DIGITAL NOTES
ON
DISTRIBUTED SYSTEMS

B.TECH III YEAR - I SEM
(2018-19)

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad– 500100, Telangana State, INDIA.

III Year B.Tech. IT - I Sem

L	T/P/D	C
4	-/-/-	3

(R15A0524)DISTRIBUTED SYSTEMS
(Elective-I)

Objectives:

- To learn the principles, architectures, algorithms and programming models used in distributed systems.
- To examine state-of-the-art distributed systems, such as Google File System.
- To design and implement sample distributed systems.

UNIT I

Characterization of Distributed Systems: Introduction, Examples of Distributed systems, Resource sharing and web, challenges.

System models: Introduction, Architectural and Fundamental models, networking and Internetworking, Interposes Communication.

UNIT II

Time and Global States: Introduction, Clocks, events and Process states, Synchronizing physical clocks, logical time and logical clocks, global States, distributed debugging.

Coordination and Agreement: Introduction, Distributed mutual exclusion, Elections, Multicast communication, consensus and related problems.

UNIT III

Inter process Communication: Introduction ,The API for the Internet Protocols, External Data Representation and Marshalling, Client –Server Communication, Group Communication, Case Study: IPC in UNIX.

Distributed Objects and Remote Invocation: Introduction, Communication between distributed objects, Remote Procedure Call, Events and Notifications, Case Study: JAVA RMI

UNIT IV

Distributed File Systems: Introduction, File Service Architecture, Case Study
1: Sun Network File System, Case Study2:The Andrew File System

Name Services: Name Services: Introduction, Name Services and the Domain Name System, Case study of the Global Name Service

Distributed Shared Memory: Introduction, Design and Implementation issues, Sequential consistency and Ivy case study, Release consistency and Munin case study, Other consistency models.

UNIT V

Transactions and Concurrency control: Introduction, Transactions, Nested Transactions, Locks, optimistic concurrency control, Timestamp ordering, Comparison of methods for concurrency control.

Distributed Transactions: Distributed Transactions: Introduction, Flat and Nested Distributed Transactions, Atomic commit protocols, Concurrency control in distributed transactions, Distributed deadlocks, Transaction recovery.

TEXT BOOKS:

1. Distributed Systems Concepts and Design, G Coulouris, J Dollimore and T Kindberg, Fourth Edition, Pearson Education. 2009.

REFERENCES:

1. Distributed Systems, Principles and paradigms, Andrew S.Tanenbaum, Maarten Van Steen, Second Edition, PHI.
2. Distributed Systems, An Algorithm Approach, Sikumar Ghosh, Chapman & Hall/CRC, Taylor & Fransis Group, 2007.

Course Outcomes:

1. Students will identify the core concepts of distributed systems: the way in which several machines orchestrate to correctly solve problems in an efficient, reliable and scalable way.
2. Students will examine how existing systems have applied the concepts of distributed systems in designing large systems, and will additionally apply these concepts to develop sample systems.

INDEX

UNIT - I

INTRODUCTION:

Definition of a Distributed System

Distributed System is a collection of autonomous computers connected through network and middleware. Users perceive the system as a single integrated computing facility.

- **G. Coulouris, J. Dollimore, T. Kindberg: A system in which hardware and software components located at networked computers communicate and coordinate their actions only by message passing**

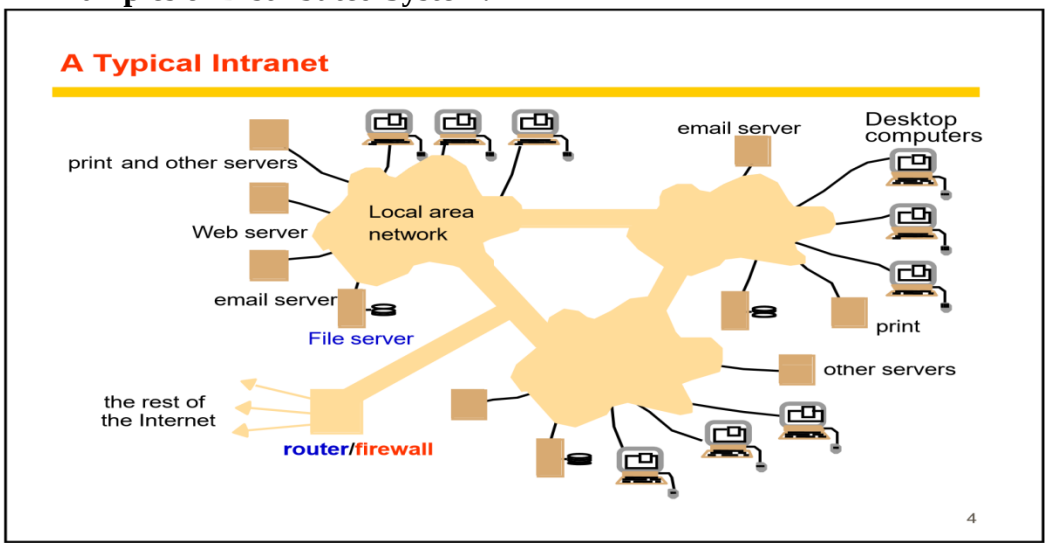
- **A. Tanenbaum and M. Steen: a collection of independent computers that appears to its uses as a single coherent system**

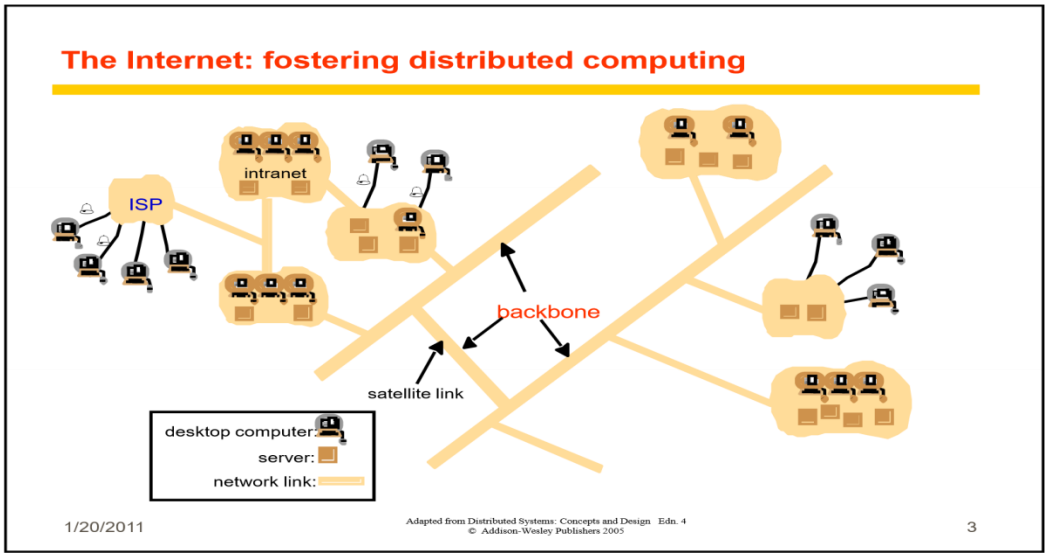
Characteristics of Distributed System:

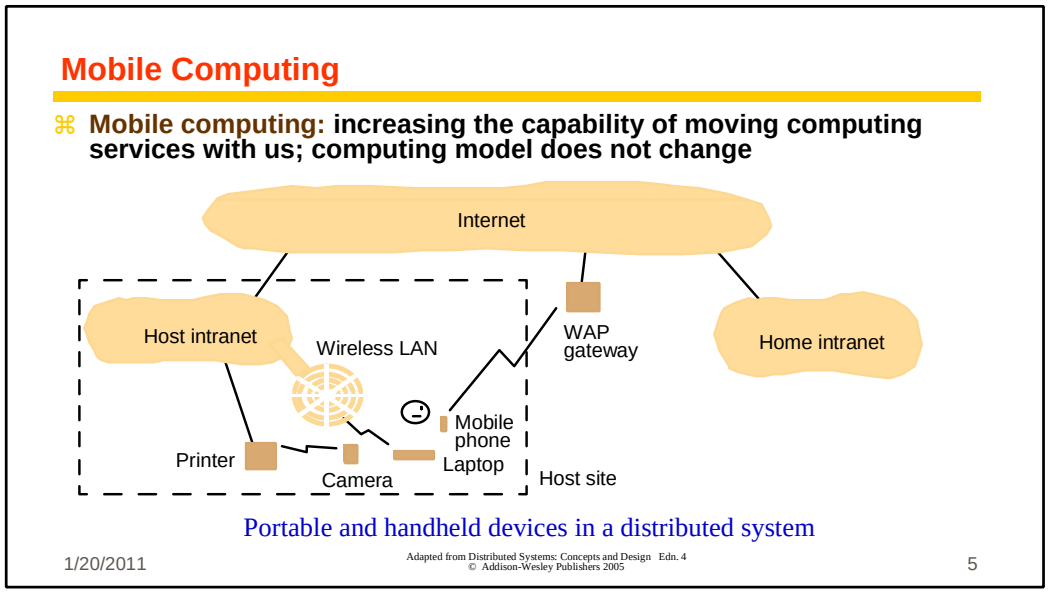
1. Concurrency of components
2. Lack of a global clock.
3. Independent failures of components

Applications of Distributed System: Telecommunication network, Network applications, Real-Time process control, Parallel computation etc.

Examples of Distributed System:







Key Characteristics of Distributed Systems

- **Raise three key characteristics of distributed systems**
- **Concurrency: concurrent program execution and resource access** How to coordinate actions?
- **No global clock: the limits to the accuracy with which the computers in a network can synchronize their clocks**
  - The only communication is messaging passing

- **Independent failures: each component of the system can fail independently, leaving the others still running**
  - How to tell if the network has failed or become unusually slow?

CHALLENGES AND ISSUES:

OPENNESS
HETEROGENEITY
SECURITY
SCALABILITY
FAILURE HANDLING
CONCURRENCY
TRANSPARENCIES

Distributed vs. Centralized Systems

- **Advantages of Distributed Systems**
  - **Resource sharing**
  - **Reliability**
  - **aggregate computing power**
  - **scalability / openness**

- **Disadvantage of Distributed Systems**
  - **Security**
  - **computing power per node is limited**

Distributed Operating Systems

- **Requirements**
  - **Provide user with a single coherent computer system**
  - **Hide distribution of resources**
  - **Mechanisms for resource protection**
  - **Secure communication**

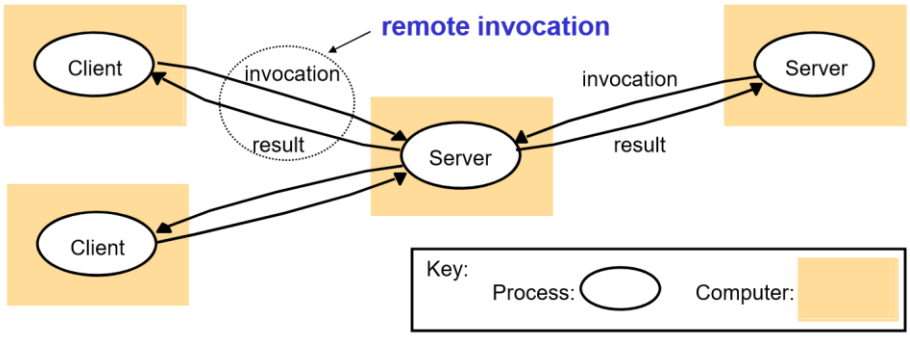
- **Definition of Distributed OS**
  - **Look to user like ordinary centralized OS, but runs on multiple and independent CPUs**
    - **use of multiple processors is invisible**
    - **user views system as a virtual uniprocessors**

System Models

Architectural Models

- The architecture of a system is its structure in terms of separately specified components
- Two main issues about an architectural model
  - the placement of the components across a network of computers
  - the inter-relationships between the components

CLIENT-SERVER SYSTEM MODEL



Interaction Model

- A distributed system are composed of interacting processes
  - Multiple server processes cooperate to provide a service

