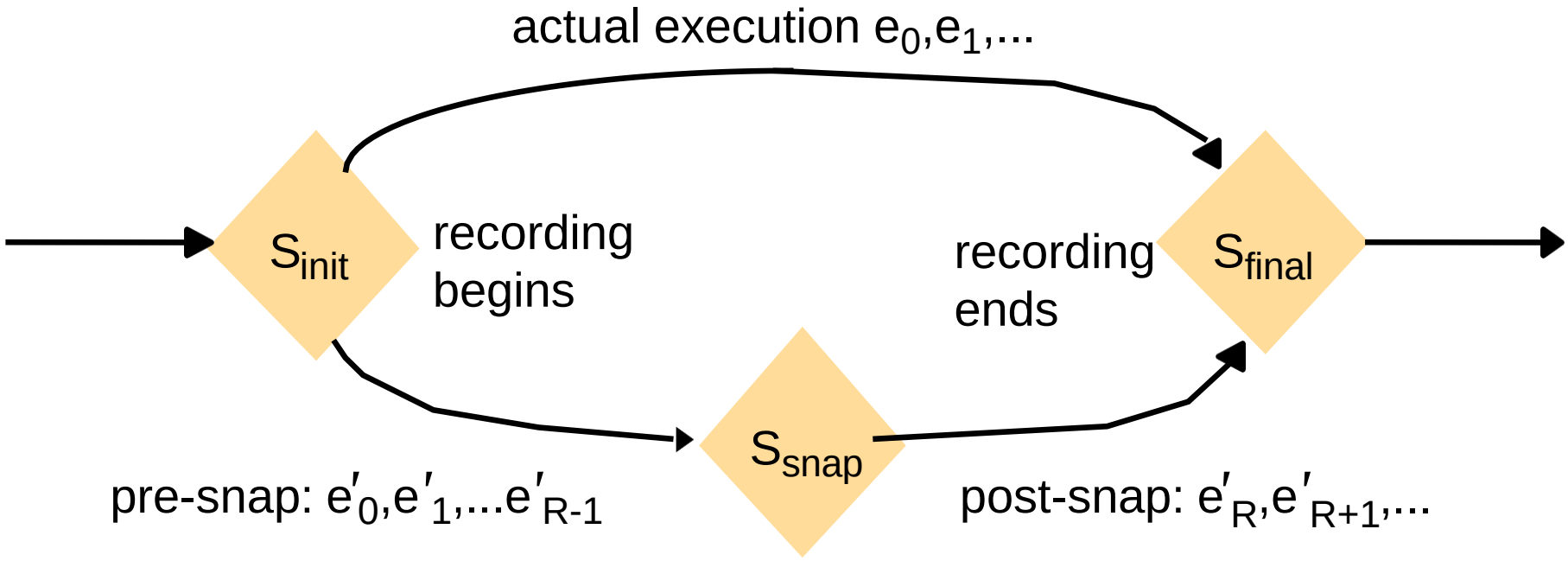


Figure 2.13 Reachability between states in the snapshot algorithm

Chapter2
Coordination and Agreement

Chapter 2 Coordination and Agreement

1. Introduction
2. Distributed mutual exclusion
   - A central server algorithm
   - A ring-based algorithm
   - An algorithm using multicast and logic clocks
   - Maekawa's voting algorithm
3. Elections
   - A ring-based election algorithm
   - The bully election algorithm
4. Multicast communication
   - Basic multicast
   - Reliable multicast: total, FIFO and causal ordering
5. Consensus and related problems (agreement)
   - Consensus, Byzantine Generals and interactive consensus
6. Summary

2.1 Introduction

- Fundamental issue: for a set of processes, how to coordinate their actions or to agree on one or more values?
  - even no fixed master-slave relationship between the components
- Further issue: how to consider and deal with failures when designing algorithms
- Topics covered
  - mutual exclusion
  - how to elect one of a collection of processes to perform a special role
  - multicast communication
  - agreement problem: consensus and byzantine agreement

Failure Assumptions and Failure Detectors

- Failure assumptions of this chapter
  - Reliable communication channels
  - Processes only fail by crashing unless state otherwise
- Failure detector: object/code in a process that detects failures of other processes
- unreliable failure detector
  - One of two values: unsuspected or suspected
    - Evidence of possible failures
  - Example: most practical systems
    - Each process sends “alive/I’m here” message to everyone else
    - If not receiving “alive” message after timeout, it’s suspected
      - maybe function correctly, but network partitioned
- reliable failure detector
  - One of two accurate values: unsuspected or failure
  - few practical systems

2.2 Distributed Mutual Exclusion

- Process coordination in a multitasking OS
  - **Race condition**: several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place
  - **critical section**: when one process is executing in a critical section, no other process is to be allowed to execute in its critical section
  - **Mutual exclusion**: If a process is executing in its critical section, then no other processes can be executing in their critical sections
- Distributed mutual exclusion
  - Provide critical region in a distributed environment
  - message passing

for example, locking files, lockd daemon in UNIX (NFS is stateless, no file-locking at the NFS level)

Algorithms for distributed mutual exclusion

- Problem: an asynchronous system of *N* processes
  - processes don't fail
  - message delivery is reliable; not share variables
  - only one critical region
  - application-level protocol: enter(), resourceAccesses(), exit()
- Requirements for mutual exclusion
  - Essential
    - [ME1] safety: only one process at a time
    - [ME2] liveness: eventually enter or exit
  - Additional
    - [ME3] happened-before ordering: ordering of enter() is the same as HB ordering
- Performance evaluation
  - overhead and bandwidth consumption: # of messages sent
  - client delay incurred by a process at entry and exit
  - throughput measured by synchronization delay: delay between one's exit and next's entry

A central server algorithm

- server keeps track of a token---permission to enter critical region
  - a process requests the server for the token
  - the server grants the token if it has the token
  - a process can enter if it gets the token, otherwise waits
  - when done, a process sends release and exits

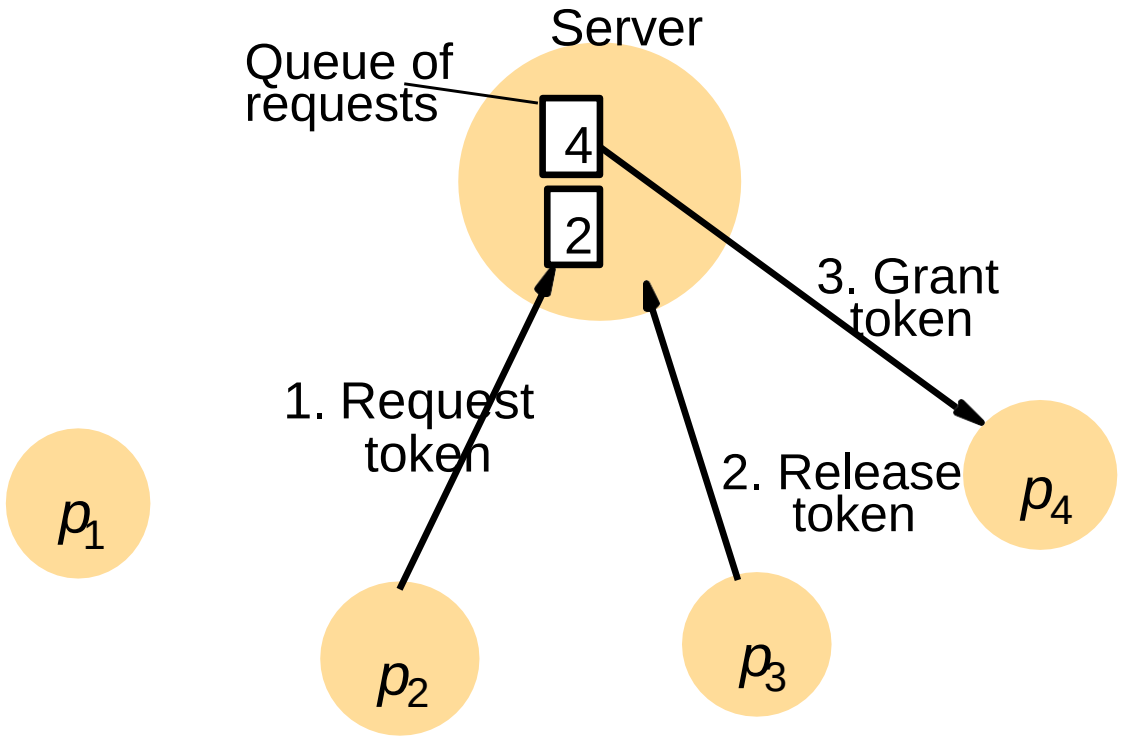


Figure 2.1 Server managing a mutual exclusion token for a set of processes

A central server algorithm: discussion

- Properties
  - safety, why?
  - liveness, why?
  - HB ordering not guaranteed, why?
- Performance
  - enter overhead: two messages (request and grant)
  - enter delay: time between request and grant
  - exit overhead: one message (release)
  - exit delay: none
  - synchronization delay: between release and grant
  - centralized server is the bottleneck

A ring-based algorithm

- Arrange processes in a logical ring to rotate a token
  - Wait for the token if it requires to enter the critical section
  - The ring could be unrelated to the physical configuration
- p_i sends messages to $p_{(i+1) \bmod N}$
  - when a process requires to enter the critical section, waits for the token
  - when a process holds the token
    - If it requires to enter the critical section, it can enter
      - when a process releases a token (exit), it sends to its neighbor
    - If it doesn't, just immediately forwards the token to its neighbor

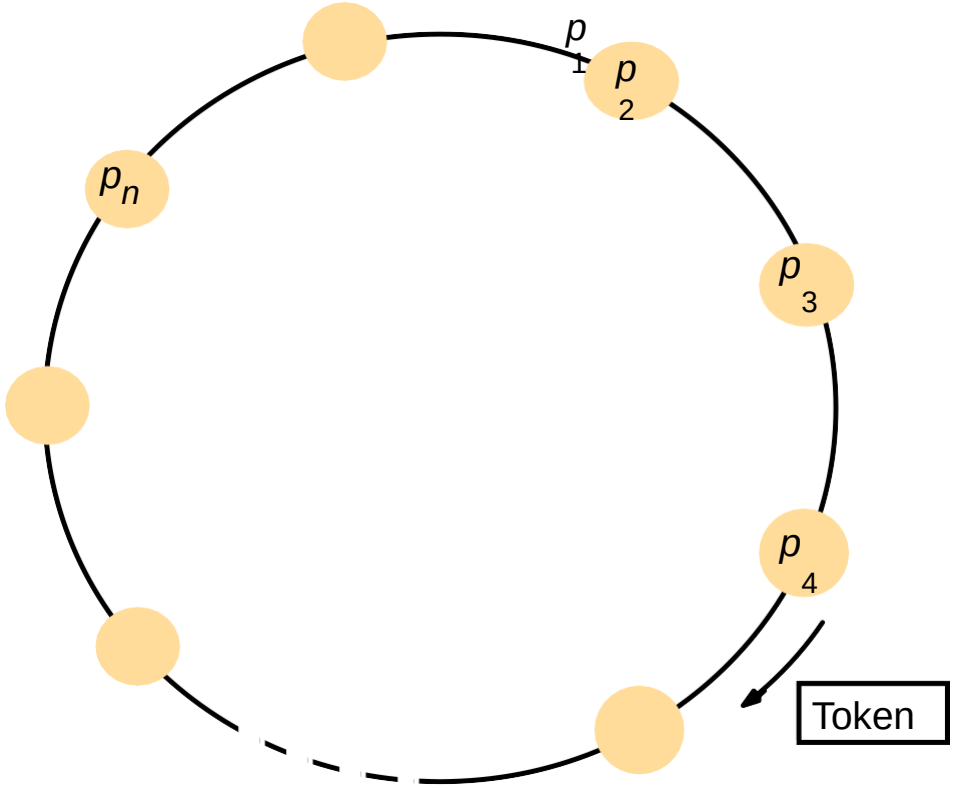


Figure 2.3 A ring of processes transferring a mutual exclusion token

A ring-based algorithm: discussion

- Properties
  - safety, why?
  - liveness, why?
  - HB ordering not guaranteed, why?
- Performance
  - bandwidth consumption: token keeps circulating
  - enter overhead: 0 to *N* messages
  - enter delay: delay for 0 to *N* messages
  - exit overhead: one message
  - exit delay: none
  - synchronization delay: delay for 1 to *N* messages

An algorithm using multicast and logical clocks

- Multicast a request message for the token (Ricart and Agrawala [1981])
  - enter only if all the other processes reply
  - totally-ordered timestamps: <T, p_i >
- Each process keeps a *state: RELEASED, HELD, WANTED*
  - if all have *state* = *RELEASED*, all reply, a process can hold the token and enter
  - if a process has *state* = *HELD*, doesn't reply until it exits
  - if more than one process has *state* = *WANTED*, process with the lowest timestamp will get all *N*-1 replies first

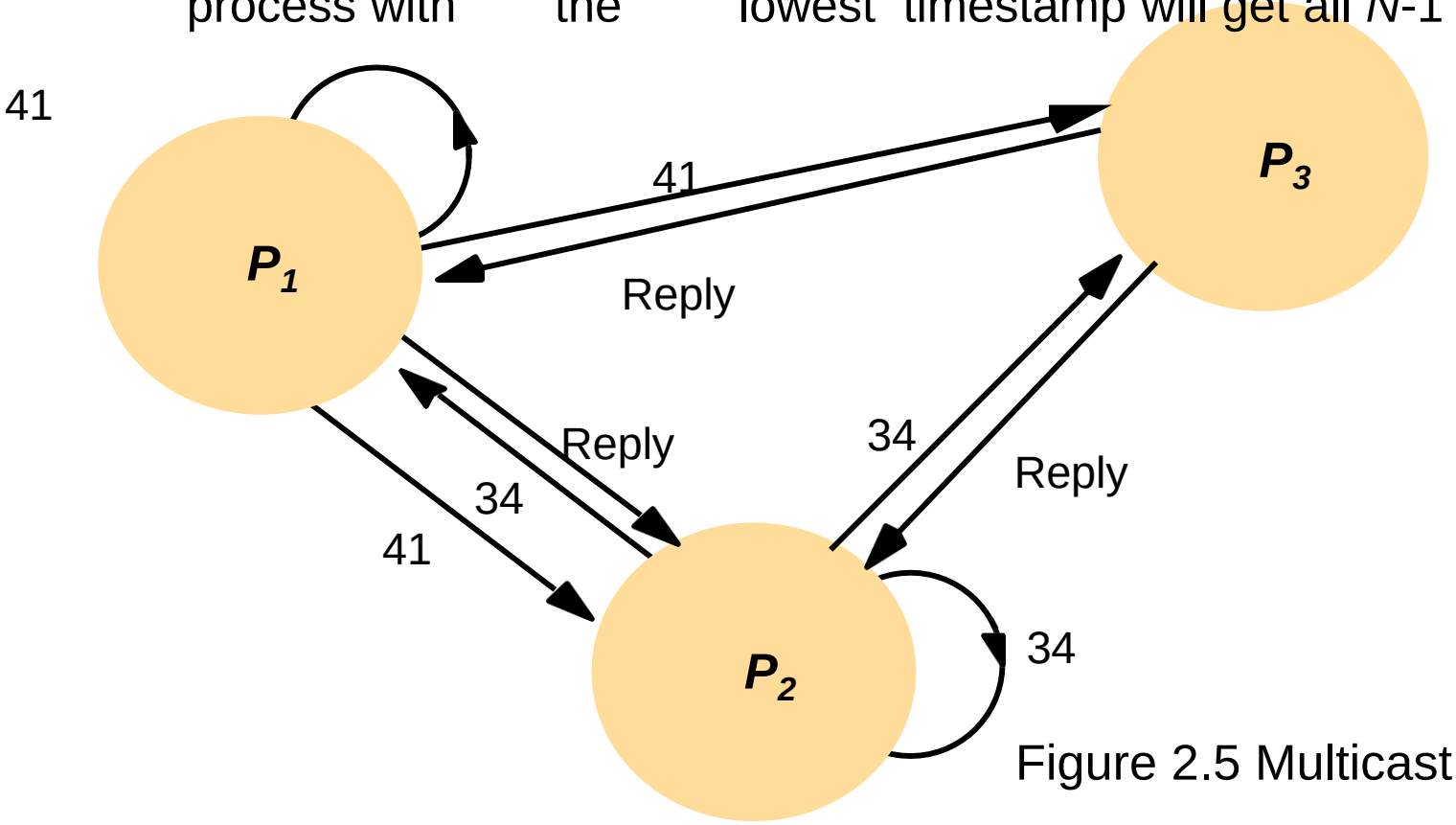


Figure 2.5 Multicast synchronization

Figure 2.4 Ricart and Agrawala's algorithm

On initialization
state := RELEASED;
To enter the section *state* := WANTED;
Multicast *request* to all processes; } request processing deferred here
T := request's timestamp;
Wait until (number of replies received = (*N* – 1));
state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$)) *then*
queue *request* from p_i without replying;
else
reply immediately to p_i; end if
To exit the critical section
state := RELEASED;
reply to any queued requests;

An algorithm using multicast: discussion

- Properties
  - safety, why?
  - liveness, why?
  - HB ordering, why?
- Performance
  - bandwidth consumption: no token keeps circulating
  - entry overhead: $2(N-1)$, why? [with multicast support: $1 + (N - 1) = N$]
  - entry delay: delay between request and getting all replies
  - exit overhead: 0 to N-1 messages
  - exit delay: none
  - synchronization delay: delay for 1 message (one last reply from the previous holder)

Maekawa's voting algorithm

- Observation: not all peers to grant it access
  - Only obtain permission from subsets, overlapped by any two processes
- Maekawa's approach
  - subsets Vi,Vj for process Pi, Pj
    - Pi $\in$Vi, Pj $\in$ Vj
    - Vi $\cap$ Vj $\neq \varnothing$, there is at least one common member
    - subset |Vi|=K, to be fair, each process should have the same size
  - Pi cannot enter the critical section until it has received all K reply messages
  - Choose a subset
    - Simple way (2$\sqrt{N}$): place processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix and let Vi be the union of the row and column containing Pi
    - Optimal ($\sqrt{N}$): non-trivial to calculate (skim here)
  - Deadlock-prone
    - V1={P1, P2}, V2={P2, P3}, V3={P3, P1}
    - If P1, P2 and P3 concurrently request entry to the critical section, then its possible that each process has received one (itself) out of two replies, and none can proceed

    - adapted and solved by [Saunders 1987]

Figure 2.6 Maekawa's algorithm

On initialization
state := RELEASED;
voted := FALSE;

For p_i *to enter the critical section* *state* := WANTED;
Multicast *request* to all processes in V_i; *Wait until* (number of replies received = K); *state* := HELD;

On receipt of a request from p_i *at* p_j
if (*state* = HELD *or voted* = TRUE)
then
queue *request* from p_i without replying;
else
send *reply* to p_i; *voted* := TRUE;
end if

For p_i *to exit the critical section* *state* := RELEASED;
Multicast *release* to all processes in V_i;

On receipt of a release from p_i *at* p_j *if* (queue of requests is non-empty) *then*
remove head of queue – from p_k, say; send *reply* to p_k;
voted := TRUE;
else
voted := FALSE;
end if

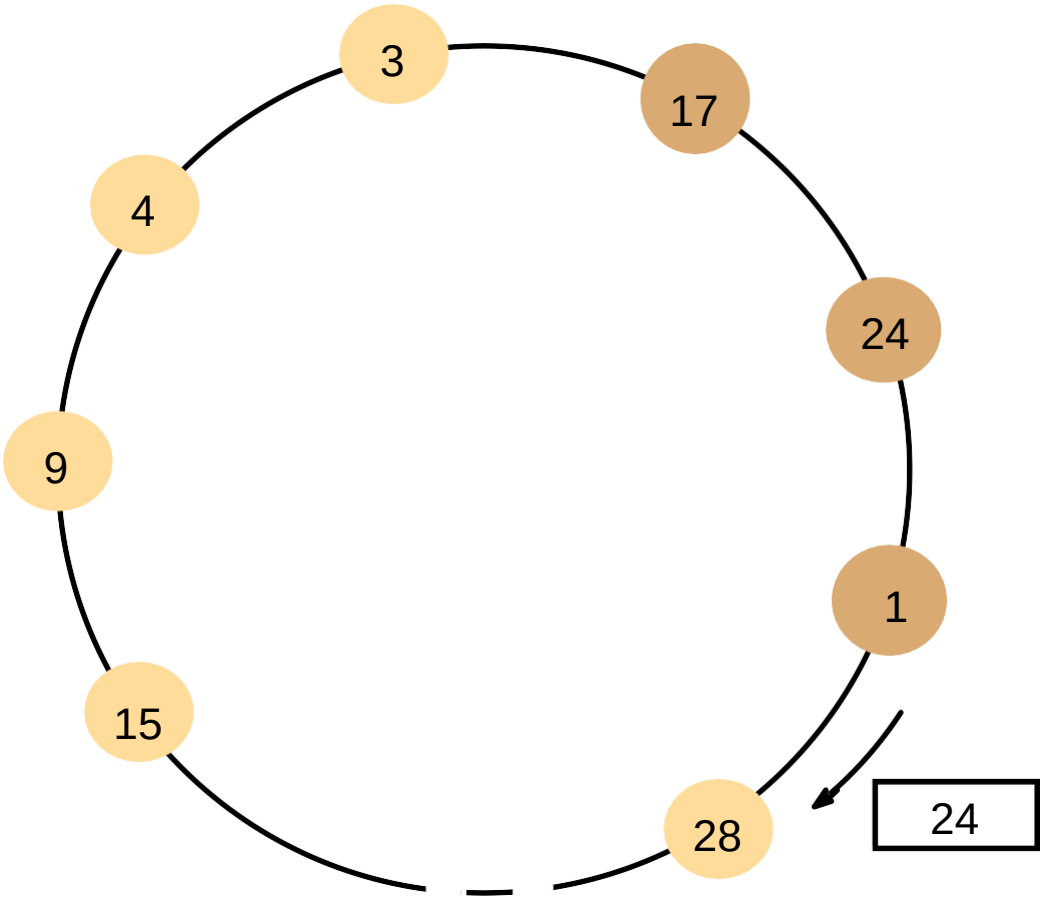
2.3 Elections

- Election: choosing a unique process for a particular role
  - All the processes agree on the ***unique*** choice
  - For example, server in dist. mutex
- Assumptions
  - Each process can call only one election at a time
  - multiple concurrent elections can be called by different processes
  - Participant: engages in an election
    - each process p_i has variable $elected_i$ = ? ⊥ (don't know) initially
    - process with the *largest* identifier wins
  - The (unique) identifier could be any useful value
- Properties
  - [E1] $elected_i$ of a "participant" process must be P (elected process=largest id) or ⊥ (undefined)
  - [E2] liveness: all processes participate and eventually set $elected_i$!= ⊥ (or crash)
- Performance
  - overhead (bandwidth consumption): # of messages
  - turnaround time: # of messages to complete an election

A ring-based election algorithm

- Arrange processes in a logical ring
  - p_i sends messages to $p_{(i+1) \bmod N}$
  - It could be unrelated to the physical configuration
  - Elect the coordinator with the largest id
  - Assume no failures
- Initially, every process is a non-participant. Any process can call an election
  - Marks itself as participant
  - Places its id in an *election* message
  - Sends the message to its neighbor
  - Receiving an election message
    - if *id* > *myid*, forward the msg, mark participant
    - if *id* < *myid*
      - non-participant: replace *id* with *myid*: forward the msg, mark participant
      - participant: stop forwarding (why? Later, multiple elections)
    - if *id* = *myid*, coordinator found, mark non-participant, $elected_i$:= *id*, send *elected* message with *myid*
  - Receiving an elected message
    - *id* != *myid,* mark non-participant, $elected_i$:= *id* forward the msg
    - if *id* = *myid*, stop forwarding

Figure 2.7 A ring-based election in progress



- Receiving an election message:
  - if *id* > *myid*, forward the msg, mark participant
  - if *id* < *myid*
    - non-participant: replace *id* with *myid*: forward the msg, mark participant
    - participant: stop forwarding (why? Later, multiple elections)
  - if *id* = *myid*, coordinator found, mark non-participant, $elected_i$:= *id*, send *elected* message with *myid*
- Receiving an elected message:
  - *id* != *myid,* mark non-participant, $elected_i$:= *id* forward the msg
  - if *id* = *myid*, stop forwarding

Note: The election was started by process 17.
The highest process identifier encountered so far is 24.

A ring-based election algorithm: discussion

- Properties
  - safety: only the process with the largest id can send an *elected* message
  - liveness: every process in the ring eventually participates in the election; extra elections are stopped
- Performance
  - one election, best case, when?
    - *N election* messages
    - *N elected* messages
    - turnaround: 2*N* messages
  - one election, worst case, when?
    - 2*N* - 1 election messages
    - *N elected* messages
    - turnaround: 3*N* - 1 messages
  - can't tolerate failures, not very practical

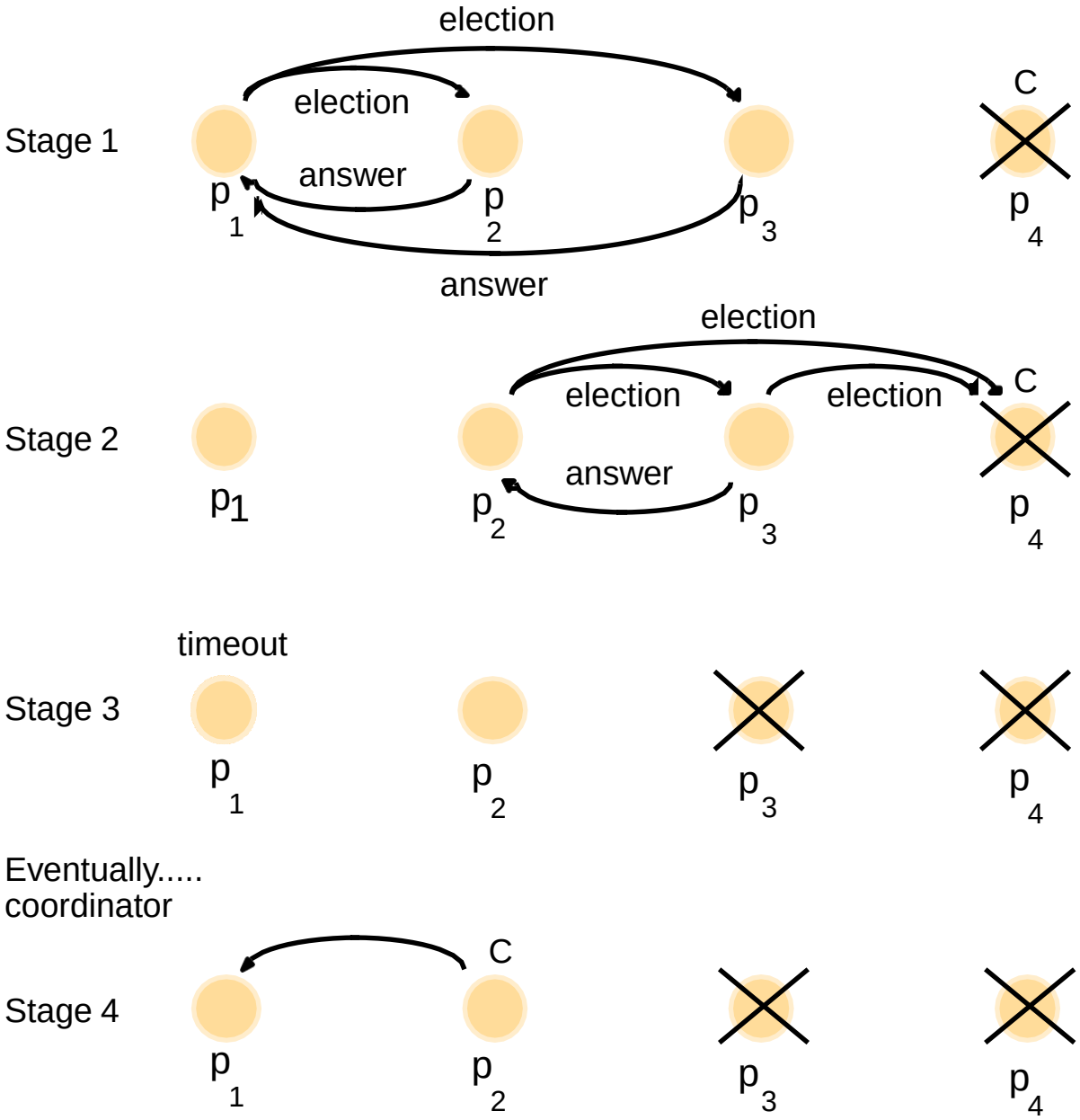
The bully election algorithm

- Assumption
  - Each process knows which processes have higher identifiers, and that it can communicate with all such processes
- Compare with ring-based election
  - Processes can crash and be detected by timeouts
    - synchronous: timeout $T = 2T_{transmitting}$ (max transmission delay) + $T_{processing}$ (max processing delay)
- Three types of messages
  - Election: announce an election
  - Answer: in response to Election
  - Coordinator: announce the identity of the elected process

The bully election algorithm: howto

- Start an election when detect the coordinator has failed or begin to replace the coordinator, which has lower identifier
  - Send an election message to all processes with higher id's and waits for answers (except the failed coordinator/process)
    - If no answers in time *T*
      - Considers it is the coordinator
      - sends coordinator message (with its id) to all processes with lower id's
    - else
      - waits for a coordinator message and starts an election if T' timeout
  - To be a coordinator, it has to start an election
    - A higher id process can replace the current coordinator (hence "bully")
      - The highest one directly sends a coordinator message to all process with lower identifiers
- Receiving an election message
  - sends an answer message back
  - starts an election if it hasn't started one—send election messages to all higher-id processes (including the "failed" coordinator—the coordinator might be up by now)
- Receiving a coordinator message
  - set $elected_i$ to the new coordinator

Figure 2.8 The bully algorithm



The election of coordinator p_2, after the failure of p_4 and then p_3

The bully election algorithm: discussion

- Properties
  - safety:
    - a lower-id process always yields to a higher-id process
    - However, it's guaranteed
      - if processes that have crashed are replaced by processes with the same identifier since message delivery order might not be guaranteed and
      - failure detection might be unreliable
  - liveness: all processes participate and know the coordinator at the end
- Performance
  - best case: when?
    - overhead: *N-2 coordinator* messages
    - turnaround delay: no *election/answer* messages
  - worst case: when?
    - overhead:
    - 1+ 2 + ...+ (*N*-2) + (*N*-2)= (*N*-1)(*N*-2)/2 + (*N*-2) *election* messages,
    - 1+...+ (*N*-2) *answer* messages,
    - *N*-2 *coordinator* messages,
    - total: (*N*-1)(*N*-2) + 2(*N*-2) = (*N*+1)(*N*-2) = O(N^2)
  - turnaround delay: delay of election and answer messages

2.4 Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
  - The set of messages that every process of the group should receive
  - On the delivery ordering across the group members
- Challenges
  - Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
  - Delivery guarantees ensure that operations are completed
- Types of group
  - Static or dynamic: whether joining or leaving is considered
  - Closed or open
    - A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
    - A group is open if processes outside the group can send to it

Reliable Multicast

- Simple basic multicasting (B-multicast) is sending a message to every process that is a member of a defined group
  - B-multicast(g, m) for each process p $\in$ group g, send(p, message m)
  - On receive(m) at p: B-deliver(m) at p
- Reliable multicasting (R-multicast) requires these properties
  - Integrity: a correct process sends a message to only a member of the group and does it only once
  - Validity: if a correct process sends a message, it will eventually be delivered
  - Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

Figure 2.10 Reliable multicast algorithm

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(*g*, *m*); // $p \in g$ is included as a destination

On B-deliver(*m*) *at process q with g = group*(*m*)

if ($m \notin$ *Received*)

then

Received := *Received* $\cup$ {*m*};

if ($q \neq p$) *then B-multicast*(*g*, *m*); *end if*

R-deliver m;

end if

Implementing reliable R-multicast over B-multicast

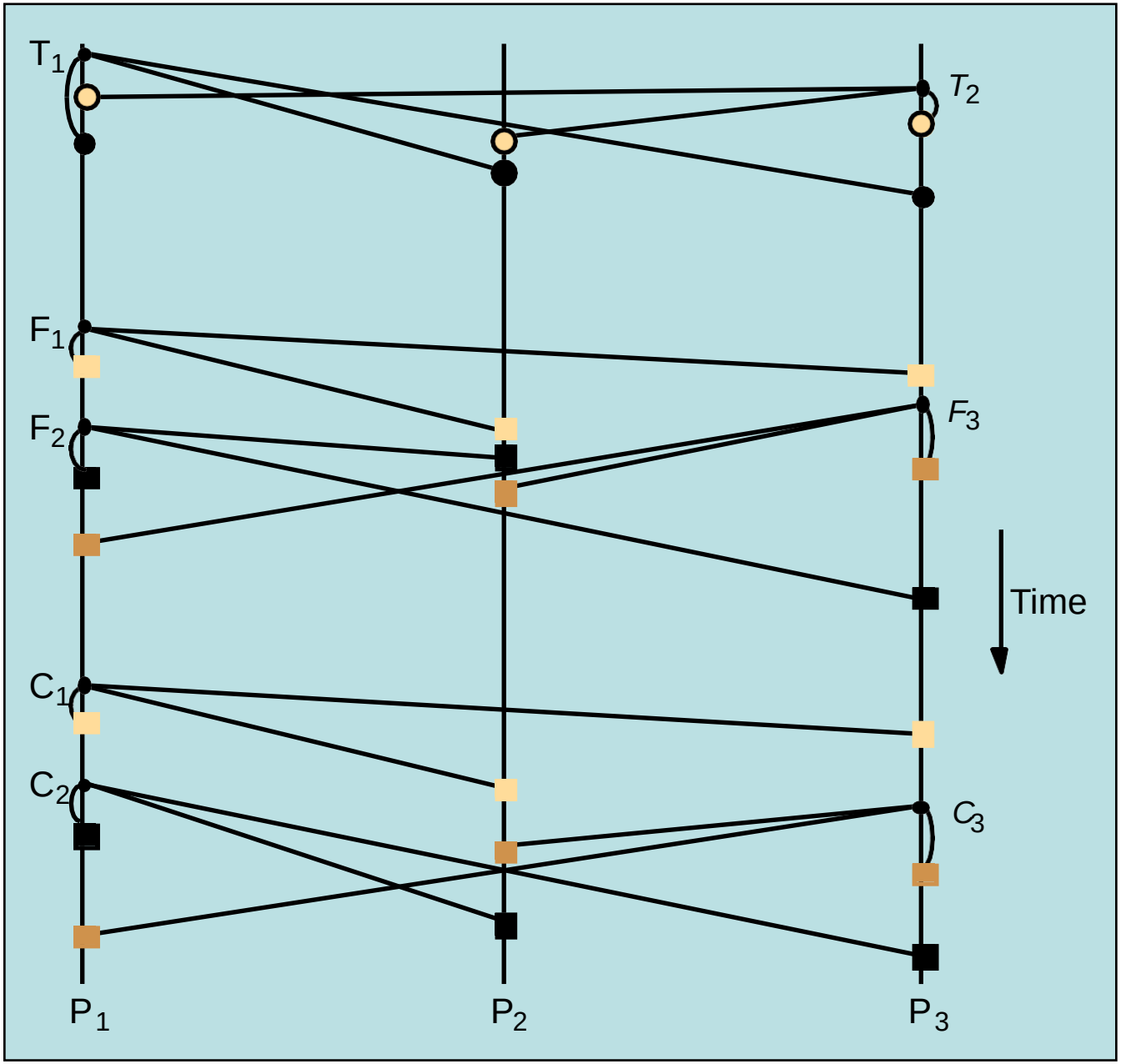
–When a message is delivered, the receiving process multicasts it

–Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering

- Three types of message ordering
  - ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message before another, every correct process will deliver the first message before the other
  - ***Casual ordering***: any correct process that delivers the second message will deliver the previous message first
  - ***Total ordering***: if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first
- Note that
  - FIFO ordering and casual ordering are only partial orders
  - Not all messages are sent by the same sending process
  - Some multicasts are concurrent, not able to be ordered by happened-before
  - Total order demands consistency, but not a particular order

Figure 2.12 Total, FIFO and causal ordering of multicast messages



Notice
- the consistent ordering of totally ordered messages T_1 and T_2,
- the FIFO-related messages F_1 and F_2 and
- the causally related messages C_1 and C_3

– and

- the otherwise arbitrary delivery ordering of messages

Note that T_1 and T_2 are delivered in opposite order to the physical time of message creation

Bulletin board example (FIFO ordering)

- A bulletin board such as Web Board at NJIT illustrates the desirability of consistency and FIFO ordering. A user can best refer to preceding messages if they are delivered in order. Message 25 in Figure 2.13 refers to message 24, and message 27 refers to message 23.
- Note the further advantage that Web Board allows by permitting messages to begin threads by replying to a particular message. Thus messages do not have to be displayed in the same order they are delivered

Bulletin board: *os.interesting*

Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 2.13 Display from bulletin board program

Implementing total ordering

- The normal approach to total ordering is to assign totally ordered identifiers to multicast messages, using the identifiers to make ordering decisions.
- One possible implementation is to use a sequencer process to assign identifiers. See Figure 2.14. A drawback of this is that the sequencer can become a bottleneck.
- An alternative is to have the processes collectively agree on identifiers. A simple algorithm is shown in Figure 2.15.

