CS5352, Distributed Computing, Summer II, 2020

Lecture 1: Introduction

(07-06-2020)

Lecture Outline

```
1. History of computer systems
2. Characteristics of distributed computing
3. Advantages of distributed systems
4. Distributed systems vs. distributed computing
5. General issues in distributed computing
6. Goals of distributed computing
7. Distributed computing vs. cloud computing
```

1. History of computer systems

   (1) Dedicated systems

   (2) Batching systems

   (3) Time-sharing systems

   (4) Distributed systems

A dedicated system is a computer system capable of performing 1 specific task an embedded system is a computer system within a larger system that performs a specific task.

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group.

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

2. Characteristics of distributed computing

   (1) Computer networks

   (2) Multiprocessor systems with shared memory

   (3) The essential features of distributed systems

      a. Multiple processing elements
      b. Intercommunication facilities
      c. Transparency to the users
      d. Fault tolerance
         i. Processing elements fail independently
         ii. Single failure will not bring down the whole systems/computing

3. Advantages of distributed systems

   (1) Convenience

   (2) Low ratio of cost/performance

   (3) Load sharing

A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.

Three aspects: concurrency of components, lack of a global clock, and independent failure of components

In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other via message passing.

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

(4) Resource sharing

(5) Availability

(6) Reliability

(7) Scalability

4. Distributed systems vs. distributed computing

(1) Distributed systems are special cases of general distributed computing. In fact distributed systems can be regarded as one of the most important applications of distributed computing.

(2) A distributed system is characterized by the existence of a distributed operating system that provide transparent distributed services to members of the distributed system.

5. General issues in distributed computing

(1) The enormous complexity

a. **Interference**. The behavior of the combination of individually well-understood components become difficult to understand.

b. Propagation of effects is affected by distributedness.

(2) Distributed control and coordination

(3) Intercommunication. Synchronization, time and clocks. Implementation of remote procedure calls.

(4) Distributed agreement and fault tolerance

(5) Managements of replicated data

(6) Naming and security. Authentication.

6. Goals of distributed computing: achieve the highest level of the following types of of transparencies

(1) **Access transparency** enables local and remote files and other objects to be accessed using identical operations.

(2) **Location transparency** enables objects to be accessed without knowing their location.

(3) **Concurrency transparency** enables several users or application programs to execute concurrently on shared data without interference with each other.

(4) **Replication transparency** enables multiple instances of files and other data to be used to increase reliability and performance without knowledge of the replicas by users or application programs.

(5) **Failure transparency** enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

(6) **Migration transparency** allows the movements of objects within a system without affecting the operation of users or application programs.

(7) **Performance transparency** allows the system to be reconfigured to improve performance as loads vary.

(8) **Scaling transparency** allows the system and applications to expand in scale without change to the system structure or the application programs.

7. Distributed computing vs. cloud computing

(1) **Cloud computing**: a new technique based on decades of research and advances of many fields of computing technology: distributed computing, utility computing, grid computing, networking techniques, web and software services, cyberinfrastructure, virtualization

(2) Essentials of cloud computing

a. Service oriented architecture: on-demand services

b. Component based system engineering

c. Virtualiztion: resources are virtualized - reduced information technology overhead

Cloud computing is a style of computing where massively scalable and flexible IT-related capabilities are delivered as a service to the users using Internet technologies, services may include: infrastructure, platform, applications, and storage space. The users pay for these services, resources they actually use. They do not need to build infrastructure of their own.

Distributed Computing can be defined as the use of a distributed system to solve a single large problem by breaking it down into several tasks where each task is computed in the individual computers of the distributed system.

Lecture 2
Elements of Operating Systems
(07-07-2020)

Lecture Outline

```
1. Sequential Processes
2. Concurrent Processes
3. Specification of Concurrency
4. Relationships among the Processes
5. Deadlocks
6. The Critical Section Problem
7. Semaphores
```

1. Sequential Processes

(1) *A sequential process is a sequential program in execution*

a. A process is an active entity, while a program is a passive entity

b. The execution of a process is sequential, i.e. following the control flow of the program

(2) State of a process

- **Running**. The instructions of the process are being executed
- **Blocked**. The process is waiting for some event to occur (such as I/O completion, communication)
- **Ready**. The process is waiting to be assigned to a processor
- **Deadlocked**. The process is waiting for some event that will never occur

2. Concurrent Processes

(0) *Concurrent* vs *parallel*: the latter implies *at the same time physically*, while the former implies *at the same time logically*.

(1) *Sequential programs might be executed concurrently*

(2) **Example 1.** Consider the following program:

$a := x + y;$
$b := z + 1;$
$c := a - b;$
$w := c + 1;$

The statement $a := x + y$ and $b := z + 1$ can be executed concurrently
This example illustrates that, within a single program:
• some statements can be executed concurrently
• there exists *precedence constraints* among the various statements

(3) **Precedence graph.** A precedence graph for a sequential program is a directed acyclic graph whose nodes correspond to individual statements in the program. An edge from node S_i to node S_j means that statement S_j can be executed only after statement S_i has completed execution.

Example 2. Consider the precedence graph below,

- S_2 and S_3 can be executed (concurrently) after S_1 completes

Refer to figure 5

- S_4 can be executed after S_2 completes
- S_5 and S_6 can be executed (concurrently) after S_4 completes
- S_7 can be executed after S_5, S_6, and S_3 complete

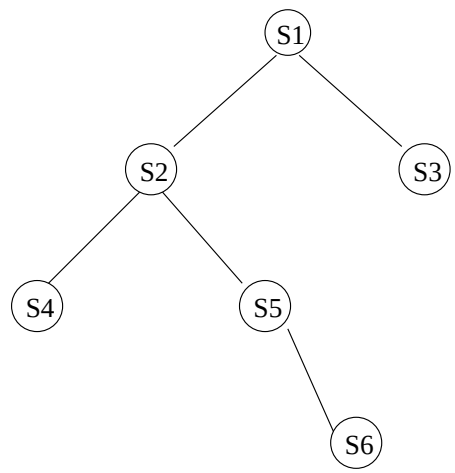


Figure 1: Process Precedence Graph

3. Specification of Concurrency

(1) The fork and join constructs

- The **fork** L instruction produces two concurrent executions in a program. One execution starts at the statement labeled L, the other is the continuation of the execution at the statement following the **fork** statement.

$$\begin{array}{rl} & S_1; \\ & \textbf{fork } L; \\ & \ldots\ldots \\ L: & S_3; \end{array}$$

- The **join** instruction combines several concurrent executions into one. All but the last computation that execute the join instruction will terminate, and the last computation will continue.

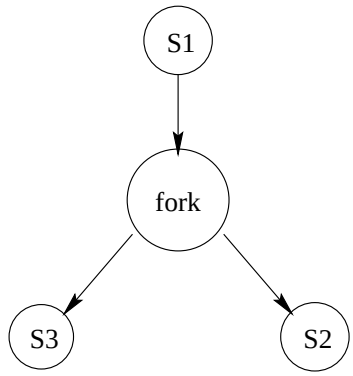


Figure 2: Effect of the Fork Statement

$count := count - 1;$
if $count \neq 0$ **then** quit;

$count := 2;$
fork L_1;
$\cdots\cdots$
S_1;
goto L_2;
$L_1 : S_2;$
$L_2 :$ **join** $count$;

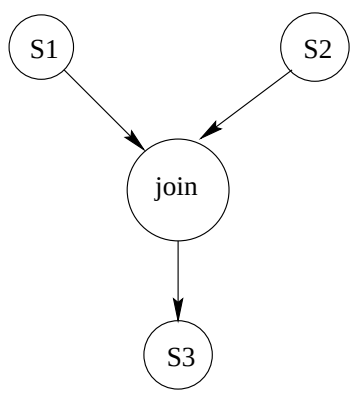


Figure 3: Effect of the join Statement

(2) The concurrent statement

parbegin $S_1; S_2; \cdots; S_n$ **parend**;