Peer processes cooperate to achieve a common goal

Synchronous Distributed Systems

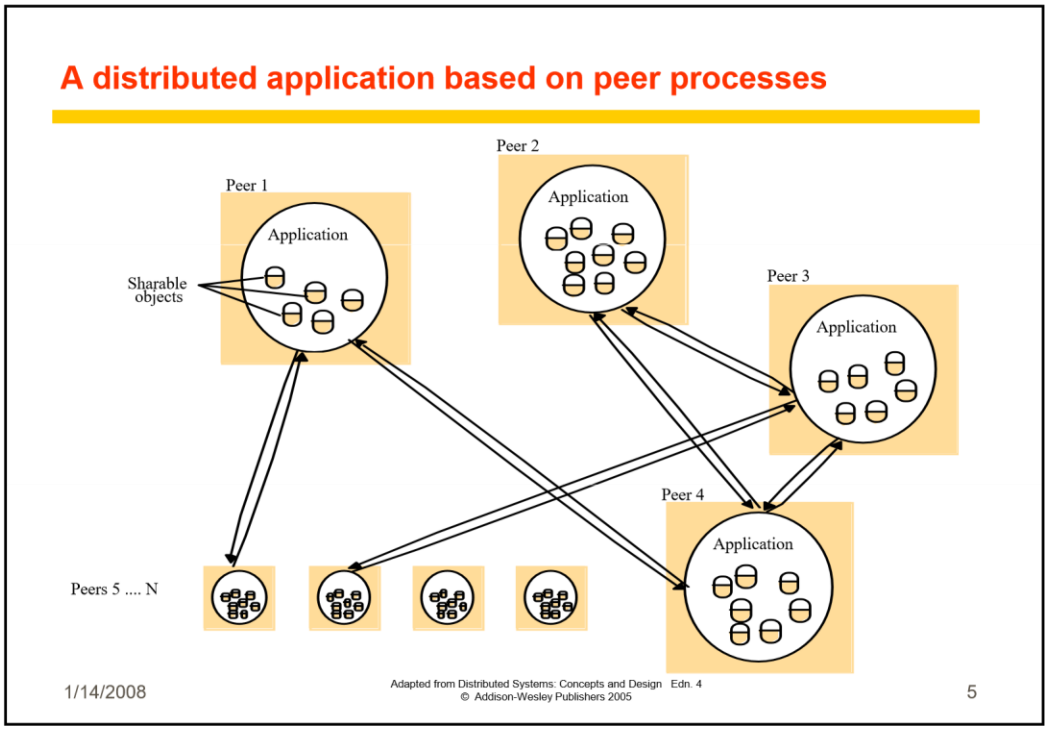
- Process execution time
  - The time to execute each step of a process has known lower and upper bounds
- Message delivery time

Asynchronous Distributed Systems

- There are no bounds on
  - Process execution time: each step of a process may take an arbitrarily long time
  - Message delivery time: a message may be received after an arbitrarily long time
  - Clock drift rate: a clock's clock drift rate is arbitrary

- Observations:
  - Actual distributed systems are very often asynchronous because of the needs to share the processors and for communication channels to share the network
  - Any solutions valid for synchronous also for asynchronous

- **Time-outs may NOT be used for failure detection, but can be used for failure masking (failure suspect)**



UNIT II

Coordination and Agreement

Failure Assumption and Failure Detectors

- Each pair of processes is connected by reliable channels
- Unreliable failure detectors
- Two values: unsuspected or suspected, both unreliable
- Failure-detection algorithm based on T + D criteria (T is the period of sending query and D is the estimated maximum message transmission)
- False positive and false negative
- Reliable failure detectors
- Feasible in synchronous

Re: Mutual Exclusion –Race Conditions

- **Race conditions: when two or more processes/threads are reading or writing some *shared data* and the final results depends on who runs precisely**
- **Critical region/section: the part of the program where the shared memory is accessed**

Thread 1	Thread 2	Balance
Read *balance*; $1000		$1000
	Read *balance*; $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update *balance* $1000 + $200		$1200
	Update *balance* $1000 + $200	$1200

time

Two threads want to deposit an account; *overwriting* issue

4

Distributed Mutual Exclusion

- Master/slave model and P2P decentralized model
- Requirements for mutual exclusion
- ME1 (safety): at most one process may execute in the critical section at a time
- ME 2 (liveness): requests to enter and exit the critical section eventually succeed (no deadlock or starvation)
- ME 3 (→ordering for more fairness): if one request to enter the critical section *happened-before* another, then entry is granted in that order
- Application-level protocol for executing a critical section
- enter() + resourceAccesses() + exit()
- Assumes each pair of processes is connected by reliable channels

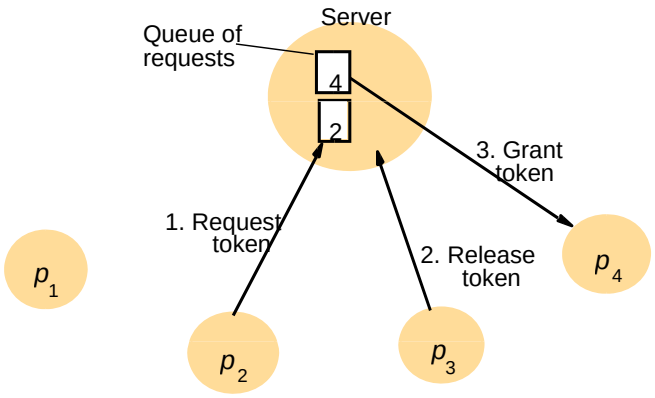
4/6/2011 5

Algorithms for Distributed Mutual Exclusion

- Master/slave model and P2P decentralized model
- Performance Metrics
- The *bandwidth* consumed, which is proportional to the number of messages sent in each entry and exit operation
- The *client delay* incurred by a process at each entry and exit operation
- The *throughput* of the system (the collection of processes), measured by the effect using the *synchronization delay* between one process exiting the critical section and the next process entering it (the shorter it is, the greater the throughput is)
- Fault tolerance
- What happens when messages are lost? • What happens when a process crashes?

6

Master/Slave Model –The Central Server Algorithm



- A central server grants permission (a token) to enter the critical section
- Discuss the ME1, ME2, and ME3 requirements (Why ME3 not met)?
- Discuss the performance of the algorithm

4/6/2011 7

Ricart and Agrawala's Multicast-based Algorithm

Basic idea? uses multicast to announce entry

On initialization request and enter it only all the other processes *state* := RELEASED; have replied to this request

To enter the section
state := WANTED;
Multicast *request* to all processes request processing deferred here
T := request's *Lamport 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;

Elections

- Election: to choose a unique process to play a particular role• In "central-server" mutual exclusion: to choose which of the processes to play the role of the server, and to choose a replacement if needed
- It is essential all the processes agree on the choice
- Action and State
- To call the election (*N* processes could call *N* concurrent elections)
- An any point of time, a process is either a participant or non-participant

The Bully Algorithm

- Assumptions
- Processes may crash during an election, but message delivery between processes is reliable
- System is synchronous: it uses timeout to detect a process failure
- Each process knows which processes have higher IDs that it can communicate with all such processes
- what is the case in the Ring-based algorithm?

- Messages
- Election: to announce an election
- Answer: a message in response to an election message
- Coordinator: a message to announce the ID of the elected process

- Failure detection
- A reliable failure detector is based on threshold T = 2 T_trans + T_process

Introduction to Multicast

- Multicast communication requires coordination and agreement. The aim is for members of a group to receive copies of messages sent to the group
- Many different delivery guarantees are possible
  - e.g. agree on the set of messages received or on delivery ordering
- A process can multicast by the use of a single operation instead of a send to each member
  - For example in IP multicast by Java *aSocket.send*(*aMessage*)
  - The single operation allows for:
    - *efficiency* i.e. send once on each link, using hardware multicast when available, e.g. multicast from a computer in London to two in Beijing
    - *delivery guarantees* e.g. can't make a guarantee if multicast is implemented as multiple sends and the sender fails (or sender fails halfway, leading to some-yes-some-no). IP multicast does not guarantee ordering or reliability, but can be enhanced

System Model of Multicast

- The system consists of a collection of processes which can communicate *reliably* over 1-1 channels
- Processes fail only by crashing (no arbitrary failures)
- Processes are members of groups - which are the destinations of multicast messages
- In general process *p* can belong to more than one group (for simplicity of discussion, at most one group at a time here)
- Operations
  - *multicast*(*g*, *m*) sends message *m* to all members of process group *g*
  - *deliver* (*m*) is called to get a multicast message delivered. It is different from *receive* as it may be delayed to allow for ordering or reliability.
- Multicast message *m* carries the *id* of the sending process *sender*(*m*) and the *id* of the destination group *group*(*m*)
- We assume there is no falsification of the origin and destination of messages

TIME AND GLOBAL STATES

Introduction

- We need to measure time accurately
  - to know the time an event occurred at a computer
  - to do this we need to synchronize its clock with an authoritative external clock
- Algorithms for clock synchronization useful for
  - concurrency control based on timestamp ordering
  - authenticity of requests e.g. in Kerberos
- There is no global clock in a distributed system
  - Issues of clock accuracy and synchronization
- Logical time is an alternative for
  - ordering of events - also useful for consistency of replicated data

Computer Clocks and Timing Events

- Each computer in a DS has its own internal clock
  - used by local processes to obtain the value of the current time
  - processes on different computers can timestamp their events
  - but clocks on different computers may give different times

Hardware & Software Clocks

We have seen how to order events at a process

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 number giving nanoseconds since some base time
- 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

- computer clocks drift from perfect time and their drift rates differ from one another.
- ***clock drift rate***: the relative amount that a computer clock differs from a perfect clock

Even if clocks on all computers in a DS are set to the same time, their clocks will eventually vary quite significantly unless corrections are applied

Coordinated Universal Time (UTC)

International Atomic Time is based on very accurate physical clocks (drift rate 10^{-13})

UTC is an international standard for time keeping

It is based on atomic time, but occasionally adjusted to astronomical time

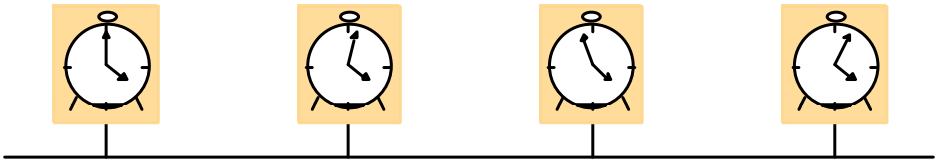
It is broadcast from radio stations on land and satellite (e.g. GPS)

Computers with receivers can synchronize their clocks with these timing signals

Signals from land-based stations are accurate to about 0.1-10 millisecond

Signals from GPS are accurate to about 1 microsecond

SKEW BETWEEN COMPUTER CLOCKS



Network

Computer clocks are not generally in perfect agreement

Skew: the difference between the times on two clocks (at any instant)

Computer clocks are subject to *clock drift* (they count time at different rates)

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

High precision quartz clocks drift rate is about 10

Synchronizing Physical Clocks: External and Internal

External synchronization

– A computer's clock C_i is synchronized with an external authoritative time source S, so that:

Cristian's Method (1989) 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

– $|S(t) - C_i(t)| < D$ for $i = 1, 2, \ldots N$ and for all real time t in I.

– The clocks C_i are accurate to within the bound D. Internal synchronization

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

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

Berkeley Algorithm

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

NTP - Synchronisation of Servers

The synchronization subnet can reconfigure if failures occur, e.g.

- a primary that loses its UTC source can become a secondary

a secondary that loses its primary can use another primaryModes

Messages Exchanged between a Pair of NTP Peers

What is the key problem in internal synchronization?

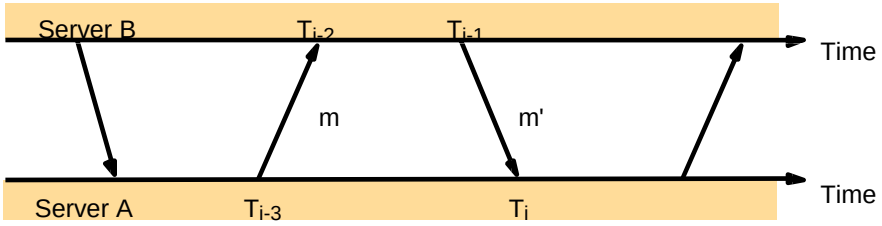
- What is the bound of the clock offset/skew between two servers?

Each message bears (3) 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 arrival of one message and the dispatch of the next



14

Clocks, Events and Process States

A distributed system is defined as 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 e.g. *Send, Receive,* change state

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 *e* occurs before *e'* at p_i

A history of process p_i is a series of events ordered by $\rightarrow_i$*history*(p_i)= $h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

The 'happened before' relation is essential to understanding logical clocks

16

Logical Time and Logical Clocks (Lamport 1978)

Instead of synchronizing clocks, event ordering can be used

Event ordering in a single process is easy

– Event ordering in a distributed system is more complex

1. If two events occurred at the same process p_i (i= 1, 2, … N) then they occurred in the order observed by p_i, that is $\rightarrow_i$
2. when a message, m is sent between two processes, *send(m)* happened before *receive(m)*
3. The *happened before relation* is transitive
   e ↛ e` and e` ↛ e``, then e ↛ e``

HB1, HB2 and HB3 (page 397) are formal statements of these 3 points

Lamport's Logical Clocks

A logical clock is a monotonically increasing software counter. It need not relate to a physical clock.