Figure 2.14 Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
    B-multicast($g \cup \{sequencer(g)\}$, <*m, i*>);

On B-deliver(<*m, i*>) *with* $g = group(m)$
    Place <*m, i*> in hold-back queue;

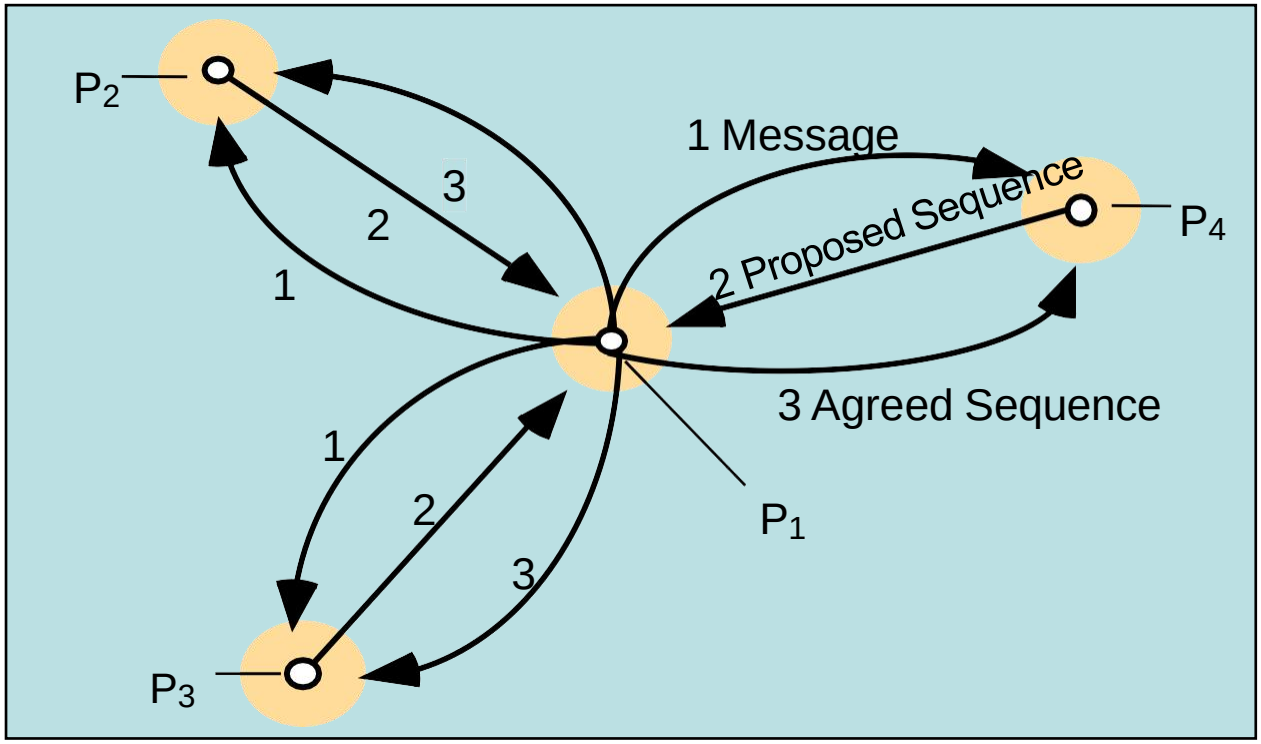
On B-deliver(m_{order} = <"order", *i, S*>) *with* $g = group(m_{order})$
    wait until <*m, i*> in hold-back queue and $S = r_g$;
    TO-deliver m; // (after deleting it from the hold-back queue)
    $r_g = S + 1$;

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver(<*m, i*>) *with* $g = group(m)$
    B-multicast(*g*, <"order", *i*, s_g>);
    $s_g := s_g + 1$;

Figure 2.15 The ISIS algorithm for total ordering



Each process q in group g keeps

- A^q_g: the largest agreed sequence number it has observed so far for the group g
- P^q_g: its own largest proposed sequence number

Algorithm for process p to multicast a message m to group g

1.p B-multicasts <m, i> to g, where i is a unique identifier for m

2.Each process q replies to the sender p with a proposal for the message's agreed sequence number of $P^q_g := Max(A^q_g, P^q_g)+1$

3.p collects all the proposed sequence numbers and selects the largest one a as the next agreed sequence number. It then B-multicasts <i, a> to g.

4.Each process q in g sets $A^q_g := Max(A^q_g, a)$ and attaches a to the message identified by i

Implementing casual ordering

- Causal ordering using vector timestamps (Figure 2.16)
  - Only orders multicasts, and ignores one-to-one messages between processes
    - Each process updates its vector timestamp before delivering a message to maintain the count of precedent messages

Algorithm for group member p_i $(i = 1, 2\ldots, N)$

On initialization

$V_i^g[j] := 0$ $(j = 1, 2\ldots, N)$;

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

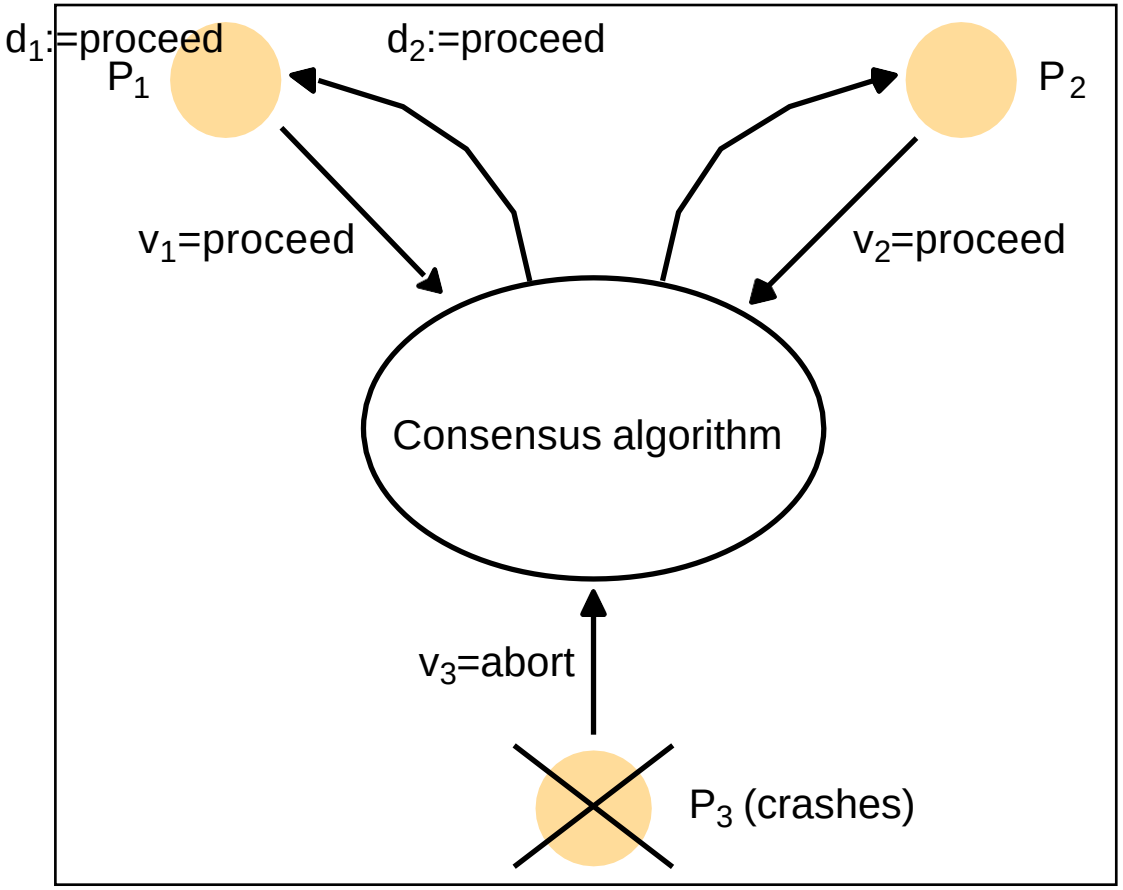
Figure 2.16 Causal ordering using vector timestamps

2.5 Consensus and related problems

- Problems of agreement
  - For processes to agree on a value (consensus) after one or more of the processes has proposed what that value should be
  - Covered topics: byzantine generals, interactive consistency, totally ordered multicast
    - The byzantine generals problem: a decision whether multiple armies should attack or retreat, assuming that united action will be more successful than some attacking and some retreating
    - Another example might be space ship controllers deciding whether to proceed or abort. Failure handling during consensus is a key concern
- Assumptions
  - communication (by message passing) is reliable
  - processes may fail
    - Sometimes up to f of the N processes are faulty

Consensus Process

1. Each process p_i begins in an undecided state and proposes a single value v_i, drawn from a set D (i=1…N)
2. Processes communicate with each other, exchanging values
3. Each process then sets the value of a decision variable d_i and enters the decided state



Two processes propose “proceed.” One proposes “abort,” but then crashes. The two remaining processes decide proceed.

Figure 2.17 Consensus for three processes

Requirements for Consensus

- Three requirements of a consensus algorithm
  - ***Termination***: Eventually every correct process sets its decision variable
  - ***Agreement***: The decision value of all correct processes is the same: if pi and pj are correct and have entered the *decided* state, then di=dj (i,j=1,2, …, N)
  - ***Integrity***: If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value

The byzantine generals problem

- Problem description
  - Three or more generals must agree to *attack* or to *retreat*
  - One general, the *commander*, issues the order
  - Other generals, the *lieutenants*, must decide to attack or retreat
  - One or more generals may be treacherous
    - A *treacherous general* tells one general to attack and another to retreat
- Difference from consensus is that a single process supplies the value to agree on
- Requirements
  - *Termination*: eventually each correct process sets its decision variable
  - *Agreement*: the decision variable of all correct processes is the same
  - *Integrity*: if the commander is correct, then all correct processes agree on the value that the commander has proposed (but the commander need not be correct)

The interactive consistency problem

- Interactive consistency: all correct processes agree on a vector of values, one for each process. This is called the decision vector
  - Another variant of consensus
- Requirements
  - *Termination*: eventually each correct process sets its decision variable
  - *Agreement*: the decision vector of all correct processes is the same
  - *Integrity*: if any process is correct, then all correct processes decide the correct value for that process

Relating consensus to other problems

- Consensus (C), Byzantine Generals (BG), and Interactive Consensus (IC) are all problems concerned with making decisions in the context of arbitrary or crash failures
- We can sometimes generate solutions for one problem in terms of another. For example
  - We can derive IC from BG by running BG N times, once for each process with that process acting as commander
  - We can derive C from IC by running IC to produce a vector of values at each process, then applying a function to the vector's values to derive a single value.
  - We can derive BG from C by
    - Commander sends proposed value to itself and each remaining process
    - All processes run C with received values
    - They derive BG from the vector of C values

Consensus in a Synchronous System

- Up to f processes may have crash failures, all failures occurring during f+1 rounds. During each round, each of the correct processes multicasts the values among themselves
- The algorithm guarantees all surviving correct processes are in a position to agree
- Note: any process with f failures will require at least f+1 rounds to agree

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \leq r \leq f + 1)$

B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f + 1)$ *rounds*

Assign d_i = *minimum*$(Values_i^{f+1})$;

Figure 2.18 Consensus in a synchronous system

Limits for solutions to Byzantine Generals

- Some cases of the Byzantine Generals problems have no solutions
  - Lamport *et al* found that if there are only 3 processes, there is no solution
  - Pease *et al* found that if the total number of processes is less than three times the number of failures plus one, there is no solution
- Thus there is a solution with 4 processes and 1 failure, if there are two rounds
  - In the first, the commander sends the values
  - while in the second, each lieutenant sends the values it received

Figure 2.19 Three Byzantine generals

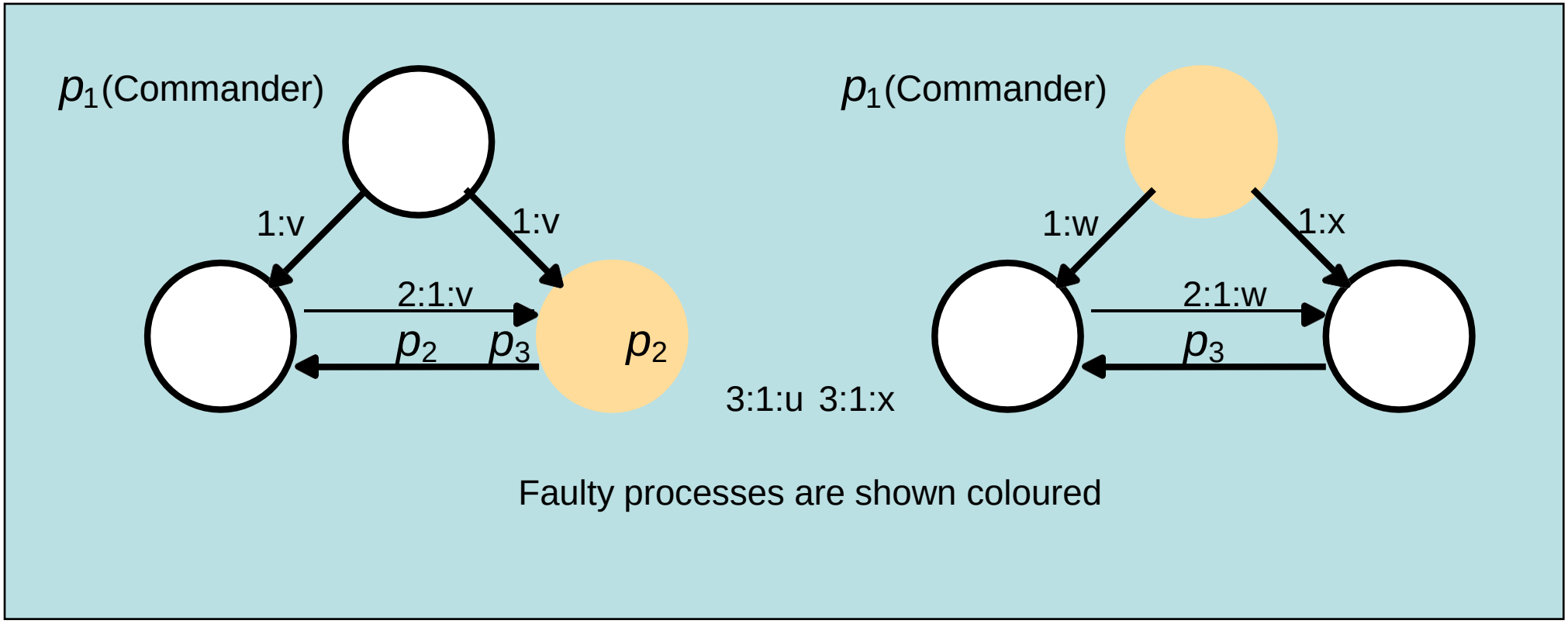
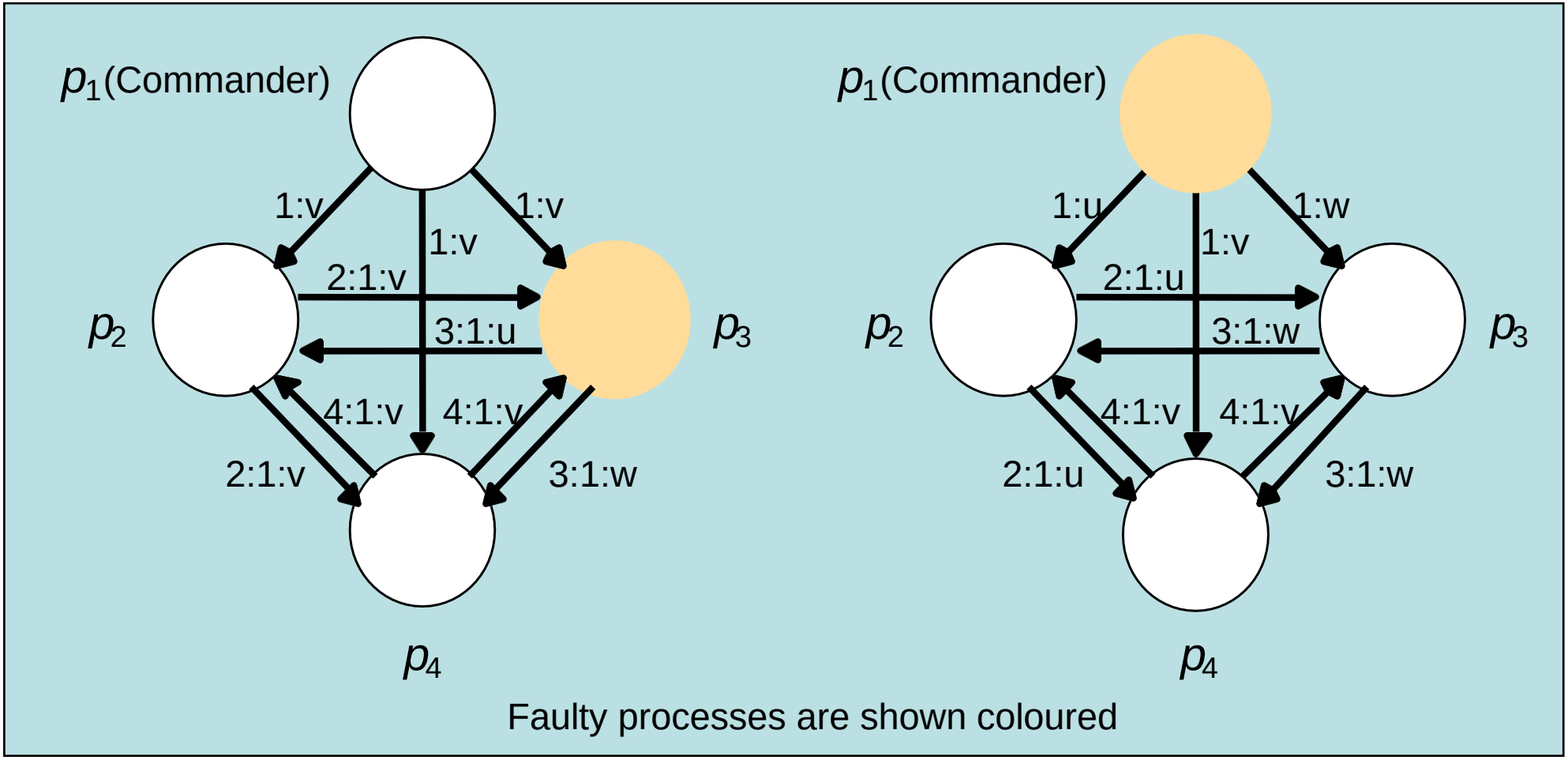


Figure 2.20 Four Byzantine generals



Asynchronous Systems

- All solutions to consistency and Byzantine generals problems are limited to synchronous systems
- Fischer *et al* found that there are no solutions in an asynchronous system with even one failure
- This impossibility is circumvented by *masking faults* or using *failure detection*
- There is also a partial solution, assuming an *adversary* process, based on *introducing random values* in the process to prevent an effective thwarting strategy. This does not always reach consensus

UNIT 3
Inter Process Communication

Chapter 1 Remote Invocation

1. Introduction
2. Request-reply protocols
3. Remote procedure call
4. Remote method invocation
5. Case study: Java RMI
6. Summary

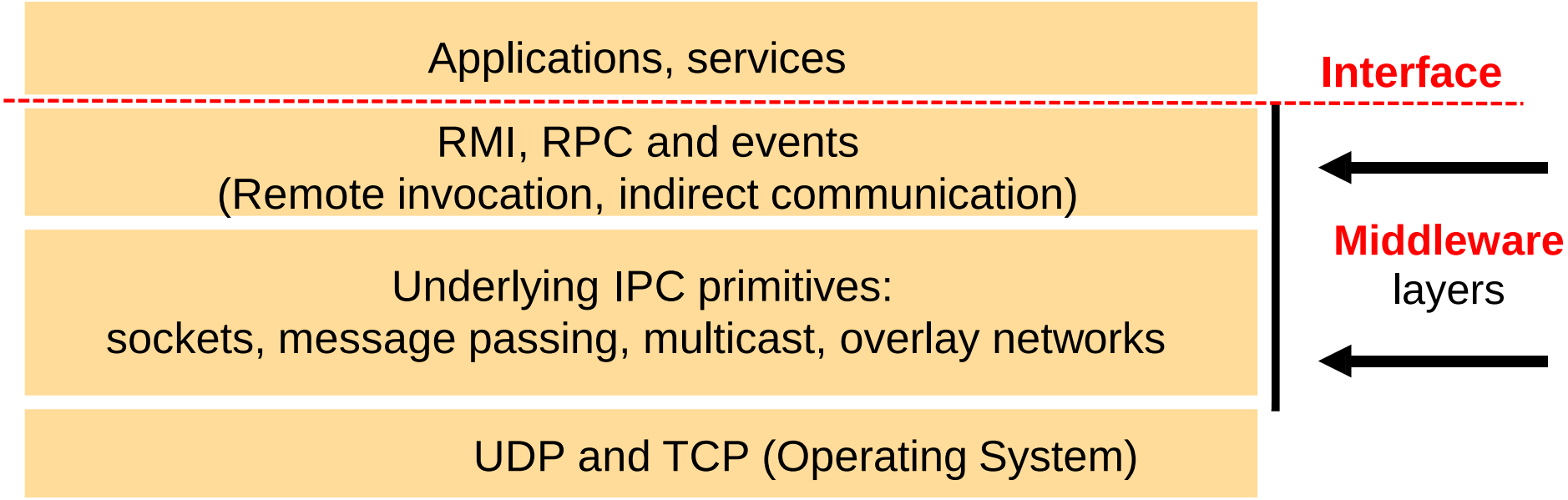
1.1 Introduction

- Programming Models for Distributed Communications
  - Request-reply protocols: message passing in client-server computing
  - Remote Procedure Calls – Client programs call procedures in server programs
  - Remote Method Invocation – Objects invoke methods of objects on distributed hosts
  - Event-based Programming Model – Objects receive notice of events in other objects in which they have interest

Middleware

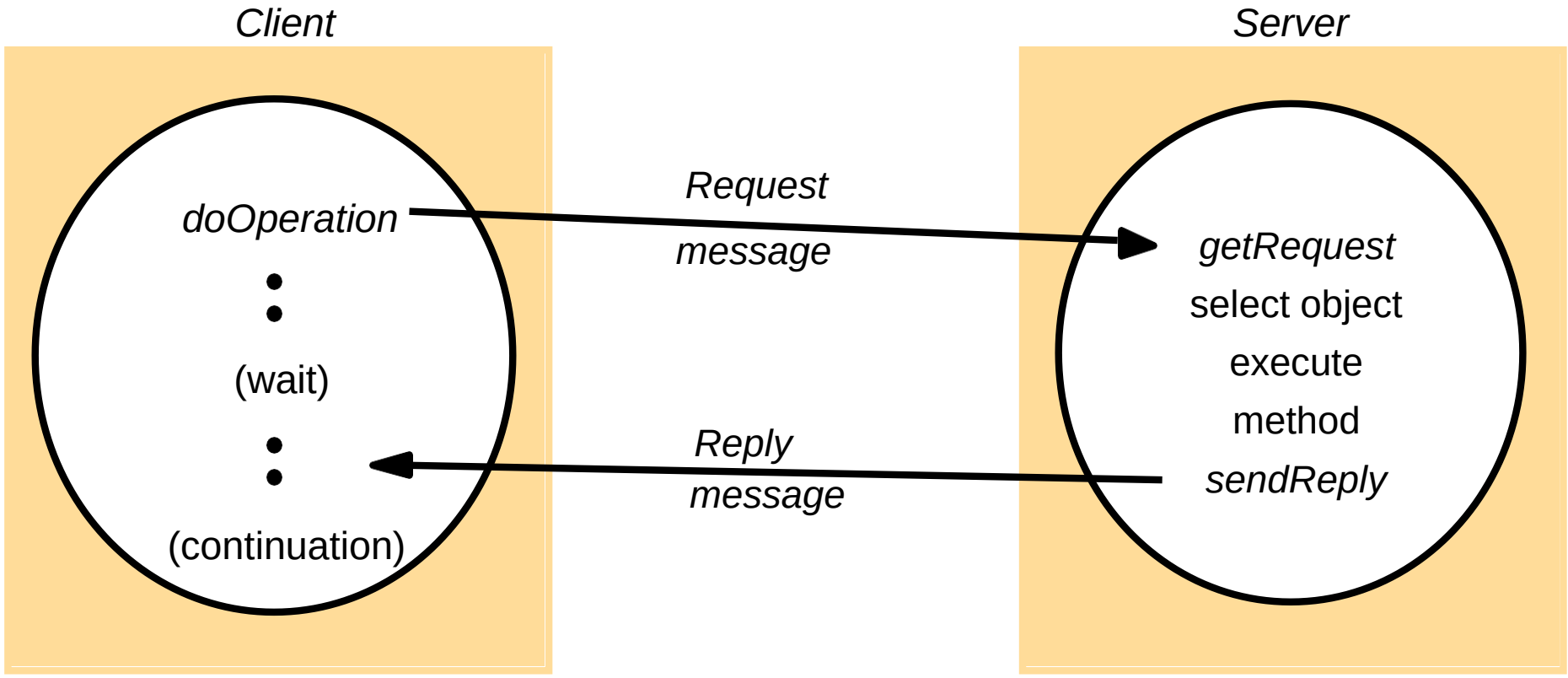
- Middleware: software that allows a level of programming beyond processes and message passing
  - Uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events
  - Supports location transparency
  - Usually uses an ***interface definition language (IDL)*** to define interfaces

Figure 1.1 Middleware Layer



1.2 Request-reply Protocols

Figure 1.2 Request-reply communication



- Typical client-server interaction Usually synchronous, reliable
- Three operations:

client: doOperation
server: getRequest, sendReply

Operations of the request-reply protocol

Figure 1.3

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.

Figure 1.4 Request-reply message structure

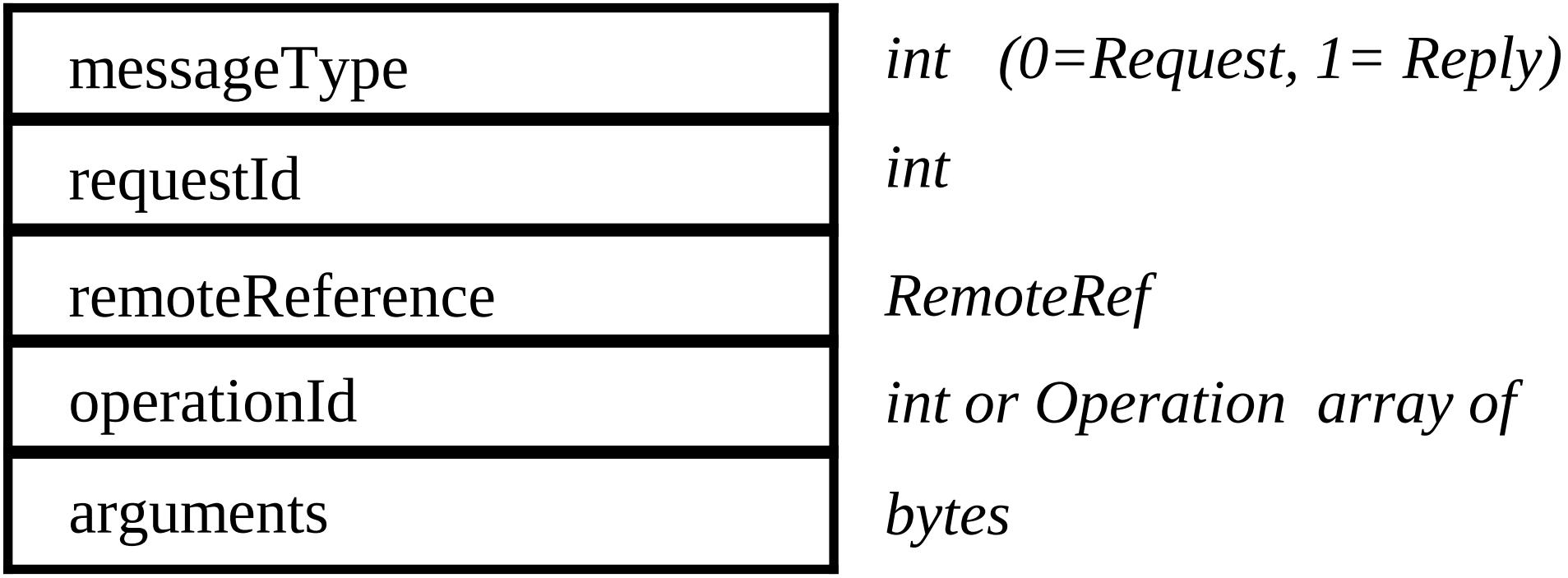


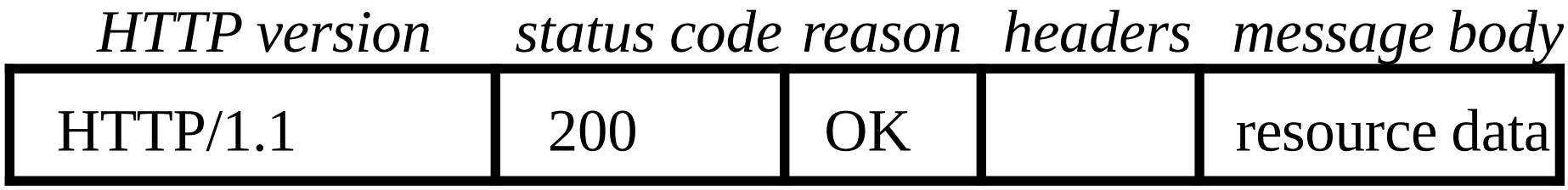
Figure 1.5 RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RR A	*Request*	*Reply*	*Acknowledge reply*

- R protocol: used when no value to be returned and the client requiring no confirmation that the operation has been executed
- RR protocol: used for most client-server exchanges
- RRA protocol: The *Acknowledge reply message* contains the requestId from the reply message being acknowledged to enable the server to discard entries from its history

Figure1.6 HTTP *Request* message

Figure 1.7 HTTP *Reply* message



1.3 Remote Procedure Call

In RPC, procedures on remote machines can be called as if they are procedures in the local address space.

1.3.1 Design Issues for RPC

- Three issues
  - Interfaces
  - Call semantics: Three main choices to provide different delivery guarantees
    - Retry request messages: Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed
    - Duplicate filtering: Controls when retransmissions are used and whether to filter out duplicate requests at the server
    - Retransmission of results: Controls whether to keep a history of result messages

to enable lost results to be retransmitted without re-executing the operations at the server

  - Transparency
    - Remote calls should be made transparent in the sense that the syntax of a remote call is the same as that of a local invocation, but differently expressed in their interfaces

Interfaces

- Interfaces in Programming Languages
  - Current PL allow programs to be developed as a set of modules that communicate with each other. Permitted interactions between modules are defined by ***interfaces***
  - A specified interface can be implemented by different modules without the need to modify other modules using the interface
- Interfaces in Distributed Systems
  - When modules are in different processes or on different hosts there are limitations on the interactions that can occur. Only actions with parameters that are fully specified and understood can communicate effectively to request or provide

services to modules in another process

  - A service interface allows a client to request and a server to provide particular services
  - A remote interface allows objects to be passed as arguments to and results from distributed modules
- Object Interfaces
  - An interface defines the signatures of a set of methods, including arguments, argument types, return values and exceptions. Implementation details are not included in an interface. A class may implement an interface by specifying behavior for each method in the interface. Interfaces do not have constructors

Invocation Semantics

- Error handling for delivery guarantees
  - Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed
  - Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server
  - Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations
- Choices of invocation semantics
  - Maybe: the method executed once or not at all (no retry nor retransmit)
  - At-least-once: the method executed **at least** once
  - At-most-once: the method executed **exactly** once

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Figure 1.9 Invocation semantics: choices of interest

1.3.2 Implementation of RPC

- client: "stub" instead of "proxy" (same function, different names)
  - local call, marshal arguments, communicate the request
- server:
  - dispatcher
  - "stub": unmarshal arguments, communicate the results back

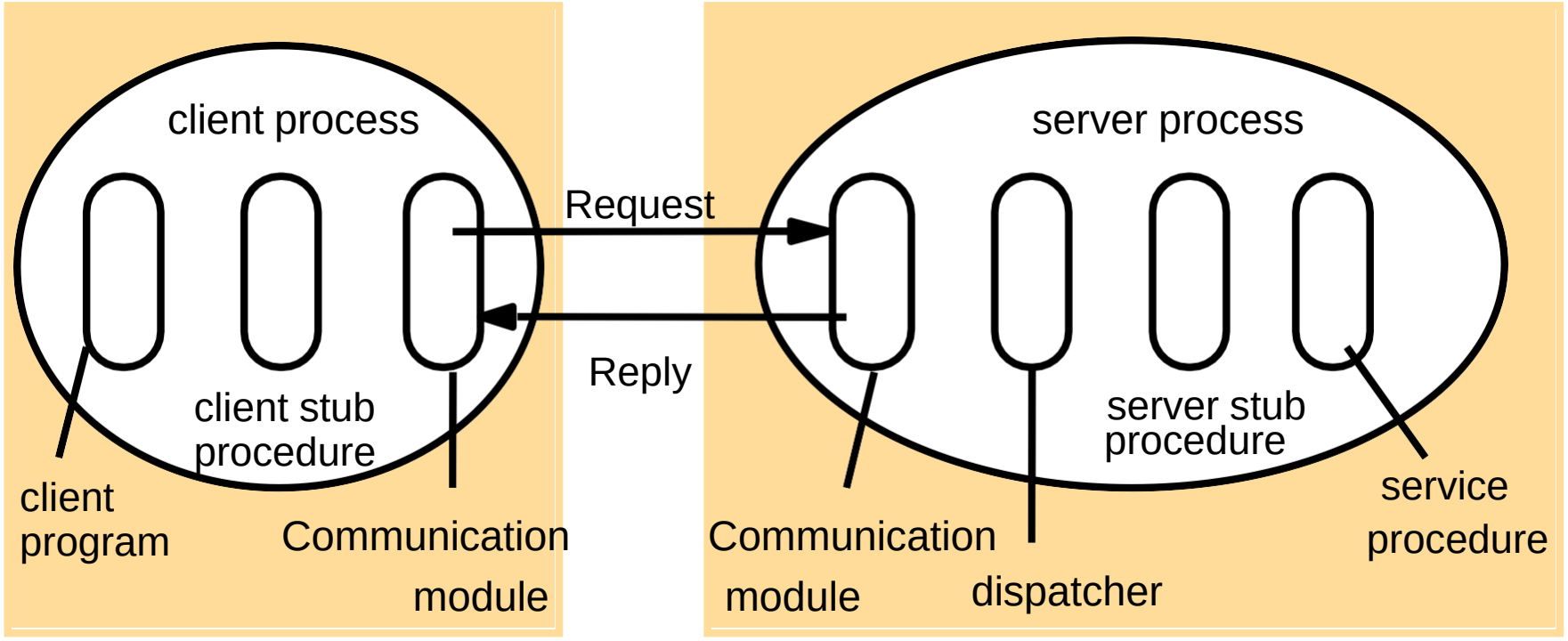


Figure 1.10 Role of client and server stub procedures in RPC in the context of a procedural language

1.3.3 Case Study: Sun RPC

- Sun RPC: client-server in the SUN NFS (network file system)
  - UDP or TCP; in other unix OS as well
  - Also called ONC (Open Network Computing) RPC
- Interface Definition Language (IDL)
  - initially XDR is for data representation, extended to be IDL
  - less modern than CORBA IDL and Java
    - program numbers instead of interface names
    - procedure numbers instead of procedure names
    - single input parameter (structs)
  - rpcgen: compiler for XDR
    - client stub; server main procedure, dispatcher, and server stub
    - XDR marshalling, unmarshaling
- Binding (registry) via a local binder - portmapper
  - Server registers its program/version/port numbers with portmapper
  - Client contacts the portmapper at a fixed port with program/version numbers to get the server port
  - Different instances of the same service can be run on different computers at different ports
- Authentication
  - request and reply have additional fields
    - unix style (uid, gid), shared key for signing, Kerberos

Figure 1.11 Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {  int length;
char buffer[MAX];
};
struct writeargs {
FileIdentifier f;
FilePointer position;
Data data;
};
```