Example 3. The program in Example 1 can be expressed by

parbegin

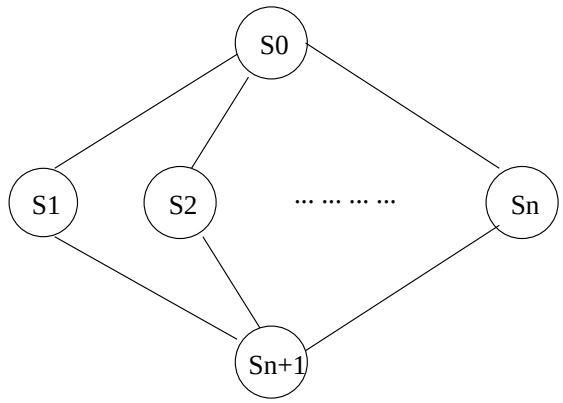


Figure 4: Illustration of the parbegin-parend Statement

$a := x + y;$
$b := z + 1;$
parend;
$c := a - b;$
$w := c + 1;$

(3) Comparison: the **fork** and **join** constructs are more powerful than the concurrent statements. The following precedence graph can be represented by the **fork** and **join** constructs, but can not be represented by concurrent statements.

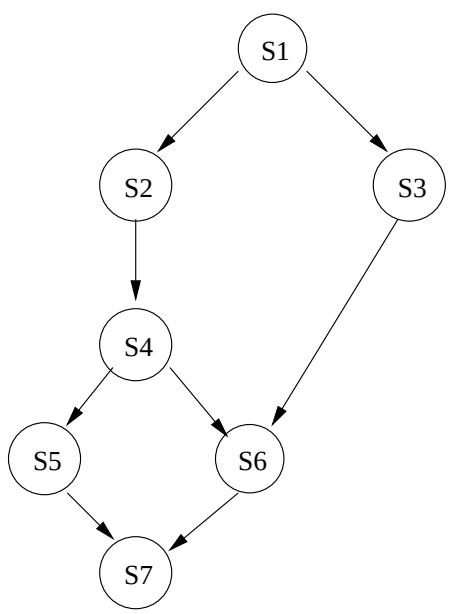


Figure 5: Comparison of fork-join and parbegin-parend Statement

4. Relationships among the Processes

(1) **Process graph.** A process graph is a directed rooted tree, whose nodes correspond to processes. An edge from node P_i to P_j means that process P_i created process P_j. P_i is called the *parent* of P_j, and P_j is called the *child* of P_i.

(2) Operations on a process

a. Process creation. When a process is created, several possible implementations are:

i. **Execution.** Concurrent versus sequential
(a) The parent continue to execute concurrently with its child
(b) The parent waits until all of its children have terminated

ii. **Sharing.** All versus partial
(a) The parent and the children share all variables in common
(b) The children share only a subset of their parent's variables

Option i.(a) and ii.(a) are adopted by the **fork** and **join** constructs. Option i.(b) and ii.(a) are adopted by the concurrent statements
Option ii.(b) is adopted by the UNIX system.

b. Process termination.
A process may terminate when it finishes its execution.
It can also be killed by its parent.

$id :=$ **fork** L;
kill id;

5. Deadlocks

(1) **Definition.** A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.

(2) **Example 4.** Two processes P_1 and P_2. Process P_1 is holding the resource printer, waiting for the resource disk drive. Process P_2 is holding the resource disk drive, waiting for the printer. P_1 and P_2 are in a deadlock state.

(3) Necessary conditions of deadlocks
A deadlock can arise if and only if the following four conditions hold simultaneously.

- **Mutual Exclusion.** At least one resource is being held in a non-sharable mode.
- **Hold and Wait.** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are being held by other processes.
- **No Preemption.** Resource cannot be preempted; i.e. a resource can only be released voluntarily by the process holding it.

- **Circular Wait.** There must exist a set $\{P_0, P_1, \cdots, P_n\}$ of waiting processes, such that P_0 is waiting for a resource held by P_1, P_1 waiting for a resource held by P_2, and P_n waiting for a resource held by P_0.

6. The Critical Section Problem

(1) Problem description. A computation consists of n cooperating processes $\{P_1, \cdots, P_n\}$. Each process has a common section of code, called *critical section*, in which a process might read or write common variables, tables, or files. The execution of critical section must be *mutually exclusive*, in the sense that at any time at most one process is allowed to execute in the critical section. The critical section problem is to design a protocol which the processes may use to cooperate.
The code for each process can be divided into four sections: *entry section*, *critical section*, *exit section*, and *remainder section*. A solution to the *cs* problem must satisfy the following three requirements:

a. **Mutual Exclusion.** If process P_i is executing its critical section, then no other process can be executing in its *cs*.

b. **Progress.** If no process is executing in its critical section and there exists some process that wishes to enter its *cs*, then only those processes that are not executing in their remainder section can participate in the decision as to who will enter the critical section, and and decision can not be postponed indefinitely.

c. **Bounded Waiting.** There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

(2) An Example
Example 5. The readers/writers problem.
A data object is to be shared among several concurrent processes. Some of the processes (called readers) may want only read the content of the shared object, while others (called writers) may want to update the object.
A reader process and a writer process or two writer processes are not allowed to access to the shared object simultaneously, otherwise inconsistency might occur. However, two reader processes are allowed to read the object simultaneously.
Assume that initially $b = 3$.

Process P_1	Process P_2
$a := b + 1;$	$b := 2;$

Example 6. Dinning philosophers problem

7. Semaphores

(1) A semaphore S is an integer variable that can only be accessed through two *atomic* operations: P and V (P stands for dutch passeren (to pass), V stands for dutch vrygeven (to release)).

$P(S)$:**while** $S \leq 0$ **do** skip;
$\quad S := S - 1;$

$V(S)$:$S := S + 1;$

(2) Usage

a. Solving critical section problem
$mutex := 1;$
repeat
$P(mutex);$
critical section;

$V(mutex);$
remainder section;
until false

b. Synchronization

Lecture 3

Introduction to Distributed Computing

(07-07,07-08-2020. Chapter 3, 1, 2)

Lecture Outline

1. Computer networks (Chapter 3)
2. Architectural models of distributed systems (Chapter 2)
3. Major challenges (Chapter 1)
4. System architectures
5. Fundamental models