– To numerically capture happened-before ordering

Each process p_i has a logical clock, L_i which can be used to apply logical timestamps to events

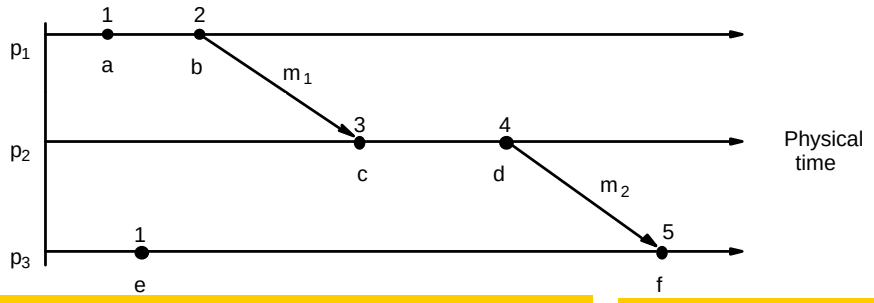
– Li(e): the timestamp of event *e* at p_i

– L(e): the timestamp of event *e* at any process

– LC1: L_i is incremented by 1 before each event is issued at process p_i – LC2:

– (a) when process p_i sends message m, it piggybacks on m 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)*

An Example of Lamport's logical clocks



each of p1, p2, p3 has its logical clock initialised to zero, the clock values are those immediately after the event. e.g. 1 for a, 2 for b.

for m_1, 2 is piggybacked and c gets $max(0,2)+1 = 3$

Give an example to show theconverse is not true

$e \rightarrow e'$ implies $L(e) < L(e')$ The converse is not true, that is $L(e) < L(e')$ does not imply $e \rightarrow e'$

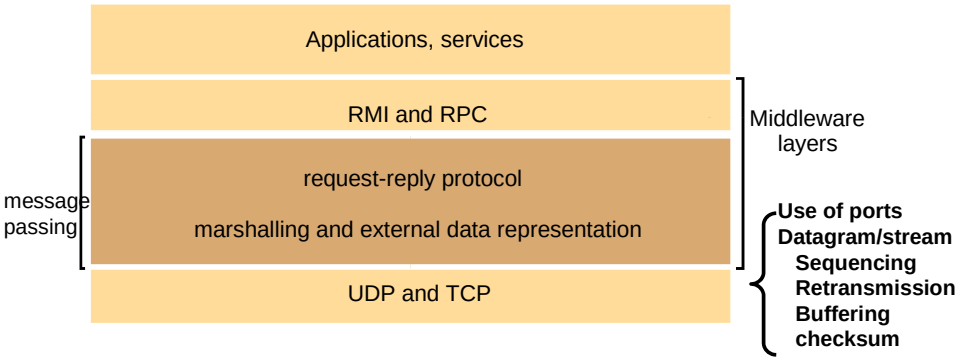
e.g. $L(b) > L(e)$ but $b \parallel e$

20

UNIT III

Interprocess Communication

MIDDLEWARE LAYERS

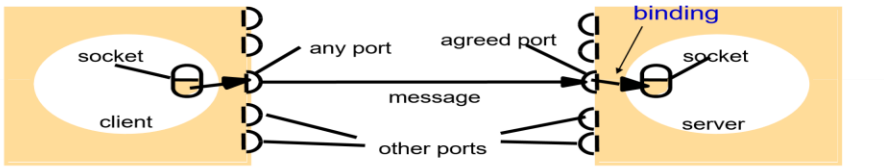


Synchronous and Asynchronous Communication

- **Synchronous communication:**
  - sending and receiving processes synchronize at every message
  - Both *send* and *receive* are *blocking* operations
- **Asynchronous communication:**
  - Messages need to be buffered
  - *send* is non-blocking, and *receive* can be blocking or non-blocking
  - Non-blocking *receive* is unusual, which involves extra complexity and requires to separately receive notification that its buffer has been filled by polling or interrupt

1/14/2008 Adapted from Distributed Systems: Concepts and Design Edn. 4 © Addison-Wesley Publishers 2005 5

Message Destinations – Sockets and ports

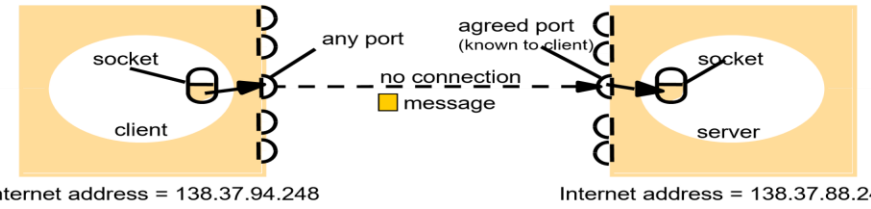


Internet address = 138.37.94.248 Internet address = 138.37.88.249

- **Internet address, local port**
  - Socket: an endpoint for communication between processes
  - **One port has one receiver (process), but can have many senders**
  - **One process can use multiple ports**

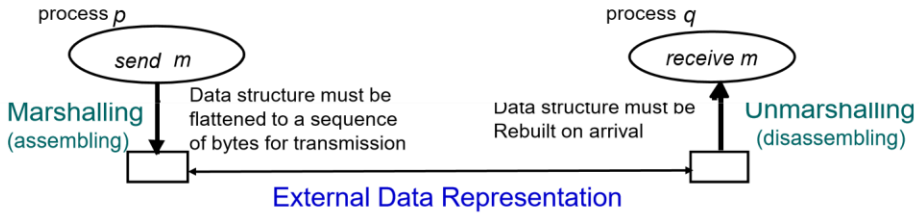
1/14/2008 Adapted from Distributed Systems: Concepts and Design Edn. 4 © Addison-Wesley Publishers 2005 7

UDP Datagram Communication



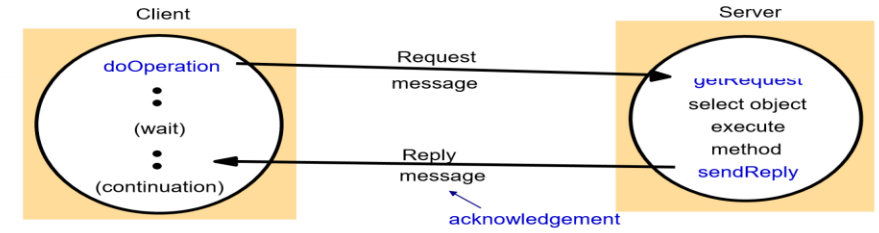
- **UDP: The delivery of the message is not guaranteed**
  - Message size: up to 2^16 B (usual restriction 8 KB)
  - Non-blocking *send* and blocking *receive*
  - Timeout: to avoid infinite wait of blocking *receive*
  - Receive from any
  - Ordering: messages can be delivered out of sender order
  - Omission failures: send-omission, receive-omission, channel-omission

External Data Representation and Marshalling



- **Heterogeneity:**
  - **Computers store primitive values such as integers & FP in different orders, big-endian vs. little-endian**
  - **Different coding schemes for character representation, ASCII vs. Unicode**
- **Who does marshalling and unmarshalling?**
  - **Middleware for transparency, and efficiency**

Client-Server Communication



- Request-reply communication (one-to-one):
  - To support message exchanges in typical client-server interactions
  - Synchronous vs. asynchronous communication
  - send and receive operations in UDP are lightweight
    - But TCP does not have limitations over the buffer size which UDP does

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

Request-Reply Message Structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

e object references:

nique identifier for a remote objec
tem

32 bits *32 bits* *32 bits* *32 bits*

Internet addre	ssport number	time	object numbe	r interface of remote object

How about failures?
- Timeout
- Discarding duplicate request messages
- Lost reply messages: *idempotent*
- History: *retransmission vs. re-execution*

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

IPC in Unix BSD: Socket Primitives for TCP/IP

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

IPC in UNIX: UDP Sockets

- **IPC primitives provided as system calls in BSD 4.x**
  - **A socket call returns a descriptor (identifier)**
  - **Binding: bind the descriptor to a socket address before communication**
    - Java API: socket creation and name binding are integrated