```
struct readargs {  FileIdentifier f;
    FilePointer position;  Length
    length;
};

program FILEREADWRITE {
    version VERSION {
void WRITE(writeargs)=1;        1
Data READ(readargs)=2;          2
}=2;
} = 9999;
```

1.4 Remote Method Invocation

Commonalities between RMI and RPC

- Both support programming with interfaces
- Both typically constructed on top of request-reply protocols and can offer a range of call semantics such as at-least-once and at-most-once
- Both offer a similar level of transparency

Differences between RMI and RPC

- Object-oriented
- All objects in an RMI-based system have (global) unique object references, such object references can be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC

1.4.1 Design Issues for RMI

Five Parts of the Object Model

–An object-oriented program consists of a collection of interacting objects

- Objects consist of a set of data and a set of methods
- In DS, object's data should be accessible only via methods

•Object References

– Objects are accessed by object references

– Object references can be assigned to variables, passedas arguments, and returned as the result of a method

– Can also specify a method to be invoked on that object

•Interfaces

– Provide a definition of the signatures of a set of methods without specifying their implementation

– Define types that can be used to declare the type of variables or of the parameters and return values of methods

The Object Model (Cont.)

- Actions
  - Objects invoke methods in other objects
  - An invocation can include additional information as arguments to perform the behavior specified by the method
  - Effects of invoking a method
    1. The state of the receiving object may be changed
    2. A new object may be instantiated
    3. Further invocations on methods in other objects may occur
    4. An exception may be generated if there is a problem encountered
- Exceptions
  - Provide a clean way to deal with unexpected events or errors
  - A block of code can be defined to *throw* an exception when errors or unexpected conditions occur. Then control passes to code that *catch*es the exception
- Garbage Collection
  - Provide a means of freeing the space that is no longer needed
  - Java (automatic), C++ (user supplied)

Distributed Objects

- Physical distribution of objects into different processes or computers in a distributed system
  - Object state consists of the values of its instance variables
  - Object methods invoked by remote method invocation (RMI)
  - Object encapsulation: object state accessed only by the object methods
- Usually adopt the client-server architecture
  - Basic model
    - Objects are managed by servers and
    - Their clients invoke their methods using RMI
  - Steps

1. The client sends the RMI request in a message to the server
2. The server executes the invoked method of the object
3. The server returns the result to the client in another message

- Other models
  - Chains of related invocations: objects in servers may become clients of objects in other servers
  - Object replication: objects can be replicated for fault tolerance and performance
  - Object migration: objects can be migrated to enhancing performance and availability

The Distributed Object Model

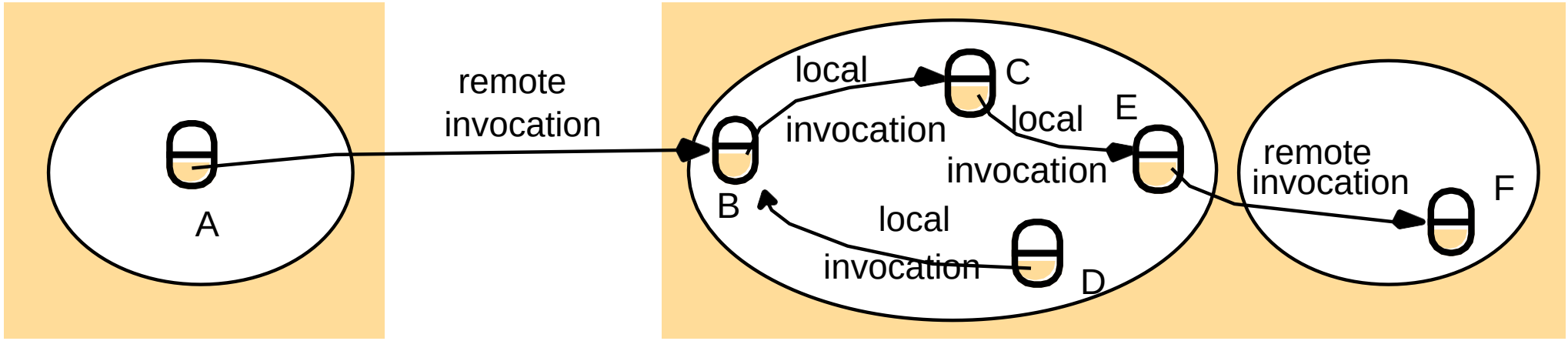


Figure 1.12 Remote and local method invocations

Two fundamentalconcepts: Remote Object Reference and Remote Interface

- Each process contains objects, some of which can receive remote invocations are called *remote objects* (B, F), others only local invocations
- Objects need to know the *remote object reference* of an object in another process in order to invoke its methods, called *remote method invocations*
- Every remote object has a *remote interface* that specifies which of its methods can be invoked remotely

Five Parts of Distributed Object Model

- Remote Object References
  - accessing the remote object
  - identifier throughout a distributed system
  - can be passed as arguments
- Remote Interfaces
  - specifying which methods can be invoked remotely
  - name, arguments, return type
  - Interface Definition Language (IDL) used for defining remote interface

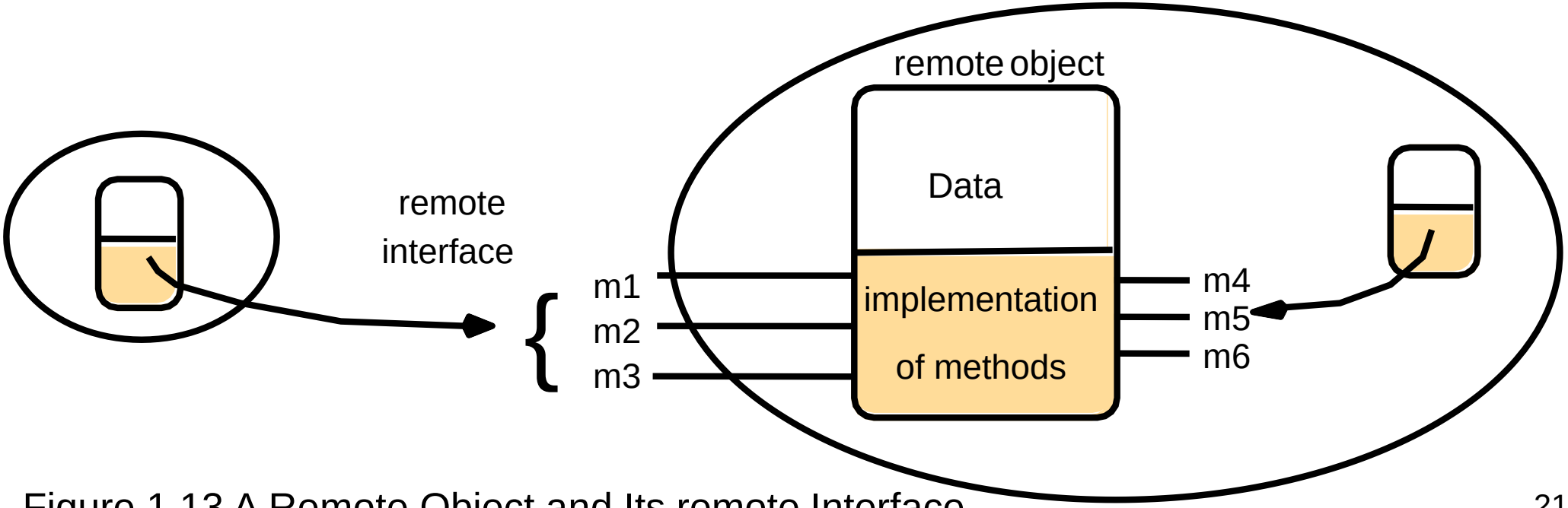


Figure 1.13 A Remote Object and Its remote Interface

Five Parts of Distributed Object Model (cont.)

- Actions
  - An action initiated by a method invocation may result in further invocations on methods in other objects located in difference processes or computers
  - Remote invocations could lead to the instantiation of new objects, ie. objects M and N of Figure 1.5
- Exceptions
  - More kinds of exceptions: i.e. timeout exception
    - RMI should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked
- Garbage Collection
- Distributed garbage collections is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting

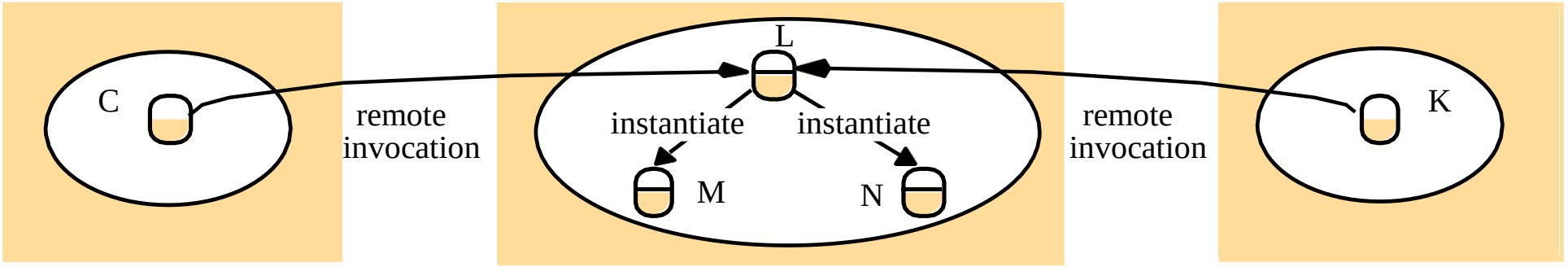


Figure 1.14 Instantiating Remote Objects

Design Issues for RMI

- Two design issues that arise in extension of local method invocation for RMI

– The choice of invocation semantics
  - Although local invocations are executed exactly once, this cannot always be the case for RMI due to transmission error
    – Either request or reply message may be lost
    – Either server or client may be crashed
  – The level of transparency
    - Make remote invocation as much like local invocation as possible

RMI Design Issues: Transparency

- Transparent remote invocation: like a local call
  - marshalling/unmarshalling
  - locating remote objects
  - accessing/syntax
- Differences between local and remote invocations
  - latency: a remote invocation is usually several order of magnitude greater than that of a local one
  - availability: remote invocation is more likely to fail
  - errors/exceptions: failure of the network? server? hard to tell
    - syntax might need to be different to handle different local vs remote errors/exceptions (e.g. Argus)
  - consistency on the remote machine:
    - Argus: incomplete transactions, abort, restore states [as if the call was never made]

1.4.2 Implementation of RMI

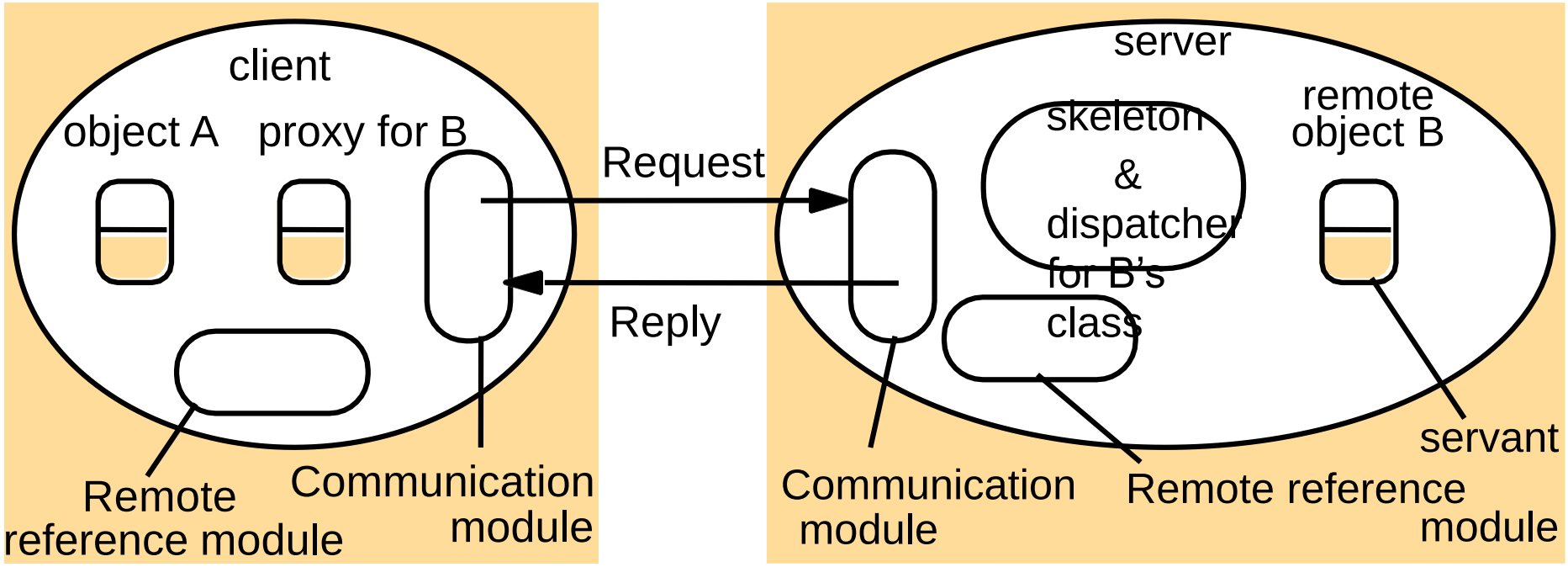


Figure 1.15 Role of proxy and skeleton in RMI

- Communication module
  - Two cooperating communication modules carry out the request-reply protocols: message type, requestID, remote object reference
    - Transmit request and reply messages between client and server
    - Implement specific invocation semantics
  - The communication module in the server
    - selects the dispatcher for the class of the object to be invoked,
    - passes on local reference from remote reference module,
    - returns request

Implementation of RMI (cont.)

- Remote reference module
  - Responsible for translating between local and remote object references and for creating remote object references
  - remote object table: records the correspondence between local and remote object references
    - remote objects held by the process (B on server)
    - local proxy (B on client)
  - When a remote object is to be passed for the first time, the module is asked to create a remote object reference, which it adds to its table
- Servant
  - An instance of a class which provides the body of a remote object
  - handles the remote requests
- RMI software
  - Proxy: behaves like a local object, but represents the remote object
  - Dispatcher: look at the methodID and call the corresponding method in the skeleton
  - Skeleton: implements the method

Generated automatically by an interface compiler

Implementation Alternatives of RMI

- Dynamic invocation
  - Proxies are static—interface complied into client code
  - Dynamic—interface available during run time
    - Generic invocation; more info in “Interface Repository” (COBRA)
    - Dynamic loading of classes (Java RMI)
- Binder
  - A separate service to locate service/object by name through table mapping for names and remote object references
- Activation of remote objects
  - Motivation: many server objects not necessarily in use all of the time
    - Servers can be started whenever they are needed by clients, similar to inetd
  - Object status: active or passive
    - active: available for invocation in a running process
    - passive: not running, state is stored and methods are pending
  - Activation of objects:
    - creating an active object from the corresponding passive object by creating a new instance of its class
    - initializing its instance variables from the stored state
  - Responsibilities of ***activator***
    - Register passive objects that are available for activation
    - Start named server processes and activate remote objects in them
    - Keep track of the locations of the servers for remote objects that it has already activated

Implementation Alternatives of RMI (cont.)

- Persistent object stores
  - An object that is guaranteed to live between activations of processes is called a ***persistent object***
  - Persistent object store: managing the persistent objects
    - stored in marshaled from on disk for retrieval
    - saved those that were modified
  - Deciding whether an object is persistent or not:
    - persistent root: any descendent objects are persistent (persistent Java, PerDiS)
    - some classes are declared persistent (Arjuna system)
- Object location
  - specifying a location: ip address, port #, …
  - location service for migratable objects
    - Map remote object references to their probable current locations
    - Cache/broadcast scheme (similar to ARP)
      - Cache locations
      - If not in cache, broadcast to find it
    - Improvement: forwarding (similar to mobile IP)

1.4.3 Distributed Garbage Collection

- Aim: ensure that an object
  - continues to exist if a local or remote reference to it is still held anywhere
  - be collected as soon as no object any longer holds a reference to it
- General approach: reference count
- Java's approach
  - the server of an object (B) keeps track of proxies
  - when a proxy is created for a remote object
    - addRef(B) tells the server to add an entry
  - when the local host's garbage collector removes the proxy
    - removeRef(B) tells the server to remove the entry
  - when no entries for object B, the object on server is deallocated

1.5 Case Study: Java RMI

```
import java.rmi.*;  import java.util.Vector;
public interface Shape extends Remote {
int getVersion() throws RemoteException;
GraphicalObject getAllState() throws RemoteException;   1

}
public interface ShapeList extends Remote {
Shape newShape(GraphicalObject g) throws RemoteException; 2
Vector allShapes() throws RemoteException;
int getVersion() throws RemoteException;
}
```

Java Remote interfaces Shape and ShapreList

Java class
ShapeListServant implements interface ShapeList

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;  import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {

   private Vector theList; // contains the list of Shapes  private int 1
      version;
   public ShapeListServant()throws RemoteException{...}
   public Shape newShape(GraphicalObject g) throws RemoteException { 2
      version++;
   Shape s = new ShapeServant(g, version);  theList.addElement(s);      3
   return s;
   }
   public Vector allShapes()throws RemoteException{...}  public int
      getVersion() throws RemoteException { ... }

}
```

Java class ShapeListServer with main method
Java client of ShapreList

```
import java.rmi.*;
public class ShapeListServer{
public static void main(String args[]){
System.setSecurityManager(new RMISecurityManager());  try{
ShapeList aShapeList = new ShapeListServant();
Naming.rebind("Shape List", aShapeList );                              1
System.out.println("ShapeList server ready");                          2
}catch(Exception e) {

            System.out.println("ShapeList server main " + e.getMessage());}
            }
            }
```

```
            import java.rmi.*;  import java.rmi.server.*;  import
            java.util.Vector;
            public class ShapeListClient{
            public static void main(String args[]){
            System.setSecurityManager(new RMISecurityManager());  ShapeList
            aShapeList = null;
            try{

    aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");          1
    Vector sList = aShapeList.allShapes();                                 2
  } catch(RemoteException e) {System.out.println(e.getMessage());
  }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

Naming class of Java RMIregistry

void rebind (String name, Remote obj)
This method is used by a server to register the identifier of a remote object by name, as shown in Figure 1.13, line 3.
void bind (String name, Remote obj)
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.
void unbind (String name, Remote obj)
This method removes a binding.
Remote lookup(String name)
This method is used by clients to look up a remote object by name, as shown in Figure 1.15 line 1. A remote object reference is returned.
String [] list()
This method returns an array of Strings containing the names bound in the registry.

Jini

- Jini
  - Jini technology is a service oriented architecture that defines a programming model which both exploits and extends Java technology to enable the construction of secure, distributed systems consisting of federations of well-behaved network services and clients
    - Allow a potential subscriber in one Java Virtual Machine (JVM) to subscribe to and receive notifications of events in an object of interest in another JVM
    - Main objects
      - event generators (publishers)
      - remote event listeners (subscribers)
      - remote events (events)
      - third-party agents (observers)
    - An object subscribes to events by informing the event generator about the type of event and specifying a remote event listener as the target for notification

Java RMI Callbacks

- Callbacks
  - server notifying the clients of events
  - why?
    - polling from clients increases overhead on server
    - not up-to-date for clients to inform users
  - how
    - remote object (callback object) on client for server to call
    - client tells the server about the callback object, server put the client on a list
    - server call methods on the callback object when events occur
  - client might forget to remove itself from the list
    - lease--client expire

Chapter 2
Java RMI

Outline

- Part I. An Overview of RMI Applications
  - A Quick Local Java RMI Example: Hello World (from Wiki)
    - RmiServerIntf, RmiServer, RmiClient
- Part II. A Behavior-based Application - Building a Generic Compute Engine

II-A. Writing an RMI Server and RMI Clients

- Step 1. Designing Remote Interfaces
- Step 2. Implementing a Remote Interface
- Step 3. Creating a Client Program
- Step 4. Implementing the Task Interface

II-B. Compiling the Example Programs II-C. Running the Example Programs

Part I. An Overview of RMI Applications

- Java Remote Method Invocation (RMI) system
  - Allows an object running in one JVM to invoke methods on an object running in another JVM
  - Provides the mechanism by which the server and the client communicate and pass information back and forth
    - Such an application is sometimes referred to as a *distributed object application*
- Often comprise two separate programs, a server and a client
  - A typical server
    - Creates some remote objects
    - Makes references to these objects accessible, and
    - Waits for clients to invoke methods on these objects
  - A typical client
    - Obtains a remote reference to one or more remote objects on a server and
    - Then invokes methods on them
- Features
  - One of the central and unique features of RMI is its ability to *on-demand download the definition of an object's class* if the class is not defined in the receiver's JVM
  - RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another JVM

Remote Interfaces, Objects, and Methods

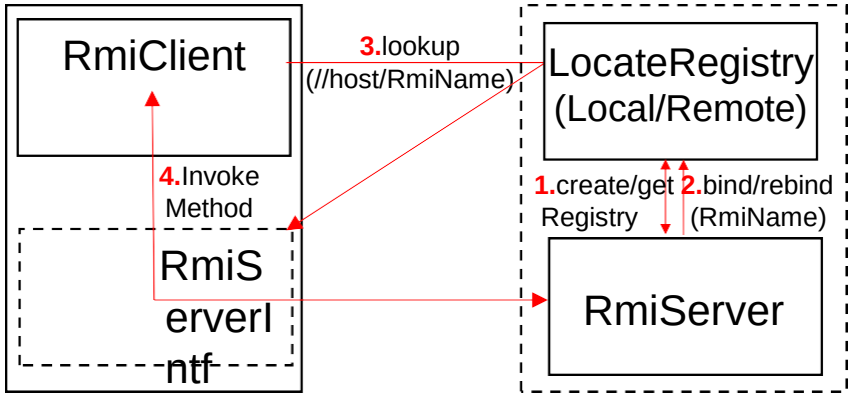
- An object becomes remote by implementing a remote interface, which has the following characteristics:
  - A remote interface extends the interface `java.rmi.Remote`
  - Each method of the interface declares `java.rmi.RemoteException` in its

throws clause, in addition to any application-specific exceptions
  - Only those methods defined in a remote interface are available to be called from the receiving Java virtual machine
- RMI passes a remote stub for a remote object
  - The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference
  - The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object

Steps to Create RMI Applications

1. Designing and implementing the components of your distributed application
   - Defining the remote interfaces
   - Implementing the remote objects
   - Implementing the clients
2. Compiling sources
   - Use the `javac` compiler to compile the source files, including
     - Declarations of the remote interfaces, their implementations, any other server classes, and the client classes
3. Making classes network accessible
   - Classes definitions are typically made network accessible through a web server
4. Starting the application
   - Starting the application includes running the RMI remote object registry, the server, and the client

A Quick Local Java RMI Example: Hello World



https://en.wikipedia.org/wiki/Java_remote_method_invocation

RmiClient.java

```
import java.rmi.Naming;

public class RmiClient {
    public static void main(String args[]) throws Exception {
        RmiServerIntf obj =
           (RmiServerIntf)Naming.lookup("//localhost/RmiServer");
        System.out.println(obj.getMessage());
    }
}
```

3. 4.

rmi://host/RmiName

RmiServerIntf.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RmiServerIntf extends Remote {
    public String getMessage() throws RemoteException;
}
```

RmiServer.java

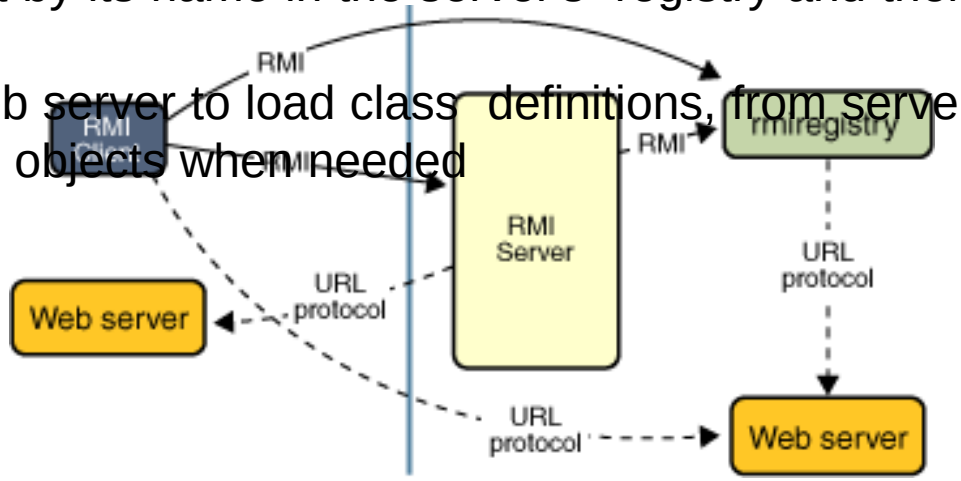
```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.*;
public class RmiServer extends UnicastRemoteObject implements RmiServerIntf {
    public static final String MESSAGE = "Hello World";

    public RmiServer() throws RemoteException {
        super(0);    // required to avoid the 'rmic' step, see below
    }

    public String getMessage() {
        return MESSAGE;
    }

    public static void main(String args[]) throws Exception {
        System.out.println("RMI server started");

        try { //special exception handler for registry creation
            LocateRegistry.createRegistry(1099);
            System.out.println("java RMI registry created.");
        } catch (RemoteException e) {
            //do nothing, error means registry already exists
            System.out.println("java RMI registry already exists.");
        }

        //Instantiate RmiServer

        RmiServer obj = new RmiServer();

        // Bind this object instance to the name "RmiServer"
        Naming.rebind("//localhost/RmiServer", obj);
        System.out.println("PeerServer bound in registry");
    }
}
```

1. 2.

Registry	LocateRegistry.createRegistry(int port)	Creates the registry on the local host with the port
Registry	LocateRegistry.getRegistry(String host, int port)	Server searches the registry on the host with the port
void	Naming.rebind/bind(String name, Remote obj)	Server binds an remote interface to the name
Remote	Naming.lookup(String name)	Client searches the name for the remote interface

Distributed Object Applications

- Generic flow
  - Locate remote objects
    - Can register its remote objects with RMI's simple naming facility, the *RMI registry*
    - Can pass and return remote object references as part of other remote invocations
  - Communicate with remote objects
  - Load class definitions for objects that are passed around
- The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object
  - The server calls the registry to associate (or bind) a name with a remote object
  - The client looks up the remote object by its name in the server's registry and then invokes a method on it
  - The RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed



Part II. A Behavior-based Application - Building a Generic Compute Engine

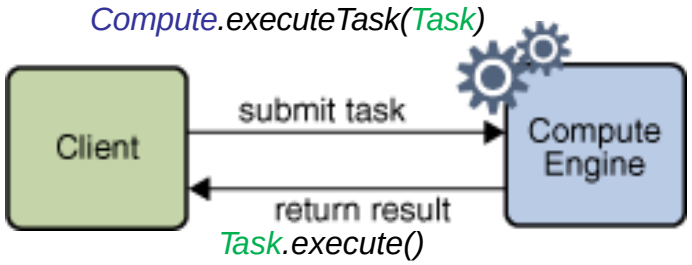
- Function: enable a number of client machines to make use of a particularly powerful machine to execute specified tasks (objects passed as arguments)
- Behavior-based application: download code dynamically
  - RMI dynamically loads the task code into the compute engine's JVM and
  - Runs the task without prior knowledge of the class that implements the task
- Steps: the compute engine is a remote object on the server that
  - Takes tasks from clients
  - Runs the tasks, and
  - Returns any results
- Novel aspect: the tasks it runs do not need to be defined when the compute engine is written or started
  - New kinds of tasks can be created at any time and then given to the compute engine to be run
  - The only requirement of a task is that its class implement a particular interface
  - The code needed to accomplish the task can be downloaded by the RMI system to the compute engine

II-A. Writing an RMI Server and RMI Clients

- Writing an RMI server
  - Step 1. Designing Remote Interfaces
    - `Compute.java` - ***`Compute`*** RMI interface: provides methods to be invoked by clients
    - `Task.java` - ***`Task`*** interface: client's submitted object to be passed and executed by server
  - Step 2. Implementing a Remote Interface
    - `ComputeEngine.java` - **`ComputeEngine`** class: implement server's ***`Compute`*** interface
      - Implements the ***`Compute`*** interface
      - Provides the rest of the code that makes up the server program, including a main method that
        - » Sets up a security manager
        - » Creates an instance of the remote object
        - » Registers it with the RMI registry
- Writing an RMI client
  - Step 3. Creating a Client Program
    - `ComputePi.java` - **`ComputePi`** class: client's main to invoke server's ***`Compute`*** interface
      - Provides the rest of the code that makes up the client program, including a main method that
        - » Sets up a security manager
        - » Synthesizes a remote reference to the registry
        - » Looks up the remote object
        - » Creates and submits the task
  - Step 4. Implementing the Task Interface
    - `Pi.java` - **`Pi`** class: a polymorphism implements the client's ***`Task`*** interface
      - The code is loaded and executed by the **ComputeEngine**

Step 1. Design a Remote Interface

- The protocol of the compute engine
  - Enables tasks to be submitted to the compute engine
  - The compute engine to run those tasks, and
  - The results of those tasks to be returned to the client
- Two interfaces
  - The compute engine's remote interface, **`Compute`**, enables different tasks to be submitted to the engine



Server's **`Compute`** interface

an RMI method

Compute.java

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
  <T> T executeTask(Task<T> t) throws RemoteException;
}
```

  - The client interface, **`Task`**, defines how compute engine executes a submitted task
    - The type of the parameter to the **`executeTask`** method in the **`Compute`** interface
      - Classes that implement the **`Task`** interface must also implement Serializable
    - The **`Task`** interface has a type parameter, **`T`**
      - Which represents the result type of the task's computation
    - This interface's execute method returns the result of the computation and thus its return type is **`T`**

Client's **`Task`** interface

Task.java

```
package compute;

public interface Task<T> {
  T execute();
}
```

Step 2. Implement a Remote Interface

ComputeEngine implements server's `Compute` interface

ComputeEngine.java

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;  import
compute.Task;

public class ComputeEngine implements Compute {  public
  ComputeEngine() {
super();
}
public <T> T executeTask(Task<T> t) {  return
  t.execute();
}
public static void main(String[] args) {  if
  (System.getSecurityManager() == null) {
System.setSecurityManager(new SecurityManager());
}
try {
String name = "Compute";
Compute engine = new ComputeEngine();
Compute stub =
(Compute) UnicastRemoteObject.exportObject(engine,0);
  Registry registry = LocateRegistry.getRegistry();
  registry.rebind(name, stub);
  System.out.println("ComputeEngine bound");
} catch (Exception e) {
System.err.println("ComputeEngine exception:");
  e.printStackTrace();
                          } } }
```

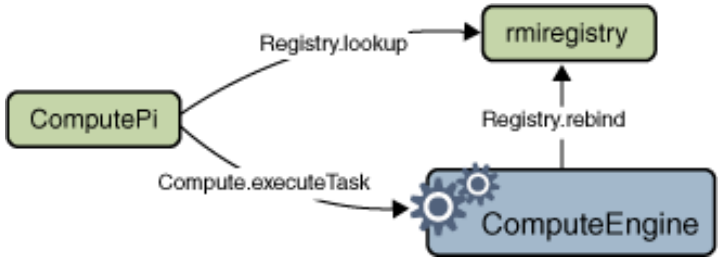
- Declare the remote interfaces being implemented
- Define the constructor for each remote object
  - Invoke superclass constructor for clarity
- Provide an implementation for each remote method in the remote interfaces
  - Remote objects are essentially passed by reference
    - A remote object reference is a stub, which is a client-side proxy that implements the complete set of remote interfaces that the remote object implements
  - Local objects are passed by copy, using object serialization
    - By default, all fields are copied except fields that are marked static or transient
    - Default serialization behavior can be overridden on a class-by-class basis
- Create and install a security manager
  - Enable passing classes for remote objects received as arguments or return values
- Create and export remote objects
  - Enable receive invocations from remote
  - 0 means anonymous TCP port
- Register remote objects with the RMI registry (or with another naming service, i.e. JNDI) for bootstrapping

Step 3. Creating a Client Program

`Client.ComputePi` invokes server's `Compute` interface

ComputePi.java

```
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
public static void main(String args[]) {  if
(System.getSecurityManager() == null) {
System.setSecurityManager(new SecurityManager());
}
try {
String name = "Compute";
Registry registry = LocateRegistry.getRegistry(args[0]);
Compute comp = (Compute) registry.lookup(name);  Pi task =
new Pi(Integer.parseInt(args[1]));  BigDecimal pi =
comp.executeTask(task);  System.out.println(pi);
} catch (Exception e) {  System.err.println("ComputePi
exception:");  e.printStackTrace();
}
}
}
```



- Create and install a security manager
  - Need to receive remote objects and download class definitions as return values
- Synthesize a remote reference to the registry on the server's host
  - `args[0]` is the name of the remote host
- Invoke the lookup method on the registry to look up the remote object
- Create a task (`Pi` object)
  - `args[1]` is the number of decimal places
- Invoke executeTask method to submit the task to the **Compute** remote object

Step 4. Implementing the Task Interface

Pi implements client's `Task` interface

```
package client;

import compute.Task;
import java.io.Serializable;  import
java.math.BigDecimal;

public class Pi implements Task<BigDecimal>,Serializable{
private static final long serialVersionUID = 227L;
/** constants used in pi computation */  private static
final BigDecimal FOUR =
BigDecimal.valueOf(4);

/** rounding mode to use during pi computation */
private static final int roundingMode =
BigDecimal.ROUND_HALF_EVEN;

/** digits of precision after the decimal point */
private final int digits;

/** Construct to calculate pi to specified precision */
public Pi(int digits) {
this.digits = digits;
}

/** Calculate pi. */
public BigDecimal execute() {  return computePi(digits);
}
/** Compute the value of pi to the specified number of
*digits after the decimal point.    The value is
*computed using Machin's formula:
   *       pi/4 = 4*arctan(1/5) - arctan(1/239)
   * and a power series expansion of arctan(x) to
   * sufficient precision.
   */
   public static BigDecimal computePi(int digits) {
   int scale = digits + 5;
   BigDecimal arctan1_5 = arctan(5, scale);
   BigDecimal arctan1_239 = arctan(239, scale);
   BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
   arctan1_239).multiply(FOUR);  return
   pi.setScale(digits,
   BigDecimal.ROUND_HALF_UP);
   }
```

- Note that all serializable classes, whether they implement the `Serializable` interface directly or indirectly, must declare a private static final field named **serialVersionUID** to guarantee serialization compatibility between versions
- Steps
  1. **ComputePi** invokes **Compute.executeTask** with a **Pi** object (**Task**)
  2. The code for the class is loaded by RMI into the **Compute** object's JVM
  3. **ComputeEngine** invokes **Task**/**Pi**'s **execute**() method

```
/**
*Compute the value, in radians, of the arctangent of
*the inverse of the supplied integer to the specified
*number of digits after the decimal point.    The value
*is computed using the power series expansion for the
*arc tangent:
*
* arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
*(x^9)/9 ...
*/
public static BigDecimal arctan(int inverseX, int scale){
 BigDecimal result, numer, term;
BigDecimal invX = BigDecimal.valueOf(inverseX);
 BigDecimal invX2 =
BigDecimal.valueOf(inverseX * inverseX);

numer = BigDecimal.ONE.divide(invX,
                  scale, roundingMode);

result = numer;  int i = 1;
do {  numer =
numer.divide(invX2, scale, roundingMode);  int denom = 2 *
 i + 1;
term =
numer.divide(BigDecimal.valueOf(denom),  scale,
 roundingMode);
if ((i % 2) != 0) {
result = result.subtract(term);
} else {
result = result.add(term);
}  i++;
} while (term.compareTo(BigDecimal.ZERO) != 0);


   return result;
} }
```

II-B. Compiling the Example Programs

- **`compute`** package – **`Compute`** and **`Task`** interfaces (compute.jar)
  - A developer would likely create a Java Archive (JAR) file that contains the Compute and Task interfaces for server classes to implement and client programs to use
- **`engine`** package – **`ComputeEngine`** implementation class
  - The developer writes an implementation of the Compute interface and deploy that service on a machine available to clients
- **`client`** package – **`ComputePi`** client code and **`Pi`** task implementation
  - Developers of client programs can use the Compute and the Task interfaces, contained in the JAR file, and independently develop a task and client program that uses a Compute service

Building Interface, Server, and Client Classes

- Build a JAR File of RMI Interface Classes
  - Build **compute.jar**

```
cd \home\rmiuser\src
javac compute\Compute.java compute\Task.java  jar cvf compute.jar
  compute\*.class
```

  - Distribute the jar file to developers or place in a network-accessible location

```
\somewhere\ or ~rmiuser\public_html\classes\
```

- Build the Server Classes

```
cd \home\server\src
javac -cp \somewhere\compute.jar engine\ComputeEngine.java
```

- Build the Client Classes
  - Compile the codes

```
cd \home\cliuser\src
javac -cp \somewhere\compute.jar client\ComputePi.java client\Pi.java
```

  - Place the task code at http://host:port/~cliuser/classes/

```
mkdir ~cliuser\public_html\classes\client
cp client\Pi.class ~cliuser\public_html\classes\client
```

II-C. Running the Example Programs

- Need to specify a security policy file so that the code is granted the security permissions it needs to run
  - `server.policy` for running server classes

```
grant codeBase "file:/home/server/src/" {  permission
  java.security.AllPermission;
};
```

  - `client.policy` for running client classes

```
grant codeBase "file:/home/cliuser/src/" {  permission
  java.security.AllPermission;
};
```

- Start the RMI Registry

– Start the `rmiregistry` command (default on port 1099)

`start rmiregistry` or `start rmiregistry 2001`

  - Use `javaw` if `start` is not available – run as background job
  - Warning: classes visible to CLASSPATH environment varilable become remote available

–Should make sure that no CLASSPATH set or no classes under CLASSPAT to be downloaded remotely

Starting the Server and the Client

- Start the server with specified system properties

```
java -cp ~server\src;\somewhere\compute.jar
-Djava.rmi.server.codebase=file:./
-Djava.rmi.server.hostname=mycomputer.example.com
-Djava.security.policy=engine\server.policy
   engine.ComputeEngine
```

a URL to the jar file (`Compute` and `Task` interfaces) or http://jarfile-host/dir-path-of-the-jarfile/

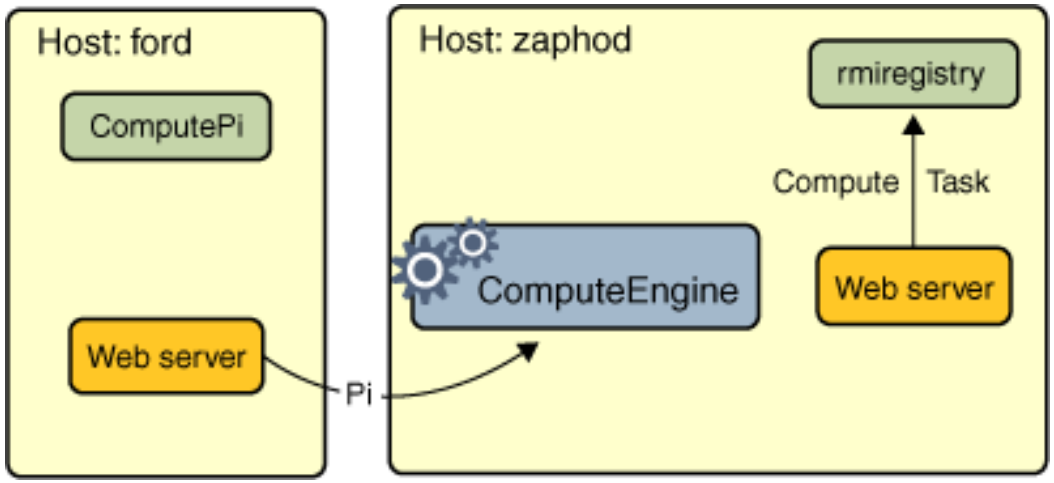
Default by java.net.InetAddress.getLocalHost

- Start the client to submit `Pi` class

```
java -cp ~cliuser\src;\somewhere\compute.jar
     -Djava.rmi.server.codebase=file:./client/
     -Djava.security.policy=client\lient.policy
        client.ComputePi mycomputer.example.com 45
```

a URL to the class definitions (`Pi`) from this the server can be downloaded (dir ended with '/')

```
3.14159265358979323846264338327950288419716939937
```



Summary

- Two RMI examples
  - A simple local “Hello World” example
  - A generic compute engine
- RMI often comprises
  - An interface
    - An RMI registry helps binding and looking up objects
    - A JAR file is usually created, and distributed via the Web
  - Two separate programs, a server and clients
    - A security manager must be configured to support remote

UNIT 4
Distributed File Systems

Chapter 1 Distributed File Systems

1. Introduction
2. File service architecture
3. Case study: Sun Network File System
4. Case study: The Andrew File System
5. Enhancements and further developments
6. Summary

1.1 Introduction

- Sharing of resources is a key goal for distributed systems
  - printers, storages, network bandwidths, memories, …
- Mechanisms for data sharing
  - Web servers
  - P2P file sharing
  - Distributed storage systems
    - Distributed file systems
    - Distributed object systems
- Goal of distributed file service
  - Enable programs to store and access remote files exactly as they do local ones

Figure 1.1 Storage systems and their properties

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system Distributed file	×	✓	×	1	UNIX file system
system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (DSM)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore

Types of consistency:
1: strict one-copy. ✓ slightly weaker guarantees. 2: considerably weaker guarantees.