1. Computer networks (Chapter 3)

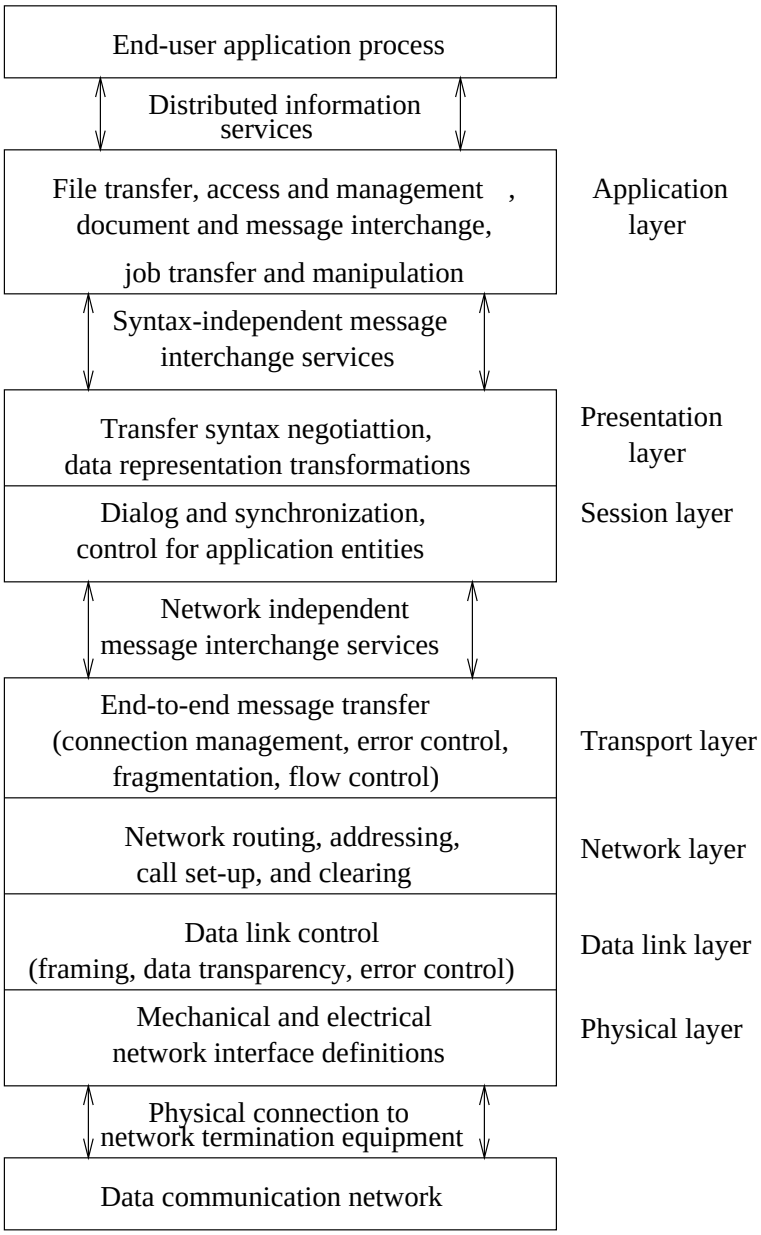


Figure 6: OSI reference model

(1) Communication protocols

- A communication protocol is a set of rules used for a designated session of communication.
- High-level protocols – used for communications occurring at high-level exchange of information.

比如 Content-Type: text/html 设有它无法显示html.

IP, TCP, Ethernet

– Low-level protocols – used for communications occurring at low physical level exchange of information.

(1) real connection like IP

(2) Connection oriented

三次握手

(2) Connection oriented vs connectionless communications

TCP

a. Connection oriented communications (telephone systems, ftp). Advantages: reliable. 比如200m不能接受delay

不用握手 UDP (有latency) 迟迟

b. Connectionless communications (post mails, emails in BITNET). Advantages: flexible, efficient. 发电邮无法确保对方是否在线

c. Datagrams (packets) and packet (data) switching vs circuit switching

checksum

(3) The Seven-layer ISO OSI Reference Model (Fig.3.5, p.95)

TCP/UDP all are datagram

a. Principles

(a) Layer creation criteria: level of abstraction;
(b) Each layer: perform a well-defined function;
(c) Layer function: easy to be standardized;
(d) Layer boundaries: minimize information flow between layers;
(e) Number of layers: reasonable.

b. The Physical Layer

Services: transmitting bits through physical channels
Design issues: ensure that a bit 1 sent will be received as 1

c. The Data Link Layer

Gen. Services: providing error-free transmission services to its upper layer
Design issues: disassemble messages and reassemble messages
detecting loss of messages, retransmitting, detecting duplicates
flooding control, piggybacking

拆 合

d. The Network Layer

Gen. Services: controlling the network operations
Design issues: routing: static, semi-dynamic, dynamic
congestion control
accounting
internetworking

e. The Transport Layer

Gen. Services: Accept and pass data
Design issues: multiple and multiplex transport
determine service types to session layer and network

(error-free point-to-point FIFO channel, no guaranteed order)
the first end-to-end layer
connection management

f. The Session Layer SSL

Gen. Services: establish sessions between users
data transport
enhanced services: remote login, file transfer
token management
synchronization

g. The Presentation Layer

Gen. Services: ensure syntax and semantics of the information
coding and encoding
data compression, cryptography

h. The Application Layer

Gen. Services: general and special user-support facilities
Examples: terminal mapping
file system mapping
electronic mails

(4) Local area networks (LANs, Sect.3.5)

a. LANs and WANs

b. Characteristics of LANs

(a) A diameter of no more than a few kilometers;
(b) A total data rate of at least several Mbps;
(c) Complete ownership by a single organization.

c. Special factors in LAN design

(a) High bandwidth; designers are not restricted to use public lines;
(b) LAN cables are more reliable than those in WANs;
(c) Criteria for protocols are different due to the availability of high bandwidth channels;
(d) Network architecture is simpler in LANs than in WANs.

d. IEEE Standard 802: Fig. 3.22, p.129

(a) Three IEEE standards for LANs: CSMA/CD, token bus, token ring, called IEEE 802.

American National Standard Institute

International Standard Organization

(b) IEEE 802 was also adopted by ANSI and ISO (ISO 8802) as international standard

(c) 802.1 gives introduction to the set of standards, and defines interface primitives. 802.2 describes the upper part of the data link layer using the LLC (Logical Link Control). 802.3 MAC Sub Layer Protocol – a bus utilizing CSMA/CD protocol.

Mac sublayer
logical link control } Data link layer.

(5) IEEE Standard 802.3 and Ethernet
802.3 is for a 1-persistent CSMA/CD LAN, the general ideas are:

a. When a station has a frame to transmit, it first checks the channel (carrier sense)

b. If the channel is busy, the station waits until it is idle

c. If the channel is detected idle, the station transmits its frame

d. If two or more stations transmit frames simultaneously, a collision occurs

e. The station stops transmission immediately once it detects the collision. It waits for a random amount of time before doing retransmission

(6) IEEE Standard 802.4: Token Bus

a. 802.4 is a collision free protocol.

b. The protocol. Basic ideas:

(a) Physically, the token bus is a linear or tree-shaped cable attached with stations.

(b) The whole network is logically organized as a ring. A station knows its left and right neighbors

(c) A special control frame is circulating around the logical ring. Only the token holder is allowed to transmit frames. When a station receives the token, it may transmit its frames for a certain amount of time, and then it must pass the token to one of its neighbor stations. If a station does not have any data to send when it receives the token, it passes the token immediately.

(d) Frames are divided into four priority classes: 0, 2, 4, and 6, with 0 being the lowest, and 6 the highest.

c. Logical ring maintenance 维护开销

(7) IEEE Standard 802.5: Token Ring

a. Why rings?

(a) Ring structures can be used for both LANs and WANs. Ring technique are well developed.

(b) IEEE 802.5 was based on the IBM's ring networks. IEEE 802.5 is called *token ring*.

b. Ring structure.

(a) A ring consists of a collection of ring interfaces connected by point-to-point lines.

why we need this

(b) Each bit arrives at an interface is copied into a 1-bit buffer and recopied out onto the ring again. The bit in the buffer can be inspected and possibly modified before being copied out. This copying introduces 1-bit delay at each interface.

c. **Token.** Like in token bus, a special bit pattern, called **token** is circulating around the ring.

(a) When a station wants to transmit its frames, it first must grab the token, and remove it from the ring. It then starts its transmission.

(b) After transmission, the station must return the token to the ring.

(c) Token rings are collision free

d. Two modes of the interface: listen and transmit. To be able to switch from listen to transmit mode in one bit time, the interface needs to buffer frames. why

e. Ring Maintenance: Token bus handles ring maintenance in a distributed fashion, in the sense no central controller is present. Token ring handles the ring maintenance in a centralized fashion. Each token ring has a **monitor station** which oversees the ring. If the monitor station goes down, the contention algorithm will elect another monitor station. Each station has the ability to become the monitor station.

Garbled frames are damaged/ corrupted frames.

Orphaned frames are frames left on the network no computer will receive it.