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress,size of CA)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress, size of CA)
•
•
amount = recvfrom(s, buffer, from)
```

- *ServerAddress* and *ClientAddress* are socket addresses (*struct sockaddr_in sa*)
- sendto() returns the actual number of bytes sent
- recvfrom() blocks if the queue is empty until a message arrives

IPC in UNIX: UDP Sockets

- **IPC primitives provided as system calls in BSD 4.x**
  - **A socket call returns a descriptor (identifier)**
  - **Binding: bind the descriptor to a socket address before communication**
    - Java API: socket creation and name binding are integrated

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress,size of CA)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message

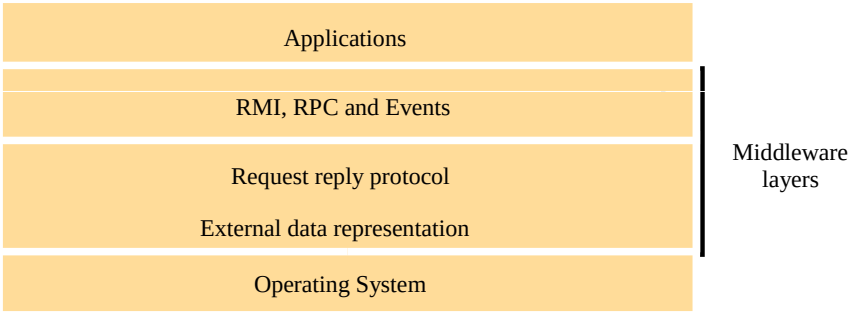
```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress, size of CA)
•
•
amount = recvfrom(s, buffer, from)
```

- *ServerAddress* and *ClientAddress* are socket addresses (*struct sockaddr_in sa*)
- sendto() returns the actual number of bytes sent
- recvfrom() blocks if the queue is empty until a message arrives

1/14/2008 Adapted from Distributed Systems: Concepts and Design Edn. 4 © Addison-Wesley Publishers 2005 23

RMI and RPC

Middleware Layers



Distributed applications composed of cooperating programs running in different processes (often in different computers)

– Programming models: how to invoke operations in other processes

Middleware Programming Models

Middleware provides a programming abstraction and masks the heterogeneity of networks and OS etc.

– Location transparency and independence from the details of communication protocols, operating systems, and computer hardware.

Distributed objects and remote object invocation –

illustrated by Java RMI

CORBA

– it provides remote object invocation between a client program written in one language and a server program written in another language (IDL)

– Illustated by Ja a COr v RBA (aoten h r la guage con mmo y used s Cnl i ++)

Other programming models

– remote event notification

– remote SQL access

– distributed transaction processing

Interface

Ainterface: specifies the procedures/methods and the variables that can be access from other modules / objects

– The implementation may be changed w/o affecting the users of the modules, if the interface remaining the same

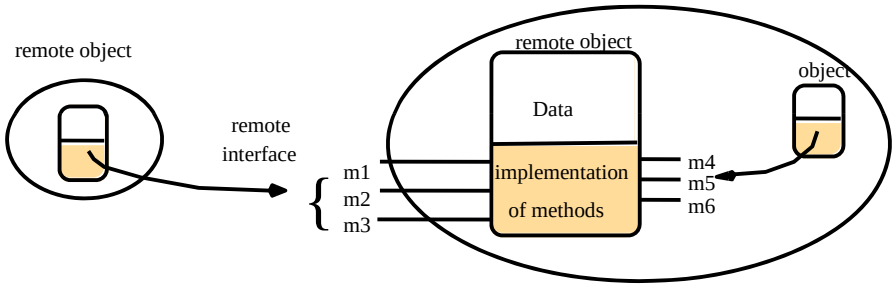
A service interface in RPC: refers to the specification of the procedures offered by a server, defining the types of the input and output arguments of each of the procedures

A remote interface in RMI: specifies the methods of an object for invocation by objects in other processes, defining the types of the input and output arguments

Differences between local and remote modules?

4

Object, Remote Object and Remote Interface



Object

– Object reference

–Interface

Distributed object system architectures

–Client-server: RMI

–Others

1/14/2008

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

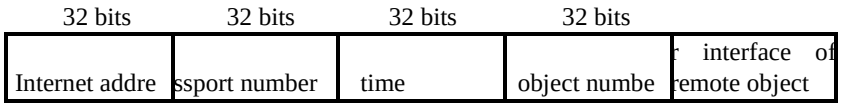
5

Remote Object References

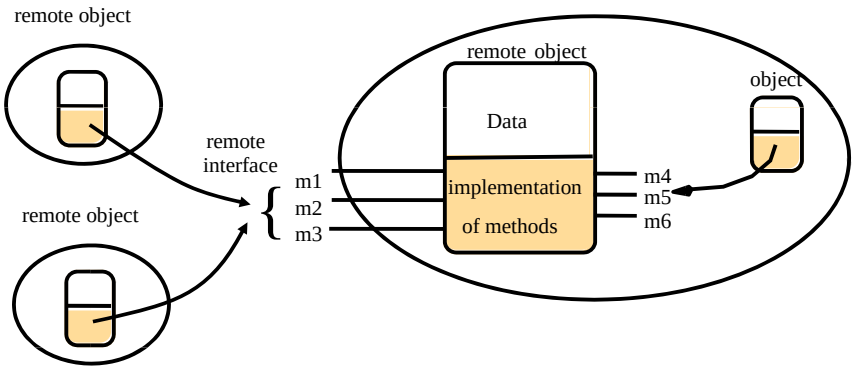
The remote object to receive a remote method invocation is specified as a remote object reference

Remote object references may be passed as arguments and results of remote method invocations

A Remote object reference must be unique as an identifier for a remote object in a distributed system



Distributed Objects & Concurrent Access



Concurrent access: an object may be accessed concurrently

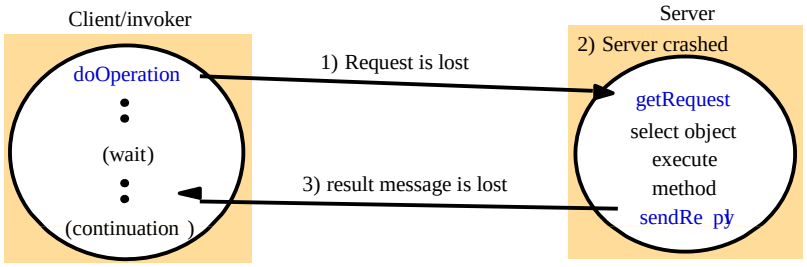
- **Encapsulation allows objects to provide methods for protecting themselves against incorrect accesses; use synchronization primitives such as condition variables to protect access to their instance variables**

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

RMI Semantics: Maybe

Invoker cannot tell if a remote method has been executed or not

- **If the result message has not been received after a timeout, how conclude? It even can come after timeout in a asynchronous system**



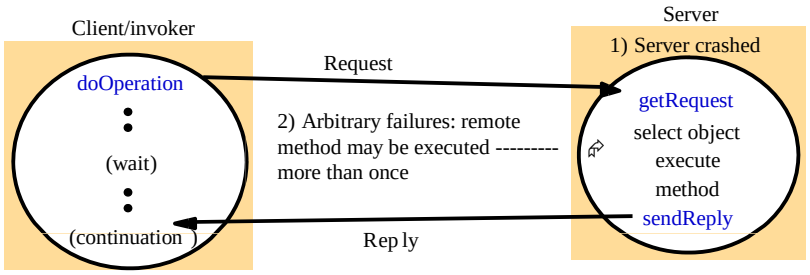
Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply	semantics
No	Not applicable	Not applicable	Maybe

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

RMI Semantics: At-least-once

Invoker receives either a result, or an exception (no results)

- **Retransmission masks omission failures of the invocation/result;**
- **Acceptable: if methods are idempotent operations ; SUN RPC**



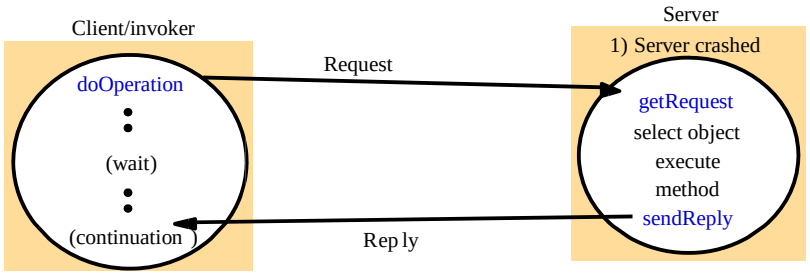
Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply	semantics
Yes	Not	Re-execute	At-least-once

12

RMI Semantics: At-most-once

Invoker receives either a result, or an exception (no results)

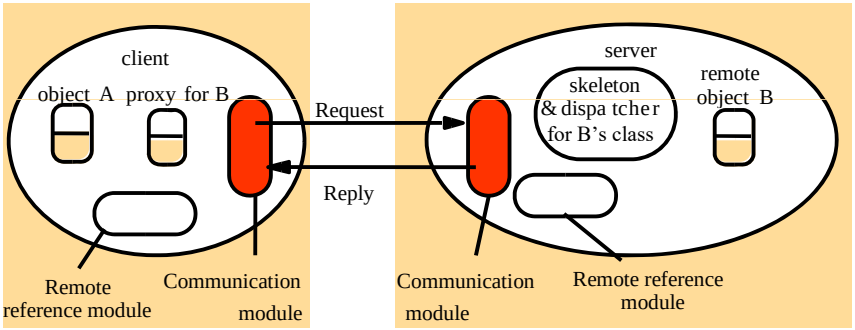
- **Retransmission masks omission failures of the invocation/result;**
- **Retransmission reply: no arbitrary failures; Java RMI & CORBA**



Retransmit request message	Duplicate filtering	Re-execute procedure or retransmit reply	semantics
Yes	Yes	Retransmit reply	At-most-once

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

The Architecture of RMI: Communication Module

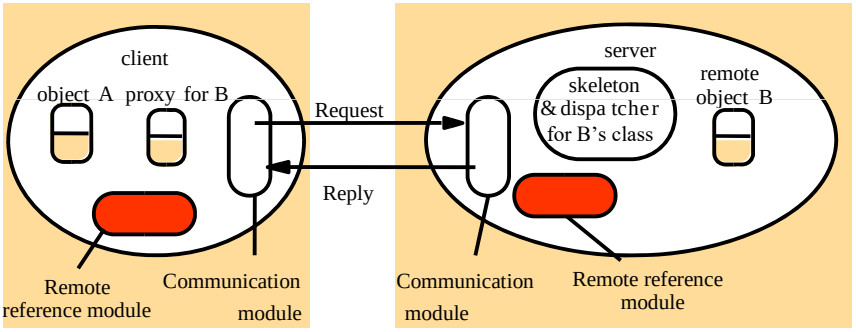


Communication modules carry out the request-reply protocol

- **Responsible for providing a specific invocation semantics**
- **The module in the server selects the dispatcher for the object to be invoked, passing on its local reference (translated from the remote object reference)**

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

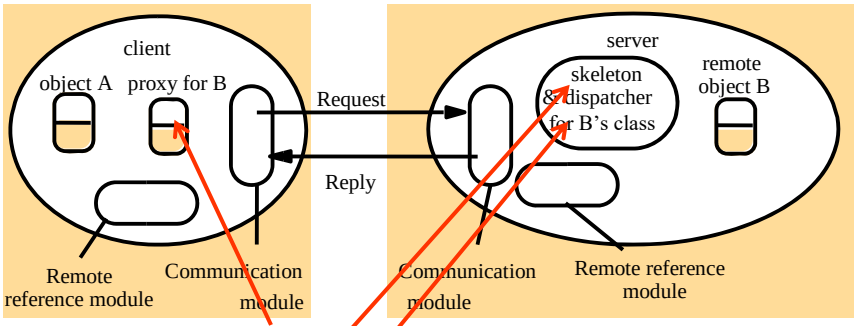
The Architecture of RMI: Remote Reference Module



Remote reference module

- **Responsible for translating between local and remote object references and creating remote object references**
- **A remote object table records the mapping between local object references and remote object references (server: object, client: proxy)**

The Architecture of RMI: RMI Middleware Software



Proxy - makes RMI transparent to client. Class implements remote interface. Marshalls requests and unmarshalls results. Sending and receiving messages from the clients.

Dispatcher - gets request from communication module and invokes method in skeleton (using methodID in message).

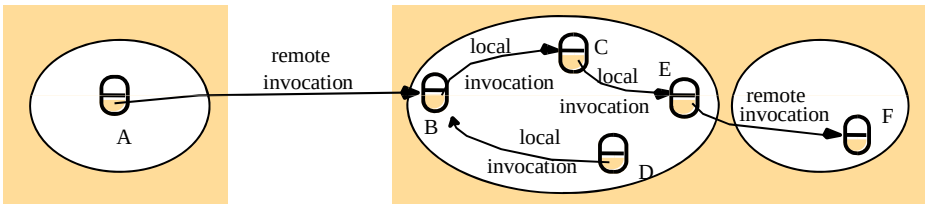
Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.

RMI software -between application level objects and communication and remote reference modules (marshalling and unmarshalling)

All generated by interface compiler

Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

Binder

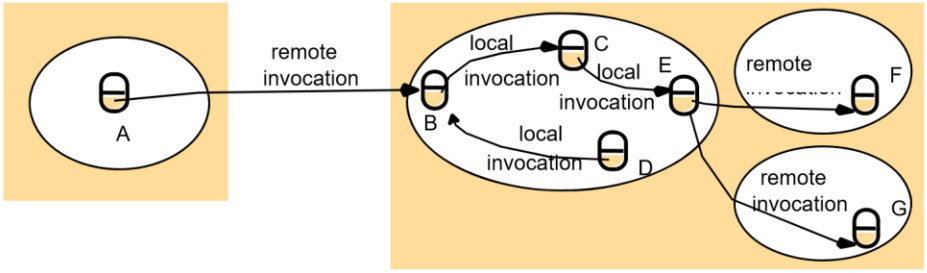


Objects need to know the remote object reference of an object in another process to invoke its methods. How do they get it?

Object A can get a remote reference to object F from object B as the results of remote method invocation. How A knows B?

Binder: a service maintains a table containing the mappings from textual names to remote object references. Used by servers to register their remote objects by name and by clients to look them up.

Server Threads and Concurrent Executions

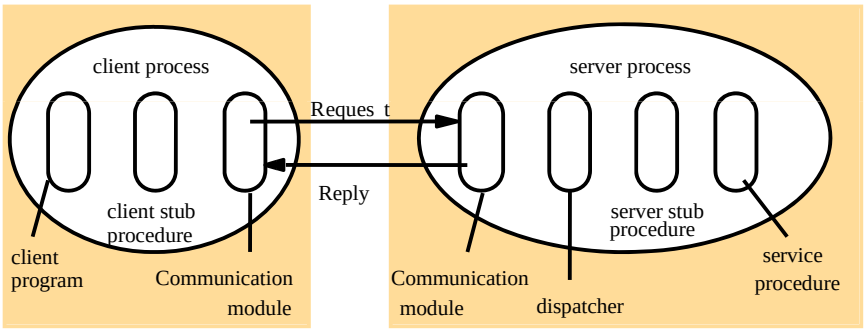


How to avoid the execution of one remote invocation delaying the execution of another?

servers generally allocate a separate thread for the execution of each remote invocation

- **The implementation of a remote object must allow for concurrent execution on its state**

Remote Procedure Call (RPC)

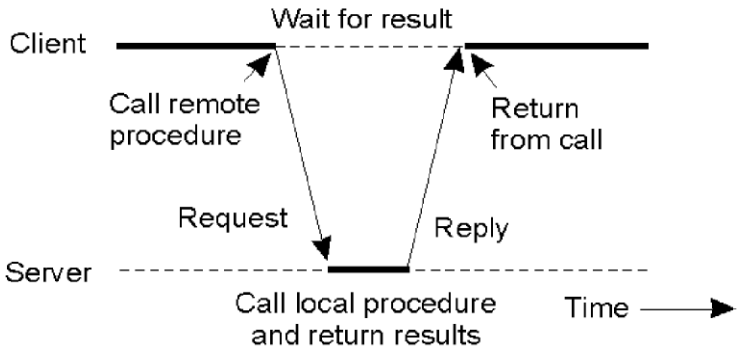


RPC vs. RMI (both rely on a request-reply protocol)

- **Procedure call, instead of object invocation**
- **Procedure ID vs. object reference**
- **Client's stub procedure is similar to a proxy; by interface compiler (rpcgen)**
- **Server's stub procedure is similar to a skeleton; by interface compiler (…)**

Client and Server Stubs

Principle of RPC between a client and server program.



Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Weslev Publishers 2005

Summary

Heterogeneity is an important challenge to designers :

- Distributed systems must be constructed from a variety of different networks, operating systems, computer hardware and programming languages.
  - The Internet communication protocols mask the difference in networks and middleware can deal with the other differences.