1.1.1 Characteristics of file systems

- File system: responsible for organization, storage, retrieval, naming, sharing and protection of files
  - file: containing data and attributes (Fig 12.3)
  - directory: mapping from text names to internal file identifiers
  - file operation: system calls in UNIX (Fig. 12.4)

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

Figure 1.2 File system modules

Figure 1.3 File attribute record structure

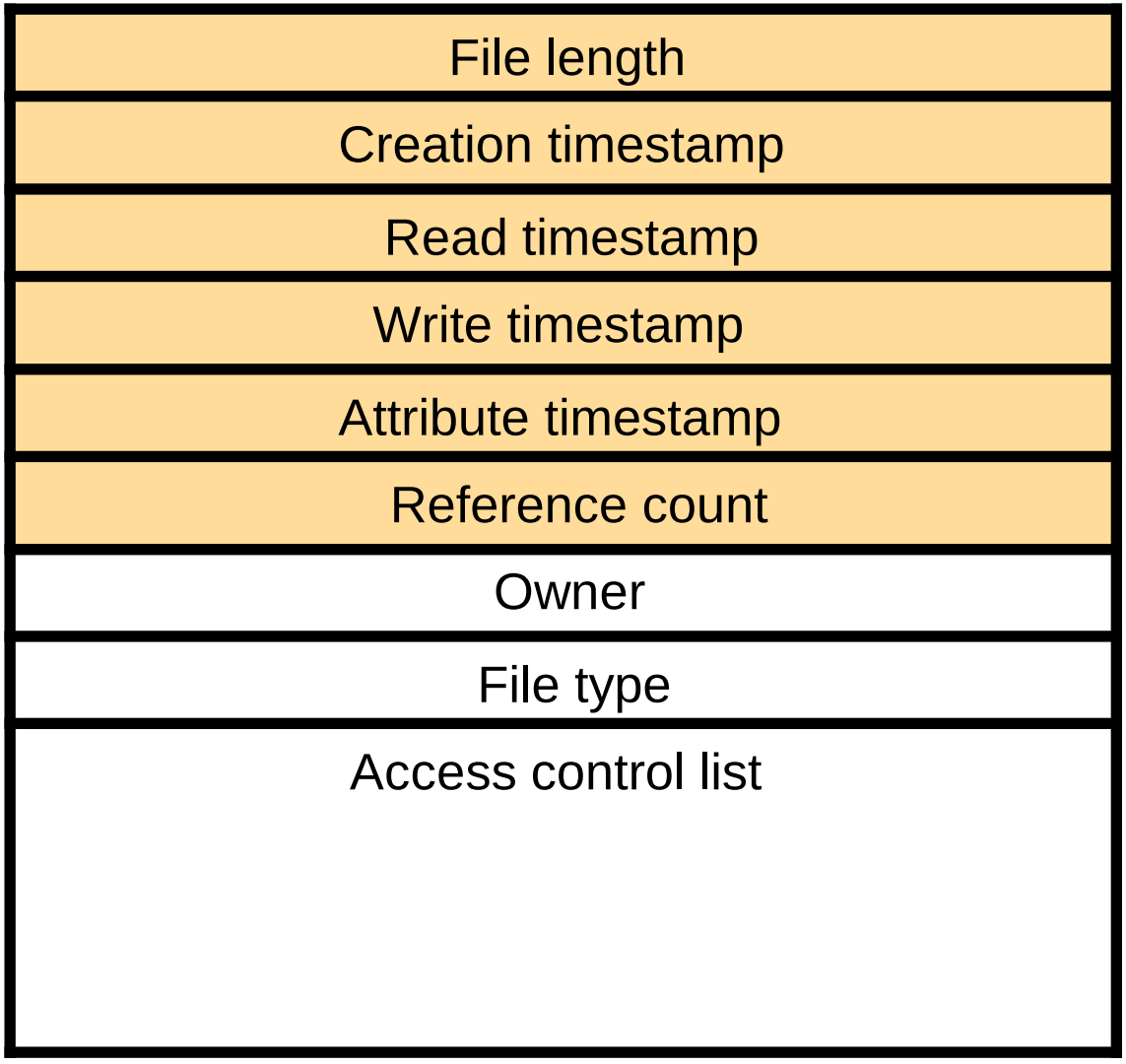


Figure 1.4 UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*.
count = write(filedes, buffer, n)	Transfers *n* bytes to the file referenced by *filedes* from buffer. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to offset (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*). Gets the file
status = stat(name, buffer)	attributes for file *name* into *buffer*.

1.1.2 Distributed file system requirements

- Transparency:
  - –access, location, mobility, performance, scaling
- Concurrent file updates
  - –changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file
  - –common services: advisory or mandatory file- or record-level locking
- File replication
  - –A file may be represented by severalcopies of its contents at different locations
  - –advantages: load sharing and fault tolerance
- Hardware and operating system heterogeneity
- Fault tolerance
  - –The service continue to operate in the face of client and server failures
- Consistency
  - –An inevitable delay in the propagation of modifications to all sites
- Security
- Efficiency

1.1.3 Case studies

- Sun NFS (Network File System)
  - introduced in 1985
  - the first file service designed as a product
  - RFC1813: NFS protocol version 3
  - Each computer can act as both a client and a server
- Andrew File System (AFS)
  - developed at CMU for use as a campus computing and information system
  - achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version

1.2 File service architecture

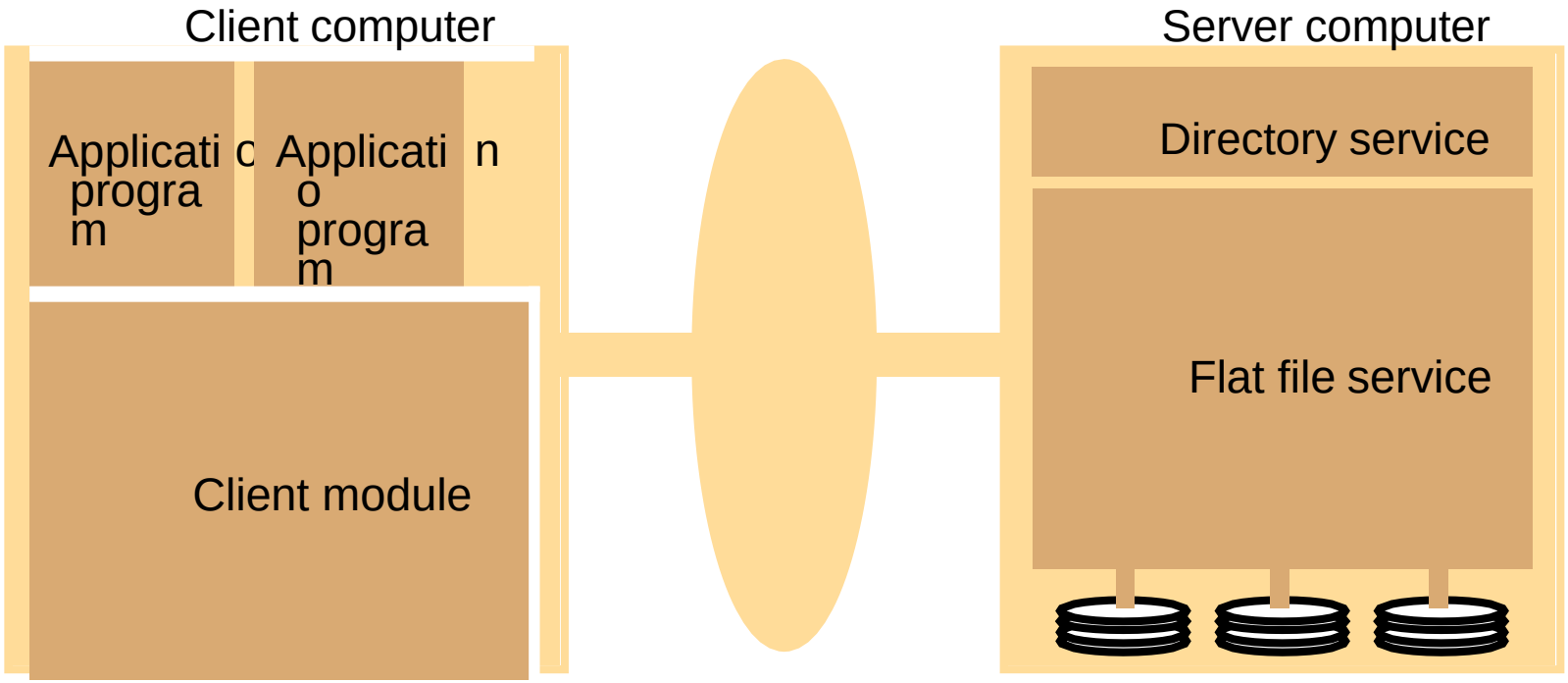


Figure 1.5 File service architecture (Author's abstract model)

- Stateless file service architecture
  - Flat file service: unique file identifiers (UFID)
  - Directory service: map names to UFIDs
  - Client module
    - integrate/extend flat file and directory services
    - provide a common application programming interface (can emulate different file interfaces)
    - stores location of flat file and directory services

Flat file service interface

- RPC used by client modules, not by user-level programs
- Compared to UNIX
  - no open/close
    - Create is not idempotent
    - at-least-once semantics
    - reexecution gets a new file
  - specify starting location in Read/Write
    - stateless server

Read(FileId, i, n) -> Data —throws *BadPosition*	If $1 \leq i \leq Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) —throws *BadPosition*	If $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() -> FileId	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

Figure 12.6 Flat file service operations

Access control

- UNIX checks access rights when a file is opened
  - subsequent checks during read/write are not necessary
- distributed environment
  - server has to check
  - stateless approaches
    1. access check once when UFID is issued
       - client gets an encoded "capability" (who can access and how)
       - capability is submitted with each subsequent request
    2. access check for each request.
  - second is more common

Directory service operations

- A directory service interface translates text names to file identifiers and performs a number of other services such as those listed among the sample commands in figure 1.7. This is the remote procedure call interface to extend the local directory services to a distributed model.

Lookup(Dir, Name) -> FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds (*Name*, *File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory: the entry containing *Name* is removed from the directory. If *Name* is not in the directory: throws an exception.
GetNames(Dir, Pattern) -> NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*.

Figure 1.7 Directory service operations

File collections

- Hierarchical file system
  - Directories containing other directories and files
  - Each file can have more than one name (pathnames)
    - how in UNIX, Windows?

- File groups
  - a logical collection of files on one server
    - a server can have more than one group
    - a group can change server
    - a file can't change to a new group (copying doesn't count)
    - filesystems in unix
    - different devices for non-distributed
    - different hosts for distributed

32 bits 16 bits file group identifier IP address date

1.3 Case Study: Sun NFS

- Sun NFS
  - Industry standard for local networks since the 1980's
  - OS independent (originally unix implementation)
  - rpc over udp or tcp

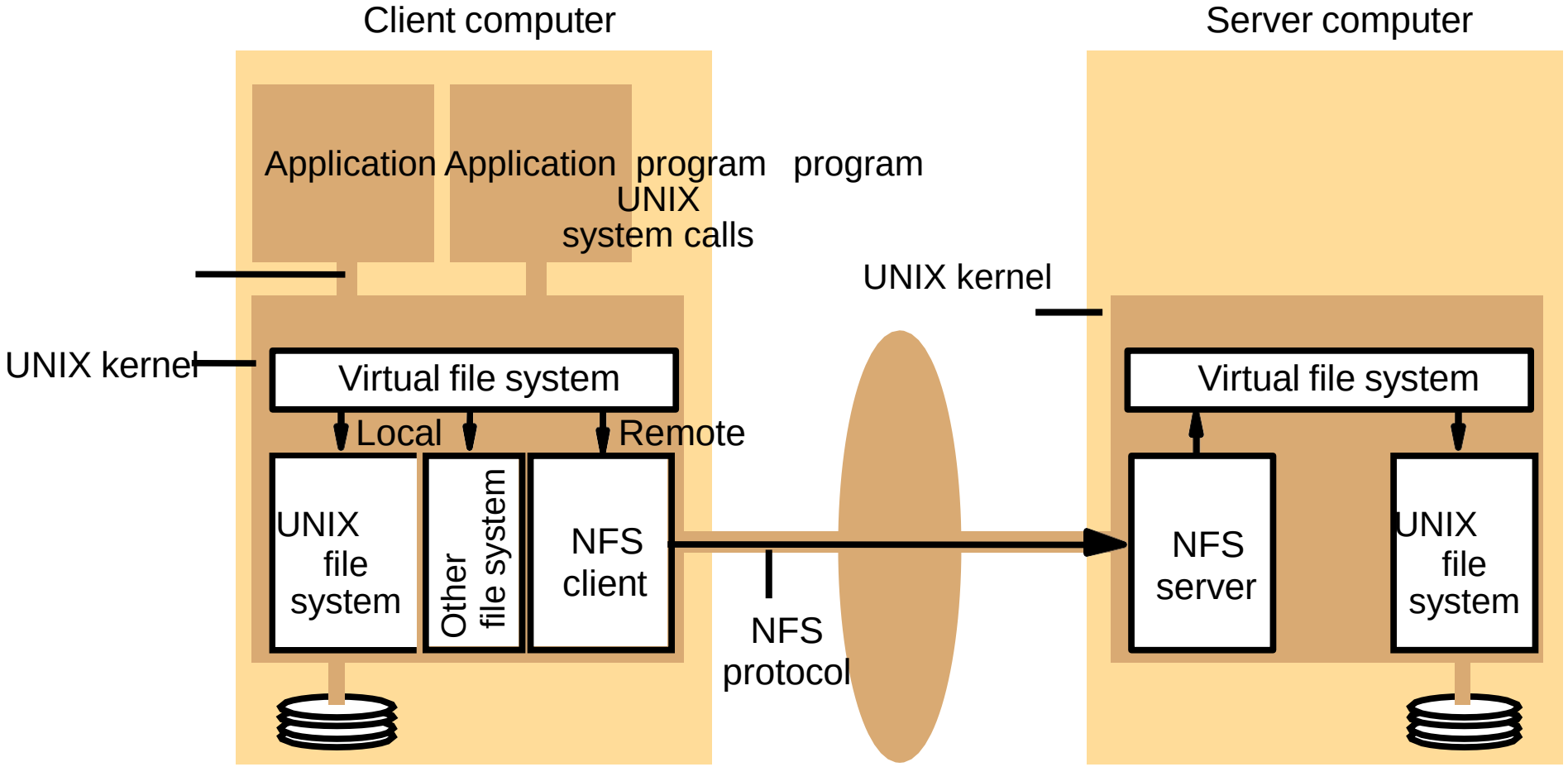
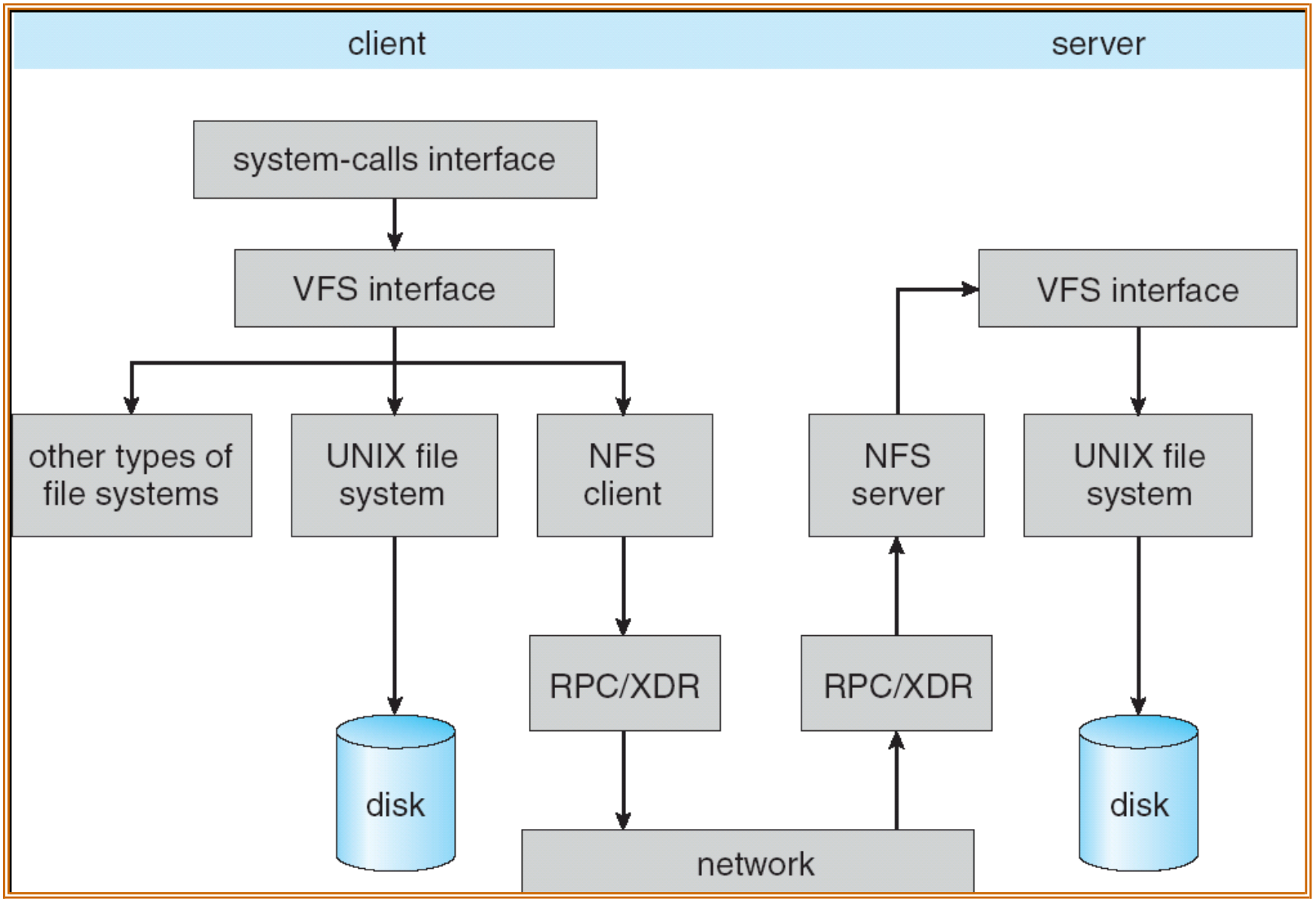


Figure 1.8 NFS architecture

Schematic view of NFS architecture



Refer to Section 1.9 of the OS Textbook

Virtual file system

- Part of unix kernel to support access transparency
- NFS file handles, 3 components:
  - i-node (index node)
    - structure for finding the file
  - filesystem identifier
    - different groups of files
  - i-node generation number
    - i-nodes are reused
    - incremented when reused
- VFS
  - struct for each file system
  - v-node for each open file
    - file handle for remote file
    - i-node number for local file

Access control, client integration, pathname translation

- Access control
  - nfs server is stateless, doesn't keep open files for clients
  - server check identity each time (uid and gid)
- Client integration
  - nfs client emulates Unix file semantics
  - in the kernel, not in a library, because:
    - access files via system calls
    - single client module for multiple user processes
    - encryption can be done in the kernel
- Pathname translation
  - pathname: /users/students/doc/abc
  - server doesn't receive the entire pathname for translation, why?
  - client breaks down the pathnames into parts
  - iteratively translate each part
  - translation is cached

Figure 1.9 server operations (simplified)

lookup(dirfh, name) -> fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) -> newfh, attr	Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) status	Removes file name from directory *dirfh*.
getattr(fh) -> attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) -> attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) -> attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) -> attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) -> status	Changes the name of file *name* in directory *dirfh* to *toname* in directory to *todir*. *fh*
link(newdirfh, newname, dirfh, name) -> status	Creates an entry *newname* in the directory *newdirfh* which refers to file *name* in the directory *dirfh*.

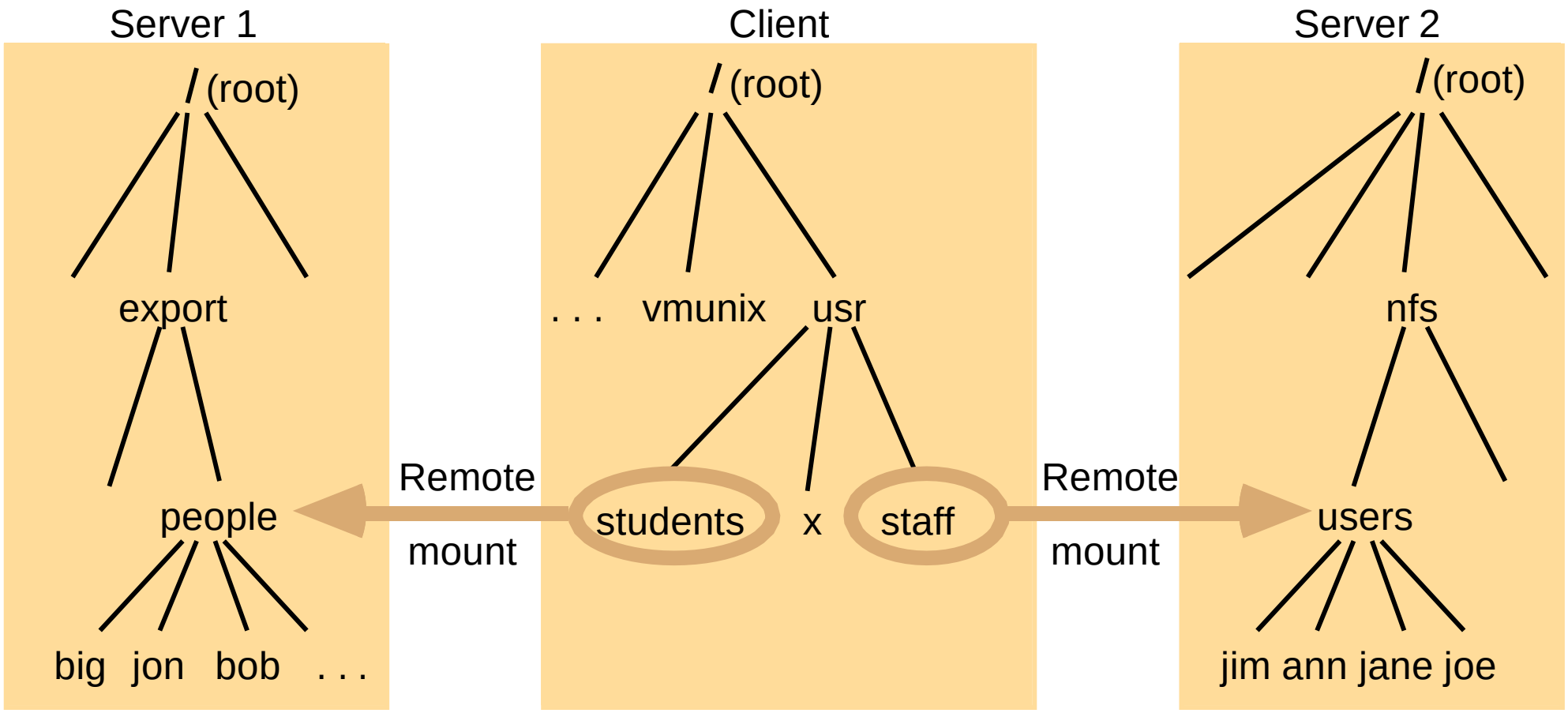
Figure 1.9 NFS server operations (simplified) *cont.*

symlink(newdirfh, newname, string) -> status	Creates an entry *newname* in the directory *newdirfh* of type symbolic link with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) -> string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) -> newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) -> status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) -> entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) -> fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

Mount

- Mount service
  - mounting: the process of including a new filesystem
  - /etc/exports has filesystems that can be mounted by others
  - clients use a modified mount command for remote filesystems
  - communicates with the mount process on the server in a mount protocol
  - hard-mounted
    - user process is suspended until request is successful
    - when server is not responding
    - request is retried until it's satisfied
  - soft-mounted
    - if server fails, client returns failure after a small # of retries
    - user process handles the failure
- Automounter
  - what if a user process reference a file on a remote filesystem that is not mounted
  - table of mount points (pathname) and servers
  - NFS client sends the reference to the automounter
  - automounter check find the first server that is up
  - mount it at some location and set a symbolic link (original impl)
  - mount it at the mount point (later impl)
  - could help fault tolerance, the same mount point with multiple replicated servers.

Accessible file systems on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Figure 1.10 Local and remote file systems accessible on an NFS client

Caching

- Server caching
  - caching file pages, directory/file attributes
  - read-ahead: prefetch pages following the most-recently read file pages
  - delayed-write: write to disk when the page in memory is needed for other purposes
  - "sync" flushes "dirty" pages to disk every 30 seconds
  - two write option
    1. write-through: write to disk before replying to the client
    2. cache and commit:
       - stored in memory cache
       - write to disk before replying to a "commit" request from the client
- Client caching
  - caches results of read, write, getattr, lookup, readdir
  - clients responsibility to poll the server for consistency

Client caching: reading

- timestamp-based methods for consistency validation
  - *Tc*: time when the cache entry was last validated
  - *Tm*: time when the block was last modified at the server
- cache entry is valid if:
  1. *T - Tc < t*, where *t* is the freshness interval
     - *t* is adaptively adjusted:
       - files: 3 to 30 seconds depending on freq of updates
       - directories: 30 to 60 seconds
  2. $Tm_{client} = Tm_{server}$
- cache validation
  - need validation for all cache accesses due to no share check
  - condition “1" can be determined by the client alone--performed first
  - Reducing getattr() to the server [for getting Tm_{server}]
    1. new value of Tm_{server} is received, apply to all cache entries from the same file
    2. piggyback getattr() on file operations
    3. adaptive alg for update *t*
  - validation doesn't guarantee the same level of consistency as one-copy

Client caching: writing

- dirty: modified page in cache
- flush to disk: file is closed or sync from client
- bio-daemon (block input-output)
  - read-ahead: after each read request, request the next file block from the server as well
  - delayed write: after a block is filled, it's sent to the server
  - reduce the time to wait for read/write

Other optimization

- UDP packet is extended to 9KB to containing entire file block (8KB for UNIX BSD FFS) and RPC message in a single packet
  - Clients and servers of NFSv3 can negotiate sizes larger than 8 KB
- Piggybacked
  - File status information cached at clients must be updated at least every 3 seconds for active files
  - All operations that refer to files or directories are taken as implicit *getattr* requests, and the current attribute values are piggybacked along with the other results of the operation

Security, Performance

- Security
  - stateless nfs server
  - user's identity in each request
  - Kerberos authentication during the mount process, which includes uid and host address
  - server maintain authentication info for the mount
  - on each file request, nfs checks the uid and address
  - one user per client
- Performance
  - overhead/penalty is low
  - main problems
    - frequent getattr() for cache validation (piggybacking)
    - relatively poor performance is write-through is used on the server (delay-write/commit in current versions)
  - write < 5%
  - lookup is almost 50% (step by step pathname translation)

Summary for NFS

An excellent example of a simple, robust, high-performance distributed service

- access transparency: same system calls for local or remote files
- location transparency: could have a single name space for all files (depending on all the clients to agree the same name space)
- mobility transparency: mount table need to be updated on each client (not transparent)
- scalability: can usually support large loads, add processors, disks, servers...
- file replication: read-only replication, no support for replication of files with updates
- hardware and OS: many ports
- fault tolerance: stateless and idempotent
- consistency: not quite one-copy for efficiency
- security: added encryption--Kerberos
- efficiency: pretty efficient, wide-spread use

1.4 Case Study: the Andrew File System

- Goal: provide transparent access to remote shared files
  - like NFS and compatible with NFS
- Differing from NFS
  - Primarily attributable to the scalability
    - Caching of whole files in client nodes
- Two unusual design characteristics
  - Whole file serving: the entire contents of directories and files are transmitted to client computers
    - 64KB chunks in AFS3
  - Whole-file caching: clients permanently cache a copy of a file or a chunk on its local disk

Scenario of AFS

- Open a new shared remote file
  - A user process issues open() for a file not in the local cache and then sends a request to the server
  - The server returns the requested file
  - The copy is stored in the client's local UNIX file system and the resulting UNIX file descriptor is returned to the client
- Subsequent read, write and other operations on the file are applied to the local copy
- When the process in the client issues close()
  - if the local copy has been updated, its contents are sent back to the server
    - server updates the contents and the timestamps on the file
    - the copy on the client's local disk is retained

Characteristics

- Good for shared files likely to remain valid for long periods
  - –infrequently updated
  - –normally accessed by only a single user Overwhelming majority of file accesses
- Local cache can be allocated a substantial proportion of the disk space
  - –should be enough for a working set of files used by one user
- Assumptions about average and maximum file size and reference locality
  - –Files are small; most are less than 10KB in size
  - –Read operations are much more common than writes
  - –Sequential access is much more common than random access
  - –Most files are written by only one user. When a file is shared, it is usually only one user who modified it
  - –Files are referenced in bursts. A file referenced recently is very probably referenced soon.
- Maybe good for distributed database applications

Venus and Vice: software components in AFS

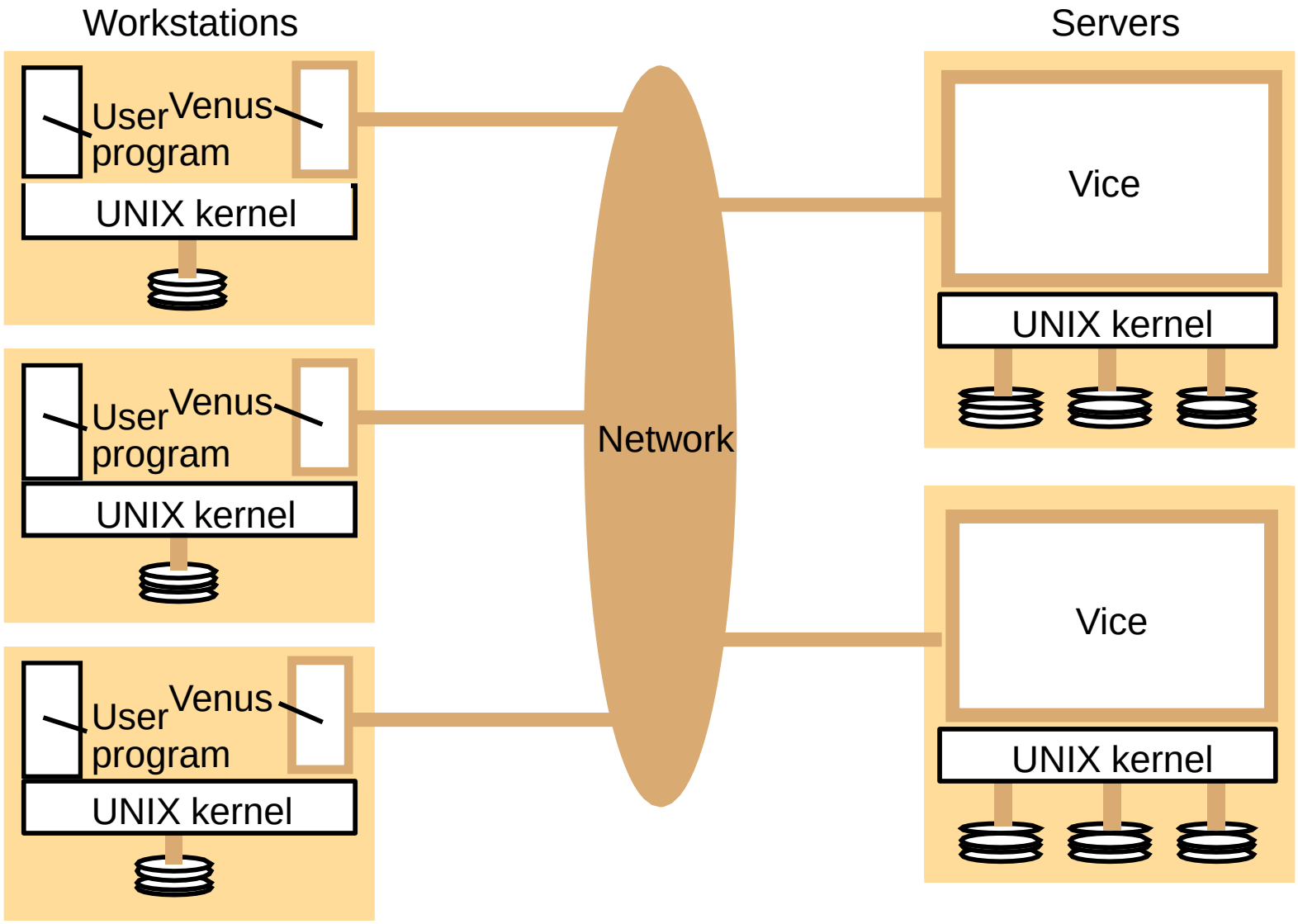
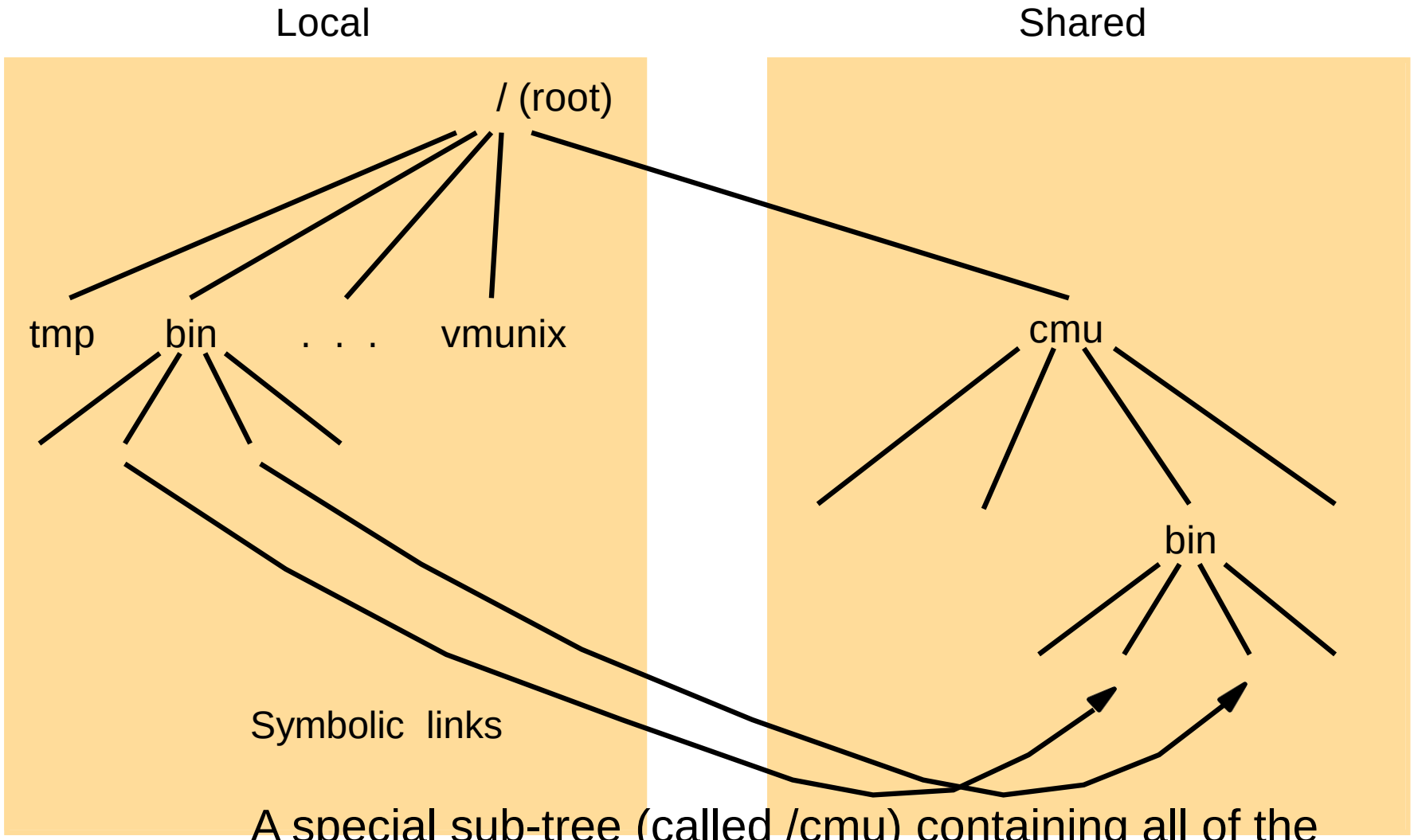


Figure 1.11 Distribution of processes in the Andrew File System

File name space seen by clients of AFS



A special sub-tree (called /cmu) containing all of the shared files

Figure 1.12 File name space seen by clients of AFS

System call interception in AFS

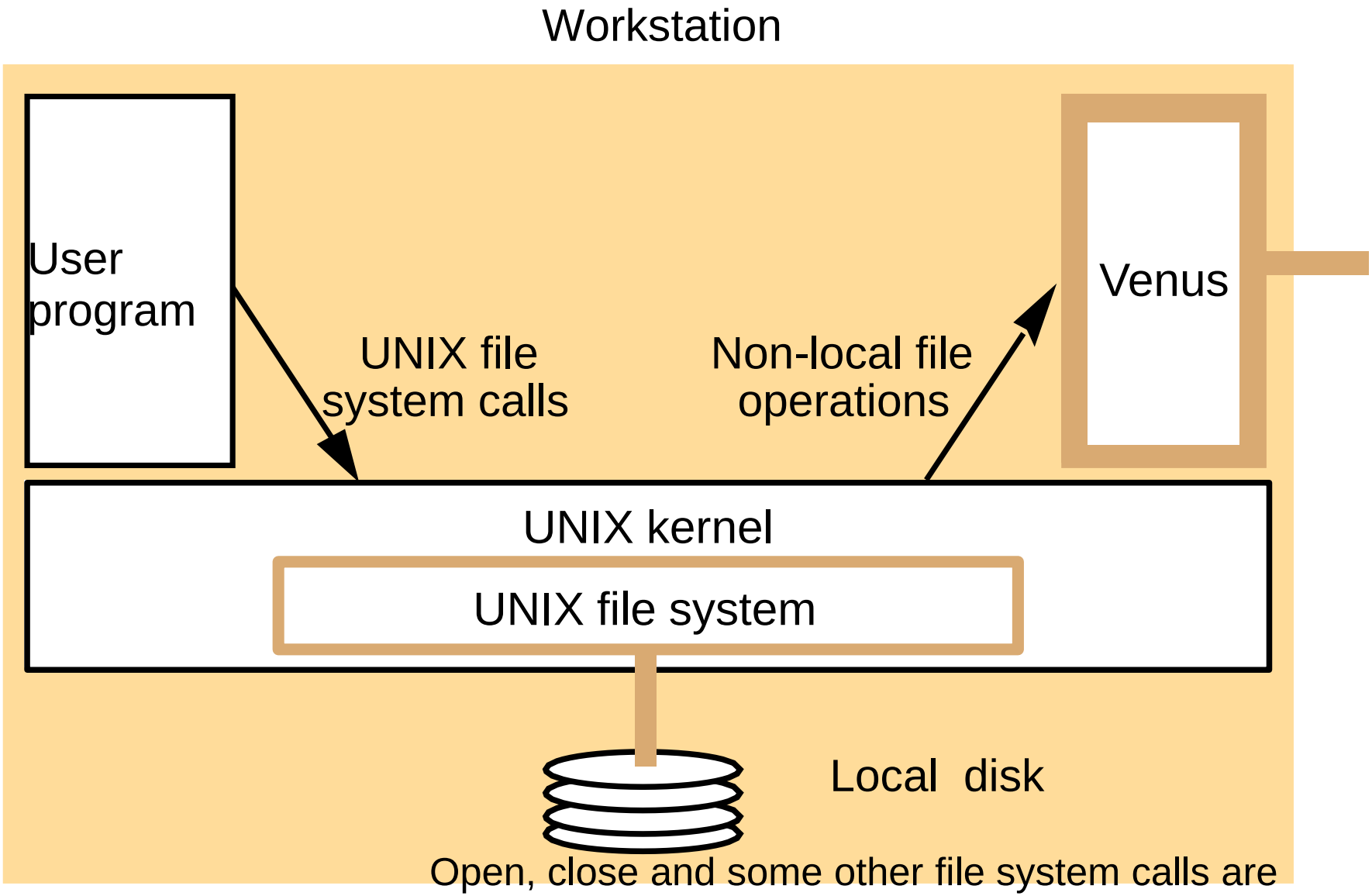


Figure 1.13 System call interception in AFS

Figure 1.14

Implementation of file system calls in AFS

User process	Unix Kernel	Venus	Net	Vice
open(FileName, mode)	If FileName refers to a file in shared file space, pass the request to Venus. Open the local file and return the file descriptor to the application.	Check list of files in local cache. If not present or there is no valid ***callback promise***, send a request for the file to the Vice server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	→ ← ←	Transfer a copy of the file and a ***callback promise*** to the workstation. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice serve r that is the custodian of the file.	→	Replace the file contents and send a ***callback*** to all other clients holding callback promises on the file.