(a) Monitor's responsibilities:

- Seeing if the token is lost;
- Taking actions if the ring breaks;
- Cleaning up the ring when garbled frames appear and watching out for orphan frames. ?

how

(b) To check for if the token is lost, the monitor has a timer that is set to sum of the longest token holding time of each station. If the timer goes off, the monitor drains the ring and issues a new token.

(c) If a frame is damaged, the monitor detects it by its invalid format or checksum. The monitor then drains the ring and issues a new token when the ring is clean.

(d) The monitor detects orphan frame by using the monitor bit in the access control byte.

how and when

(e) Monitor introduce extra delay if necessary.

Token

Start Delimiter	Access Control	End Delimiter

infrared : 430F-300 G

microwave : 300G → 300M

(8) IEEE 802.11 (WiFi) wireless LAN

a. LAN with wireless: Fig.3.24, p.136

b. Physical level:

(a) Radio signals of 2.4 GHz and 5 GHz band

(b) Infrared signalling

what is it exactly

(c) Guardian intervals (GI): are used to ensure distinct transmissions do not interfere with each other, namely to introduce immunity to propagation delays, echoes, and reflections. Longer GI: more echo tolerance, lower channel efficiency (data rate).

c. IEEE 802.11b: first standard (1999); 2.4 GHz band; data rate up to 11Mbps

d. IEEE 802.11g: 2.4 GHz band; different signaling technique; data rate up to 54Mbps; not an official IEEE standard.

e. IEEE 802.11a: 5 GHz band; data rate up to 54Mbps; range shorter than 802.11g.

why

????

f. IEEE 802.11n (2009): 5 GHz band; data rate up to 54Mbps; range shorter than 802.11g.

(a) Radio signals of 2.4 GHz and 5 GHz band (dual band)

(b) MIMO (multiple-input multiple-output): a technology that uses multiple antennas to send/receive more information coherently than possible using a single antenna.

SDMA空分多址

(c) Spatial streams: spatial division multiplexing, spatially multiplexes multiple independent data streams, transferred simultaneously within one spectral channel of bandwidth. Up to 4 streams.

(d) Bandwidth of channels: 20MHz or 40MHz

(e) Data rate: 6.5Mbps to 600Mbps, depending on channel bandwidth allocated (20 or 40 MHz), GI (guardian interval, 800ns vs 400ns), and encoding (modulation, BPSK, QPSK, 16-QAM, 64-QAM) technique.

h. IEEE 802.11ac (2011 -): on going effort

(a) Radio signals of 5 GHz band

(b) Channels up to 160MHz (20, 40, 80, 160 MHz) 这是什么意思.....?

(c) More MIMO spatial streams (up to 8) 这是什么意思.....?

(d) Higher density modulation: upto 256 QAM 越高代表什么?

(e) Data rate at least 1Gbps for multi-station WLAN and at least 500Mbps for a single link 什么是multi-station WLAN，然后什么是 a single link?

(9) TCP/IP Protocol Suite: Sect.3.4

a. TCP/IP layers: Fig.3.12, p.106, Fig.3.13, p.107

The specification has multi-station throughput of at least 1 gigabit per second (1 Gbit/s) and single-link throughput of at least 500 megabits per second (0.5 Gbit/s)

b. IPv4 vs. IPv6

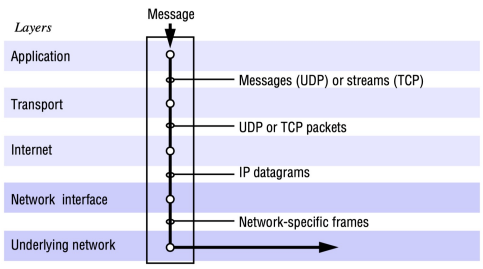
(a) IPv4 number: 32 bit integer, divided into 4 decimal integers.

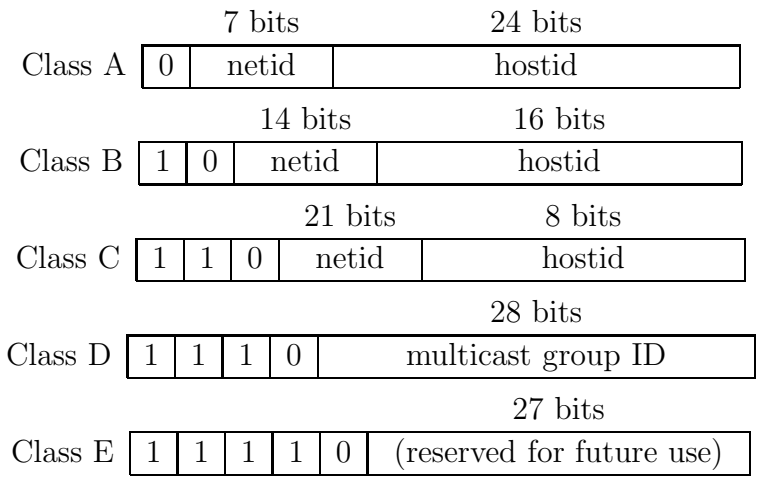
(b) IPv6 number: 128 bit integer, divided into 32 hexadecimal numbers. Example:

5f1b:df00:ce3e:e200:0020:0800:2078:e3e3

c. IPv4 addresses (Fig.3.15, p.109, Fig.3.16, p.110)

(a) An IPv4 Internet address occupies 32 bits, and encode both the network ID and host ID. The host ID is relative to the network ID. Every host on a TCP/IP Internet must have a unique 32 bit address.





(b) Address ranges:

Class	Range
A	**0**.0.0.0 to **127**.255.255.255
B	**128**.0.0.0 to **191**.255.255.255
C	**192**.0.0.0 to **223**.255.255.255
D	**224**.0.0.0 to **239**.255.255.255
E	**240**.0.0.0 to **247**.255.255.255

d. IP, UDP, and TCP

e. IP routing 底层通信datagram的方式。stream都是用ip routing

2. Architectural models of distributed systems (Chapter 2)

(1) The whole system can be viewed as consisting of four layers (Fig.7).

a. Applications and services

(a) Server: a process that accept service requests from other processes

(b) Service: a service can be provided by more than one server process

(c) Example:

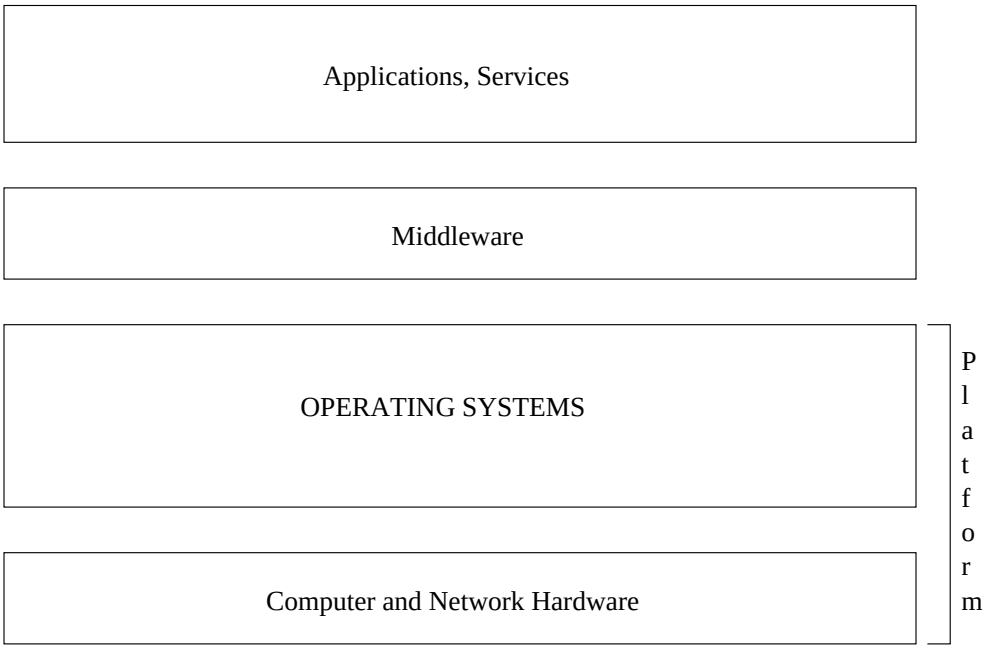


Figure 7: Software and hardware service layers in distributed systems (Fig.2.7,p.52)