External data representation and marshalling

- CORBA marshals data for use by recipients that have prior knowledge of the types of its components. It uses an IDL specification of the data types
- Java serializes data to include information about the types of its contents, allowing the recipient to reconstruct it. It uses reflection to do this.

RMI (& RPC)

- each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely.
- local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once
- Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers.

UNIT IV

Distributed File Systems

Storage Systems and Their Properties

What is persistence? How it is achieved?
Types of consistency between copies: 1 - strict one-copy consistency
√ - approximate consistency
X - no automatic consistency

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (Ch. 16)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Persistent distributed object store	✓	✓	✓	✓	PerDiS, Khazana

4

What is a File System? *1*

- Stored data sets
  - Responsible for organization, storage, retrieval, naming, sharing, and protection of files
- Hierarchic name space visible to all processes
- API with the following characteristics:
  - access and update operations on persistently stored data sets
  - Sequential access model (with additional random facilities) • Sharing of data between users, with access control
- Concurrent access:
  - certainly for read-only access – what about updates? • Other features:
  - mountable file stores
  - more? ... What are key requirements in a file system?

2/21/2011 5

UNIX File System 2

UNIX file system operations (system calls accessed via library procedures)

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name (and mode)*. 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) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes*to *buffer*. 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*).
status = stat(name, buffer)	Gets the file attributes for file *name* into *buffer*.

2/21/2011 6

File System Modules 3

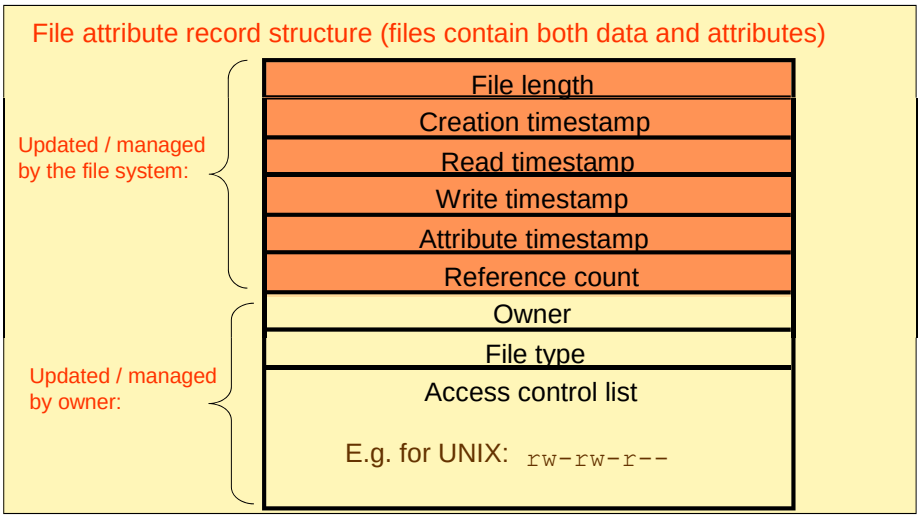
Layered modules of non-distributed file systems

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

- What additional components a distributed file system needs?

2/21/2011 7

File Attribute Record Structure 4



8

DFS Requirements: Transparency and Concurrency

Transparencies

Access: Same operations; unaware of distributed of files

Location: Same name space after relocation of files or processes; w/o changing pathnames

Mobility: Automatic relocation of files is possible

Performance: Satisfactory performance across a specified range of system loads

Scaling: Service can be expanded to meet additional loads

Concurrency properties

Isolation of one operation from a simultaneous one

UNIX File-level or record-level locking

Other forms of concurrency control to minimize contention to shared data

2/21/2011 9

DFS Requirements: Replication and Heterogeneity

Replication properties

File service maintains multiple identical copies of files, major advantages:

- Load-sharing between servers makes service more scalable
- Local access has better response (lower latency)
- Fault tolerance major issues:

  Full replication is difficult to implement.

  Caching (of all or part of a file) gives most of the benefits (except fault tolerance)

Heterogeneity properties

Service can be accessed by clients running on (almost) any OS or hardware platform.

Design must be compatible with the file systems of different OSs

Service interfaces must be *open* - precise specifications of APIs are published.

1/2011

10

DFS Requirements: Fault-tolerance and Consistency

Fault tolerance

Service must continue to operate even when clients make errors or crash.

- at-most-once semantics
- at-least-once semantics

  requires idempotent operations

  Service must resume after a stateless server crashes.

  If the service is replicated, it can continue to operate even during a server crash.

Consistency

Unix offers one-copy update semantics for operations on local files - caching is completely transparent.

Difficult to achieve the same for distributed file systems while maintaining good performance and scalability.

DFS: Security and Efficiency

Security

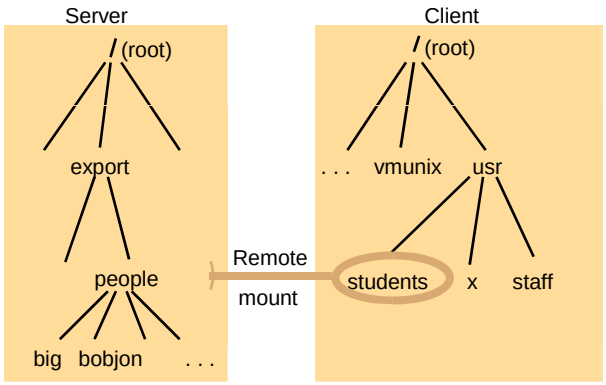
Must maintain access control and privacy as for local files (access control lists).

- based on identity of user making request
- identities of remote users must be authenticated
- privacy requires secure communication

Efficiency

Goal for distributed file systems is usually performance comparable to local file system.

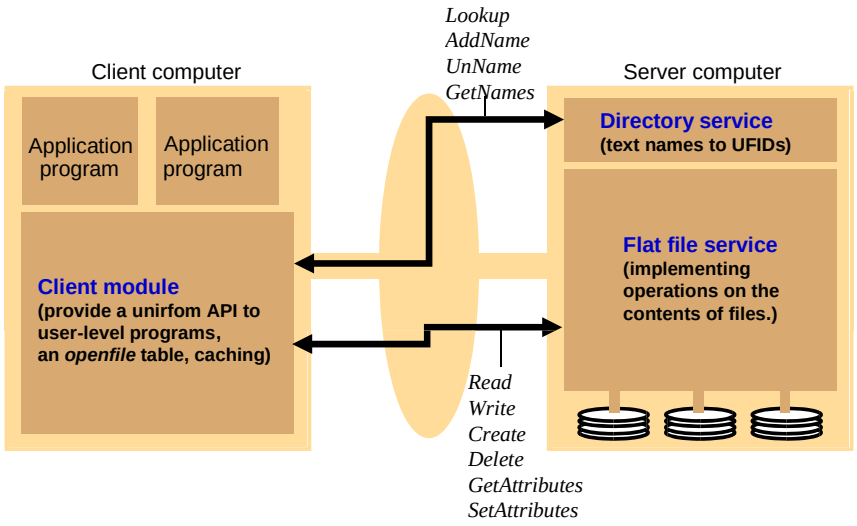
DFS History



- In first generation of distributed systems (1974-95), file systems (e.g. NFS) were the only networked storage systems.
- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.

2/21/2011 13

Model File Service Architecture



14

Flat File Service Interface and Operations

Read(FileId, i, n) -> Data throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at position *i* and returns it in *Data*.
Write(FileId, i, Data) throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at position *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).

- RPC interface to *Client Module*, not interface to user-level programs
- UNIX File System
  - Has *open* and *close* operations
  - read and *write* operations start at the current position; *seek* to reposition.

Are UNIX file operations Idempotent? Repeatable? Stateless?

2/21/2011 15

How about access control?

- Access control in Unix file system
  - User's access rights are checked against the access mode requested in the *open* call
  - The esu r lting access g rihts a e ea ed u r r t in ntil the file s c osedi l
  - No further check will be required when subsequent operations on the same file requested
- What have to be done in a distributed file system?
- Access control in a distributed file system
  - Access rights checks have to be performed at the server whenever a file name is converted to UFID; why?
  - A user identity has to be submitted with every client request; why?

16

Directory Service Interface & Operations

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, File) 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*.

- Mapping from text file names to UFIDs, each directory is stored as a conventional file with a UFID, thus, directory service is a client of the *Flat file service*

2/21/2011 17

Example

Show how each of following UNIX file operations would be executed at the *Client Module*using the operations of the Model File Service, both *Flat file service* and *directory file service*.

fd = open('/usr/class/CS551/UDP.c", READ) n

= read(fd, buf, BUFSIZE) fdnew = creat("/foo",

READ)

18

File Group

A collection of files that can be located on any server or moved between servers while maintaining the same names.

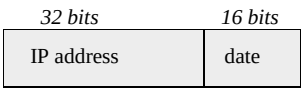
- Similar to a UNIX *filesystem*
- A UFID includes a file group id
  - *Enabling the client module to dispatch requests to the server holding the relevant file group.*
- File groups have identifiers which are

unique throughout the system (and hence for an open system, they must be globally unique).

- *Used to refer to file groups and files*

To construct a globally unique ID we use some unique attribute of the machine on which it is created, e.g. IP number, even though the file group may move subsequently (so how to locate the group?)

Example File Group ID:



2/21/2011

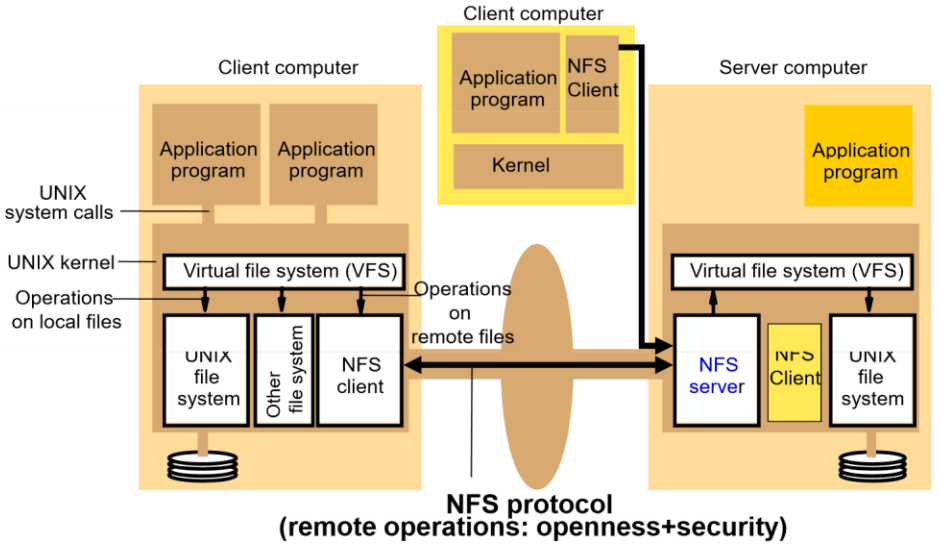
19

Case Study: Sun NFS

- An industry standard for file sharing on local networks since the 1980s
- An open standard with clear and simple interfaces
- Closely follows the abstract file service model defined above
- Supports many of the design requirements already mentioned:
  - transparency
  - heterogeneity
  - efficiency
  - fault tolerance
- Limited achievement of:
  - concurrency
  - replication
  - consistency
  - security

20

NFS Architecture



2/21/2011

21

Virtual File System (VFS)

- VFS distinguishes local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers used by UNIX and other file systems.
- *File handle* in NFS
  - Opaque to clients, used by servers to distinguish an individual file

Filesystem identifier	i-node number	i-node generation