Main components of Vice service interface

Fetch(fid) -> attr, data	Returns the attributes (status) and, optionally, the contents of file identified by the *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() -> fid	Creates a new file and records a callback promise on it. Deletes the
Remove(fid)	specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

Figure 1.15

1.5 Enhancements and further developments

NFS enhancements

WebNFS - NFS server implements a web-like service on a well-known port. Requests use a 'public file handle' and a pathname-capable variant of *lookup().* Enables applications to access NFS servers directly, e.g. to read a portion of a large file.

One-copy update semantics (Spritely NFS, NQNFS) - Include an *open()* operation and maintain tables of open files at servers, which are used to prevent multiple writers and to generate callbacks to clients notifying them of updates. Performance was improved by reduction in *gettattr()* traffic.

Improvements in disk storage organisation

RAID - improves performance and reliability by striping data redundantly across several disk drives

Log-structured file storage - updated pages are stored contiguously in memory and committed to disk in large contiguous blocks (~ 1 Mbyte). File maps (in memory with a persistent backup) are modified whenever an update occurs. Garbage collection to recover disk space.

New design approaches

- Distribute file data across several servers
  - Exploits high-speed networks (ATM, Gigabit Ethernet)
  - Layered approach, lowest level is like a 'distributed virtual disk'
  - Achieves scalability even for a single heavily-used file
- 'Serverless' architecture
  - Exploits processing and disk resources in all available network nodes
    - Service is distributed at the level of individual files
  - Examples:
    - xFS (section 12.5): Experimental implementation demonstrated a substantial performance gain over NFS and AFS
    - Frangipani (section 12.5): Performance similar to local UNIX file access
    - Tiger Video File
    - P2P systems: Napster, OceanStore (UCB), Farsite (MSR), Publius (AT&T research) - see web for documentation on these very recent systems
- Replicated read-write files
  - High availability
  - Disconnected working
    - re-integration after disconnection is a major problem if conflicting updates have occurred
  - Examples:
    - Bayou system (Section 14.4.2)
    - Coda system (Section 14.4.3)

Chapter 2
Peer-to-Peer Systems

Chapter 2 Peer-to-Peer Systems

1. Introduction
2. Napster and its legacy
3. Peer-to-peer middleware
4. Routing overlays
5. Overlay case studies: Pastry, Tapestry
6. Application case studies
   - Squirrel web cache
   - OceanStore file store
   - Ivy file system
7. Summary

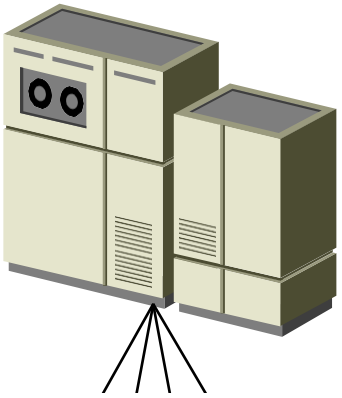
2.1 Introduction

Paradigm Shift of Computing System Models

1980~

Terminal-Mainframe

(Super-computing)



RS-232

VT100/DOS

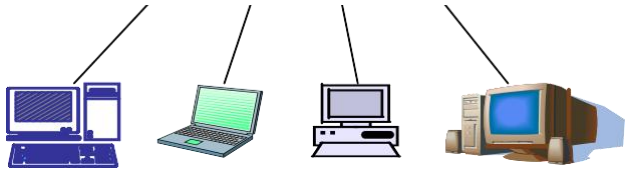
1990~

Client-Server

(Micro-computing

/Personal Computer)

Dialup/10M Ethernet

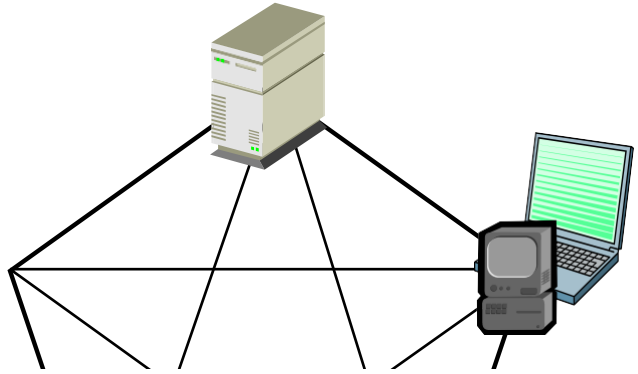


Windows 31/95

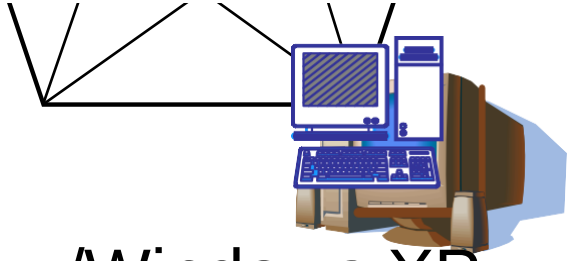
2000~

Peer-to-Peer

(Macro-computing)



ADSL/100M+ Ethernet

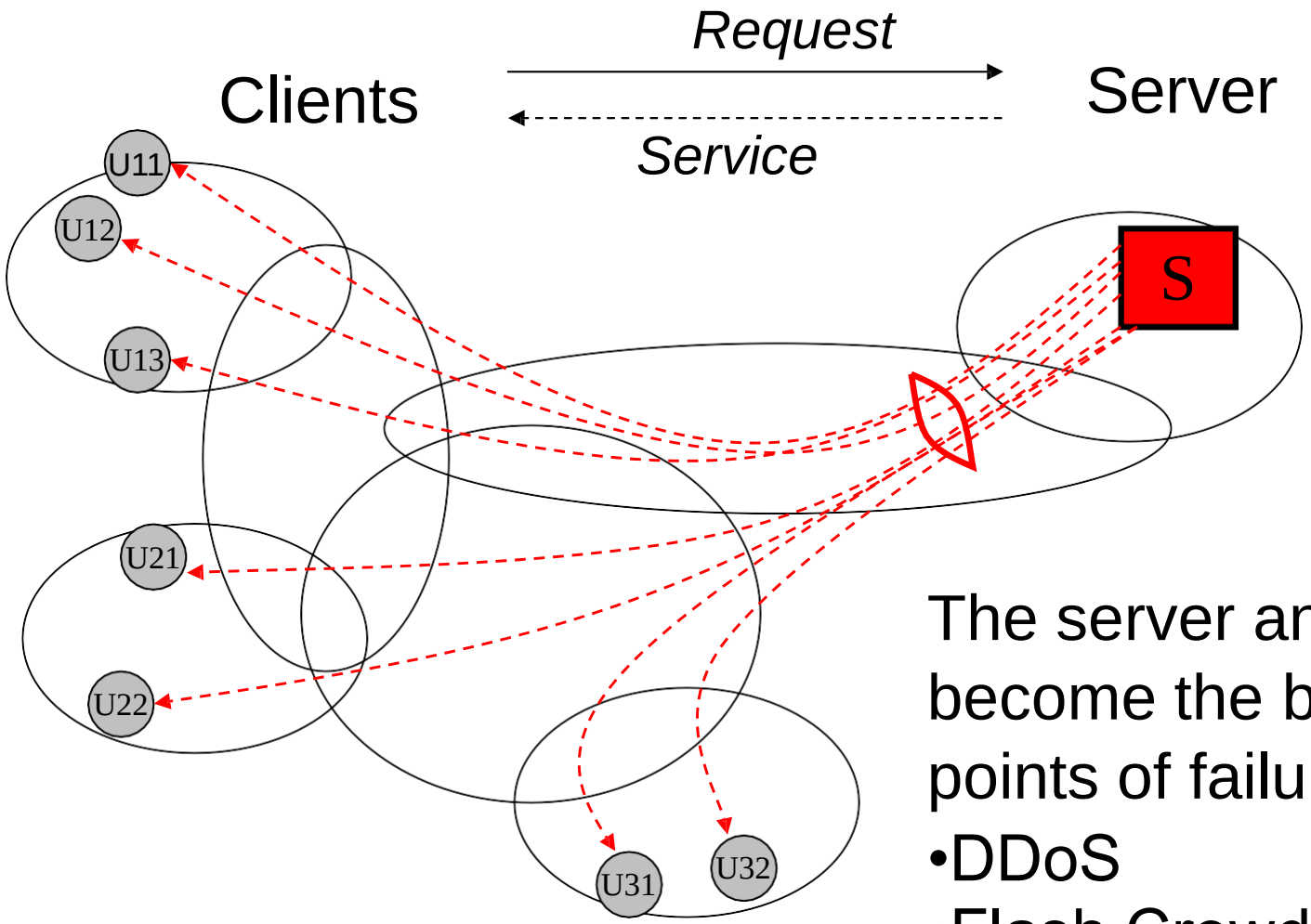


Linux/Windows XP

Figure 2.1

Client-server Model

Clients and servers each with distinct roles



The server and the network become the bottlenecks and points of failure

- DDoS
- Flash Crowd

Figure 2.2

Peer-to-peer Model

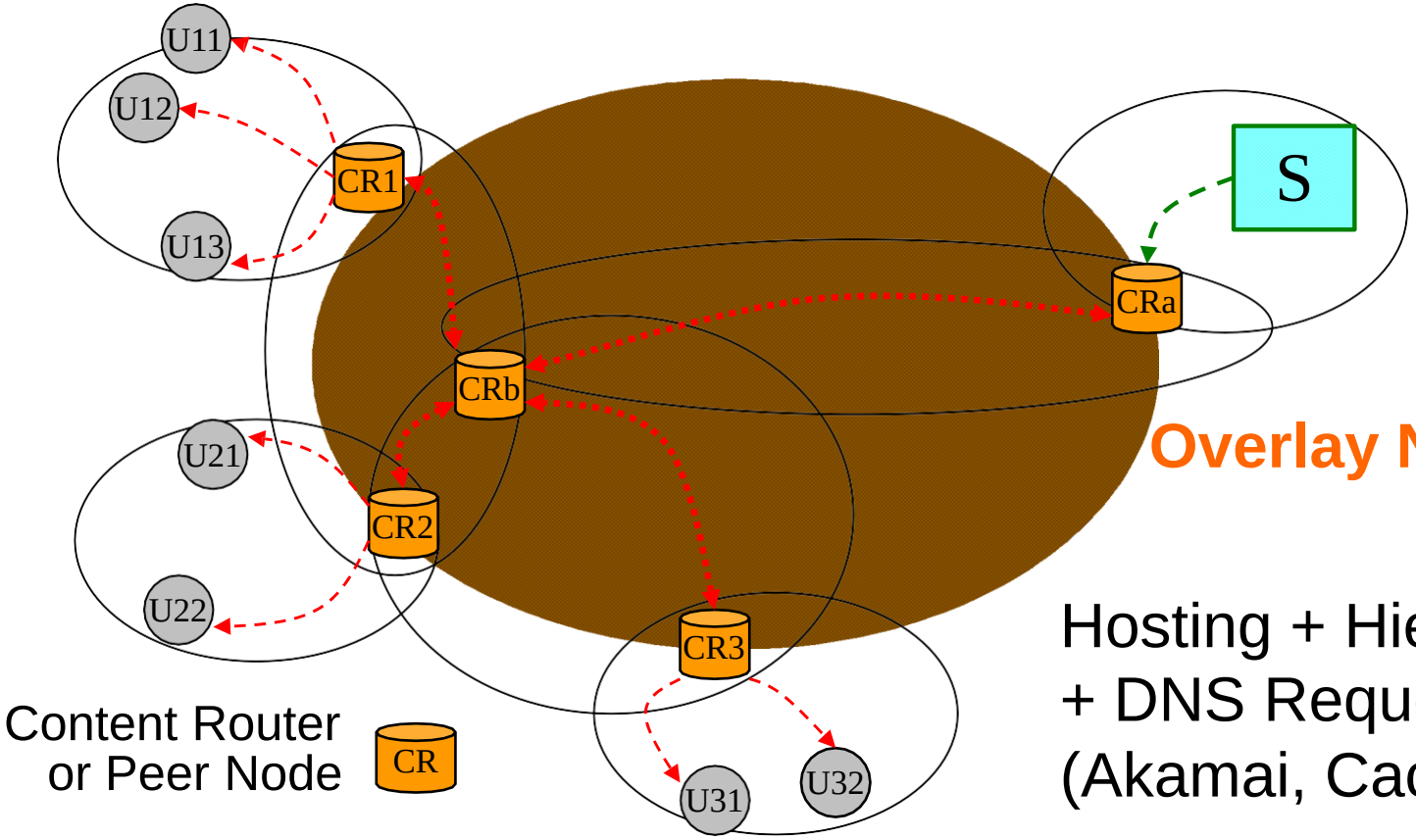
“ Peer-to-Peer (P2P) is a way of structuring distributed applications such that the individual nodes have symmetric roles. Rather than being divided into clients and servers each with quite distinct roles, in P2P applications a node may act as both a client and a server.”

Excerpt from the Charter of Peer-to-Peer Research Group, IETF/IRTF, June 24, 2003

http://www.irtf.org/charters/p2prg.html

Peers play similar roles No distinction of responsibilities

Content Distribution Networks



Hosting + Hierarchical Proxies
+ DNS Request Routing
(Akamai, CacheFlow, etc.)

Name: lb1.www.ms.akadns.net
Addresses: 207.46.20.60, 207.46.18.30, 207.46.19.30, 207.46.19.60, 207.46.20.30
Aliases: **www.microsoft.com**, toggle.www.ms.akadns.net, g.www.ms.akadns.net

Name: www.yahoo.akadns.net
Addresses: 66.94.230.33, 66.94.230.34, 66.94.230.35, 66.94.230.39, 66.94.230.40, …
Aliases: **www.yahoo.com**

Name: e96.g.akamaiedge.net 202.177.217.122
Address: **www.gio.gov.tw**, www.gio.gov.tw.edgekey.net
Aliases:

Name: a1289.g.akamai.net 203.133.9.9, 203.133.9.11
Addresses: **www.whitehouse.gov**, www.whitehouse.gov.edgesuite.net
Aliases:

Figure 2.3

Characteristics of P2P Systems

- Ensures that each user contributes resources to the system
- All the nodes have the same functional capabilities and responsibilities
- Their correct operation does not depend on the existence of any centrally-administered systems
- They can be designed to offer a limited degree of anonymity to the providers and users of resources
- A key issue: placement of data across many hosts
  - efficiency
  - load balance
  - availability

Generations

- Early services
  - DNS, Netnews/Usenet
  - Xerox Grapevine name/mail service
  - Lamport's part-time parliament algorithm
  - Bayou replicated storage system
  - classless inter-domain IP routing algorithm
- 1st generation – centralized search
  - Napster
- 2nd generation – decentralized, unstructured
  - Freenet, Gnutella, Kazaa, BitTorrent
- 3rd generation – decentralized, structured
  - P2P middleware: Pastry, Tapestry, CAN, Chord, Kademlia

Example of Centralized P2P Systems: Napster

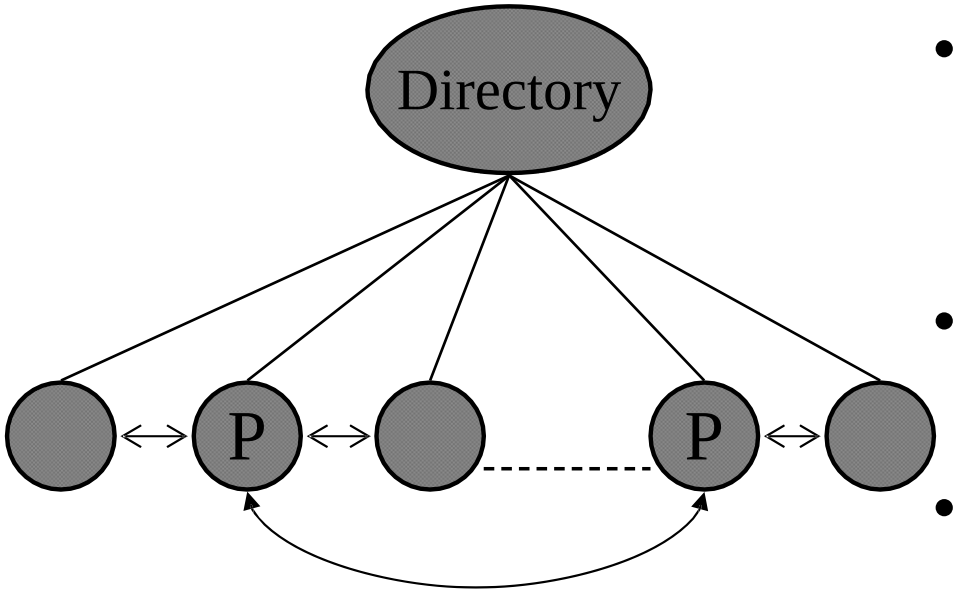
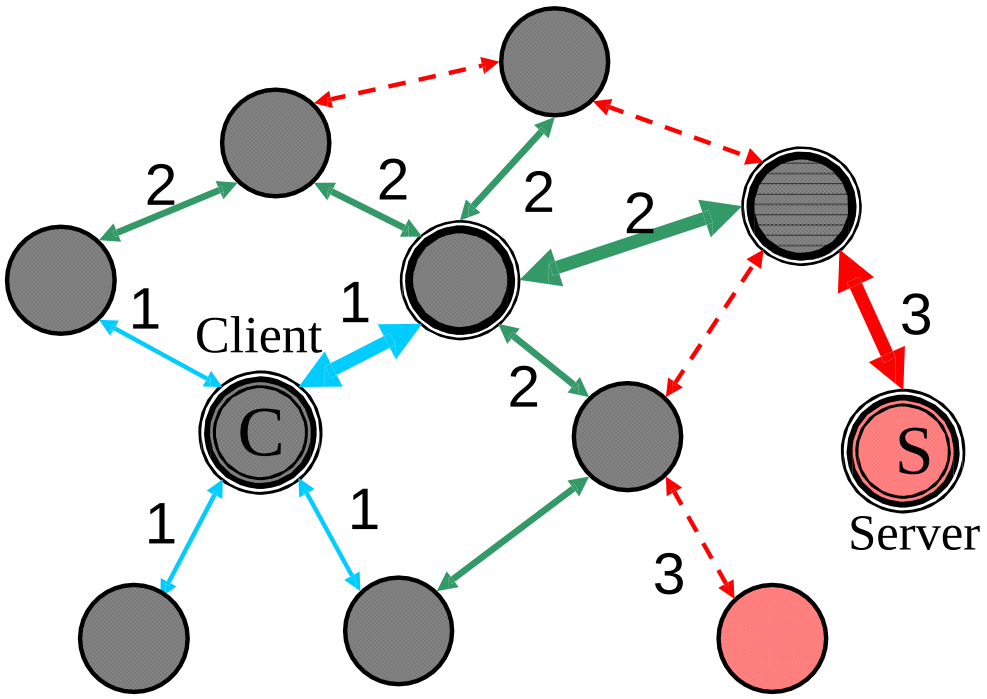


Figure 2.4

- Announced in January 1999 by Shawn Fanning for sharing MP3 files and pulled plug in July 2001
- Centralized server for search, direct file transfer among peer nodes
- Proprietary client-server protocol and client-client protocol
- Relying on the user to choose a ‘best’ source
- Disruptive, proof of concepts
- IPR and firewall issues

Example of Decentralized P2P Systems: Gnutella

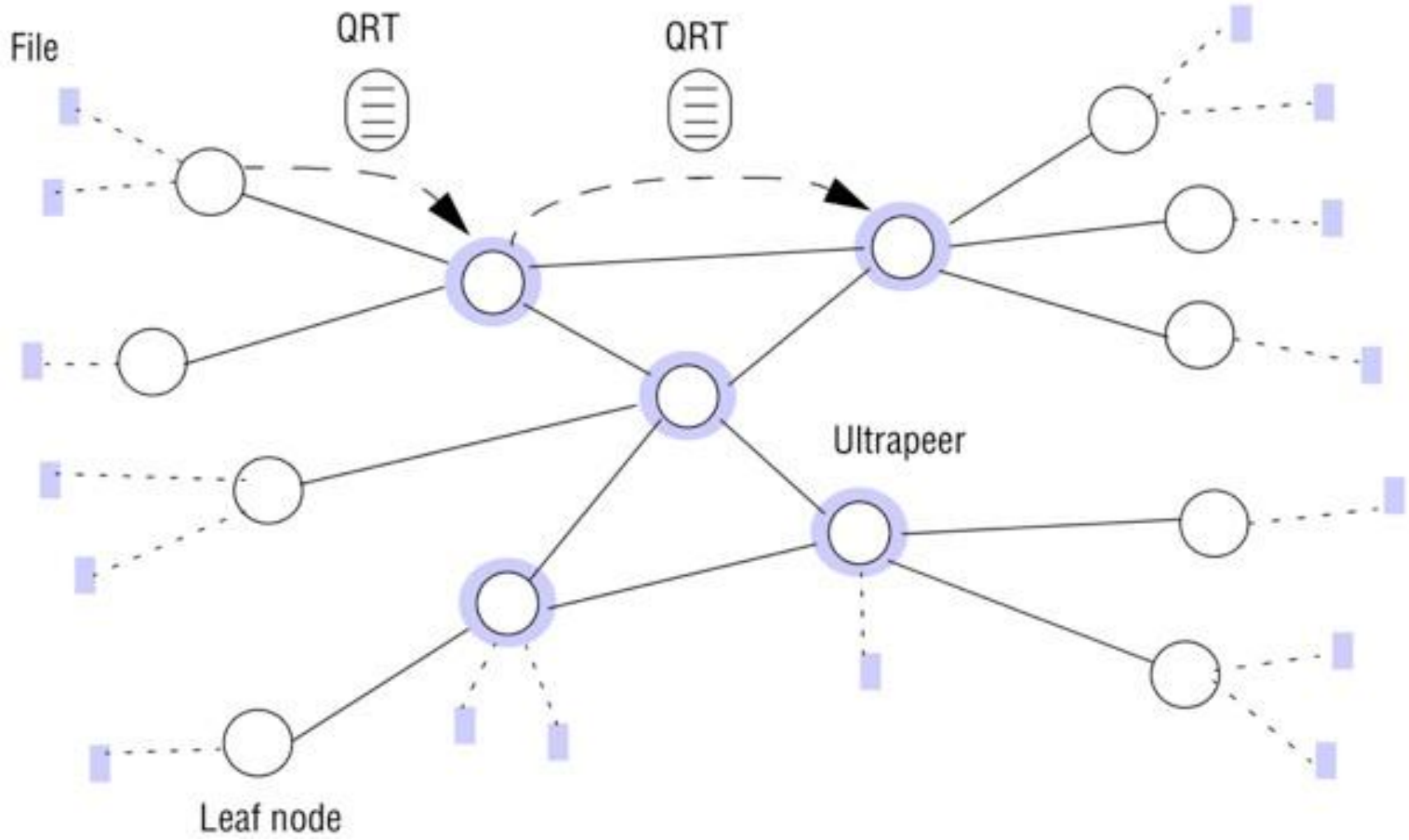


Servent (=***Serv*** er + Cli ***ent***)

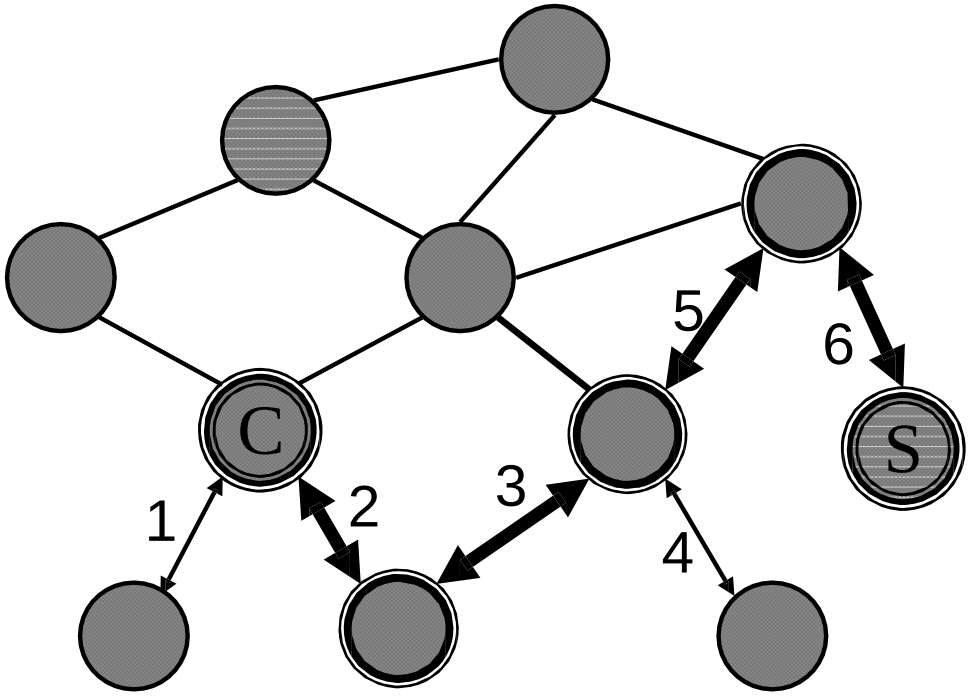
Figure 2.5

- Open source
- 3/14/2000: Released by NullSoft/AOL, almost immediately withdrawn, and became open source
- Message flooding: *serverless, decentralized* search by message broadcast, direct file transfer using HTTP
- Limited-scope query

Figure 2.6: Key elements in the Gnutella protocol



Example of Unstructured P2P Systems: Freenet



- Ian Clarke, Scotland, 2000
- Distributed depth-first search, Exhaustive search
- File hash key, lexicographically closest match
- *Store-and-forward* file transfer
- Anonymity
- Open source

Figure 2.7

Example of Hybrid P2P Systems: FastTrack / KaZaA

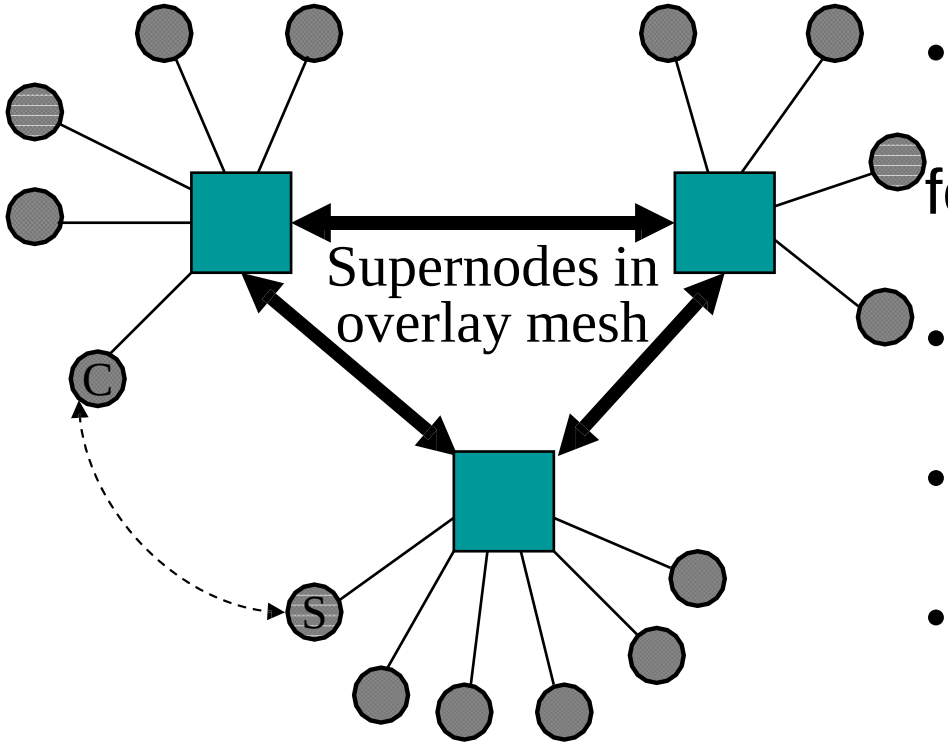
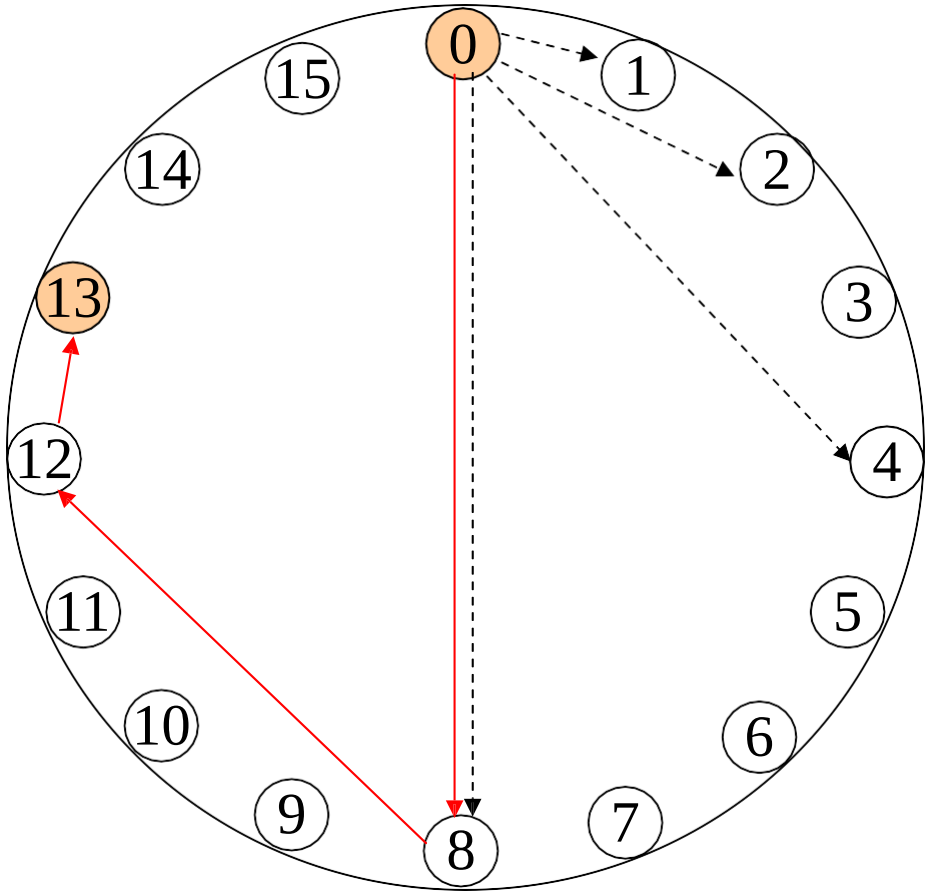


Figure 2.8

- Proprietary software developed by FastTrack in Amsterdam and licensed to many companies
- Summer 2001, Sharman networks, founded in Vanuatu, acquires FastTrack
- Hierarchical supernodes (Ultra-peers)
- Dedicated authentication server and supernode list server
- From user's perspective, it's like Google.
- Encrypted files and control data transported using HTTP
- Parallel download
- Automatically switch to new server

Example of Structured P2P Systems: Chord



- Frans Kaashoek, et. al., MIT, 2001
- IRIS: Infrastructure for Resilient

Internet Systems, 2003

- Distributed Hashing Table
- Scalable lookup service
- Hyper-cubic structure

Figure 2.9

Figure 2.10: Structured versus unstructured peer-to-peer systems

	Structured peer-to-peer	*Unstructured peer-to-peer*
Advantages	Guaranteed to locate objects (assuming they exist) and can offer time and complexity bounds on this operation; relatively low message overhead.	Self-organizing and naturally resilient to node failure.
Disadvantages	Need to maintain often complex overlay structures, which can be difficult and costly to achieve, especially in highly dynamic environments.	Probabilistic and hence cannot offer absolute guarantees on locating objects; prone to excessive messaging overhead which can affect scalability.

Benefits from P2P

- Theory
  - Dynamic discovery of information
  - Better utilization of bandwidth, processor, storage, and other resources
  - Each user contributes resources to network
- Practice examples
  - Sharing browser cache over 100Mbps lines
  - Disk mirroring using spare capacity
  - Deep search beyond the web

Figure 2.11 Distinctions between IP and overlay routing for P2P applications

	IP	*Application-level routing overlay*
Scale	IPv4 is limited to 232 addressable nodes. The IPv6 name space is much more generous (2128), but addresses in both versions are hierarchically structured and much of the space is pre-allocated according to administrative requirements.	Peer-to-peer systems can address more objects. The GUID name space is very large and flat (>2128), allowing it to be much more fully occupied.
Load balancing	Loads on routers are determined by network topology and associated traffic patterns.	Object locations can be randomized and hence traffic patterns are divorced from the network topology.
Network dynamics (addition/deletion of objects/nodes) Fault tolerance	IP routing tables are updated asynchronously on a best-efforts basis with time constants on the order of 1 hour. Redundancy is designed into the IP network by its managers, ensuring tolerance of a single router or network connectivity failure. *n*-fold replication is costly.	Routing tables can be updated synchronously or asynchronously with fractions of a second delays. Routes and object references can be replicated *n*-fold, ensuring tolerance of *n* failures of nodes or connections.
Target identification	Each IP address maps to exactly one target node.	Messages can be routed to the nearest replica of a target object.
Security and anonymity	Addressing is only secure when all nodes are trusted. Anonymity for the owners of addresses is not achievable.	Security can be achieved even in environments with limited trust. A limited degree of anonymity can be provided.

Distributed Computation

- Only a small portion of the CPU cycles of most computers is utilized. Most computers are idle for the greatest portion of the day, and many of the ones in use spend the majority of their time waiting for input or a response.
- A number of projects have attempted to use these idle CPU cycles. The best known is the SETI@home project, but other projects including code breaking have used idle CPU cycles on distributed machines.

- SETI@home project: a scientific experiment that uses Internet-connected computers to analyze radio telescope data in the Search for Extraterrestrial Intelligence (SETI) http://setiathome.ssl.berkeley.edu/

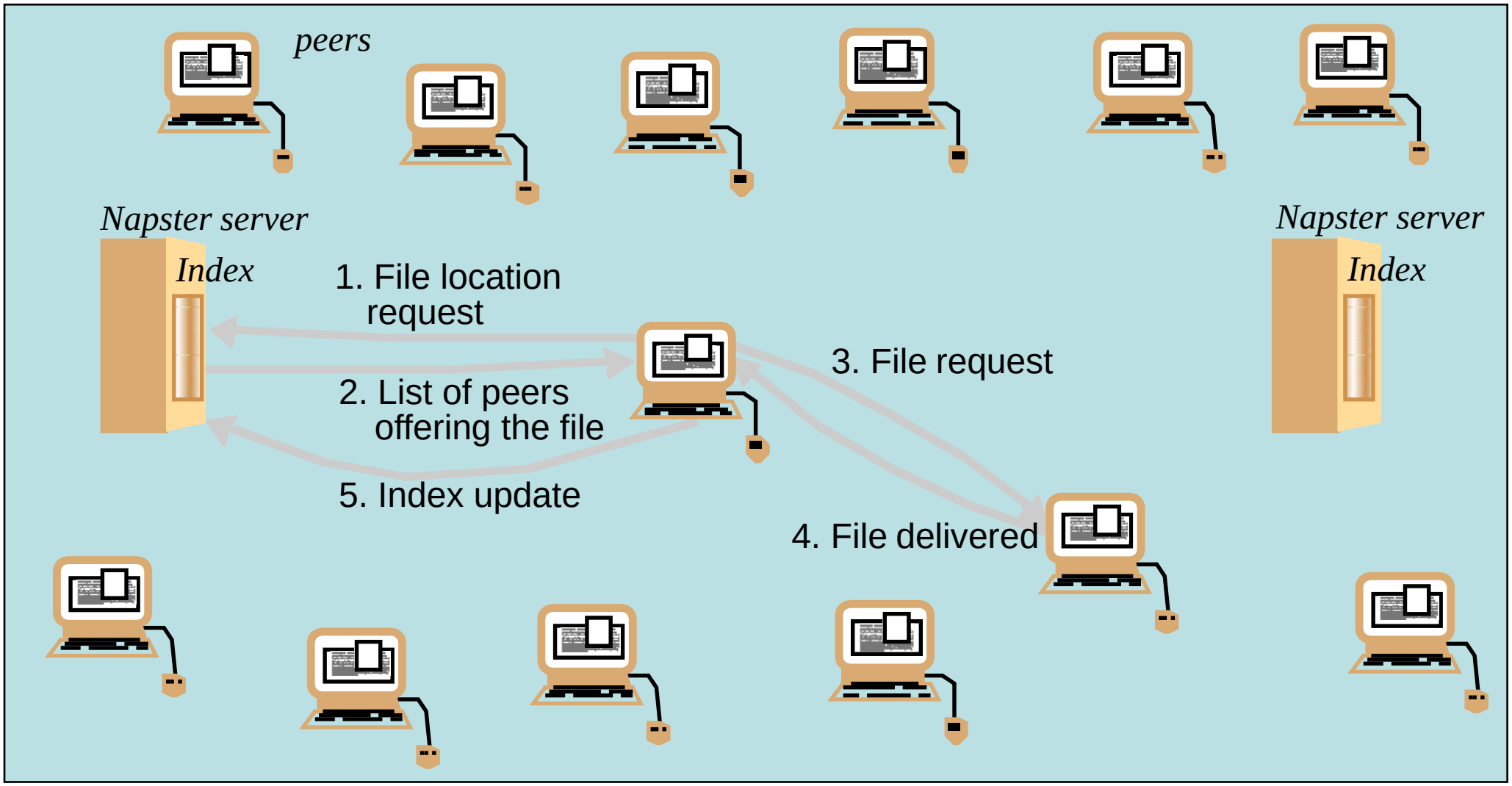
Dangers and Attacks on P2P

- Poisoning (files with contents different to description)
- Polluting (inserting bad packets into the files)
- Defection (users use the service without sharing)
- Insertion of viruses (attached to other files)
- Malware (originally attached to the files)
- Denial of Service (slow down or stop the network traffic)
- Filtering (some networks don’t allow P2P traffic)
- Identity attacks (tracking down users and disturbing them)
- Spam (sending unsolicited information)

2.2 Napster and its legacy

- The first large scale peer-to-peer network was Napster, set up in 1999 to share digital music files over the Internet. While Napster maintained centralized (and replicated) indices, the music files were created and made available by individuals, usually with music copied from CDs to computer files. Music content owners sued Napster for copyright violations and succeeded in shutting down the service. Figure 10.2 documents the process of requesting a music file from Napster.

Figure 2.12 Napster: peer-to-peer file sharing



Napster: Lessons Learned

- Napster created a network of millions of people, with thousands of files being transferred at the same time.
- There were quality issues. While Napster displayed link speeds to allow users to choose faster downloads, the fidelity of recordings varied widely.
- Since Napster users were parasites of the recording companies, there was some central control over selection of music. One benefit was that music files did not need updates.

•There was no guarantee of availability for a particular item of music.