- Web service
- NTP (network time protocol) service

b. Middleware

(a) Middleware is a software layer that provides programming abstraction and masking of heterogeneity of the underlying networks, hardware components, and programming languages. Example middleware:

- RPC facility
- Java RMI (remote method invocation)
- CORBA (Common Object Request Broker Architecture)

(b) Forms of components of middleware: they can be processes (services) or objects in a number of computers in a DC

(c) Main functionalities of middleware:

最流行 比如 DB的connector

- providing high level building blocks that facilitate construction of software components. High level communication facility (that hide low level comm. details) in particular.
- providing services for use by application programs. Examples: naming services, transaction processing services, event reporting services. NTP

(d) Limitations of middleware

- Example: Email transfer. Typically TCP based. TCP is reliable. In case TCP cannot establish connections, or email transfers of large emails are

interrupted, a middleware component can be added to do retransmission (therefore providing more transparently reliable email services to users). However, if communications cannot go through for a prolonged period of time, the added middleware still cannot abstract the nature of comm. from users.

- The nature of distributedness of DC requires communications for many essential functions provided by a DC.
- The problems that can occur in communications cannot be easily controlled or cannot be controlled at all by communication systems.
- Thus, in many cases middleware will only be able provide incomplete abstraction of services to users.

c. Operating systems

(a) Distributed operating systems (DOS) running on individual computers in a DS provide essential services to the middleware. Examples:

- File locking and data replication
- Process scheduling and management

(b) DOSs also interact with hardware components to hide hardware dependent features from middleware.

(c) Some DOSs services are part of middleware too

d. Computer and network hardware Arm is technology used for cell phone

(a) They are vendor specific. Different computers may have different characteristics. Different network hardware may have different comm. capabilities.

(b) Examples: 大部分用intel的cpu，有些用amd

- FDDI (a high speed LAN) is faster than a traditional Ethernet. FDDI is ring based while Ethernet is bus based.
- The architecture of a Digital UNIX Alpha workstation is 64-bit. A Pentium III is 32-bit.

(2) The ISO OSI reference model is not suitable for distributed systems/computing

a. The model is a very general model for computer networks. DC is based on but is on top of computer networks. A DC expects the underlying computer networks to provide all functions (such as connection-oriented comm.) that are supported by these networks. A DC is not concerned with how these functions are implemented.

b. The nature of DC demands higher level support of abstraction and transparency. Distributed applications in a DC utilizes all functions by computers and network hardware to implement *functional* services. The structure and interrelations of these services have little to do with the layered structure of the underlying computer networks.

3. Major challenges (Chapter 1)

(1) Heterogeneity 不均匀性;

a. Questions:

(a) Should different types of computers be allowed in a DC?

(b) Should the users be allowed to know the existence of such machines?

(c) If yes, how to define and provide transparent services?

b. Types of heterogeneity:

(a) Computer hardware heterogeneity: different instruction sets, incompatible data representations (high-low, low-high orders). Can be handled relatively easily, except the last one. **Example**: different UNIX systems. 什么意思

(b) Network heterogeneity: transmission media, signaling techniques, computer-media interfaces, lower-level protocols (eg. data link layer protocols).

(c) Operating system heterogeneity: most troublesome. UNIX OSs can hardly talk with VAX/VMS.

(d) Programming language heterogeneity.

c. Causes of heterogeneity

(a) Different machines were purchased at different times.

(b) Special needs or services.

(c) Personal preference.

d. Possible solutions: heterogeneity is a major problem that prevents efficient utilization of resources and management in DC. A significant portion of middleware is dedicated to handling heterogeneity.

(2) Autonomy and interdependence

a. Two types of systems:

(a) The first type: each system controls its own resources and provides resource services to other systems (through RPC, RMI, and etc.).

(b) The second type: computers depend on networked services, such as filing and authentication (NFS in UNIX). 什么意思?

b. Advantages and problems of the 1st type DC.

(a). Advantages: (i) Relative independence (autonomy). The crash of a single machine will cause a relative minor inconvenience. **Example**: Text editors; 什么意思
(ii) Higher security; (iii) Easier to manage.
不是那么依赖别人。其中一个挂了，也可以活

(b) Problems: (i) Redundancy of functions and resources; (ii) Heterogeneity problem; (iii) Higher cost of resource maintenance.

c. Advantages and problems of the 2nd type DC.

(a) Advantages: (i) Higher ratio of resource sharing; (ii) Lower cost of hardware and software maintenance.

(b) Problems: interdependence. The crash of a single machine (the file server) will cause the unavailability of many resources; **Example**: Text editors again; (ii) Complexity in software. **Example**: Yellow page services in UNIX. Must ensure security; (iii) Much harder to manage. 这跟黄页有什么关系

跟2nd type DC 什么关系

d. Summary:

(a) Autonomy and interdependence are both desired (as justified by the advantages listed above).

(b) Unnecessary interdependence should be avoided (or we lose the very purpose of having DC).

(3) Openness

a. Openness: the degree by which a DC can be extended, re-implemented, and communicated with.

b. Openness can be achieved by using standard interfaces that are based on well-known standards

b. Summary

(a) Open systems are characterized by the fact that the interfaces are published, discussed and preferably well adopted.

(b) The communication interfaces and interfaces for accessing shared resources are essential.

(c) Open systems can be constructed from heterogeneous software and hardware through well designed middleware.

(4) Security

a. Due to distributedness, a DC is subject to security attacks that computer networks face. Examples:

(a) Financial information transfer

(b) Confidential personal information

b. Two major security issues

(a) Correctly identifying a remote user or other agent;

(b) Protecting the contents of sensitive information.

c. Security issues still under research:

(a) Denial of service attack: an attack may disrupt a service by making it not available

- by exploring bugs in server software 这是什么意思?
- by other means (eg. flood the computer hosting the service with requests)

(b) Security of mobile code. Eg: email virus. Difficult to prevent.

(5) Scalability

a. Explosion of computers and web servers in Internet. Computers vs. web servers in the Internet (Fig.1.6, p.20)

Date	Computers	Web servers	Percentage
1973, July	1,776,000	130	0.008
1995, July	6,642,00	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
2003, July	200,000,000	42,298,371	21
2005, July	353,284,187	67,571,581	

Figure 1.6 Growth of the Internet (computers and web servers

b. Major scalability challenges:

(a) Controlling the cost of physical resources:

- Grow the DC at reasonably cost as demand for a resource grows.
- Grow the system linearly in response to demand. Eg.: a file server can server 20 users. A newly added same file server should server another 20 users.

(b) Controlling the performance loss:

- As the size of a resource (such as a database file) grows, the time needed to access the resource will grow.
- A well designed DC should utilize good hierarchical algorithms so that the time taken to access a resource of size n will grow as a function of $\lg n$ (i.e. $O(\lg n)$)

(c) Preventing exhaustion of software resources:

- A resource not well structured may become unavailable after a period of growth.
- Example: (i) IPv4 addresses. (ii) a database designed with a fixed up limit size.

(d) Avoiding performance bottleneck:

- A hot resource in a DC may cause comm. congestion and slow response. Such resources then become performance bottlenecks that hinder performance.
- Example: DNS database (distributed across Internet. Not easily becomes a bottleneck). 怎么理解这个例子?