- One VFS structure for each mounted file system, and one *vnode*per each open file
  - V-node contains an indicator to show if a file is local or remote; why?
- VFS is implemented in the UNIX kernel

How transparency provided?

22

NFS Server Interface & Operations (simplified)

- *read(fh, offset, count) ->attr, data*
- *write(fh, offset, count, data) ->attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) ->attr*
- *setattr(fh, attr) ->attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) > entries*
- *symlink(newdirfh, newname, string) -> status*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) ->fsstats*

Filesystem identifier	i-node number	i-node generation

Model flat file service
Read(FileId, i, n) -> Data
Write(FileId, i, Data)
Create() ->FileId
Delete(FileId)
GetAttributes(FileId) ->Attr
SetAttributes(FileId, Attr)

Model directory service
Lookup(Dir, Name) ->FileId
AddName(Dir, Name, File)
UnName(Dir, Name)
GetNames(Dir, Pattern) ->NameSeq

2/21/2011 25

NFS Performance

- Early measurements (1987) established that:
  - *write()* operations are responsible for only 5% of server calls in typical UNIX environments; but more *getattr()* calls, what implied?
  - *lookup()* accounts for 50% of operations due to step-by-step pathname resolution necessitated by the naming and mounting semantics.
- More recent measurements (1993) show high performance:
  1 x 450 MHz Pentium III: > 5000 server ops/sec, < 4 millisec. average latency
  24 x 450 MHz IBM RS64: > 29,000 server ops/sec, < 4 millisec. average latency
  seewww.spec.org for more recent measurements
- Provides a good solution for many environments including:
  - large networks of UNIX and PC clients
  - multiple web server installations sharing a single file store

NFS Summary 1

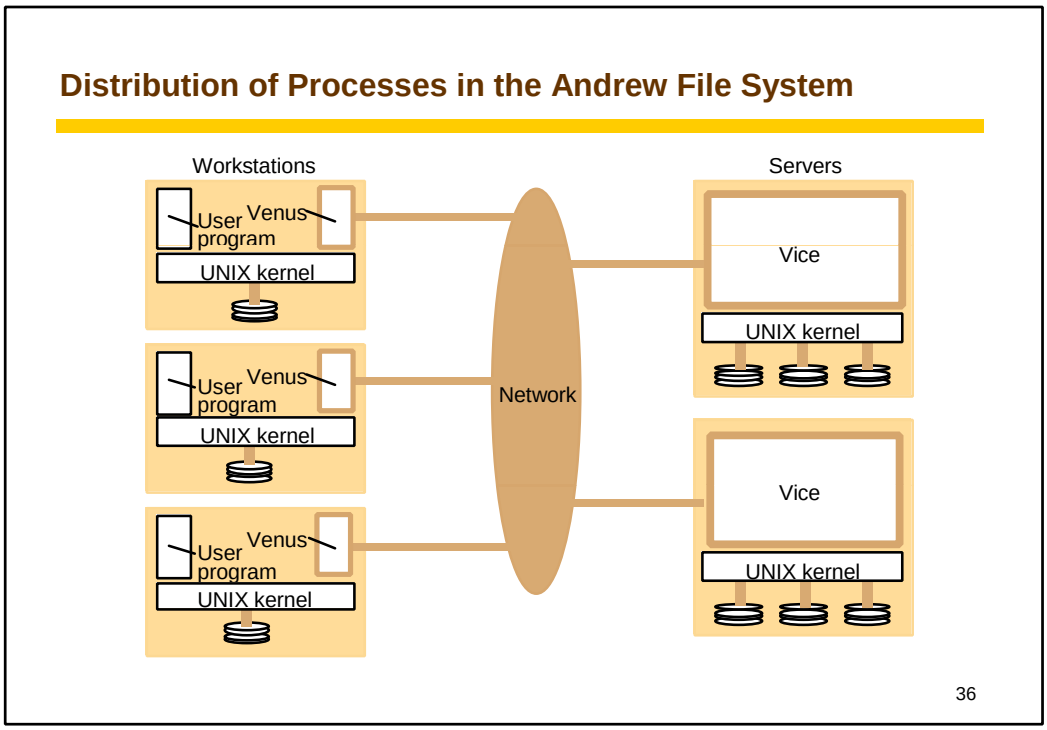
- An excellent example of a simple, robust, high-performance distributed service.
- Achievement of transparencies (*See section 1.4.7*):

  Access: *Excellent*; the API is the UNIX system call interface for both local and remote files.

  Location: *Not guaranteed but normally achieved*; naming of filesystems is controlled by client mount operations, but transparency can be ensured by an appropriate system configuration.

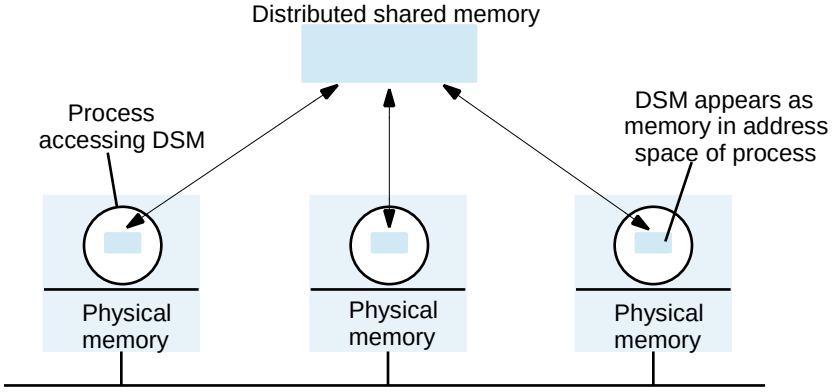
  Concurrency: *Limited but adequate for most purposes*; when read-write files are shared concurrently between clients, consistency is not perfect.

  Replication: *Limited to read-only file systems*; for writable files, the SUN Network Information Service (NIS) runs over NFS and is used to replicate essential system files, see Chapter 14.



DISTRIBUTED SHARED MEMORY

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another. It is as though the processes access a single shared memory, but in fact the physical memory is distributed (see Figure 18.1). The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client-server systems, where clients normally view server-held resources as abstract data and access them by request (for reasons of modularity and protection). However, servers can provide DSM that is shared between clients.



Message passing versus DSM

As a communication mechanism, DSM is comparable with message passing rather than with request-reply-based communication, since its application to parallel processing, in particular, entails the use of asynchronous communication.

Implementation approaches to DSM

Distributed shared memory is implemented using one or a combination of specialized hardware, conventional paged virtual memory or middleware:

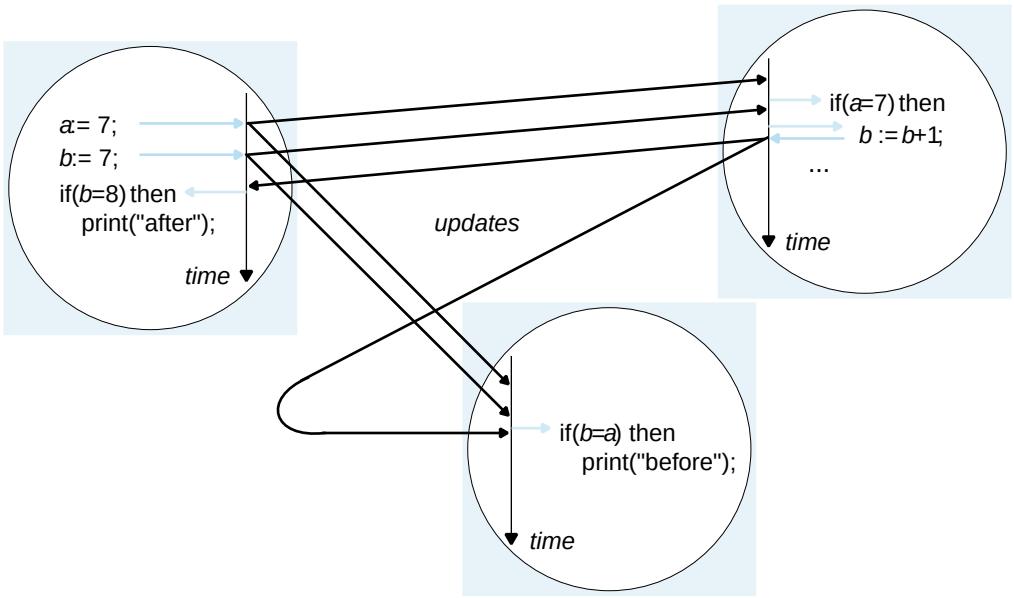
Hardware:Shared-memory multiprocessor architectures based on a NUMA architecture (for example, Dash [Lenoski*et al*. 1992] and PLUS [Bisiani and Ravishankar 1990]) rely on specialized hardware to provide the processors with a consistent view of shared memory.
Paged virtual memory:implement DSM as a region of virtual memory occupying the same address range in the address space of every participating process.

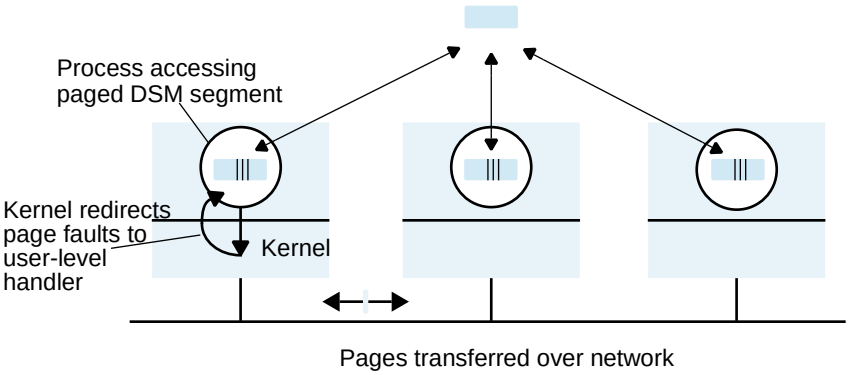
Middleware:. In this type of implementation, sharing is implemented by communication between instances of the user-level support layer in clients and servers. Processes make calls to this layer when they access data items in DSM.

Synchronization model

Many applications apply constraints concerning the values stored in shared memory. This is as true of applications based on DSM as it is of applications written for sharedmemory multiprocessors (or indeed for any concurrent programs that share data, such as operating system kernels and multi-threaded servers).

DSM using write-update





Munin

The Munin DSM design [Carter *et al*. 1991] attempts to improve the efficiency of DSM by implementing the release consistency model. Furthermore, Munin allows programmers to annotate their data items according to the way in which they are shared, so that optimizations can be made in the update options selected for maintaining consistency. It is implemented upon the V kernel [Cheriton and Zwaenepoel 1985], which was one of the first kernels to allow user-level threads to handle page faults and manipulate page tables.

The following points apply to Munin's implementation of release consistency:

- Munin sends update or invalidation information as soon as a lock is released.
- The programmer can make annotations that associate a lock with particular data items. In this case, the DSM runtime can propagate relevant updates in the same message that transfers the lock to a waiting process – ensuring that the lock's recipient has copies of the data it needs before it accesses them.

Name Services

The role of Names and Name services

Resources are accessed using identifier or reference
- An identifier can be stored in variables and retrieved from tables quickly
- Ide t enifi r incudes o cal r n be t a s o r n f rmed to a n add ess o ar f r n object
  - E.g. NFS file handle, Corba remote object reference
- A name, bounded to attributes of the named object, is human-readable value (usually a string) that can be resolved to an identifier or address
  - Internet domain name, file pathname, process number
  - E.g ./etc/passwd, http://www.cdk3.net/

For many purposes, names are preferable to identifiers
- because t e h b d gin in o t e a ed esouce to a p ys ca ocat of h n m r r h i l li n is de e ed f rr and can be changed; address can relocated
- because they are more meaningful to users

Resource names are resolved by name services
- to give identifiers and other useful attributes; such as address, remote object reference

Requirements for Name Spaces

A name space is the collection of all valid names recognized by a particular service
- Allow simple but meaningful names to be used

Potentially infinite number of names

Structured (often hierarchical)
- to allow similar subnames without clashes
- to group related names

Allow re-structuring of name trees
- for some types of change, old programs should continue to work

Management of trust

Names and Resources

Currently, different name systems are used for each type of resource: resource name identifies

filepathname file within a given file system process process id process on a given computer port port number IP port on a given computer

Uniform Resource Identifiers (URI) offer a general solution for any type of resource. There are two main classes:

URL Uniform Resource Locator

- typed by the protocol field (http, ftp, nfs, etc.)
- part of the name is service-specific
- resources cannot be moved between domains

URN Uniform Resource Name

- requires a universal resource name lookup service - a DNS-like system for all resources: given URN and provide URL; resource movable

DNS - The Internet Domain Name System

A distributed naming database vs. original Internet naming database

Name structure reflects administrative structure of the Internet

– Organizations and departments can manage their own naming data

Rapidly resolves domain names to IP addresses

– exploits caching heavily

– typical query time ~100 milliseconds

Scales to millions of computers

– Hierarchical partitioned database

– Replication of naming data

– caching

Resilient to failure of a server

– Distribution, replication, and caching

Basic DNS algorithm (host/domain name -> IP number)