2.3 Middleware for Peer-to-Peer

- A key problem in Peer-to-Peer applications is to provide a way for clients to access data resources efficiently.
  - Similar needs in client/server technology led to solutions like NFS.
  - However, NFS relies on pre-configuration and is not scalable enough for peer-to-peer.
- Peer clients need to locate and communicate with any available resource, even though resources may be widely distributed and configuration may be dynamic, constantly adding and removing resources and connections.

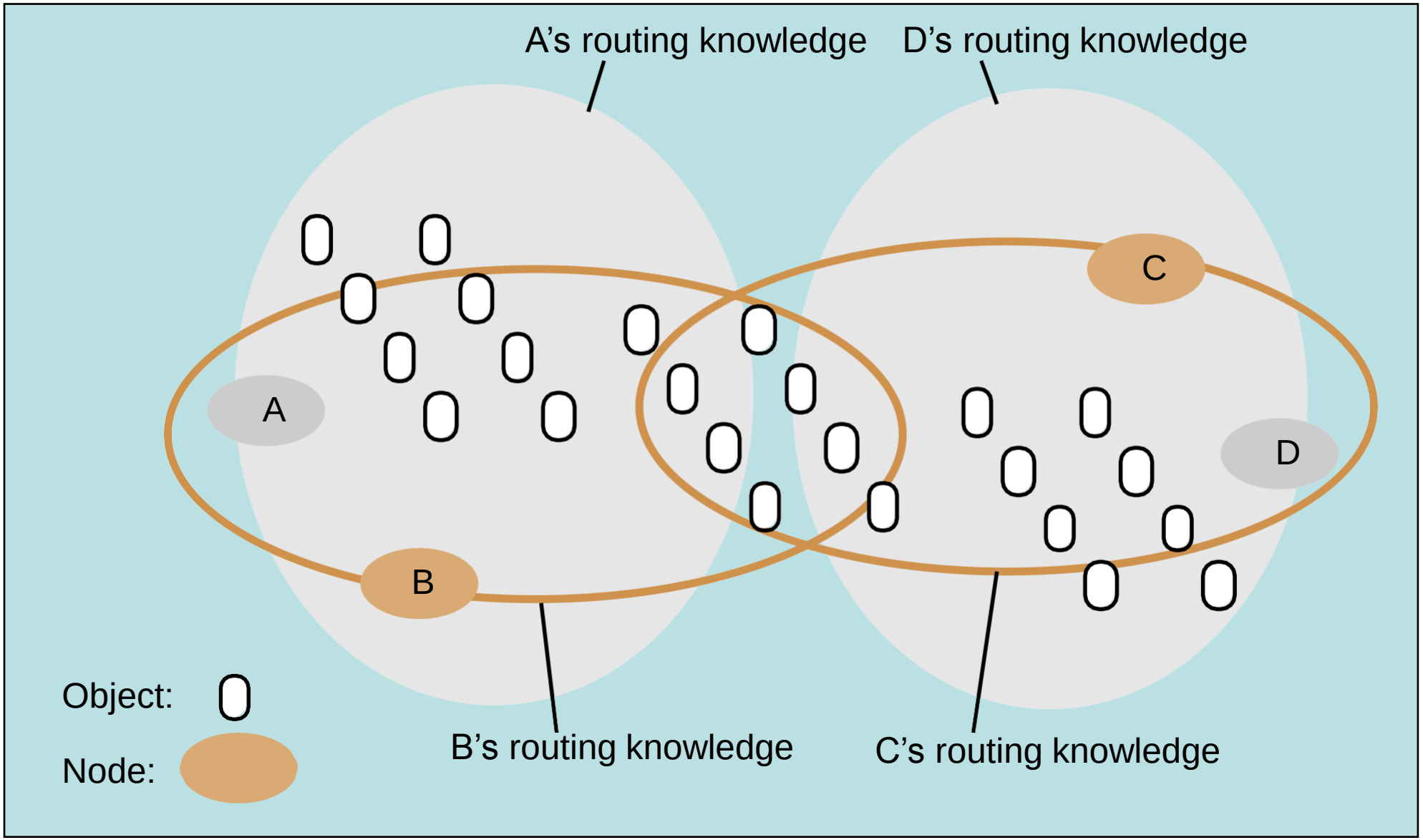
Non-Functional Requirements for Peer-to-Peer Middleware

- Global Scalability
- Load Balancing
- Local Optimization
- Adjusting to dynamic host availability
- Security of data
- Anonymity, deniability, and resistance to censorship (in some applications)

2.4 Routing Overlays

- A routing overlay is a distributed algorithm for a middleware layer responsible for routing requests from any client to a host that holds the object to which the request is addressed. Main tasks:
  - Routing of requests to objects: locating nodes and objects
  - Insertion and deletion of objects
  - Node addition and removal
- Any node can access any object by routing each request through a series of nodes, using information in the intermediate nodes to locate the destination object. Global User IDs (GUID) also known as opaque identifiers are used as names, but do not contain location information

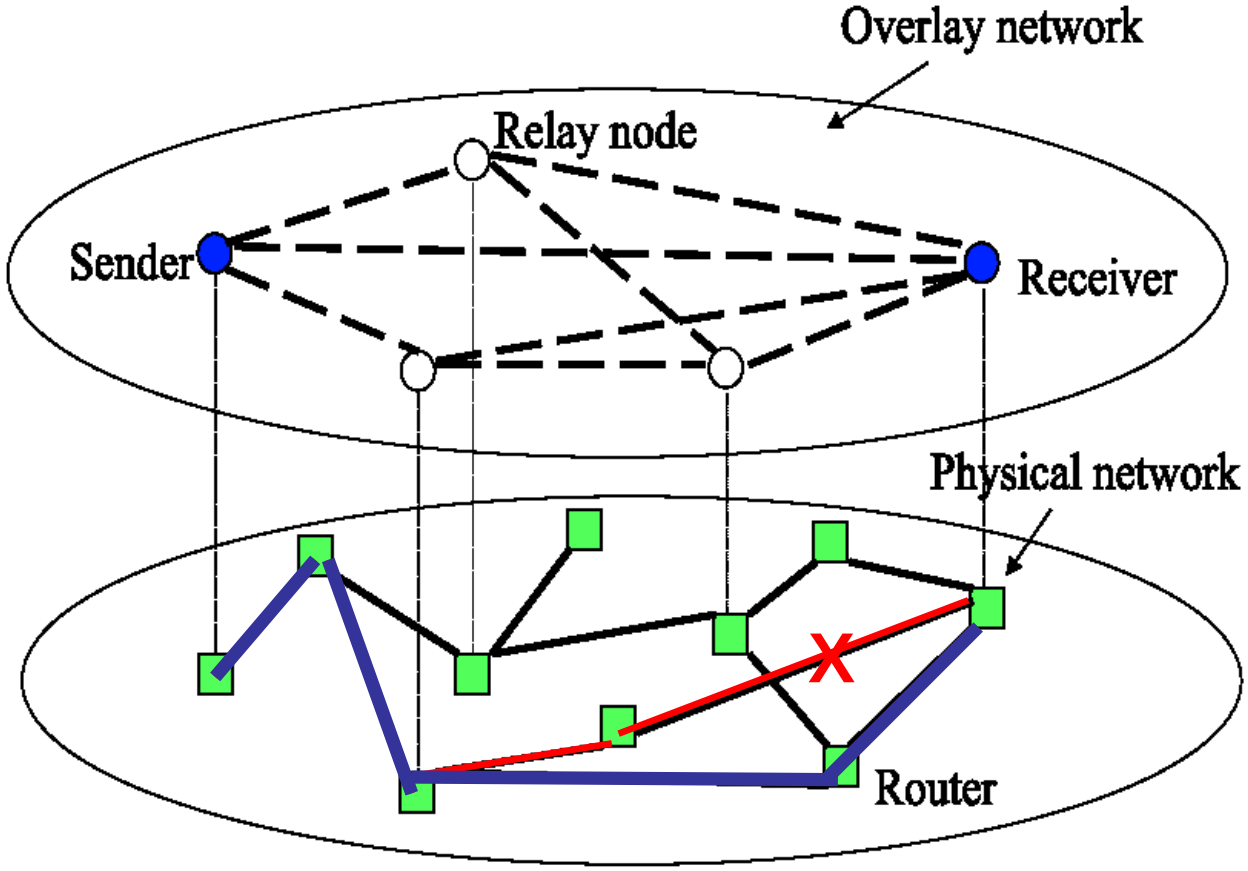
Figure 2.13 Distribution of information in a routing overlay



Overlay Networks

Overlay: a network on top of another (IP) networks

- links on one layer are network segments of lower layers
- tunneling, application adaptor, transparent router



Make the application control the routing

Routing Inefficiencies

- Principal categories of the routing inefficiencies
  - Poor routing metrics
    - These routers make decisions by minimizing the number of independent autonomous systems (ASs)
  - Restrictive routing policies
    - Private relationships
  - Single-path routing
- It can forward packets along non-optimal routes
- It can spread load unequally

2.5 Overlay case studies: Pastry, Tapestry

2.5.1 Pastry

Information on Pastry from the Microsoft Web Site (in Bibliography)

- Each node in a Pastry network has a unique, uniform random identifier (nodeId) in a circular 128-bit identifier space. When presented with a message and a numeric 128-bit key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes.
- The expected number of forwarding steps in the Pastry overlay network is *O(log N)*, while the size of the routing table maintained in each Pastry node is only *O(log N)* in size (where *N* is the number of live Pastry nodes in the overlay network). At each Pastry node along the route that a message takes, the application is notified and may perform application-specific computations related to the message.

Pastry

- Each Pastry node keeps track of its *L* immediate neighbors in the nodeId space (called the *leaf set*), and notifies applications of new node arrivals, node failures and node recoveries within the leaf set.
- Pastry takes into account locality (proximity) in the underlying Internet; it seeks to minimize the distance messages travel, according to a scalar proximity metric like the ping delay. Pastry is completely decentralized, scalable, and self-organizing; it automatically adapts to the arrival, departure and failure of nodes.

Pastry Applications

- P2P applications built upon Pastry can utilize its capabilities in many ways, including:
  - Mapping application objects to Pastry nodes
  - Inserting objects
  - Accessing objects
  - Availability and persistence
  - Diversity
  - Load balancing
  - Object caching
  - Efficient, scalable information dissemination

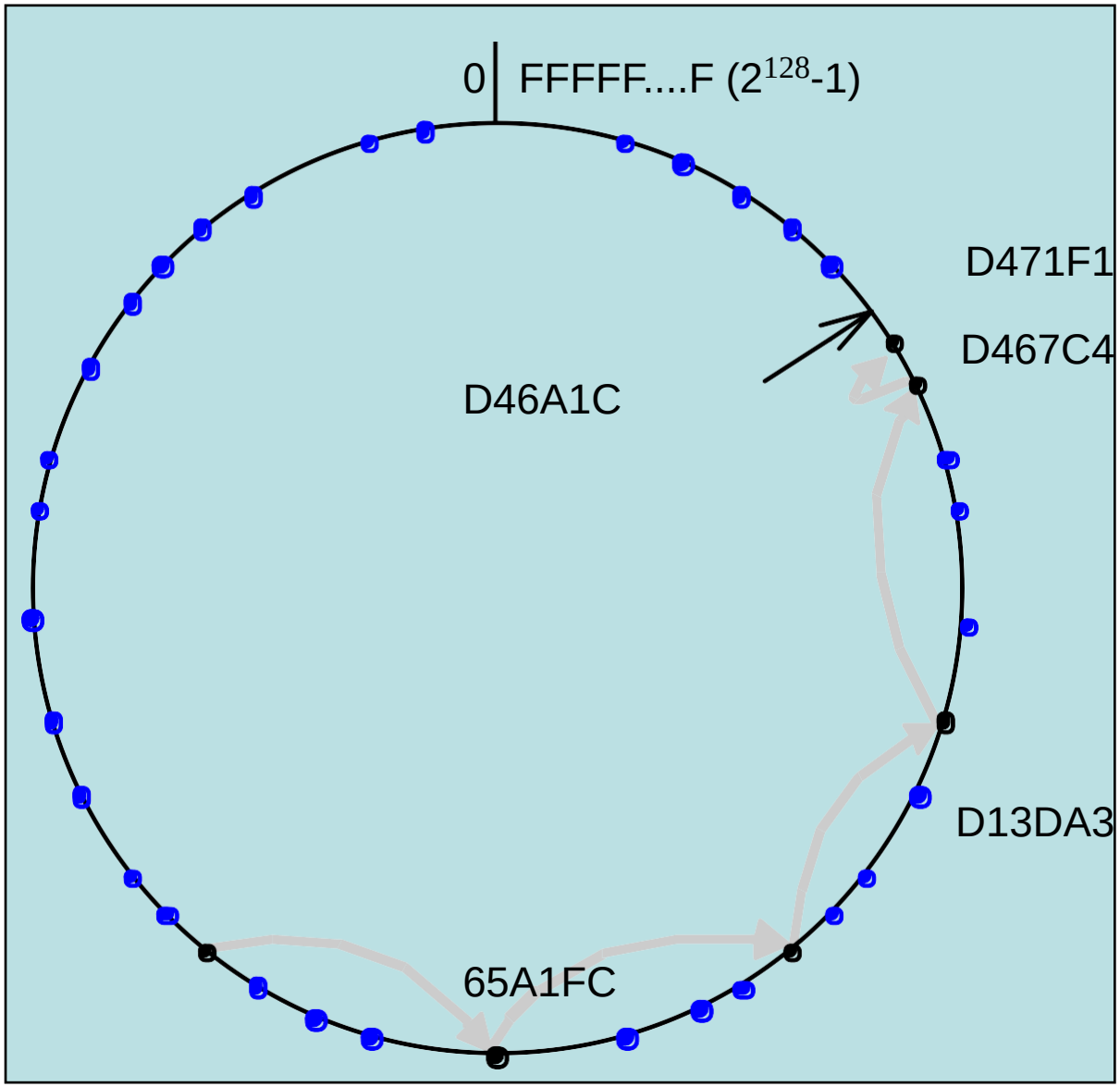
Figure 2.14: Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry

put(GUID, data)
The *data* is stored in replicas at all nodes responsible for the object identified by *GUID*.

remove(GUID)
Deletes all references to *GUID* and the associated data.

value = get(GUID)
The data associated with *GUID* is retrieved from one of the nodes responsible it.

Figure 2.15 Circular routing is correct but inefficient



The dots depict live nodes. The space is considered as circular: node 0 is adjacent to node (2^{128}-1). The diagram illustrates the routing of a message from node 65A1FC to D46A1C using leaf set information alone, assuming leaf sets of size 8 (l = 4). This is a degenerate type of routing that would scale very poorly; it is not used in practice.

Pastry Routing Tables

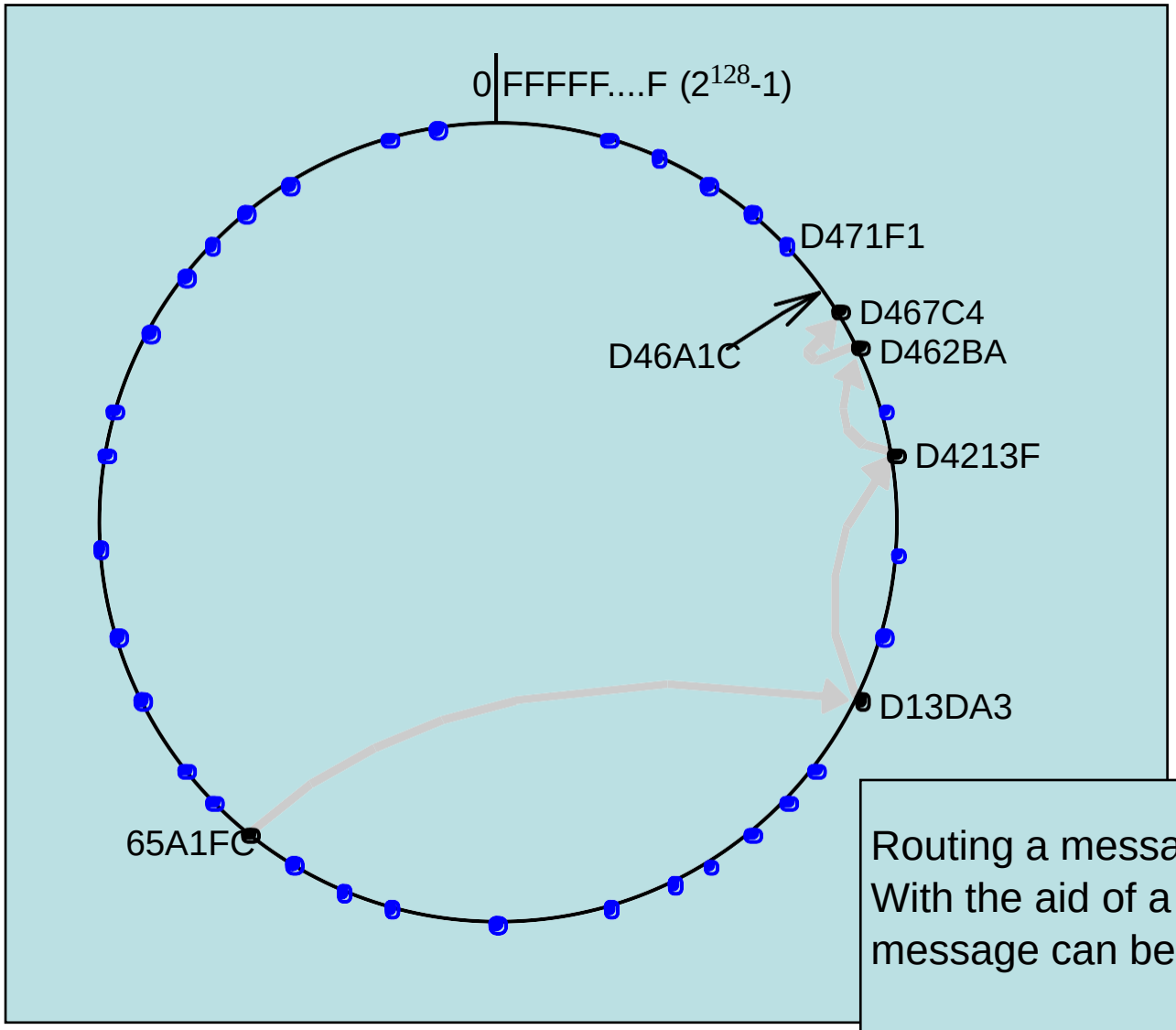
- To increase the efficiency of the Pastry system, a routing table is maintained at each node. Figure 10.7 shows a portion of a routing table, while figure 10.8 shows how that information can reduce the number of hops shown in figure 10.6. Figure 10.9 shows the Pastry Routing Algorithm.

Figure 2.16 First four rows of a Pastry routing table

p =	*GUID prefixes and corresponding nodehandles n*															
0	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
	n	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
1	*60*	*61*	*62*	*63*	*64*	*65*	*66*	*67*	*68*	*69*	*6A*	*6B*	*6C*	*6F*	*6E*	*6F*
	n	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
2	*650*	*651*	*652*	*653*	*654*	*655*	*656*	*657*	*658*	*659*	*65A*	*65B*	*65C*	*65D*	*65E*	*65F*
	n	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*
3	*65A0*	*65A1*	*65A2*	*65A3*	*65A4*	*65A5*	*65A6*	*65A7*	*65A8*	*65A9*	*65AA*	*65AB*	*65AC*	*65AD*	*65AE*	*65AF*
	n		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*

The routing table is located at a node whose GUID begins 65A1. Digits are in hexadecimal. The *n* s represent [GUID, IP address] pairs specifying the next hop to be taken by messages addressed to GUIDs that match each given prefix. Grey-shaded entries indicate that the prefix matches the current GUID up to the given value of *p*: the next row down or the leaf set should be examined to find a route. Although there are a maximum of 128 rows in the table, only $\log_{16} N$ rows will be populated on average in a network with *N* active nodes.

Figure 2.17 Pastry routing example



Routing a message from node 65A1FC to D46A1C. With the aid of a well-populated routing table the message can be delivered in ~ $\log_{16}(N)$ hops.

Figure 2.18 Pastry's routing algorithm

To handle a message *M* addressed to a node *D* (where *R*[*p*,*i*] is the element at column *i*, row *p* of the routing table):

1.If ($L_{-l} < D < L_l$) { //
the destination is within the leaf set or is the current node.
2.Forward *M* to the elementL_i of the leaf set with GUID closest to*D* or the current node *A*.
3.} else { // use the routing table to despatch *M* to a node with a closer GUID
4.find *p*, the length of the longest common prefix of

D and *A*. and *i*, the (p+1)th
hexadecimal digit of*D*.
5.If (*R*[*p*,*i*] ? *null*) forward*M* to *R*[*p*,*i*] // route *M* to a node with a longer common prefix.
6.else { // there is no entry in the routing table
7.Forward *M* to any node in *L* or *R* with a common prefix of length *i*, but a GUID that is numerically closer.
}

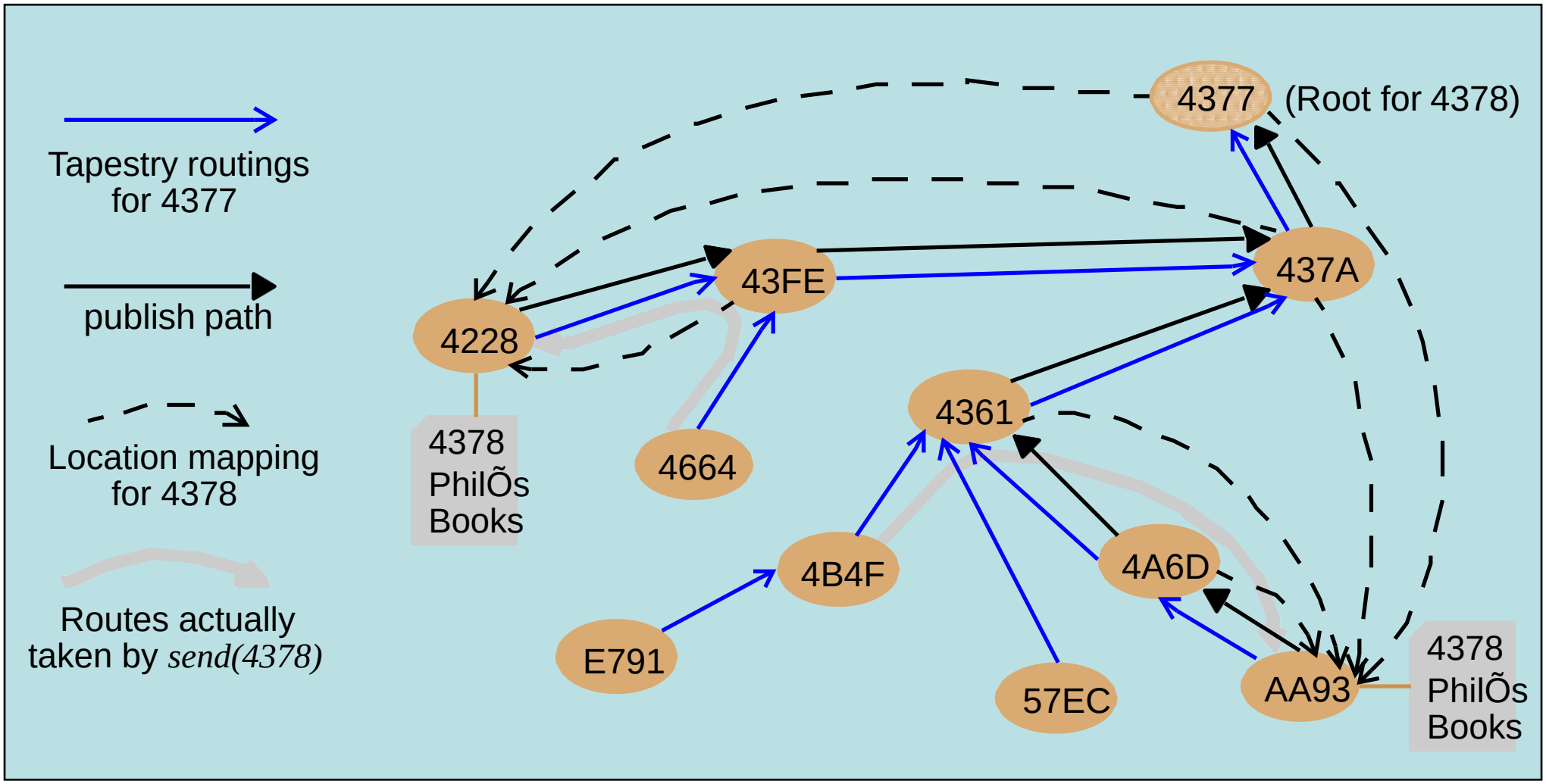
Tapestry

- Tapestry is another peer-to-peer model similar to Pastry.
- It hides a distributed hash table from applications behind a Distributed object location and routing (DOLR) interface to make replicated copies of objects more accessible by allowing multiple entries in the routing structure.
- This allows for a more direct access to a nearby copy of data resources, as shown in Figure 10.10.

Figure 2.19: Basic programming interface for distributed object location and routing (DOLR) as implemented by Tapestry

publish(GUID)
GUID can be computed from the object (or some part of it, e.g. its name). This function makes the node performing a *publish* operation the host for the object corresponding to *GUID*.
unpublish(GUID)
Makes the object corresponding to *GUID* inaccessible.
sendToObj(msg, GUID, [n])
Following the object-oriented paradigm, an invocation message is sent to an object in order to access it. This might be a request to open a TCP connection for data transfer or to return a message containing all or part of the object's state. The final optional parameter *[n]*, if present, requests the delivery of the same message to n replicas of the object.

Figure 2.20: Tapestry routing



2.6 Application case studies: Squirrel, OceanStore, Ivy

1. Squirrel web cache

- Computers on a local network form a Pastry overlay
  - SHA-1 applied to the URL to produce a 128-bit GUID
- Client nodes include a local Squirrel proxy process
  - If object not in the local cache, Squirrel routes a Get request via Pastry to the home node
  - If the home node has a fresh copy, it directly responds to the client. If it has a stale copy or no copy, it issues a get to the origin server.
- Evaluation

  105 active clients in Cambridge and more than 36,000 in Redmond

  •Each client contributes 100MB of disk storage

  •Hit ratios: centralized - 29% (Redmond) and 38% (Cambridge)
  - Similar simulation results achieved by the overlay

  •Latency: mean 4.11 hops (Redmond) and 1.8 hops (Cambridge)
  - Local transfer take only a few milliseconds

  •Computation: average number of cache requests served for other nodes by each node: 0.31 per minute

OceanStore

- The developers of Tapestry built OceanStore as a prototype for a very large scale, incrementally scalable persistent data store for multiple data objects. It allows persistent storage of both mutable and immutable data objects.
- Objects are structured in a manner similar to Unix files, as illustrated in figure 10.13.

Figure 2.21: OceanStore object storage

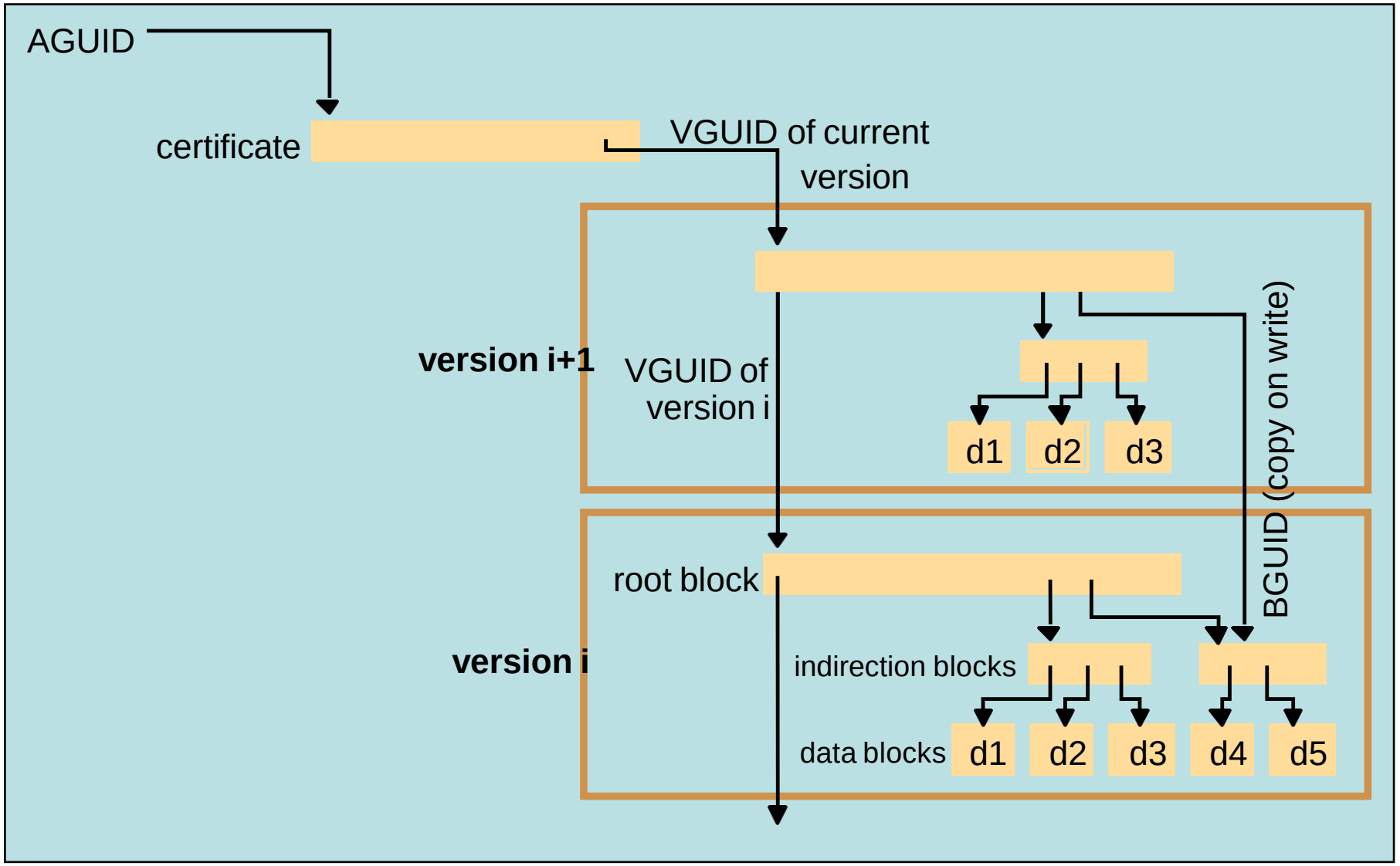


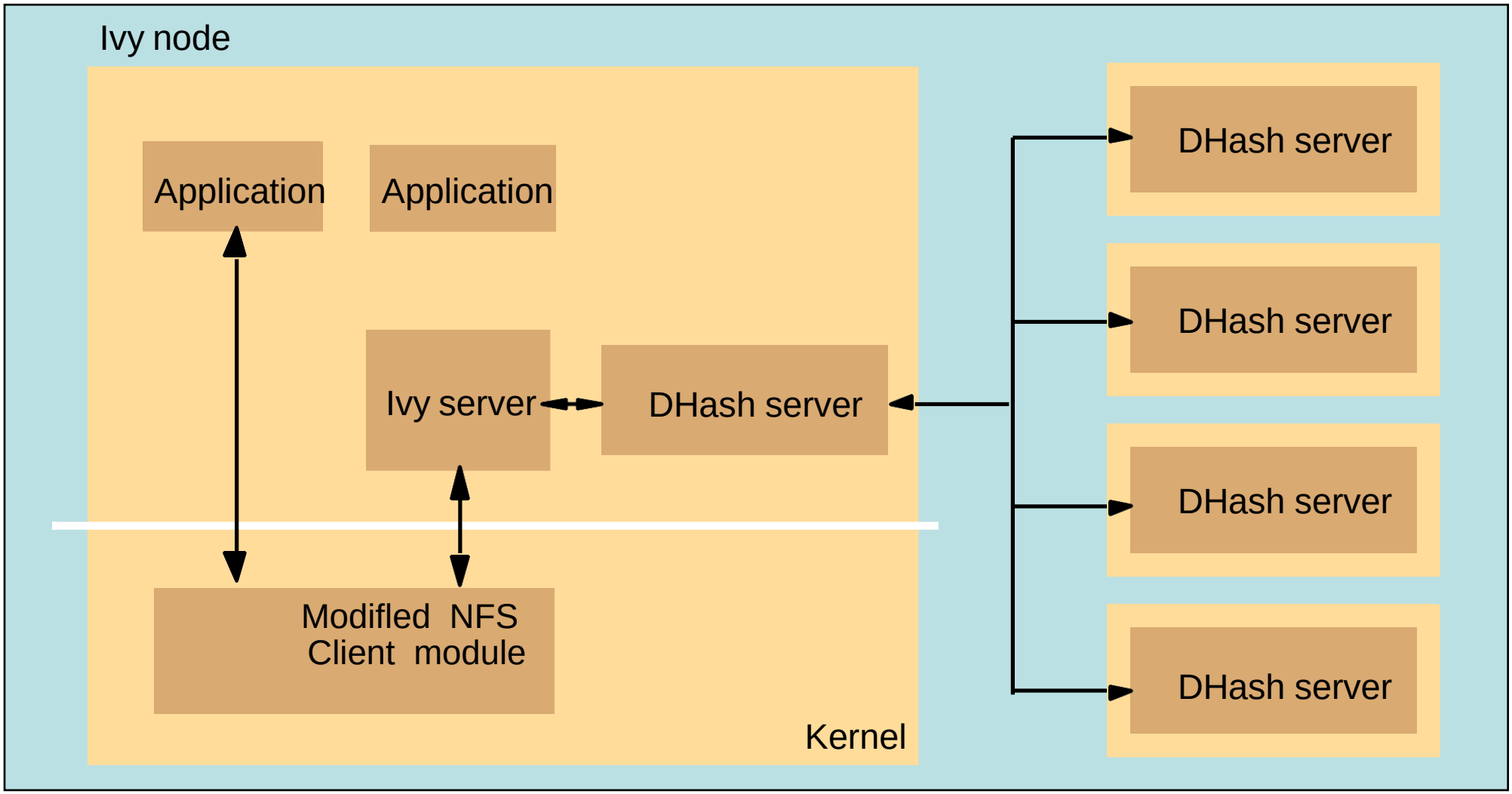
Figure 2.22: Identifier types in OceanStore

Name	*Meaning*	*Description*
BGUID	block GUID	Secure hash of a data block
VGUID	version GUID	BGUID of the root block of a version Uniquely
AGUID	active GUID	identifies all the versions of an object

Ivy

•The Ivy file system emulates a Sun NFS server. Ivy maintains a store of file updates in logs and reconstructs a particular state from the logs. It does not allow file locking to allow for the failure or withdrawal of nodes or connections. For protection against malicious attacks, file changes are associated with nodes and can omit a node when performing a reconstruction. Conflicts in file contents due to partitions in a network are dealt with by algorithm. Ivy architecture is shown in figure 2.22.

Figure 2.23: Ivy system architecture



UNIT 5

Transactions and Concurrency Control

Chapter 1 Transactions and Concurrency Control

1. Introduction
2. Transactions
3. Nested transactions
4. Locks
5. Optimistic concurrency control
6. Timestamp ordering
7. Comparison of methods for concurrency control
8. Summary

1.1 Introduction

- Transaction
  - Definition: a sequence of sever operations that is guaranteed by the server to be ***atomic*** in the presence of multiple clients and server crashes
- The goal of transactions
  - the objects managed by a server must remain in a consistent state
    - when they are accessed by multiple transactions and
    - in the presence of server crashes
- Recoverable objects
  - can be recovered after their server crashes (recovery in Chapter 14)
  - objects are stored in permanent storage
- Failure model
  - transactions deal with crash failures of processes and omission failures of communication
- Designed for an asynchronous system
  - It is assumed that messages may be delayed

Figure 1.1 Operations of the *Account* interface

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() → *amount*
return the balance of the account
setBalance(amount)
set the balance of the account to amount

Used as an example. Each *Account* is represented by a remote object whose interface *Account* provides operations for making deposits and withdrawals and for setting and getting the balance.

Operations of the *Branch* interface

create(name) → *account*
create a new account with a given name
lookUp(name) → *account*
return a reference to the account with the given name
branchTotal() → *amount*
return the total of all the balances at the branch

and each *Branch* of the bank is represented by a remote object whose interface *Branch* provides operations for creating a new account, looking one up by name and enquiring about the total funds at the branch. It stores a correspondence between account names and their remote object references

Atomic operations at server

- Simple synchronisation: without transactions
  - when a server uses multiple threads it can perform several client operations concurrently
  - if we allowed *deposit* and *withdraw* to run concurrently we could get inconsistent results
- Objects should be designed for safe concurrent access e.g. in Java use synchronized methods, e.g.
  - *public synchronized void deposit(int amount) throws RemoteException*
- *Atomic operations* are free from interference from concurrent operations in other threads
  - use any available mutual exclusion mechanism (e.g. mutex)

Client cooperation by means of synchronizing server operations

- In some applications clients share resources via a server and depend on one another to progress
  - e.g. some clients update server objects and others access them
  - e.g. one is a producer and another a consumer
  - e.g. one sets a lock and the other waits for it to be released
- Servers implementation with multiple threads
  - Not a good idea for a waiting client to poll the server to see whether a resource is yet available
    - Unfair (later clients might get earlier turns)
  - Java *wait* and *notify* methods allow threads to communicate with one another and to solve these problems
    - e.g. when a client requests a resource, the server thread waits until it is notified that the resource is available

Failure model for transactions

Lampson's failure model deals with failures of disks, servers and communication

– algorithms work correctly when predictable faults occur
– but if a disaster occurs, we cannot say what will happen

- Writes to permanent storage may fail
  - e.g. by writing nothing or a wrong value (write to wrong block is a disaster)
  - reads can detect bad blocks by checksum
- Servers may crash occasionally
  - when a crashed server is replaced by a new process its memory is cleared and it carries out a recovery procedure to get its objects' state
  - faulty servers are made to crash so they do not produce arbitrary failures
  - recipient can detect corrupt messages (by checksum)
  - forged messages and undetected corrupt messages are disasters

1.2 Transactions

- Some applications require a sequence of client requests to a server to be atomic in the sense that
  1. they are free from interference by operations being performed on behalf of other concurrent clients; and
  2. either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.
- Retrospect:

– Transactions originate from database management systems
– Transactional file servers were built in the 1980s
– Transactions on distributed objects late 80s and 90s
– Middleware components e.g. CORBA Transaction service

- Transactions apply to recoverable objects and are intended to be atomic

– Servers 'recover' - they are restated and get their objects from permanent storage

A client's banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Figure 16.2

- This transaction specifies a sequence of related operations involving bank accounts named *A*, *B* and *C* and referred to as *a, b* and *c* in the program
- The first two operations transfer $100 from *A* to B
- The second two operations transfer $200 from *C* to *B*

Atomicity of transactions

The atomicity has two aspects

1.All or nothing

- It either completes successfully, and the effects of all of its operations are recorded in the objects or
- It has no effect at all (if it fails or is aborted)

Two further aspects of its own

- failure atomicity: effects are atomic even when the server crashes;
- durability: after a transaction has completed successfully, all its effects are saved in permanent storage.