(6) Failure handling and recovery

Failure model page 67

a. Many types of failures are possible in A DC (Ch.2 has more details on types of failures)

b. Most failures in a DC are partial failures: the failure will not disrupt the whole system.

c. Major failure handling issues:

(a) Failure detection.
- Eg.: using checksum to detect corrupted packets.
- Not all failures can be easily detected. Eg.: non-response of a remote server may be caused by many possible reasons.
- Challenge: detect failures earlier and accurately

(b) Failure masking: failure transparency. 不懂
- Some failures can be masked from users. A file may be replicated. 什么意思
- Effects of some other failures can be made less severe. Eg.: a corrupted message can be retransmitted.

(c) Failure tolerance and redundancy. Services can be made redundant to allow failure tolerance.
- A disk can be duplicated.
- A raid array can be used to tolerate single disk failure.
- Between any two computers in a DC there should be at least two routes.
- For a service, more than one server running on different computers can be created.

(d) Failure recovery:
- In case a failure cannot be tolerated, a DC should undo (roll back) to non-permanent changes at the point of failure and return to its most recent consistent state before failure occurs.
- Check points can be used to log incremental consistent state. 什么意思?

(7) Concurrent control: access to shared resources must be controlled to ensure that such resources are always in consistent states. A issue of distributed coordination:

An

a. Conventional concurrency control techniques s.a. semaphores can be used within a single computers. But they are not adequate in a DC environment.

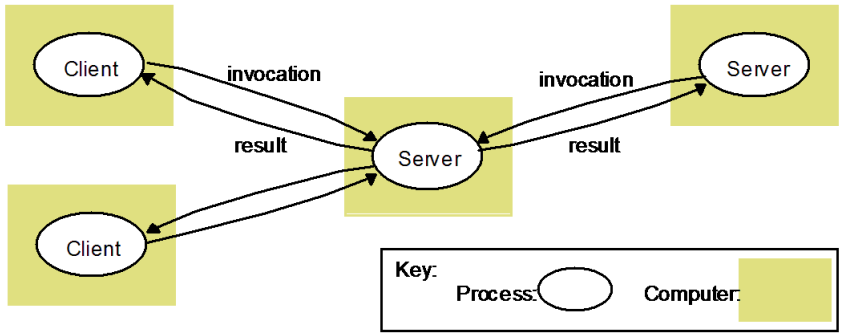


Figure 8: Clients invoke individual servers (Fig.2.3,p.57)

b. Common techniques:

(a) Locks and tokens

(b) Timestamps

(8) Transparency: the ultimate challenge. Cf Lecture 1

4. System architectures

(1) **Client-server**

a. General ideas: control to individual resources is distributed among various processes (servers) across the DC. Trend: reduce the OS kernel, and move part of the kernel up to higher levels (middleware) as system services. Processes are classified as **servers** and **clients**. Each server process is in charge of a particular resource and client processes request the use of the resources through corresponding server processes. (Fig.2.3,p.47)

b. Clients compete the use of resources

c. Problems with the client-server architecture:

i. Control of individual resources is centralized in a single server. If the computer supporting the server fails, that resource becomes inaccessible. Compromise reliability of DC.

ii. Each server is a potential bottleneck, particularly when more clients demanding the same few resources are added. Static. 啥意思?

iii. Multiple implementation of similar controls may be added to improve reliability and availability. Consistency then becomes an issue. 怎么理解

d. Intuitive and most popular

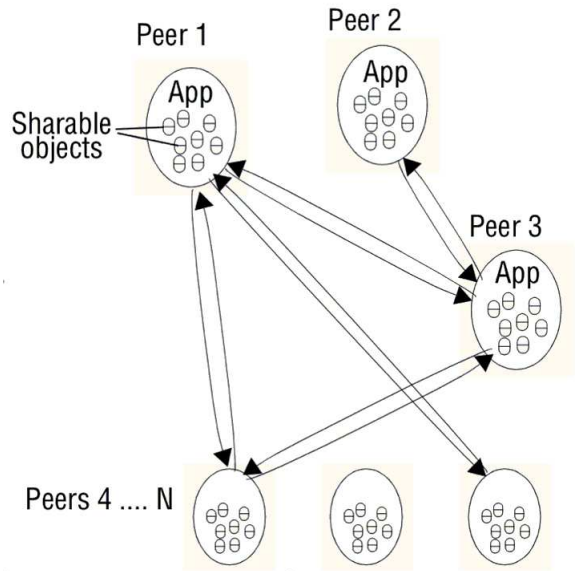


Figure 9: A distributed application based on peer-to-peer architecture (Fig.2.4a, but not shown in textbook)

(2) **Peer-to-peer** (peer processes) (Fig.2.4a, not shown in textbook, but in teaching slides provided by the book)

a. Basic architecture

(a) All processes in this model play similar roles – they are peers to each other. These processes play similar roles and interact with each other. They cooperate as *peers* to perform a distributed activity/computation. There is no clear distinction of clients and servers of these peer processes.

(b) The peer processes are coded in such a way that consistency of application-level resources (such as a replicated file) is maintained and actions on these resources are synchronized if necessary.

b. Examples: *whiteboard* application: a group of programmers on several different computers view and modify an image. A peer process runs on each computer. Modification of the image by an individual programmer is communicated through middleware. Advantages: better response. 可以详细解释一下吗?

(3) Variations

a. Multiple servers (Fig.2.4, p.48)

(a) Also called *process pools*. A service is implemented by multiple server processes (possible running in separate host computers).

(b) The resources (objects) to be managed by the service could be partitioned among the servers. Alternatively, resources could be replicated to improve availability and performance

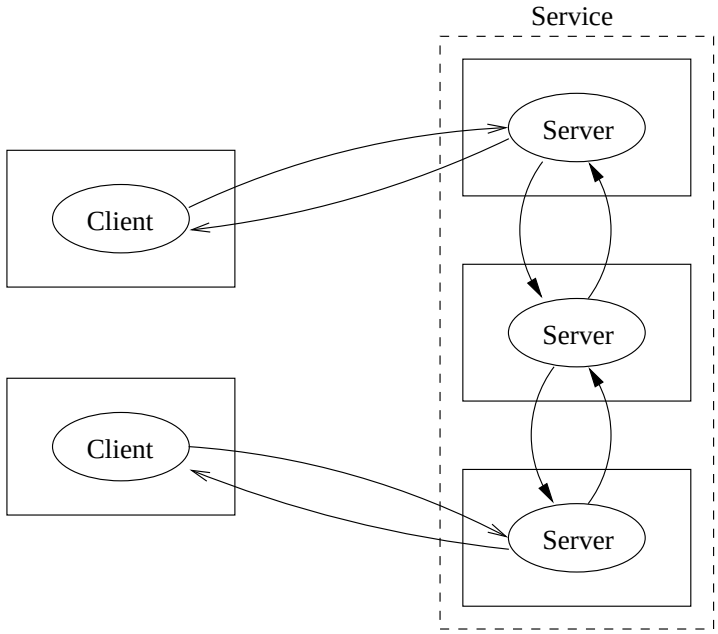


Figure 10: A service provided by multiple servers (Fig.2.4, p.48)

b. **Proxy servers and caches** (Fig.2.5, p.49)

(a) Cache: a store of recently referenced objects that are closer than the objects themselves.

(b) Web proxy servers: a server that keep a shared cache to be referenced by client sites.

(c) Proxy servers: allow access of shared cache through firewall.

(d) Characteristics: cached data is read only to client sites.

c. Mobile code (p.17, Chapter 1, *Heterogeneity and mobile code* for details)

(a) Definition: code that can be sent from one computer to another and run at the destination computer.