- Look for the name in the local cache
- Try a superior DNS server, which responds:
  – another recommended DNS server – the IP address (which may not be entirely up to date due to caching)

1/14/2008

ems: Concepts and Design Edn. 4
esley Publishers 2005

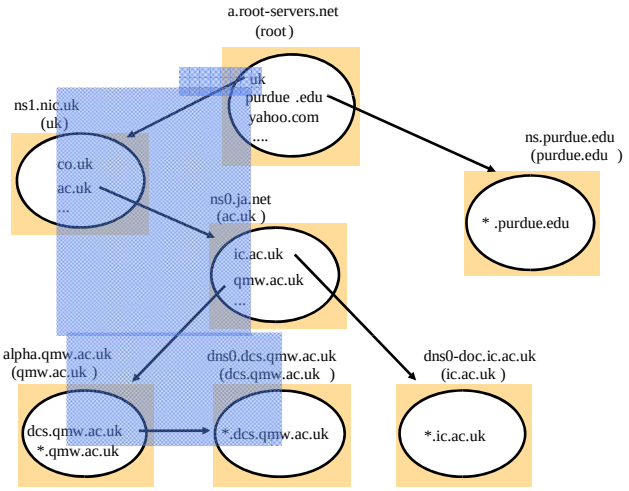
10

DNS Name Servers

Each name server holds:
part of the naming database
(mainly data of local domain)
+ doma in names an d a ddress
of other name servers

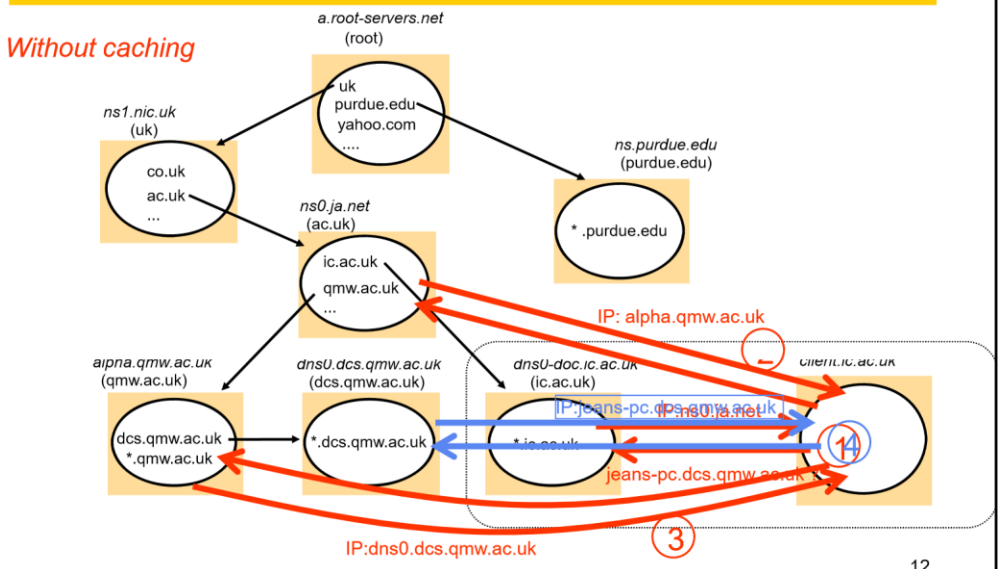
Note : Name server names are in italics, and the corresponding domains are in parentheses. Arrows denote name server entries

authoritative path to lookup:
jeans-pc.dcs.qmw.ac.uk



Adapted from Distributed Systems: Concepts and Design Edn. 4
© Addison-Wesley Publishers 2005

DNS in Typical Operation



12

DNS Server Functions and Configuration

Main function is to resolve domain names for computers, i.e. to get their IP addresses

- caches the results of previous searches until they pass their 'time to live'

Other functions:

- get mail host for a domain
- reverse resolution - get domain name from IP address
- Host information - type of hardware and OS; not suggested due to security (may be used to gain unauthorized access to computers)
- Well-known services - a list of well-known services offered by a host (telnet, FTP, etc)
- Other attributes can be included (optional)

DNS Issues

Name tables change infrequently, but when they do, caching can result in the delivery of stale data.

- Clients are responsible for detecting this and recovering

Its design makes changes to the structure of the name space difficult. For example:

- merging previously separate domain trees under a new root
- moving subtrees to a different part of the structure (e.g. if Scotland became a separate country, its domains should all be moved to a new country-level domain.

See Section 9.4 on GNS, a research system that solves the above issues.

14

Directory and Discovery Services

Directory service:-'yellow pages' for the resources in a network

- Attributed-based name services: retrieves the set of names that satisfy a given description (attribute); name is an attribute too!
- e.g. X.500, LDAP, MS Active Directory Services
  - (DNS holds some descriptive data <name, attribute>, but:
    - the data is very incomplete
    - DNS isn't organised to search it)

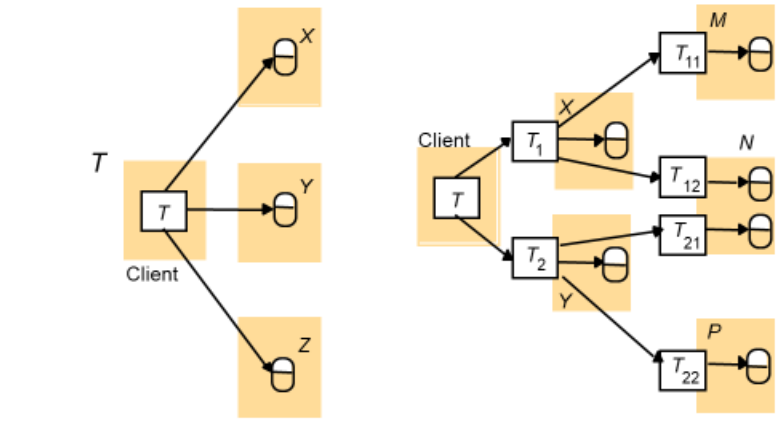
Discovery service:-a directory service that also:

- Is a database of services with lookup based on service description or type, location and other criteria
- is automatically updated as the network configuration changes
- meets the needs of clients in spontaneous networks (Section 2.2.3)
- discovers services required by a client (who may be mobile) within the current scope, for example, to find the most suitable printing service for image files after arriving at a hotel.
- Examples of discovery services: Jini discovery service, the 'service location protocol', the 'simple service discovery protocol' (part of UPnP), the 'secure discovery service'.

UNIT V

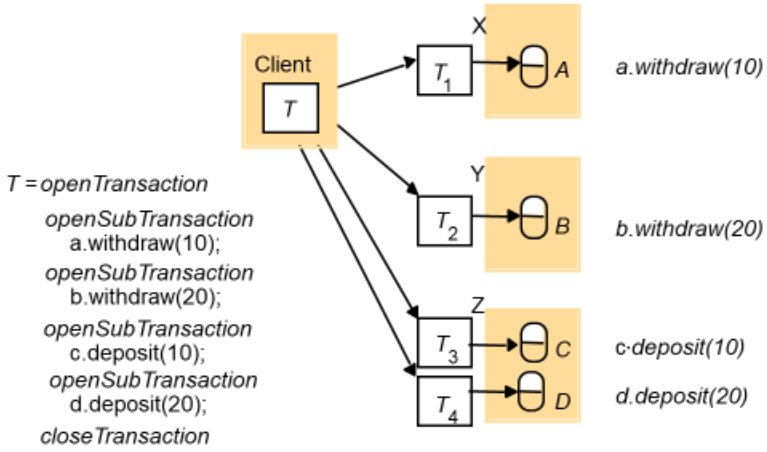
Distributed transactions

Figure 17.1
Distributed transactions



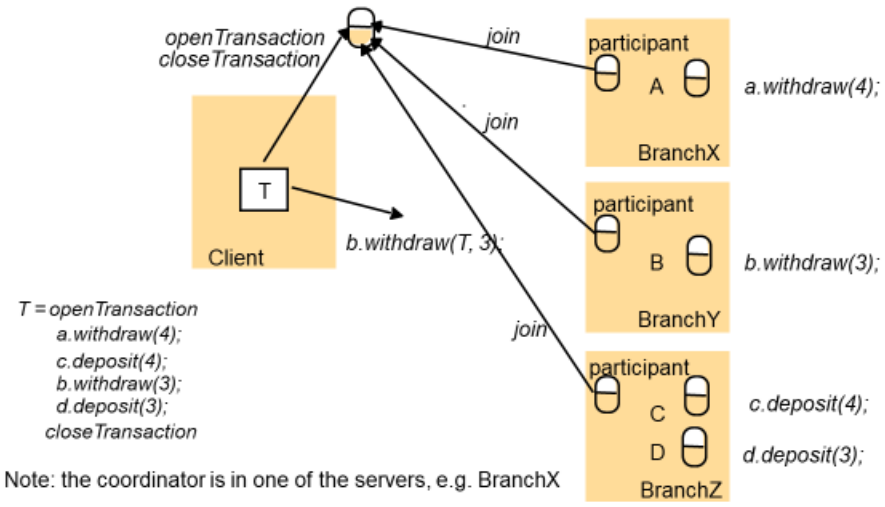
Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.2
Nested banking transaction



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.3
A distributed banking transaction



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.4
Operations for two-phase commit protocol

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.

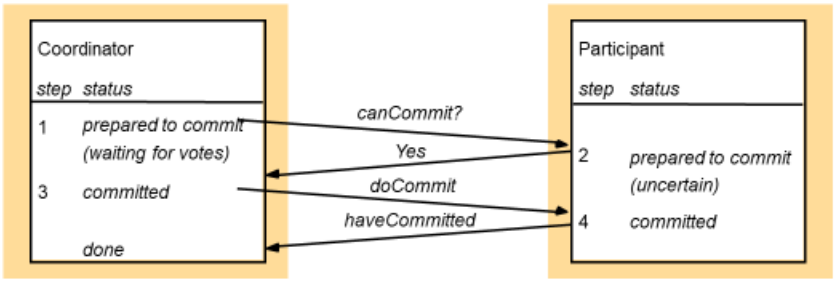
doAbort(trans)
Call from coordinator to participant to tell participant to abort its part of a transaction.

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.

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.6
Communication in two-phase commit protocol



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.5
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.

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

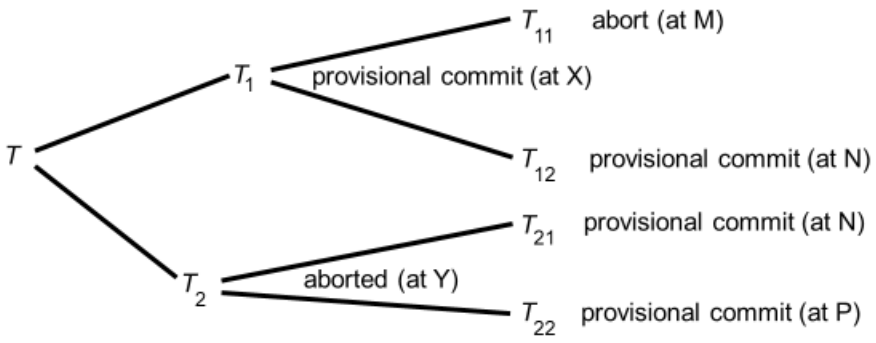
Figure 17.7
Operations in coordinator for nested transactions

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*.

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.8
Transaction *T* decides whether to commit



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.9
Information held by coordinators of nested transactions

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_{21}'s parent has aborted

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.10
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*.

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

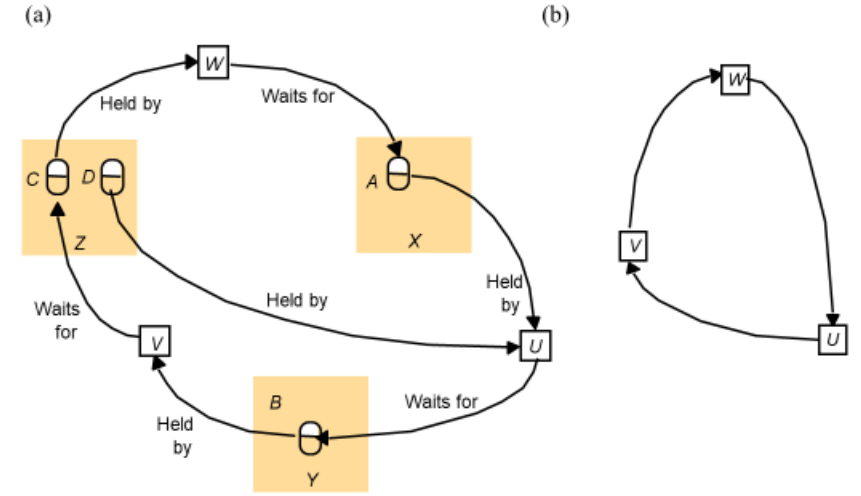
Figure 17.11
canCommit? for flat two-phase commit protoco

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*.

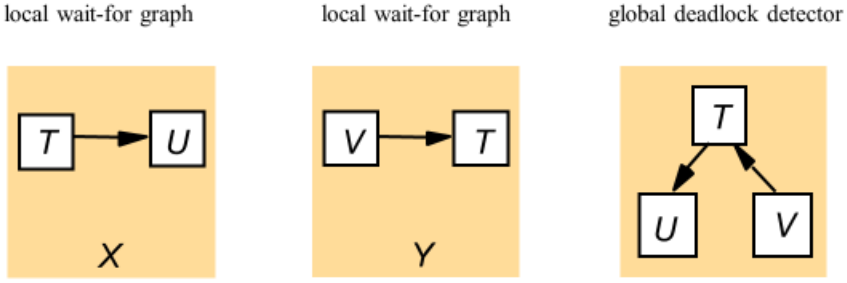
Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.13
Distributed deadlock



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.14
Local and global wait-for graphs



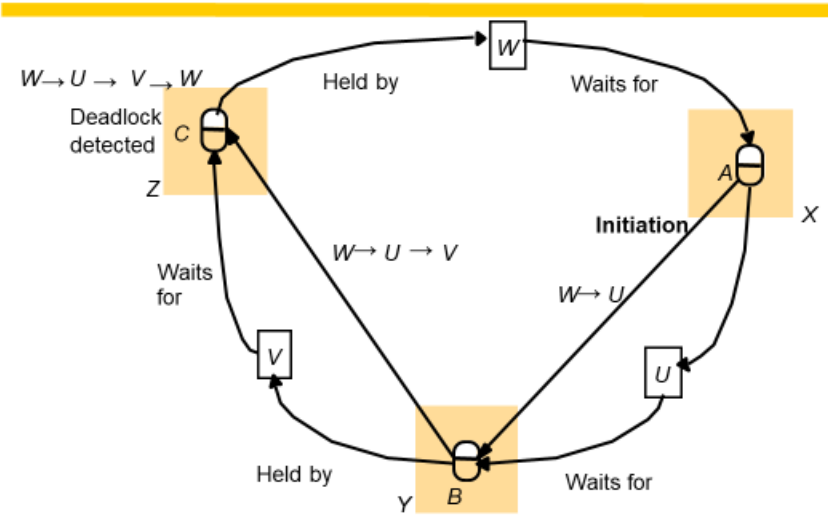
Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.12
Interleavings of transactions *U*, *V* and *W*

U		*V*		*W*	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B* at *Y*		
a.deposit(20)	lock *A* at *X*				
				c.deposit(30)	lock *C* at *Z*
b.withdraw(30)	wait at *Y*				
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.15
Probes transmitted to detect deadlock



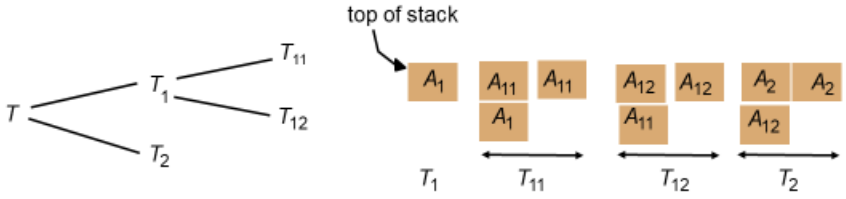
Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.22
Recovery of the two-phase commit protocol

Role	*Status*	*Action of recovery manager*
Coordinator	*prepared*	No decision had been reached before the server failed. It sends *abortTransaction* to all the servers in the participant list and adds the transaction status *aborted* in its recovery file. Same action for state *aborted*. If there is no participant list, the participants will eventually timeout and abort the transaction.
Coordinator	*committed*	A decision to commit had been reached before the server failed. It sends a *doCommit* to all the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4 (Fig 13.5).
Participant	*committed*	The participant sends a *haveCommitted* message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	*uncertain*	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a *getDecision* to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	*prepared*	The participant has not yet voted and can abort the transaction.
Coordinator	*done*	No action is required.

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Figure 17.23
Nested transactions



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Transactions and Concurrency

A Banking Example

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

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

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

- first we consider the synchronization of client operations 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:
    - *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)

Enhancing Client Cooperation by Synchronizing Server Operations

- Clients share resources via a server
  - e.g. some clients update server objects and others access them
  - servers with multiple threads require atomic objects
- but in some applications, clients depend on one another to progress
  - 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
- 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
- Wait-notify vs. rejection-retry
  - it would not be a good idea for a waiting client to poll the server to see whether a resource is yet available
  - it would also be unfair (later clients might get earlier turns)

6

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 then it carries out a recovery procedure to get its objects' state
  - faulty servers are made to crash so that they do not produce arbitrary failures
- There may be an arbitrary delay before a message arrives. A message may be lost, duplicated or corrupted.
  - recipient can detect corrupt messages (by checksum)
  - forged messages and undetected corrupt messages are disasters

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.
- 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 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);