1.Isolation

- Each transaction must be performed without interference from other transactions
  - There must be no observation by other transactions of a transaction's intermediate effects
  - Concurrency control ensures isolation

Operations in the *Coordinator* interface

- Transaction capabilities may be added to a server of recoverable objects
  - each transaction is created and managed by a *Coordinator* object whose interface follows:

Figure 16.3

openTransaction() -> trans;
starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);
ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);
aborts the transaction.

Transaction life histories

Figure 1.4

Successful	*Aborted by client*		*Aborted by server*
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•	server aborts	•
•	•	transaction →	•
operation	*operation*		*operation ERROR*
			reported to client
closeTransaction	*abortTransaction*		

- A transaction is either successful (it commits)
  - the coordinator sees that all objects are saved in permanent storage
- or it is aborted by the client or the server
  - make all temporary effects invisible to other transactions
  - how will the client know when the server has aborted its transaction?
  - the client finds out next time it tries to access an object at the server

1.2.1 Concurrency control

- Two well-known concurrent transaction problems
  - Lost update
    - a lost update occurs when two transactions both read the old value of a variable and use it to calculate a new value
  - Inconsistent retrievals
    - inconsistent retrievals occur when a retrieval transaction observes values that are involved in an ongoing updating transaction
- Assumption
  - the operations *deposit*, *withdraw*, *getBalance* and *setBalance* are *synchronized* operations
  - that is, their effect on the account balance is atomic

The lost update problem

Figure 1.5

Transaction *T* :	**Transaction *U*:**
balance = b.getBalance();	*balance = b.getBalance();*
*b.setBalance(balance*1.1);*	*b.setBalance(balance*1.1);*
a.withdraw(balance/10)	*c.withdraw(balance/10)*
balance = b.getBalance(); $200	
	balance = b.getBalance(); $200
	*b.setBalance(balance*1.1);* $220
*b.setBalance(balance*1.1);* $220	
a.withdraw(balance/10) $80	
	c.withdraw(balance/10) $280

- The initial balances of accounts A, B, C are $100, $200, $300
  Both transfer transactions increase B's balance by 10%

The net effect should be to increase *B* by 10% twice - 200, 220, 242, but it only gets to 220. *T*'s update is lost.

The inconsistent retrievals problem

Figure 1.6

Transaction *V*:		**Transaction *W*:**		
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*		
a.withdraw(100);	$100	*total = a.getBalance()*	$100	*total =*
		total+b.getBalance()	$300	*total =*
		total+c.getBalance()		
b.deposit(100)	$300	⋮		

- *V* transfers $100 from *A* to *B* while *W* calculates branch total (which should be $600)

we see an inconsistent retrieval because *V* has only done the withdraw part when *W* sums balances of *A* and *B*

Serial equivalence

- The ***same effect*** means
  - the read operations return the same values
  - the instance variables of the objects have the same values at the end
- If each one of a set of transactions has the correct effect when *done on its own*

then if they are *done one at a time in some order* the effect will be correct

- A ***serially equivalent interleaving*** is one in which the combined effect is the same as if the transactions had been

done one at a time in some order

  - The transactions are scheduled to avoid overlapping access to the accounts accessed by both of them

A serially equivalent interleaving of *T* and *U* (lost updates cured)

Figure 1.7

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

- if one of *T* and *U* runs before the other, they can't get a lostupdate,
- the same is true if they are run in a serially equivalent ordering

their access to *B* is serial, the other part can overlap

A serially equivalent interleaving of *V* and *W* (inconsistent retrievals cured)

Figure 1.8

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); b.deposit(100)		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		total = total+c.getBalance()	
		...	

- if *W* is run before or after *V*, the problem will not occur
- therefore it will not occur in a serially equivalent ordering of *V* and *W*
- the illustration is serial, but it need not be

we could overlap the first line of *W* with the second line of *V*

Read and *write* operation conflict rules

Figure 1.9

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair o*fread* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a*read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair o*wf rite* operations depends on the order of their execution

- Conflicting operations: a pair of operations conflicts if their combined effect depends on the order in which they were performed
  - e.g. *read* and *write* (whose effects are the result returned by *read* and the value set by *write)*

Serial equivalence defined in terms of conflicting operations

- For two transactions to be ***serially equivalent***, it is necessary and sufficient that

all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access

- Consider
  - T and U access i and j
    - *T*: *x* = *read*(*i*); *write*(*i*, 10); *write*(*j*, 20);
    - *U*: *y* = *read*(*j*); *write(*j, 30); z = *read* (*i*);
  - serial equivalence requires that either
    - *T* accesses *i* before *U* and *T* accesses *j* before *U*. or
    - U accesses *i* before *T* and *U* accesses j before *T*
- Serial equivalence is used as a criterion for designing concurrency control schemes. Three alternative approaches
  - Locking: used by most practical systems
  - Optimistic concurrency control
  - Timestamp ordering

A non-serially equivalent interleaving of operations of transactions *T* and *U*

Transaction *T*:	Transaction *U*:
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Figure 1.10

- Each transaction's access to *i* and *j* is serialized w.r.t one another, but
  - *T* makes all accesses to *i* before *U* does
  - *U* makes all accesses to *j* before *T* does

therefore this interleaving is not serially equivalent

1.2.2 Recoverability from aborts

If a transaction aborts, the server must make sure that other concurrent transactions do not see any of its effects, we study two problems:

- 'dirty reads'
  - an interaction between a *read* operation in one transaction and an earlier *write* operation on the same object (by a transaction that then aborts)
  - a transaction that committed with a 'dirty read' is not recoverable
- 'premature writes'
  - interactions between *write* operations on the same object by different transactions, one of which aborts

For illustration, assume *getBalance* is a read operation and *setBalance* a write operation

A dirty read when transaction *T* aborts

Figure 1.11

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$160
		commit transaction	
abort transaction			

U has committed, so it cannot be undone => dirty read

Premature writes – overwriting uncommitted values

Figure 1.12

Transaction *T*:		**Transaction *U*:**	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

Some database systems keep ‘before images’ and restore them after aborts

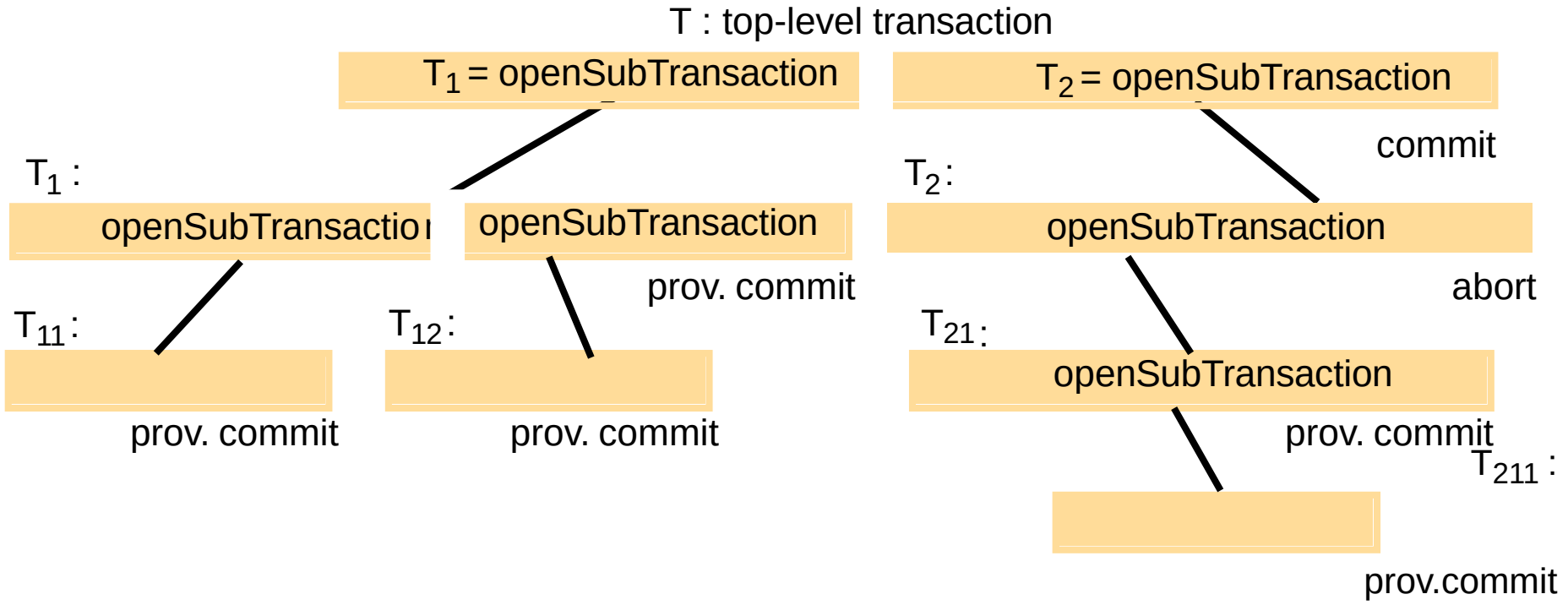
- e.g. $100 is before image of *T*’s write, $105 is before image of *U*’s write
- if *U* aborts we get the correct balance of $105,
- But if *U* commits and then *T* aborts, we get $100 instead of $110

Strict executions of transactions

- Curing premature writes:
  - if a recovery scheme uses before images
    - write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted
- Strict executions of transactions
  - to avoid both ‘dirty reads’ and ‘premature writes’.
    - delay both read and write operations
  - executions of transactions are called *strict* if both *read* and *write* operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted.
  - the strict execution of transactions enforces the desired property of isolation
- *Tentative versions* are used during progress of a transaction
  - objects in tentative versions are stored in volatile memory

1.3 Nested transactions

Figure 1.13



- Transactions may be composed of other transactions

– Several transactions may be started from within a transaction
– We have a ***top-level transaction*** and ***subtransactions*** which may have their own subtransactions

Nested transactions

- To a parent, a subtransaction is atomic with respect to failures and concurrent access
  - Transactions at the same level (e.g. *T*1 and *T*2) can run concurrently but access to common objects is serialised
  - A subtransaction can fail independently of its parent and other subtransactions
    - When it aborts, its parent decides what to do, e.g. start another subtransaction or give up
- Commitment
  - A transaction may commit or abort only after its child transactions have completed
  - A subtransaction decides independently to *commit provisionally* or to *abort*. Its decision to abort is final
  - When a parent aborts, all of its subtransactions are aborted
  - When a subtransaction aborts, parent can decide whether to abort or not
  - If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted

Advantages of nested transactions (over *flat* ones)

- Subtransactions may run concurrently with other subtransactions at the same level
  - –this allows additional concurrency in a transaction.
  - –when subtransactions run in different servers, they can work in parallel.
    - e.g. consider the *branchTotal* operation
    - it can be implemented by invoking *getBalance* at every account in the branch.
  - – these can be done in parallel when the branches have different servers
  - •Subtransactions can commit or abort independently
    - – this is potentially more robust
    - – a parent can decide on different actions according to whether a subtransaction has aborted or not

Summary on transactions

We consider only transactions at a single server, they are:

- atomic in the presence of concurrent transactions
  - which can be achieved by serially equivalent executions
- atomic in the presence of server crashes
  - they save committed state in permanent storage (recovery Ch.14)
  - they use strict executions to allow for aborts
  - they use tentative versions to allow for commit/abort
- nested transactions are structured from sub-transactions
  - they allow concurrent execution of sub-transactions
  - they allow independent recovery of sub-transactions

1.4 Locks

- Transactions must be scheduled so that their effect on shared objects is serially equivalent
  a) all access by a transaction to a particular object must be serialized with respect to another transaction's access
  b) all pairs of conflicting operations of two transactions should be executed in the same order

– A server can achieve serial equivalence by serializing access to objects, e.g. by the use of locks
– to ensure (b), a transaction is not allowed any new locks after it has released a lock

- *Two-phase locking* - has a 'growing' and a 'shrinking' phase
  – growing phase: new locks are acquired
  – shrinking phase: the locks are released

A simple serializing mechanism: exclusive locks

- The server attempts to lock any object that is about to be used by any operation of a client's transaction

– If a client requests access to an object that is locked by another client's transaction, the request is suspended and the client must wait until the object is unlocked

Transaction *T*: :		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	•••	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

Figure 1.14
(same as 1.7)

Strict two-phase locking

- Any locks applied during the progress of a transaction are held until the transaction commits or aborts
  - Strict executions prevent dirty reads and premature writes (if transactions abort)
  - A transaction that reads or writes an object must be delayed until other transactions that wrote the same object have committed or aborted
  - For recovery purposes, locks are held until updated objects have been written to permanent storage
- Granularity - apply locks to small things e.g. bank balances
  - There are no assumptions as to granularity in the schemes we present
- Read operations of different transactions do not conflict, so exclusive locks reduce concurrency more than necessary
  - The 'many reader/single writer' scheme allows several transactions to read an object or a single transaction to write it (but not both)
  - It uses read locks and write locks
    - read locks are sometimes called shared locks

Lock compatibility

The operation conflict rules tell us that:

1.If a transaction *T* has already performed a *read* operation on a particular object, then a concurrent transaction *U* must not *write* that object until *T* commits or aborts.

2.If a transaction *T* has already performed a *write* operation on a particular object, then a concurrent transaction *U* must not *read* or *write* that object until *T* commits or aborts.

to enforce 1, a request for a write lock is delayed by the presence of a read lock belonging to another transaction

to enforce 2, a request for a read lock or write lock is delayed by the presence of a write lock belonging to another transaction

Figure 1.15

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Lock promotion

- Locking prevents the inconsistent retrievals problem
  - If the retrieval transaction comes first, its read locks delay the update transaction
  - If the retrieval transaction comes second, its request for read locks causes it to be delayed until the update transaction has completed
- *Lock promotion* is required to prevent the lost update problem
  - Lost updates occur when two transactions read an object and then use it to calculate a new value
  - Lost updates are prevented by making later transactions delay their reads until the earlier ones have completed
    - Each transaction sets a read lock when it reads and then promotes it to a write lock when it writes the same object
    - when another transaction requires a read lock it will be delayed
  - ***Lock promotion***: the conversion of a lock to a stronger lock – that is, a lock that is more exclusive

Lock implementation

- The granting of locks will be implemented by a separate object in the server that we call the *lock manager*
- The lock manager holds a set of locks, for example in a hash table
- Each lock is an instance of the class *Lock* (Fig 1.17) and is associated with a particular object
  - its variables refer to the object, the holder(s) of the lock and its type
- The lock manager code uses *wait* (when an object is locked) and *notify* when the lock is released
- The lock manager provides *setLock* and *unLock* operations for use by the server

Figure 1.17 *Lock* class

```
public class Lock {
private Object object;  private Vector          // the object being protected by the lock
holders;  private LockType lockType;            // the TIDs of current holders
                                                // the current type
        public synchronized void acquire(TransID trans,     LockType aLockType ){
                while(/*another transaction holds the lock in conflicing mode*/) {
        try {
                                                        wait();
        }catch ( InterruptedException e){/*...*/ }
        }
        if(holders.isEmpty()) { // no TIDs hold lock  holders.addElement(trans);  lockType =
                aLockType;
        } else if(/*another transaction holds the lock, share it*/ ) ){
        if(/* this transaction not a holder*/) holders.addElement(trans);
        } else if (/* this transaction is a holder but needs a more exclusive lock*/)
                                                lockType.promote();
        }


        }
public synchronized void release(TransID trans ){
                holders.removeElement(trans);           // remove this holder
                // set locktype to none
                notifyAll();
        }
}
```

Figure 1.18 *LockManager* class

```
public class LockManager {  private Hashtable theLocks;

public void setLock(Object object, TransID trans, LockType lockType){  Lock
    foundLock;
synchronized(this){
// find the lock associated with object
// if there isn't one, create it and add to the hashtable
}
foundLock.acquire(trans, lockType);
}

// synchronize this one because we want to remove all entries  public
    synchronized void unLock(TransID trans) {
Enumeration e = theLocks.elements();
while(e.hasMoreElements()){
Lock aLock = (Lock)(e.nextElement());
if(/* trans is a holder of this lock*/ ) aLock.release(trans);
}
}
}
```

Deadlock with write locks

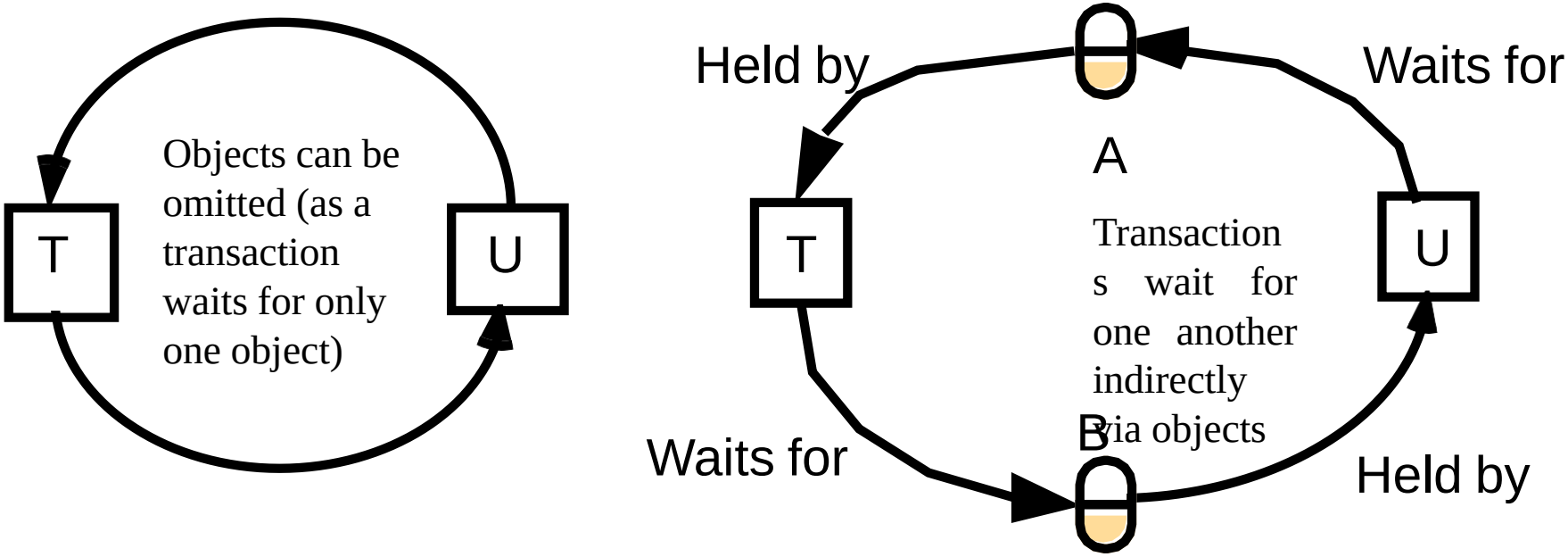
Figure 1.19

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
••• waits for *U*'s		*a.withdraw(200);*	waits for *T*'s
lock on *B*		••• lock on *A*	
•••		•••	
•••		•••	

The *deposit* and *withdraw* methods are atomic. Although they read as well as write, they acquire write locks. T accesses A → B, U accesses B → A

When locks are used, each of T and U acquires a lock on one account and then gets blocked when it tries to access the account the other one has locked. We have a 'deadlock'.

Figure 1.20 The wait-for graph for Figure 1.19

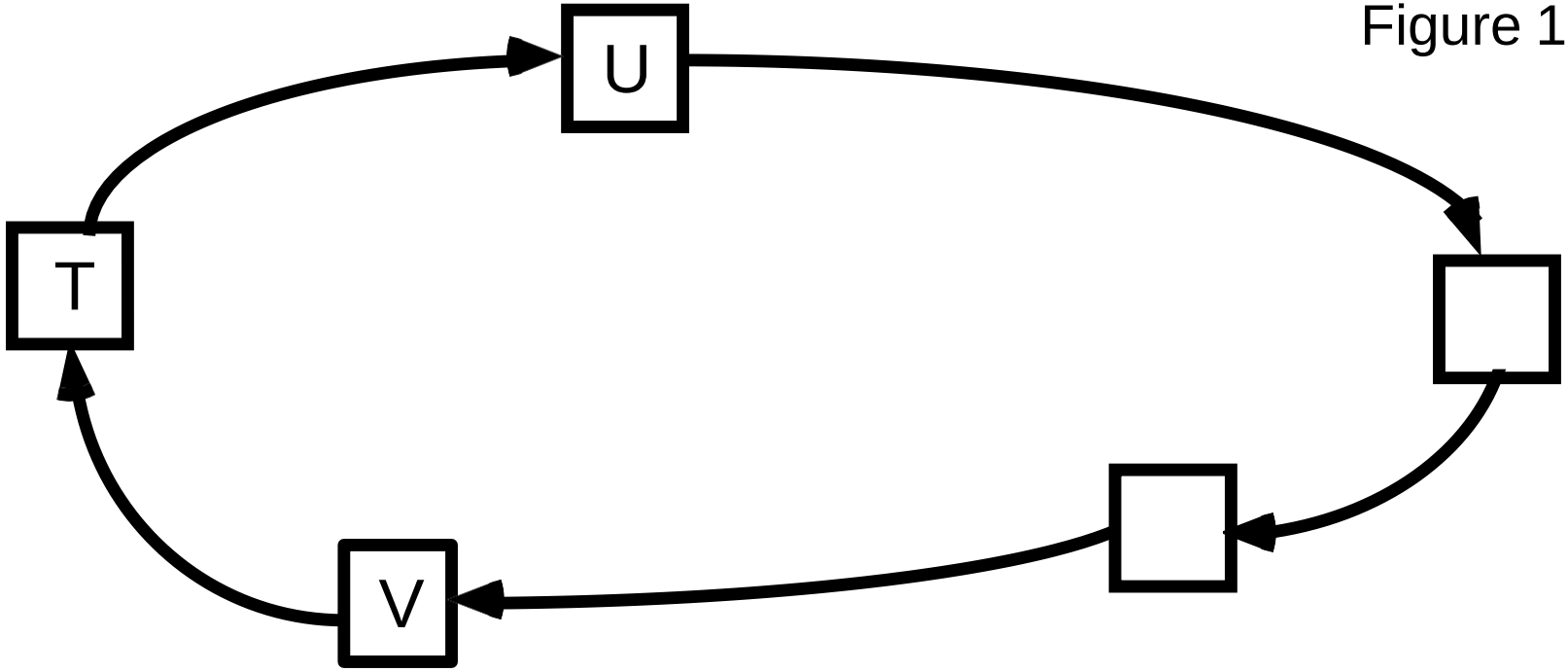


- Definition of deadlock
  - deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.
  - a *wait-for graph* can be used to represent the waiting relationships between current transactions

In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions

A cycle in a wait-for graph

Figure 1.21



Suppose a wait-for graph contains a cycle T …→ U → … → V → T

–Each transaction waits for the next transaction in the cycle

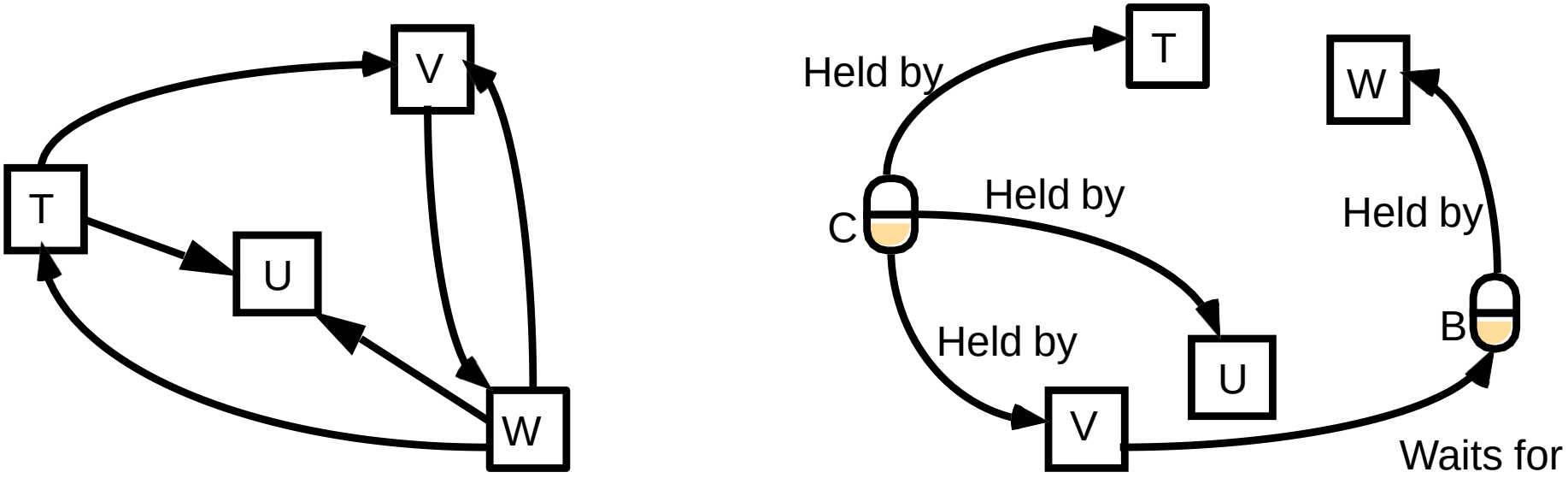
–All of these transactions are blocked waiting for locks

–None of the locks can ever be released (the transactions are deadlocked)

–If one transaction is aborted, then its locks are released and that cycle is broken

Another wait-for graph

Figure 1.22



- *T*, *U* and *V* share a read lock on *C* and
- *W* holds write lock on *B* (which *V* is waiting for)
- *T* and *W* then request write locks on *C* and deadlock occurs
  - e.g. *V* is in two cycles - look on the left

Deadlock prevention

- Deadlock prevention is unrealistic
  - e.g. lock all of the objects used by a transaction when it starts
    - unnecessarily restricts access to shared resources.
    - it is sometimes impossible to predict at the start of a transaction which objects will be used.
- Deadlock can also be prevented by requesting locks on objects in a predefined order
  - but this can result in premature locking and a reduction in concurrency

Deadlock detection

Deadlock detection: finding cycles in the wait-for graph

–After detecting a deadlock, a transaction must be selected to be aborted to break the cycle

–The software for deadlock detection can be part of the lock manager

–It holds a representation of the wait-for graph so that it can check it for cycles from time to time

–Edges are added to the graph and removed from the graph by the lock manager's *setLock* and *unLock* operations

–When a cycle is detected, choose a transaction to be aborted and then remove from the graph all the edges belonging to it

–It is hard to choose a victim - e.g. choose the oldest or the one in the most cycles

Timeouts on locks

Lock timeouts can be used to resolve deadlocks

–Each lock is given a limited period in which it is ***invulnerable***

- after this time, a lock becomes ***vulnerable***

–Provided that no other transaction is competing for the locked object, the vulnerable lock is allowed to remain

–But if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken

- (that is, the object is unlocked) and the waiting transaction resumes

–The transaction whose lock has been broken is normally aborted

Problems with lock timeouts

•Locks may be broken when there is no deadlock

•If the system is overloaded, lock timeouts will happen more often and long transactions will be penalised

•It is hard to select a suitable length for a timeout

1.5 Optimistic concurrency control

- The likelihood of two transactions conflicting is low
  - a transaction proceeds without restriction until the *closeTransaction* (no waiting, therefore no deadlock)
  - it is then checked to see whether it has come into conflict with other transactions
  - when a conflict arises, a transaction is aborted
- Each transaction has three phases
  - Working phase
    - the transaction uses a tentative version of the objects it accesses (dirty reads can't occur as we read from a committed version or a copy of it)
    - the coordinator records the *readset* and *writeset* of each transaction
  - Validation phase
    - at *closeTransaction* the coordinator validates the transaction (looks for conflicts)
    - if the validation is successful the transaction can commit.
    - if it fails, either the current transaction, or one it conflicts with is aborted
  - Update phase
    - If validated, the changes in its tentative versions are made permanent.
    - read-only transactions can commit immediately after passing validation

Validation of transactions

We use the read-write conflict rules

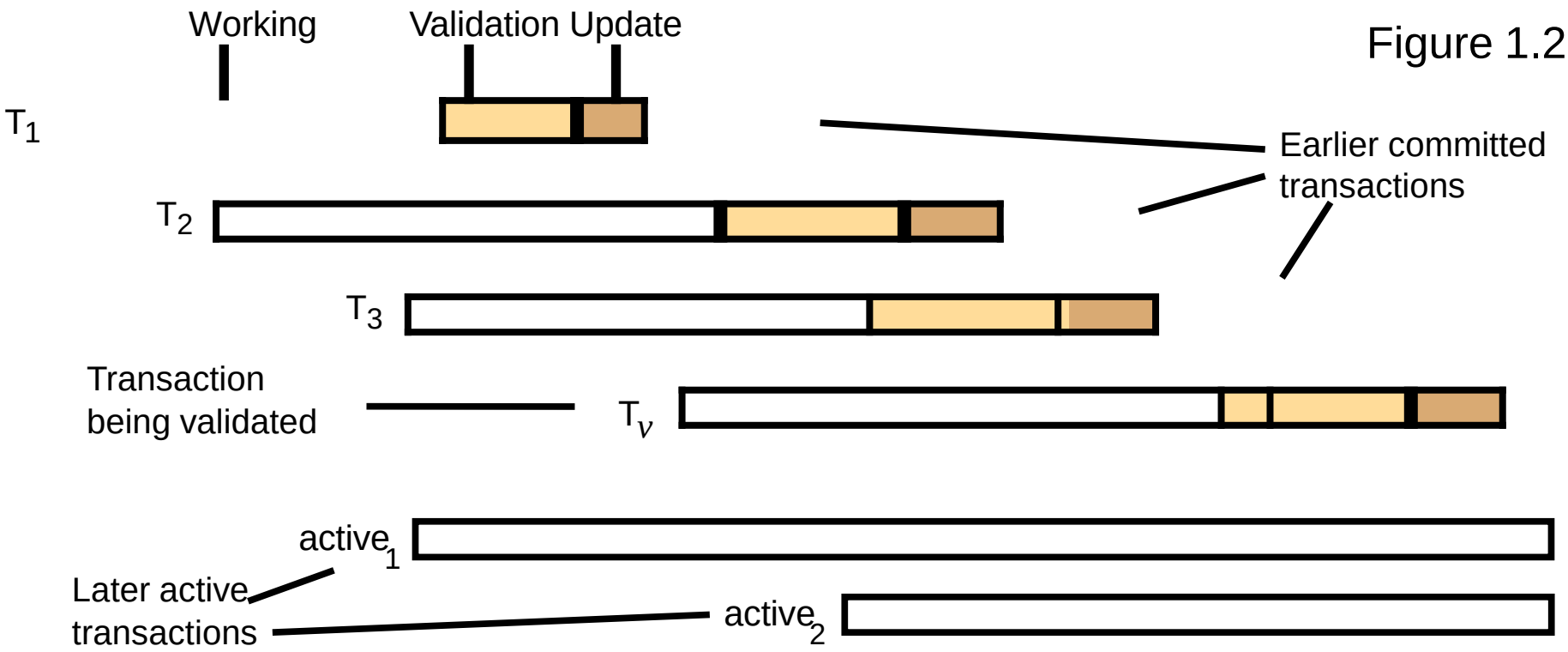
– to ensure a particular transaction is serially equivalent with respect to all other overlapping transactions

- Each transaction is given a transaction number when it starts validation (the number is kept if it commits)
- The rules ensure serializability of transaction T_v (transaction being validated) with respect to transaction T_i

T_v	T_i	*Rule*	
write	*read*	1. T_i must not read objects written by T_v	***forward***
read	*write*	2. T_v must not read objects written by T_i	***backward***
write	*write*	3. T_i must not write objects written by T_v and T_v mustnot write objects written by T_i	

Validation can be simplified by *omitting rule 3*
(if no overlapping of validate and update phases)

Validation of transactions



The earlier committed transactions are *T*1, *T*2 and *T*3. *T1* committed before *Tv* started.(*earlier* means they started validation earlier)

•Backward validation: check T_v with preceding overlapping transactions

–Rule1(Tv's*write*vsTi's*read*)is satisfied because reads of earlier transactions were done before *Tv* entered validation (and possible updates)

–Rule 2 - check if *T*v's read set overlaps with write sets of earlier *Ti*

–*T*2 and *T*3 committed before *Tv* finished its working phase.

– Rule3 - (*write*vs *write*) assume no overlap of validate and commit.

Backward Validation of Transactions

Backward validation of transaction T_v

boolean valid = true; for (int T_i

= *startTn*+1; T_i <= *finishTn*; T_i++){

if (read set of T_v intersects write set of T_i) valid = false;

}

- *startTn* is the biggest transaction number assigned to some other committed transaction when T_v started its working phase
- *finishTn* is biggest transaction number assigned to some other committed transaction when *Tv* started its validation phase
- In figure, *StartTn* + 1 = T_2 and *finishTn* = T_3. In backward validation, the read set of T_v must be compared with the write sets of T_2 and T_3.
- the only way to resolve a conflict is to abort T_v

to carry out this algorithm, we must keep write sets of recently committed transactions

Forward validation

- Rule 1. the write set of T_v is compared with the read sets of all overlapping active transactions
  - In Figure 16.28, the write set of T_v must be compared with the read sets of *active1* and *active2*.
- Rule 2. (read T_v vs write T_i) is automatically fulfilled because the active transactions do not write until after T_v has completed.

```
Forward validation of transaction Tv  boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
if (write set of Tv intersects read set of Tid)      valid = false;
}
```

- Read only transactions always pass validation
- The scheme must allow for the fact that read sets of active transactions may change during validation
- As the other transactions are still active, we may abort them or *Tv*
- if we abort *Tv*, it may be unnecessary as an active one may anyway abort

Comparison of forward and backward validation

- In conflict, choice of transaction to abort
  - forward validation allows flexibility, whereas backward validation allows only one choice (the one being validated)
- In general read sets > than write sets.
  - backward validation
    - compares a possibly large read set against the old write sets
    - overhead of storing old write sets
  - forward validation
    - checks a small write set against the read sets of active transactions
    - need to allow for new transactions starting during validation
- Starvation
  - after a transaction is aborted, the client must restart it, but there is no guarantee it will ever succeed
- In both cases, aborted transactions are not guaranteed future success
- Deadlock is less likely than starvation because locks make transactions wait

1.6 Timestamp ordering concurrency control

Each operation in a transaction is validated when it is carried out

–if an operation cannot be validated, the transaction is aborted

–each transaction is given a unique timestamp when it starts.

- The timestamp defines its position in the time sequence of transactions.

–requests from transactions can be totally ordered by their timestamps.

- Basic timestamp ordering rule (based on operation conflicts)
  - A request to write an object is valid only if that object was last read and written by earlier transactions.
  - A request to read an object is valid only if that object was last written by an earlier transaction
- This rule assumes only one version of each object
- Refine the rule to make use of the tentative versions
  - to allow concurrent access by transactions to objects

Operation conflicts for timestamp ordering

- Refined rule
  - tentative versions are committed in the order of their timestamps (wait if necessary) but there is no need for the client to wait
  - but read operations wait for earlier transactions to finish
    - only wait for earlier ones (no deadlock)
  - each read or write operation is checked with the conflict rules

Figure 1.29

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.