(b) Applet: a special mobile code that originally resides on a Web server. A client can select through a Web link the mobile code and download it from server and run it (Fig.2.6,p.50).

(c) Advantage: running of the downloaded code locally by a client is much faster and smoother than running it in realtime on a server.

(d) Potential security problems

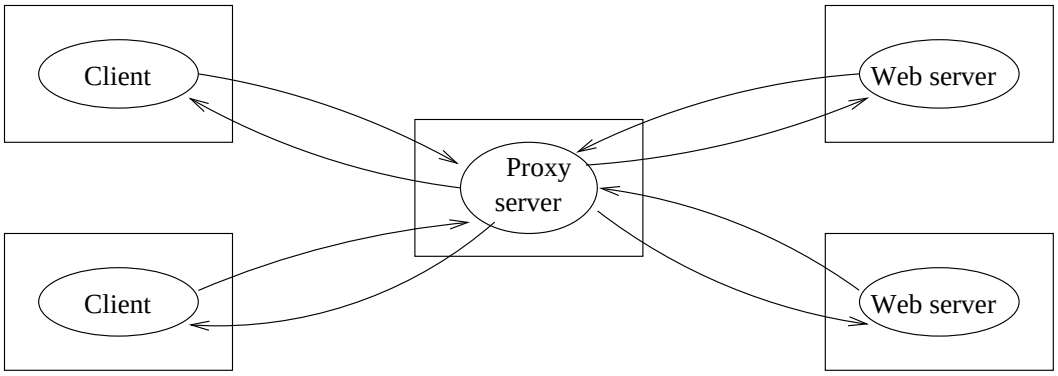
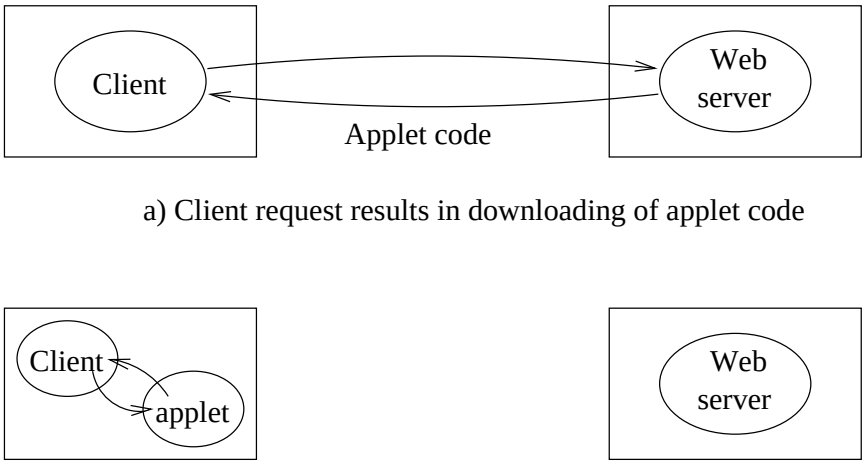


Figure 11: Web proxy server (Fig.2.5, p.49)



a) Client request results in downloading of applet code

b) Client interacts wtih applet

Figure 12: Web applets (Fig.2.6,p.50)

d. Mobile agent (p.50-51) Mobile code 的高级版
   (a) Definition: a mobile agent is a running program that can move (migrate) from one computer to another in a network while carrying task on behalf of some applications.
   (b) Examples
      · Software system maintenance 可以解释一下吗?
      · Information collection (user statistics in a DC) 可以解释一下吗?
   (c) Potential security problems (similar as mobile agent)
e. Network computers
   (a) Motivation: OS and application software require large amount of code. More importantly management and maintenance of the software require sophisticated skills.

(b) Definition: a network computer is one that does not have its own OS and any other application software. All of them have to be downloaded from remote servers. Applications are run locally while resources such as files are managed by remote servers. Users of NC only have to operate a remote controller similar to the one of a typical DVD player.

(c) Cache storage is used to reduce need of data transfer.

f. Thin clients (p.56)

(a) Definition: a thin client is a software layer that supports a window-based user interface on a remote computer (Fig.2.10, p.56)

(b) An X-Window software that runs on a window based PC. The software allows a user to interact with a remote X-window based UNIX computer on a local window based PC. The user can launch X-window based applications locally.

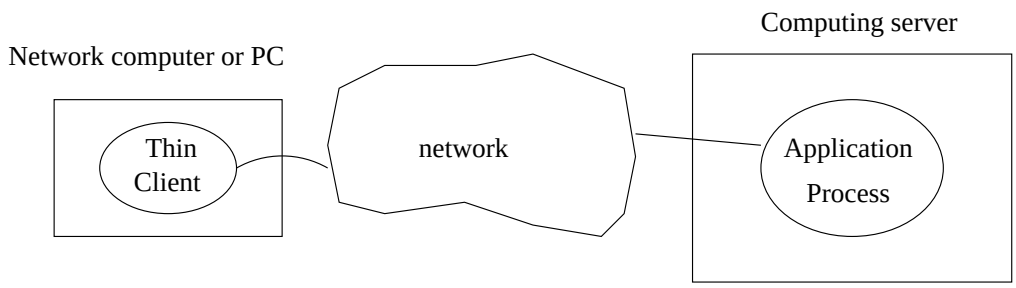


Figure 13: Thin clients and computing servers (Fig.2.10,p.56)

g. Discussions:

(a) The selection of an architecture and the distribution of responsibilities between processes and between computers is an important design issue in DS.

(b) Often a DS may adopt more than architectures to provide maximum flexibility and performance.

(4) Interfaces and objects

a. The set of functions available for invocation in a process (servers or peers) are defined in interface specification.

b. Interfaces in basic client-server architecture: fixed and static. 这是什么?

c. Interfaces supported by object-oriented model: dynamic. The number and types of objects hosted by each process may vary. The location of objects may also migrate (location transparency).

(5) Design requirements of distributed architectures

a. Performance

(a) Distribution of resources in DSs creates challenges of managing concurrent updates of distributed resources.

(b) Performance of a DS is affected by communication capacities, processing capabilities, concurrency, and failures:

- Responsiveness. Hindered by concurrency, communications, and failures. Example: Web clients browsing a Web site.
- Throughput. The amount of computation that can be completed in certain units of time. In addition to other facts listed above, it can also be hindered by processing speed of individual computers.
- Load balancing. To eliminate bottleneck. May involve migration of computation activities. 什么是load balancing

b. Quality of service (QoS)

(a) Main quality properties: performance, reliability, security, and adaptability.

(b) Adaptability: change the system configuration and resource allocation to meet the changing system environment. Eg. changing the location of a database used by a Web client in response to a change of access pattern to that database.

c. Use of caching and replication

(a) Motivations: to improve performance and reliability

(b) Web-cache protocol.

- Web browser and proxy servers cache most referenced contents.
- A browser and proxy server can validate a cached response by checking *freshness* of cached data. 这是什么?
- An expiry time may be associated to each cached piece of data.

(c) Caching and replication may hinder performance if not properly managed or mis-used (cf. section for Web proxy server). ; 这是什么, cf. section for web proxy server

d. Dependability issue

(a) Dependability: correctness, security, and fault tolerance.

(b) Fault tolerance. A DS application should continue to function correctly in the presence of faults in hardware, software, and networks (cf. section for failure handling and recovery).

- Redundancy is the primary technique. Redundancy can be at different levels. 什么意思， Redundancy
- Architecture level: Multiple hardware components (e.g. computers and disk drives)
- Communication protocol level.