- 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 (if it fails or is aborted) it has no effect at all. This all-or-nothing effect has two further aspects of its own:
   - failure atomicity:
     - the effects are atomic even when the server crashes; –

       durability:
     - after a transaction has completed successfully, all its effects are saved in permanent storage.
2. 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:

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

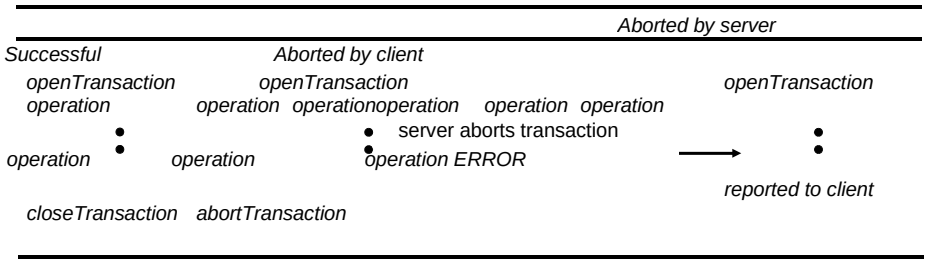
closeTransaction(trans) -> (commit, abort); ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);
aborts the transaction.

Operations in the *Coordinator* Interface (cont.)

- the client uses *OpenTransaction*to get *TID* from the coordinator
  - the client passes the *TID* with each request in the transaction
  - e.g. as an extra argument
    - *deposit(trans, amount)*
    - or transparently if provided as middleware; The CORBA transaction service does uses 'context' to do this.
- **To *commit***
  - the client uses *closeTransaction*and the coordinator ensures that the recoverable objects are saved in permanent storage
- **To *abort*** The client asks either to commit or abort
  - the client uses *abortTransaction*and the recoverable objects and the coordinator must ensure that all temporary effects are invisible to other transactions

Transaction Life Histories



- 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 server know when the client crashes?
  - 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.

Concurrency Control

- We will illustrate the '*lost update*' and the '*inconsistent retrievals*' problems which can occur in the absence of appropriate concurrency control
  - 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 occur when a retieval transaction observes values that are involved in an ongoing updating transaction
- we show how *serial equivalent executions* of concurrent transactions can avoid these problems
- we assume that the operations *deposit*, *withdraw*, *getBalance*and *setBalance*are *synchronized* operations - that is, their effect on the account balance is *atomic*.

The Lost Update Problem

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

The Inconsistent Retrievals Problem

Transaction *V*:		Transaction *W*:
a.withdraw(100)	*// initial $200*	*aBranch.branchTotal()*
b.deposit(100)	*// initial $200*	
b.deposit(100)	$300	

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

Serial Equivalence Interleaving

- 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 same effect means
  - the read operations return the same values
  - the instance variables of the objects have the same values at the end

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

4/4/2011 17

Read and *Write* Operation - Conflict Rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* 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 of *write* 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)*

20

Serial Equivalence Defined by 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*: *x* = *read*(*i*); *write*(*i*, 10); *write*(*j*, 20);
  - *U*: *y* = *read*(*j*); *write*(j, 30); z = *read* (*i*);

  T and U access i and j

  Which of their operations conflict?

  –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 criteriafor designing schemes for concurrency control

A Non-serially Equivalent Interleaving

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

What is its serially equivalent Interleaving ?
Any possible concurrency?

- Each transaction's access to *i* and *j* is serialised 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

Recoverability from Aborts

- Servers must record the effects of all committed transactions and none of the effects of aborted transactions
  - 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
- (*getBalance*is a read operation; *setBalance*a write operation)

A *Dirty Read* When Transaction *T* Aborts

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100	•*U* reads *A*'s balance (which was set by *T*) and then commits	
a.setBalance(balance + 10)	$110	*balance = a.getBalance()*	$110
		a.setBalance(balance + 20)	$130
T subsequently aborts.		*commit transaction*	
abort transaction			

These executions are serially equivalent

What is the problem? *U* has performed a *dirty read*

Recoverability of Transactions

- If a transaction (like *U*) commits after seeing the effects of a transaction that subsequently aborted, it is not recoverable

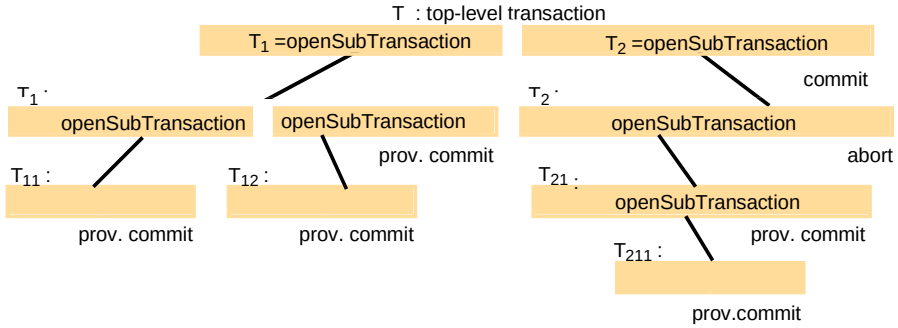
  For recoverability:
  A commit is delayed until after the commitment of any other transaction whose state has been observed

- e.g. *U* waits until *T* commits or aborts
- if *T* aborts then *U* must also abort

So what is the potential problem?

4/4/2011 25

Nested Transactions



- 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

4/4/2011 29

Introduction to Concurrency Control

- Transactions must be scheduled so that their effect on shared objects is serially equivalent

for serial equivalence,
(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 serialising 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 (also useful to avoid cascading aborts)

- *Two-phase locking* - has a 'growing' and a 'shrinking' phase

Transactions T and U with Exclusive Locks (two-phase)

Transaction T		Transaction U	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(bal*1.1)*		*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)		*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*

when *T* is about to use *B*, it is locked for *T*

when *U* is about to use *B*, it is still locked for *T* and *U* waits

U can now continue

when *T* commits, it unlocks *B*

the use of the lock on *B* effectively serialises access to *B*

- initially the balances of A, B and C unlocked

4/4/2011 33

Read-Write Conflict Rules

- concurrency control protocols are designed to deal with *conflicts* between operations in different transactions on the same object
- we describe the protocols in terms of *read* and *write* operations, which we assume are atomic
- Read operations of different transactions do not conflict
  - therefore 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

What decides whether a pair of operations conflict?

Lock Implementation

- The granting of locks will be implemented by a separate object in the server that we call the *lock manager*
  - The client program has no access to operations for locking/unlocking, but the coordinator
- the lock manager holds a set of locks, for example in a hash table.
- each lock is an instance of the class *Lock* 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

Deadlock with Write Locks

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

Then what must we do with T and U?

Is this serially equivalent? Can both T and U be allowed to commit?

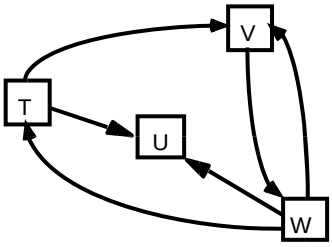
The *deposit* and *withdraw* methods are atomic. Although in practice they read as well as write, they acquire write locks.

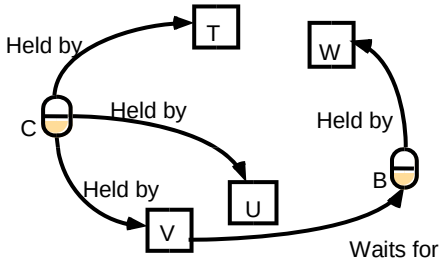
When locks are used, each of T and U acquires a lock on one account and then gets blocked when it tries to access the account the other one has locked. We have a 'deadlock'. The lock manager must be designed to deal with deadlocks.

What can a lock manager do about deadlocks?

Deadlock Prevention vs. Deadlock Detection

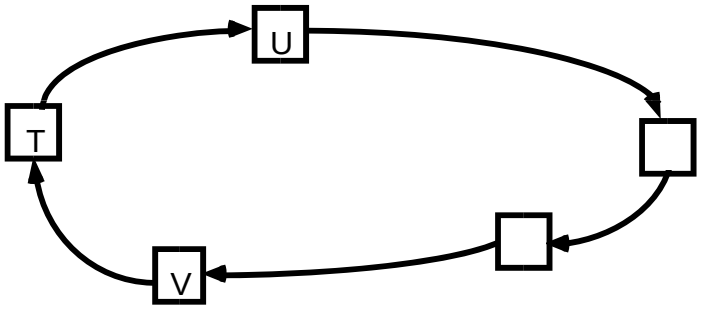
- Is deadlock prevention realistic? How about lock-all-at-once?
- How about deadlock detection with abortion?





4/4/2011 47

A Cycle in a Wait-for Graph



- 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

4/4/2011 45

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 (working phase finished; 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 write	*write*	2. T_v must not read objects written by T_i	backward
		3. T_i must not write objects written by T_v and T_v must not write objects written by T_i	

•*Validation can be* simplified by *omitting rule 3* (if no overlapping of validate and update phases – make sure only one in the validation and update)

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

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.		

66

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.