- As usual write operations are in tentative objects
- Each object has a write timestamp and a set of tentative versions
- Each with its own write timestamp and a set of read timestamps
- When a *write* operation is accepted it is put in a tentative version and given a write timestamp
- When a *read* operation is accepted it is directed to the tentative version with the maximum write timestamp less than the transaction timestamp *Tc* is the current transaction, *Ti* are other transactions
- *Ti*>*Tc* means *Ti* is later than *Tc*
- When a *write* operation is accepted it is put in a tentative version and given a write timestamp

Write operations and timestamps

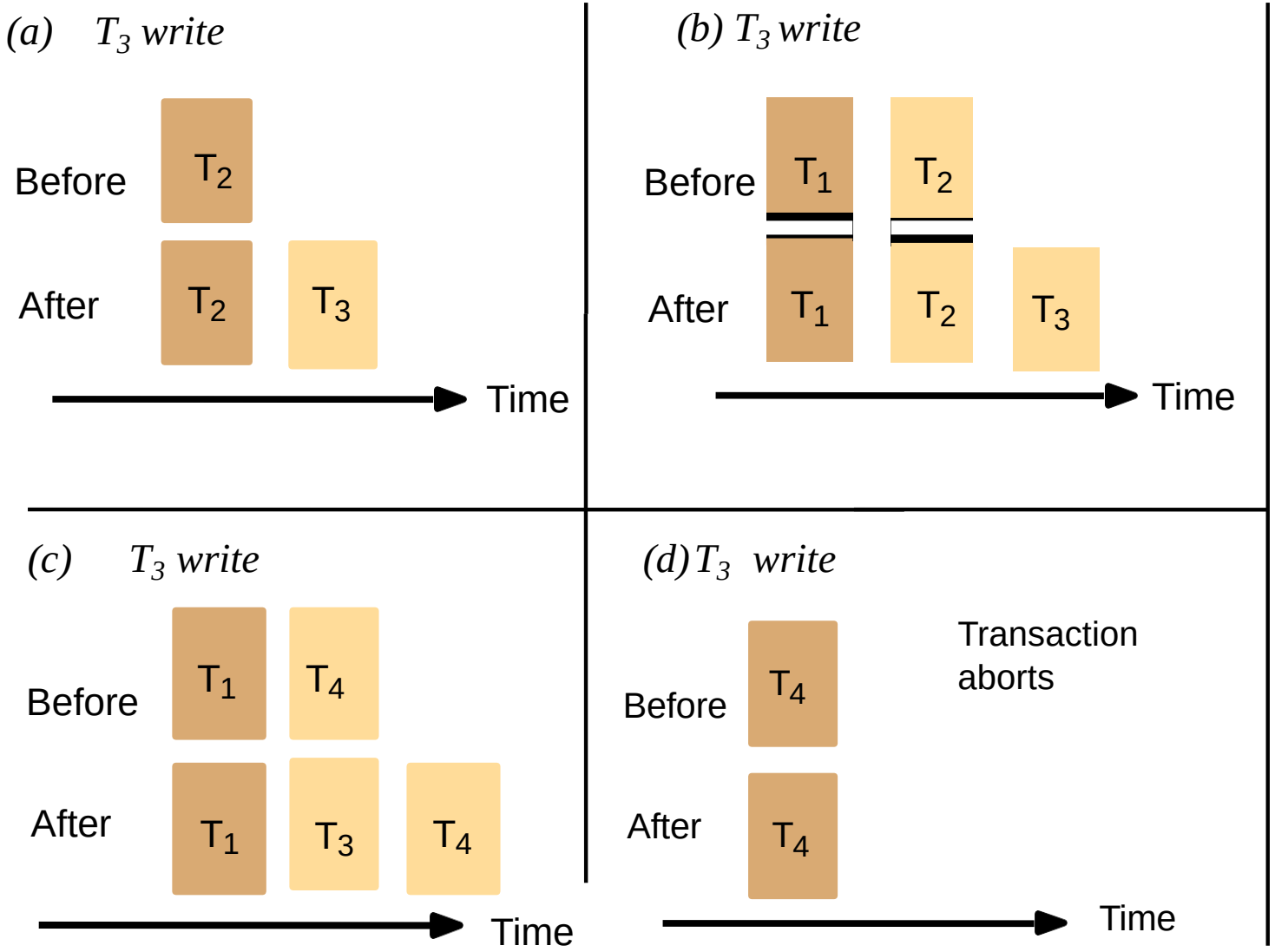


Figure 1.30

Key:

T_i

Committed

T_i

Tentative

object produced by transaction T_i (with write timestamp T_i)

$T_1 < T_2 < T_3 < T_4$

- this illustrates the versions and timestamps, when we do *T3 write*. for write to be allowed, T3≥ maximum read timestamp (not shown)
- In cases (a), (b) and (c) T3> w.t.s on committed version and a tentative version with w.t.s T3 is inserted at an appropriate place in the list of versions
- In case (d), T3< w.t.s on committed version and the transaction is aborted

Timestamp ordering write rule

- by combining rules 1 (write/read) and 2 (write/write) we have the following rule for deciding whether to accept a write operation requested by transaction T_c on object D

– rule 3 does not apply to writes

if ($T_c \geq$ maximum read timestamp on D &&
$T_c >$ write timestamp on committed version of D)
perform write operation on tentative version of D with write timestamp T_c
else /* write is too late */ Abort transaction T_c

Timestamp ordering read rule

- by using Rule 3 we get the following rule for deciding what to do about a read operation requested by transaction T_c on object *D.* That is, whether to
  - accept it immediately,
  - wait or
  - reject it

if (T_c > write timestamp on committed version of D) {
let D_{selected} be the version of D with the maximum write timestamp $\leq T_c$
if (D_{selected} is committed)
perform *read* operation on the version D_{selected}
else
Wait until the transaction that made version D_{selected} commits or aborts then reapply the *read* rule
} else
Abort transaction T_c

Page 504

Read operations and timestamps

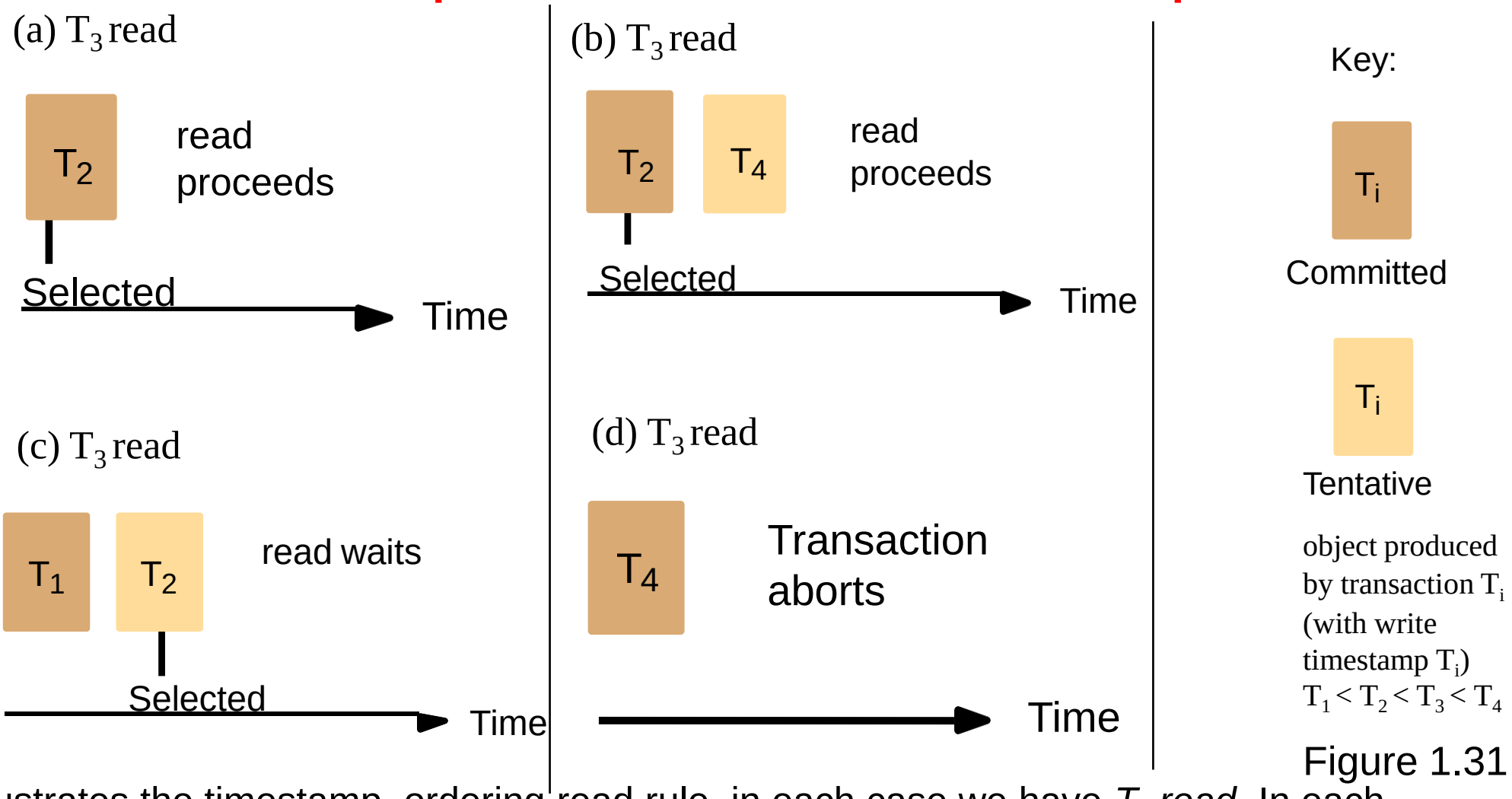


Figure 1.31

- Illustrates the timestamp, ordering read rule, in each case we have *T_3 read.* In each case, a version whose write timestamp is <= *T*3 is selected
- In cases (a) and (b) the read operation is directed to a committed version,
  - in (a) this is the only version. In (b) there is a later tentative version
- In case (c) the read operation is directed to a tentative version and the transaction must wait until the maker of the tentative version commits or aborts
- in case (d) there is no suitable version and T3 must abort

Transaction commits with timestamp ordering

- when a coordinator receives a commit request, it will always be able to carry it out because all operations have been checked for consistency with earlier transactions
  - committed versions of an object must be created in timestamp order
  - the server may sometimes need to wait, but the client need not wait
  - to ensure recoverability, the server will save the 'waiting to be committed versions' in permanent storage
- the timestamp ordering algorithm is strict because
  - the read rule delays each read operation until previous transactions that had written the object had committed or aborted
  - writing the committed versions in order ensures that the write operation is delayed until previous transactions that had written the object have committed or aborted

Remarks on timestamp ordering concurrency control

The method avoids deadlocks, but is likely to suffer from restarts

- modification known as 'ignore obsolete write' rule is an improvement
  - If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway.
  - However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the item
- multiversion timestamp ordering
  - allows more concurrency by keeping multiple committed versions
    - late read operations need not be aborted
  - there is not time to discuss the method now

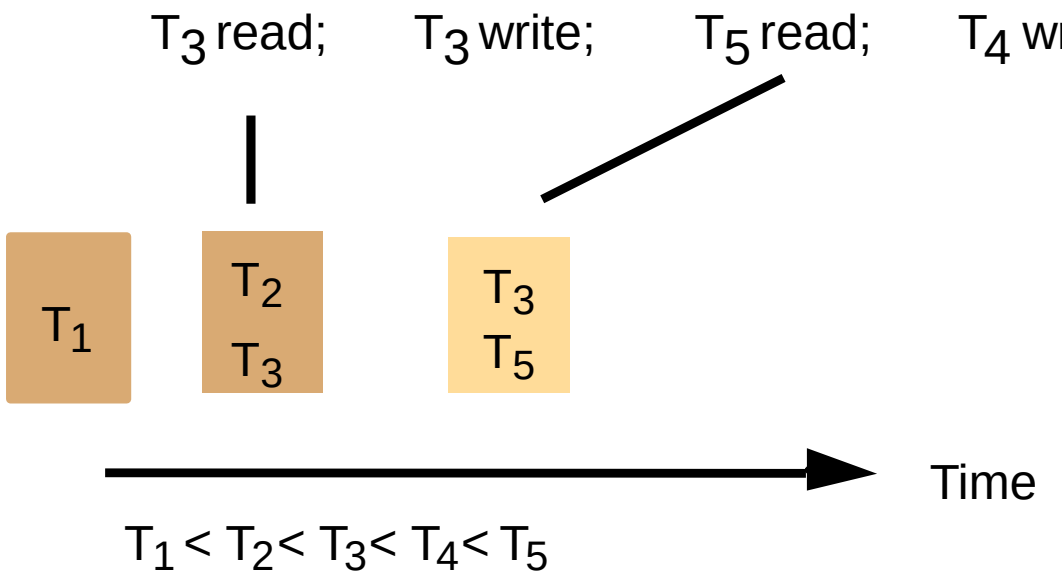
Timestamps in transactions *T* and *U*

Figure 1.32

T	*U*	*Timestamps and versions of objects* *A*	*B*	*C*
		RTS WTS	*RTS WTS*	*RTS WTS*
		{}	***S***	{} ***S***
openTransaction			{}	***S***
bal = b.getBalance()				
openTransaction			{*T*}	
*b.setBalance(bal*1.1)*				
bal = b.getBalance()	*wait for T*		***S***, *T*	
a.withdraw(bal/10)	*commit*			
bal = b.getBalance()	● ● ●	***S***, *T*		
*b.setBalance(bal*1.1)*	*c.withdraw(bal/10)*	***T***	***T***	
			{*U*}	
			T, *U*	
				S, *U*

Late *write* operation would invalidate a *read*

Figure 1.33



Key:

Committed Tentative

object produced by transaction T_i (with write timestamp T_i and read timestamp T_k)

1.7 Comparison of methods for concurrency control

- pessimistic approach (detect conflicts as they arise)
  - timestamp ordering: serialisation order decided statically
  - locking: serialisation order decided dynamically
  - timestamp ordering is better for transactions where reads >> writes,
  - locking is better for transactions where writes >> reads
  - strategy for aborts
    - timestamp ordering – immediate
    - locking– waits but can get deadlock
- optimistic methods
  - all transactions proceed, but may need to abort at the end
  - efficient operations when there are few conflicts, but aborts lead to repeating work
- the above methods are not always adequate e.g.
  - in cooperative work there is a need for user notification
  - applications such as cooperative CAD need user involvement in conflict resolution

Summary

- Operation conflicts form a basis for the derivation of concurrency control protocols.
  - protocols ensure serializability and allow for recovery by using strict executions
  - e.g. to avoid cascading aborts
- Three alternative strategies are possible in scheduling an operation in a transaction:
  - (1) to execute it immediately, (2) to delay it, or (3) to abort it
  - strict two-phase locking uses (1) and (2), aborting in the case of deadlock
    - ordering according to when transactions access common objects
  - timestamp ordering uses all three - no deadlocks
    - ordering according to the time transactions start.
  - optimistic concurrency control allows transactions to proceed without any form of checking until they are completed.
    - Validation is carried out. Starvation can occur.

Chapter 2
Distributed Transactions

Chapter 2 Distributed Transactions

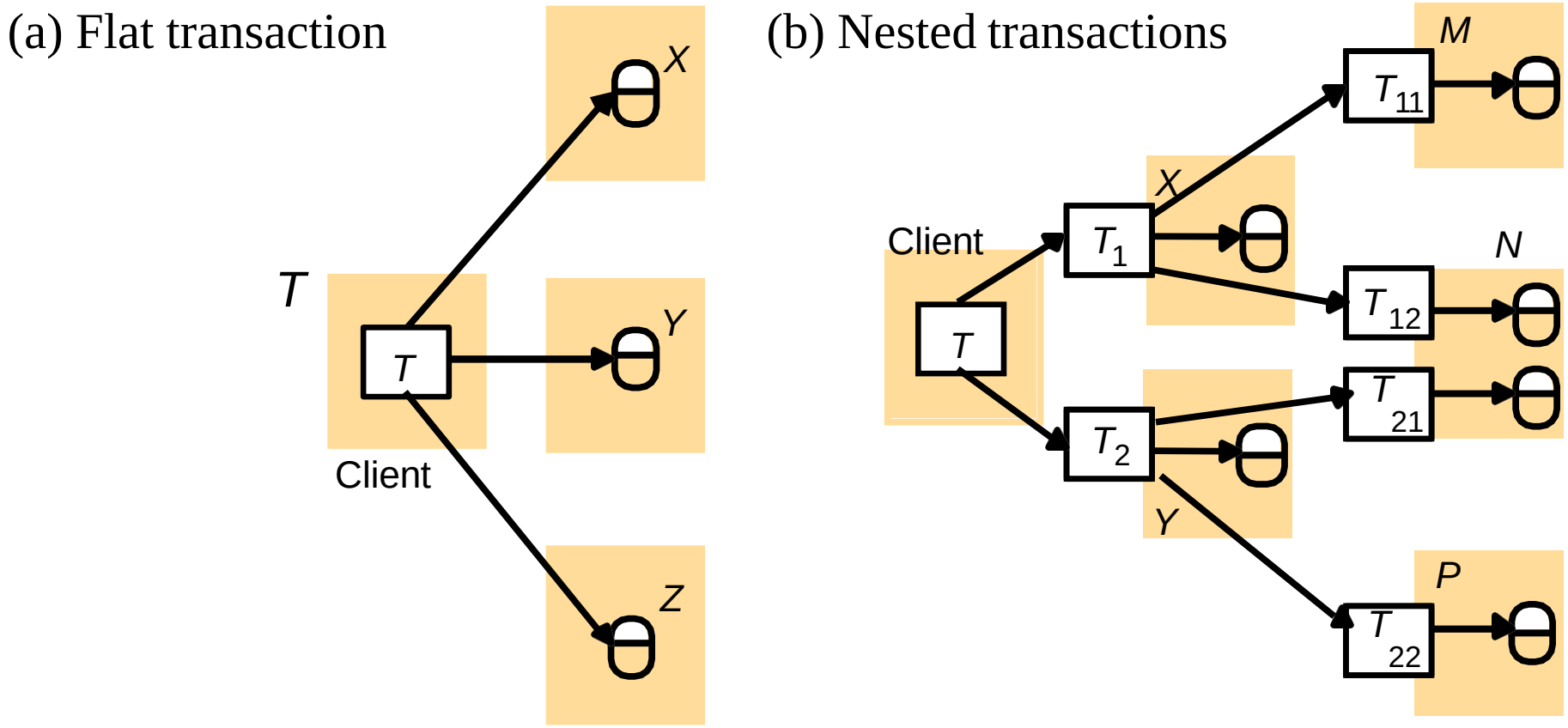
1. Introduction
2. Flat and nested distributed transactions
3. Atomic commit protocols
4. Concurrency control in distributed transactions
5. Distributed deadlocks
6. Transaction recovery
7. Summary

2.1 Introduction

- *distributed transaction*: a flat or nested transaction that accesses objects managed by multiple servers
- When a distributed transaction comes to an end
  - either ***all*** of the servers commit the transaction
  - or ***all*** of them abort the transaction
- One of the servers is *coordinator*, it must ensure the same outcome at all of the servers
- The 'two-phase commit protocol' is the most commonly used protocol for achieving this

2.2 Flat and nested distributed transactions

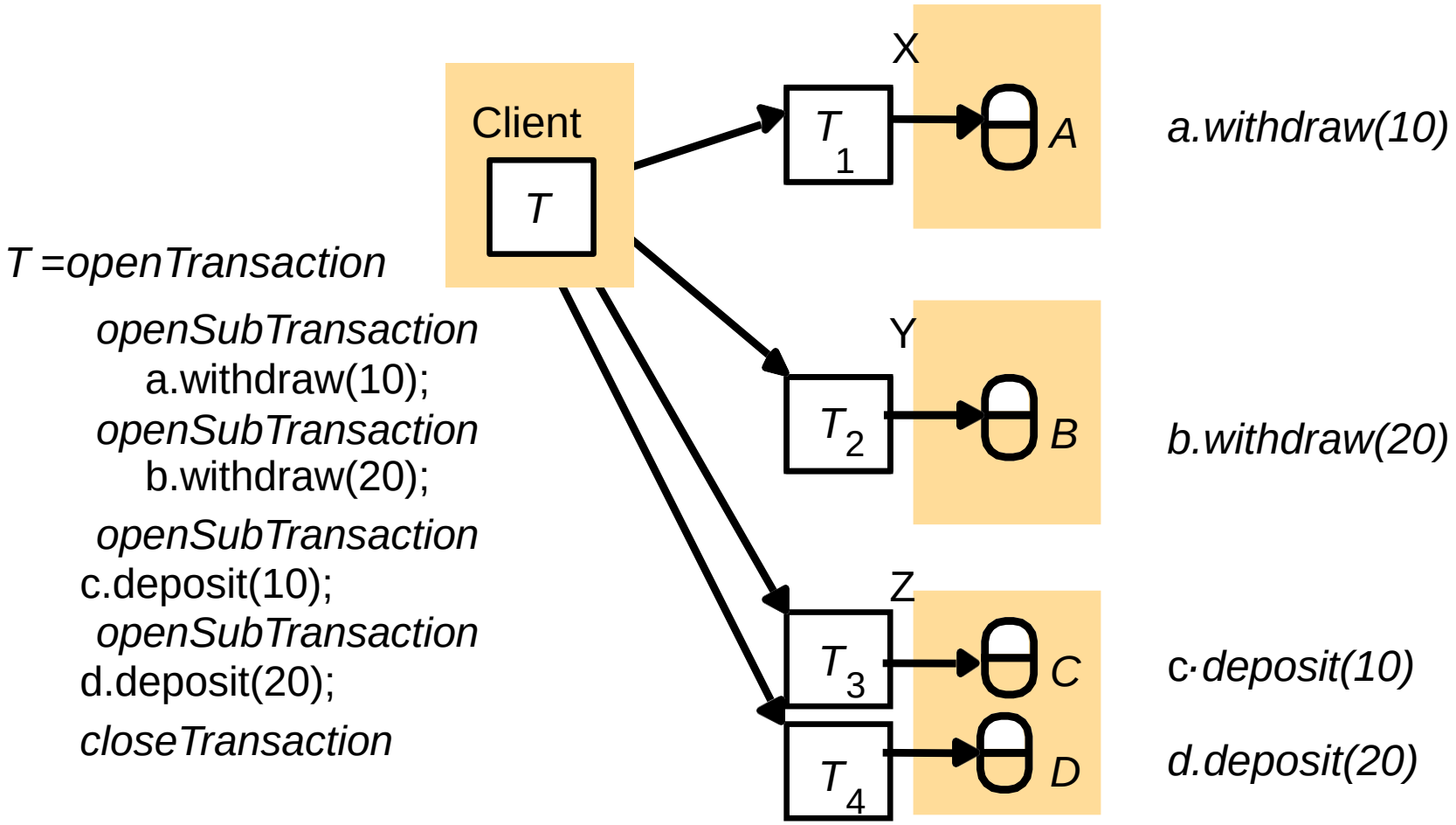
Figure 2.1



- A flat client transaction completes each of its requests before going on to the next one. Therefore, each transaction accesses servers' objects sequentially
- In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting

– In the nested case, subtransactions at the same level can run concurrently, so T1 and T2 are concurrent, and as they invoke objects in different servers, they can run in parallel

Figure 2.2 Nested banking transaction



- Client transfers $10 from A to C and then transfers $20 from B to D
- requests can be run in parallel- with several servers, the nested transaction is more efficient

The coordinator of a flat distributed transaction

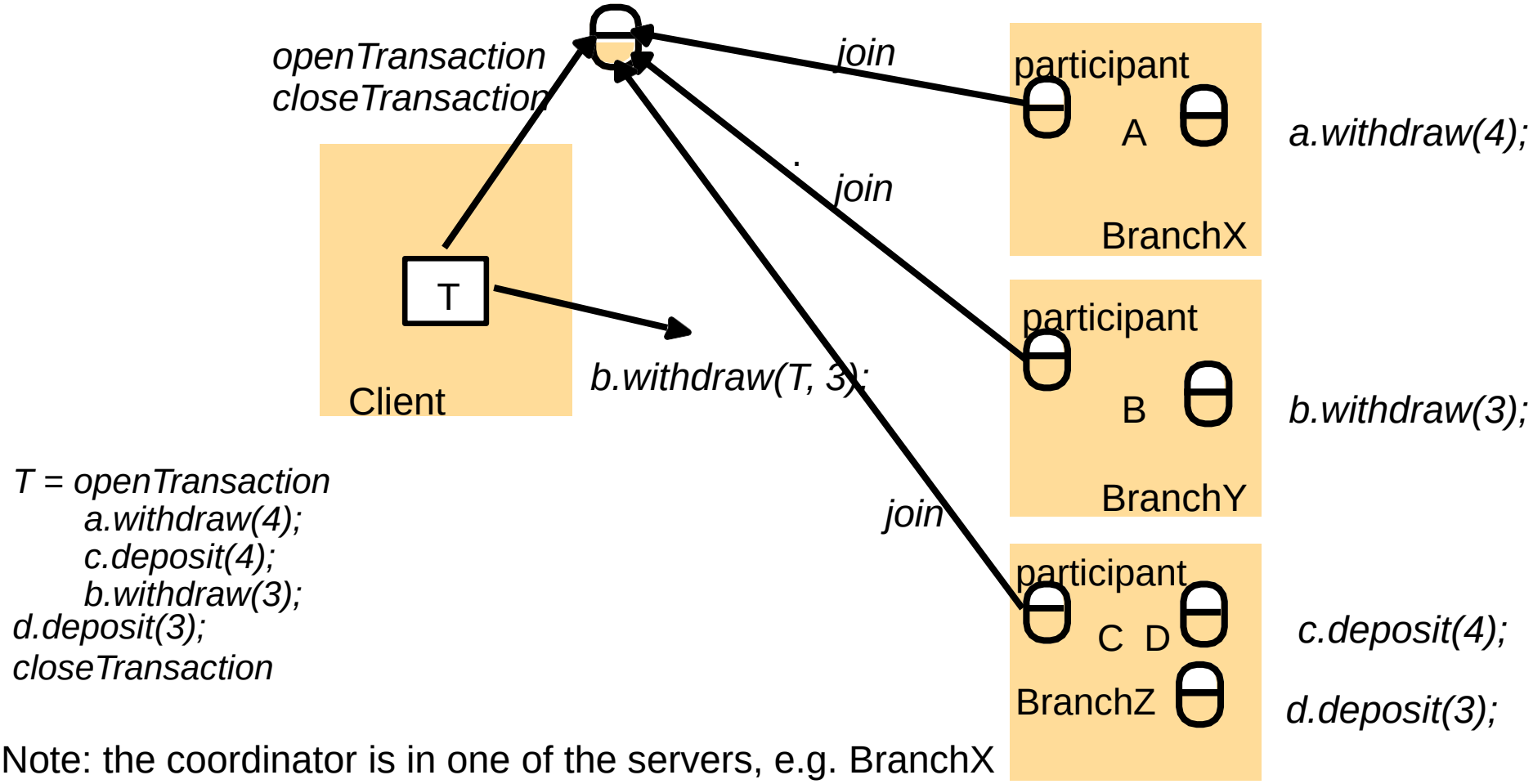
- Servers execute requests in a distributed transaction
  - when it commits they must communicate with one another to coordinate their actions
  - a client starts a transaction by sending an ope*nTransaction*

request to a *coordinator* in any server

- it returns a TID unique in the distributed system(e.g. server ID + local transaction number)
- at the end, it will be responsible for committing or aborting it

  - each server managing an object accessed by the transaction is a *participant* - it joins the transaction (next slide)
    - a participant keeps track of objects involved in the transaction
    - at the end it cooperates with the coordinator in carrying out the commit protocol
  - note that a participant can call *abortTransaction* in coordinator

Figure 2.3 A flat distributed banking transaction



- Note that the TID (*T*) is passed with each request e.g. withdraw(T,3)
- a client's (flat) banking transaction involves accounts *A*, *B*, *C* and *D* at servers *BranchX*, *BranchY* and *BranchZ*
- Each server is shown with a *participant*, which joins the transaction by invoking the *join* method in the coordinator. *openTransaction* goes to the coordinator

The join operation

- The interface for *Coordinator* is shown in Figure 2.3
  - it has *openTransaction*, *closeTransaction* and *abortTransaction*
  - *openTransaction* returns a *TID* which is passed with each operation so that servers know which transaction is accessing its objects
- The *Coordinator* interface provides an additional method, *join*, which is used whenever a new participant joins the transaction:
  - *join*(*Trans, reference to participant*)
  - informs a coordinator that a new participant has joined the transaction *Trans*.
  - the coordinator records the new participant in its participant list.
  - the fact that the coordinator knows all the participants and each participant knows the coordinator will enable them to collect the information that will be needed at commit time.

2.3 Atomic commit protocols

- Transaction atomicity requires that at the end,
  - either all of its operations are carried out or none of them
- In a distributed transaction, the client has requested the operations at more than one server
- One-phase atomic commit protocol
  - the coordinator tells the participants whether to commit or abort
  - this does not allow one of the servers to decide to abort – it may have discovered a deadlock or it may have crashed and been restarted
- Two-phase atomic commit protocol
  - is designed to allow any participant to choose to abort a transaction
  - *phase 1* - each participant votes. If it votes to commit, it is *prepared.* It cannot change its mind. In case it crashes, it must save updates in permanent store
  - phase 2 - the participants carry out the joint decision
- The decision could be *commit* or *abort* - participants record it in permanent store

2.3.1 The two-phase commit protocol

- During the progress of a transaction, the only communication between coordinator and participant is the *join* request
  - The client request to commit or abort goes to the coordinator
    - if client or participant request abort, the coordinator informs the participants immediately
    - if the client asks to commit, the 2PC comes into use
- Two-phase commit (2PC)
  - *voting phase*: coordinator asks all participants if they can commit
    - if yes, participant records updates in permanent storage and then votes
  - *completion phase*: coordinator tells all participants to commit or abort

Figure 2.4 Operations for two-phase commit protocol

Participant interface

- *canCommit?(trans)-> Yes / No*
  - Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote
- *doCommit(trans)*
  - Call from coordinator to participant to tell participant to commit its part of a transaction
- *doAbort(trans)*
  - Call from coordinator to participant to tell participant to abort its part of a transaction

Coordinator interface

- *haveCommitted(trans, participant)*
  - Call from participant to coordinator to confirm that it has committed the transaction
- *getDecision(trans) -> Yes / No*
  - Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages

Figure 2.5 The two-phase commit protocol

Phase 1 (voting phase):

1.The coordinator sends a *canCommit*? request to each of the participants in the transaction.

2.When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

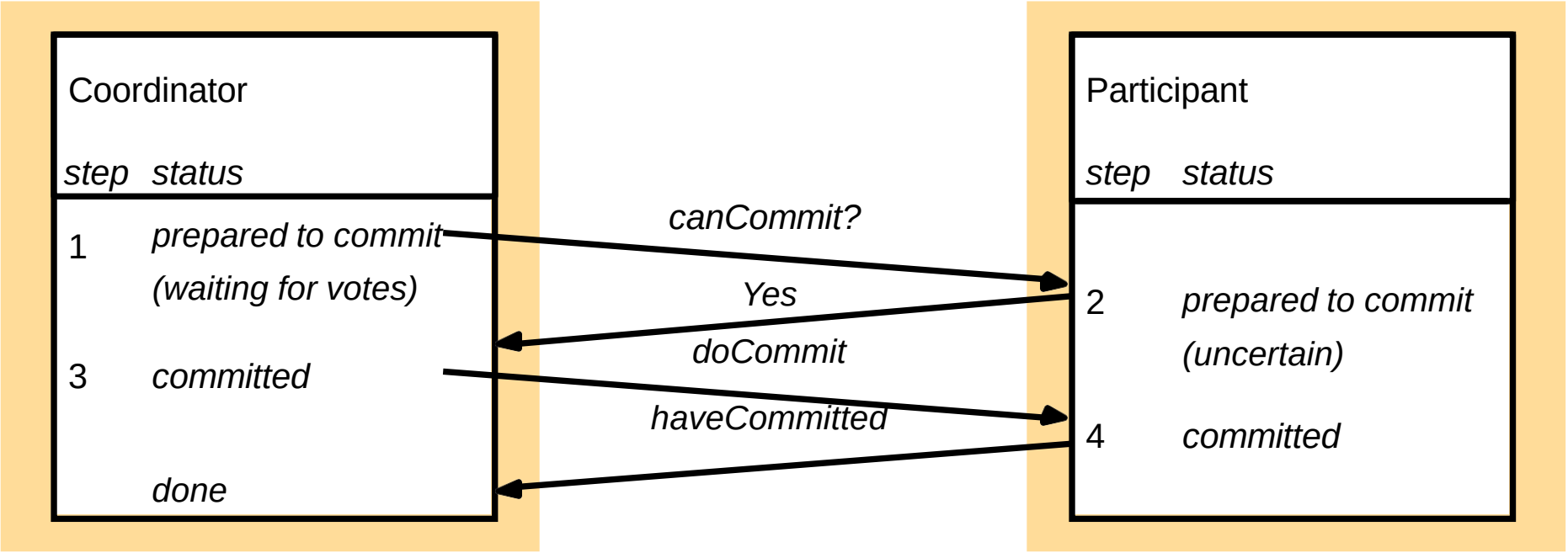
3.The coordinator collects the votes (including its own).

(a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.

(b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.

4.Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Figure 2.6
Communication in two-phase commit protocol



•Time-out actions in the 2PC

- to avoid blocking forever when a process crashes or a message is lost
  - –*uncertain* participant (step 2) has voted yes. it can't decide on its own
- it uses *getDecision* method to ask coordinator about outcome
  - –participant has carried out client requests, but has not had a *Commit?*from the coordinator. It can abort unilaterally
  - –coordinator delayed in waiting for votes (step 1). It can abort and send *doAbort* to participants.

Performance of the two-phase commit protocol

- If there are no failures, the 2PC involving *N* participants requires
  - *N canCommit?* messages and replies, followed by *N doCommit* messages.
    - the cost in messages is proportional to 3*N*, and the cost in time is three rounds of messages.
    - The *haveCommitted* messages are not counted
  - there may be arbitrarily many server and communication failures
  - 2PC is guaranteed to complete eventually, but it is not possible to specify a time limit within which it will be completed
    - delays to participants in uncertain state
    - some 3PCs designed to alleviate such delays
      - they require more messages and more rounds for the normal case

2.3.2 Two-phase commit protocol for nested transactions

- Recall Fig 17.1(b), top-level transaction T and subtransactions T_1, T_2, T_{11}, T_{12}, T_{21}, T_{22}
- A subtransaction starts after its parent and finishes before it
- When a subtransaction completes, it makes an independent decision either to *commit provisionally* or to abort.
  - A provisional commit is not the same as being prepared: it is a local decision and is not backed up on permanent storage.
  - If the server crashes subsequently, its replacement will not be able to carry out a provisional commit.
- A two-phase commit protocol is needed for nested transactions
  - it allows servers of provisionally committed transactions that have crashed to abort them when they recover.

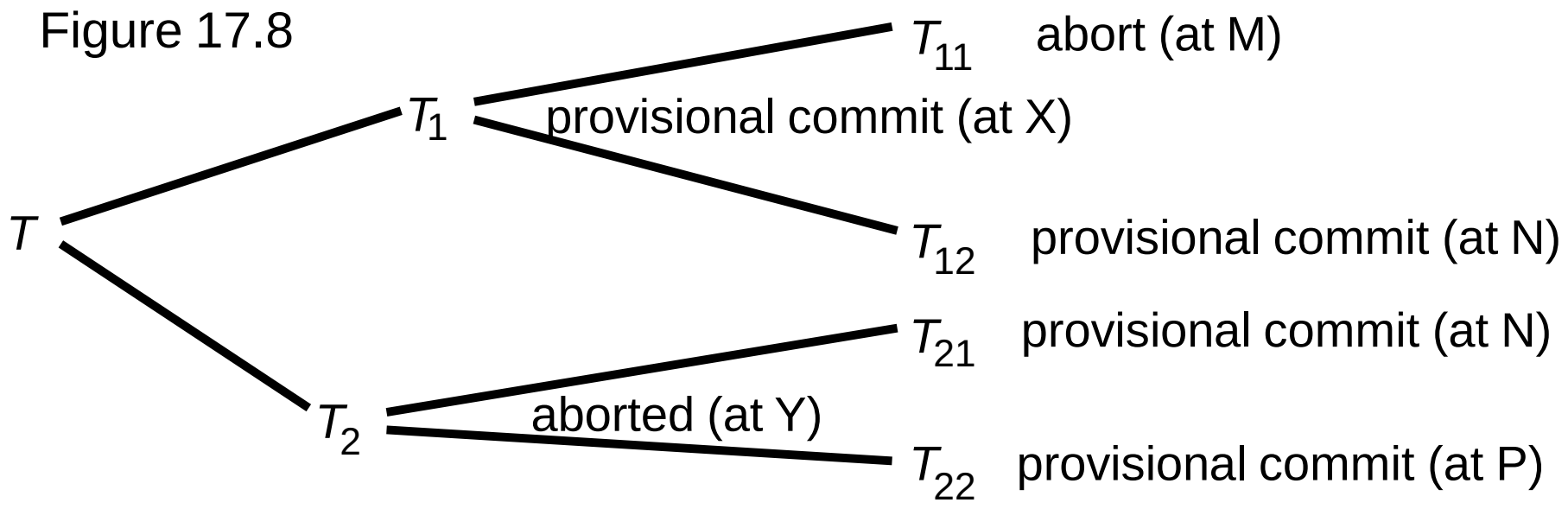
Figure 2.7
Operations in coordinator for nested transactions

openSubTransaction(trans) -> subTrans
Opens a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.
getStatus(trans)-> committed, aborted, provisional
Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed*, *aborted*, *provisional*.

- This is the interface of the coordinator of a subtransaction.
  - It allows it to open further subtransactions
  - It allows its subtransactions to enquire about its status
- Client starts by using *OpenTransaction* to open a top-level transaction.
  - This returns a TID for the top-level transaction
  - The TID can be used to open a subtransaction
    - The subtransaction automatically *joins* the parent and a TID is returned.
    - The TID of a subtransaction is an extension of its parent's TID, so that a subtransaction can work out the TID of the top-level transaction.
- The client finishes a set of nested transactions by calling *closeTransaction* or *abortTransacation* in the top-level transaction.

Transaction *T* decides whether to commit

Figure 17.8



- Recall that
  1. A parent can commit even if a subtransaction aborts
  2. If a parent aborts, then its subtransactions must abort
- In figure, each subtransaction has either provisionally committed or aborted
- T_{12} has provisionally committed and T_{11} has aborted, but the fate of T_{12} depends on its parent T_1 and eventually on the top-level transaction, T.
- Although T_{21} and T_{22} have both provisionally committed, T_2 has aborted and this means that T_{21} and T_{22} must also abort
- Suppose that T decides to commit although T_2 has aborted, also that T_1 decides to commit although T_{11} has aborted

Figure 2.9 Information held by coordinators of nested transactions

- When a top-level transcation commits it carries out a 2PC
- Each coordinator has a list of its subtransactions
- At provisional commit, a subtransaction reports its status and the status of its descendents to its parent
- If a subtransaction aborts, it tells its parent

Coordinator of transaction	*Child transactions*	*Participant*	*Provisional commit list*	*Abort list*
T	T_1, T_2	yes	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	yes	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	no (aborted)		T_2
T_{11}		no (aborted)		T_{11}
T_{12}, T_{21}		T_{12} but not T_{21}	T_{21}, T_{12}	
T_{22}		no (parent aborted)	T_{22}	

- T_{12} and T_{21} share a coordinator as they both run at server N
- When T_2 is aborted it tells T (no information about descendents)
- A subtransaction (e.g. T_{21} and T_{22}) is called an *orphan* if one of its ancestors aborts

Figure 2.10 *canCommit?* for hierarchic two-phase commit protocol

canCommit?(trans, subTrans) -> *Yes / No*
Call a coordinator to ask coordinator of child subtransaction whether it can commit a subtransaction *subTrans*. The first argument *trans* is the transaction identifier of top-level transaction. Participant replies with its vote *Yes* / *No*.

- Top-level transaction is coordinator of 2PC. Participant list:
  - the coordinators of all the subtransactions that have provisionally committed
  - but do not have an aborted ancestor
  - E.g. T, T1 and T12 in Figure 2.8
  - if they vote *yes*, they *prepare* to commit by saving state in permanent store
    - The state is marked as belonging to the top-level transaction
- The 2PC may be performed in a hierarchic or a flat manner
- Hierarchic 2PC - T asks *canCommit?* to T1 and T1 asks *canCommit?* to T12
- The *trans* argument is used when saving the objects in permanent storage
- The *subTrans* argument is use to find the subtransaction to vote on. If absent, vote *no*.

Figure 2.11 *canCommit?* for flat two-phase commit protocol

canCommit?(trans, abortList) -> *Yes / No*
Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote *Yes* / *No*.

- Flat 2PC
  - the coordinator of the top-level transaction sends *canCommit*? messages to the coordinators of all of the subtransactions in the provisional commit list.
    - in our example, T sends to the coordinators of T_1 and T_{12}.
    - the *trans* argument is the TID of the top-level transaction
    - the *abortList* argument gives all aborted subtransactions
      - e.g. server N has T_{12} prov committed and T_{21} aborted
    - On receiving *canCommit*, participant
      - looks in list of transactions for any that match *trans* (e.g. T_{12} and T_{21} at N)
      - it *prepares* any that have provisionally committed and are not in *abortList* and votes *yes*
      - if it can't find any it votes *no*

Time-out actions in nested 2PC

- With nested transactions delays can occur in the same three places as before
  - when a *participant* is prepared to *commit*
  - when a participant has finished but has not yet received *canCommit*?
  - when a coordinator is waiting for votes
- Fourth place:
  - provisionally committed subtransactions of aborted subtransactions e.g. T22 whose parent T2 has aborted
  - use *getStatus* on parent, whose coordinator should remain active for a while
  - If parent does not reply, then abort

Summary of 2PC

- A distributed transaction involves several different servers.
  - A nested transaction structure allows
    - additional concurrency and
    - independent committing by the servers in a distributed transaction.
- Atomicity requires that the servers participating in a distributed transaction either all commit it or all abort it.
  - Atomic commit protocols are designed to achieve this effect, even if servers crash during their execution.
  - Ehe 2PC protocol allows a server to abort unilaterally.
    - it includes timeout actions to deal with delays due to servers crashing.
    - 2PC protocol can take an unbounded amount of time to complete but is guaranteed to complete eventually.