(c) Security: cf. Section 2 of this lecture

5. Fundamental models

(1) Fundamental properties of DSs: distributed processes that communicate with one another by sending messages over a network.

a. Models based on these fundamental properties allow closer examination of characteristics of issues.

b. A system model is a paradigm regarding the way a certain aspect of the system interacts and behaves.

c. Questions to be addressed in a system model:

(a) Main entities in the system

(b) Patterns of interactions of the entities

(c) Characteristics that affect their behaviors

(2) Interaction model

a. *Distributed algorithm* and *distributed processes*

(a) A distributed process is a member of a set of processes running in a DS that cooperate/interact to complete a specific task/computation. A distributed process only has partial info about the overall state and progress of the whole computation.

(b) Distributed algorithms determine the steps taken by a distributed process, including transmission of messages and access/update of remote resources.

b. Interaction model defines/captures the methods in which distributed processes interact/cooperate with one another.

c. Two major factors that affect process interactions:

(a) Performance of communication channels

(b) Lack of accurate global time (and state)

重点

d. Two variants of interaction models:

(a) Synchronous DS.

i. A DS is a *synchronous* DS if (p.64):

- The time to execute each step of a process has known lower and upper bound.
- Each message transmitted over a comm. channel is received within a known bounded time.
- Each process has a local clock whose drift rate from real time has known bound.

这是什么

ii. Obtaining realistically accurate values for these three bounds is practically very difficult. DSs designed based on inaccurate bounds are not reliable.

iii. Modeling a distributed algorithm as a synchronous system can provide useful info as how the algorithm may behave under real DSs.

(b) Asynchronous DS

i. A DS is an *asynchronous* DS if (p.64):

- The time to execute each step of a process can vary and may take arbitrarily long. There is no known upper bound.
- Each message transmitted over a comm. channel may experience arbitrarily long delays. There is no known upper bound on a transmission.
- Each process has a local clock whose drift rate from real time can be arbitrary.

ii. The asynchronous model makes no assumptions about upper bounds of execution speeds, message transmission delays, and clock drift amount. Well suitable for Internet. 为什么?

iii. Any distributed algo that is valid under an asynchronous model is also valid under any synchronous model. The reverse does not hold.

iv. Asynchronous model is easier to implement than the synchronous model. A DS whose interaction model is asynchronous can/have to use timeout to limit the effects of arbitrary delays.

Most distributed systems and distributed computing are asynchronous.

e. Event ordering in DSs

(a) Knowing the execution order of two events from two cooperating processes is useful in many cases. For example, a process P_1 has requested updating a file managed by a process P_2. We want to know transactions.

T_1	T_2
$\cdots$	$\cdots$
$x := x + 1;$	$y := x + z$
$\cdots$	$\cdots$
commit;	commit;

The copy of value modified by T_1 is termed *dirty* before T_1 commits. We are interested in knowing whether T_2 performed the operation $y := x + z$ before or after T_1 commits.

(b) Again, due to nature of distributedness, events that occur physically in sequence may not be observed by individual users/processes in the same sequence as they occur.

i. Example (p.52,53) An email message and two of responses. They may arrive in a physical mailbox out of ordering. 什么意思?

ii. If physical clocks can be accurately synchronized, then physical timestamp can be used to order the email messages.

(c) Logical clocks (to be covered in next chapter) are an alternative.

i. Maintaining accurate physical time is difficult and expensive (the more accurate, the more expensive). No algorithm can guarantee 100% accuracy.

ii. Some times accurate physical occurrence time of events is not very important. What are important are their relative ordering: which one *happens before* before which one.

(d) Real-time ordering of events: Fig.2.13, p.66

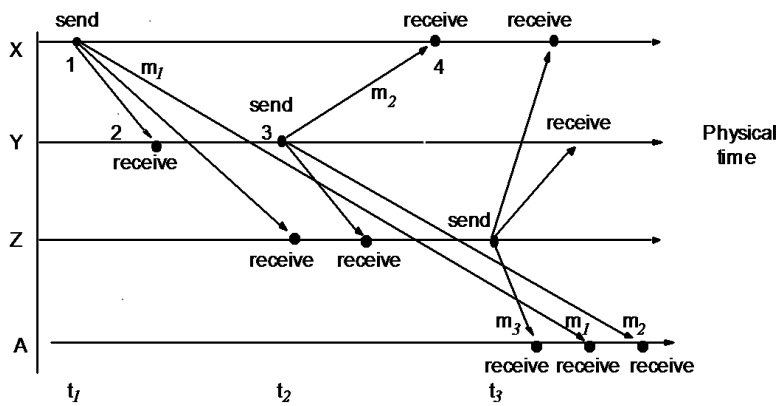


Figure 14: Real-time ordering of events (Fig.2.13, p.66)

(3) Failure model

a. A failure model defines methods by which failures may occur in a DS. It provides guidance as how to detect, solve, and prevent failures

b. *Omission failures*: a process or comm. channel fails to perform actions that are supposed to be done

(a) *Process omission failures*:

i. Main type of process omission failures: *process crash*

ii. Process crash that can be detected certainly by other processes is called *fail-stop*.

(b) *Communication omission failures*:

i. A COF occurs if the comm. channel does not transport a message from a sender's outgoing buffer to the receiver's incoming buffer (Fig.2.9,p.55).

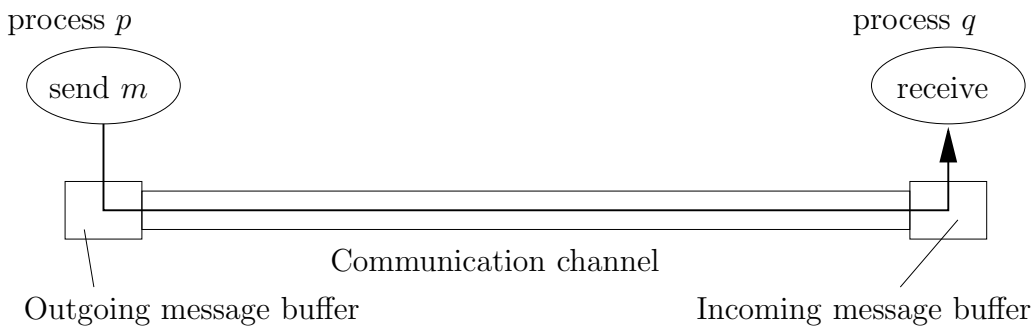


Figure 15: Processes and channels (Fig.2.14, p.69)

ii. *send-omission failure*: loss of message between the sending process and the outgoing message buffer.

iii. *receive-omission failure*: loss of message between the incoming message buffer and the receiving process.

iv. *channel-omission failure*: loss of message between the two buffers.

c. Arbitrary failures: the worst possible failures. It can be any type and can happen any where at any time.

   (a) Both processes and comm. channels can have arbitrary failures.

   (b) A process may omit intended processing steps or take unwarranted steps any time and any where.

   (c) A comm. channel may corrupt a message or generate a duplicate copy of a message.

d. Summary of omission and arbitrary failures: Fig.2.15 (p.70)

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

Figure 2.15 Omission and arbitrary failures

e. *Timing failures*. They only apply to synchronous DSs. A timing failure occurs when a time limit set on process execution time, message delivery, or clock drift rate is exceeded. A timing failure may also cause a performance failure. Fig.2.16 (p.71) summarizes timing failures.

Class of Failures	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between steps. 什么意思
Performance	Channel	A message's transmission takes longer than the set timeout period

Figure 2.16 Timing failures

这是在说什么?

Benign failures: omission failures, timing failures, and performance failures.

f. *Masking failures*.

(a) It is possible to provide reliable services even in the presence of failures.

(b) A failure is *masked* by a service if the service can hide it altogether by converting it into a more acceptable type of failures. Examples:

- Checksums can be used to convert an arbitrary failure into an omission failure.
- Omission failures could be masked through use of retransmission (plus timeout). 什么?

g. Reliability of one-to-one comm. A *reliable communication* is defined in terms of *validity* and *integrity*:

(a) Validity: any message (in an outgoing buffer) is eventually delivered to the corresponding incoming buffer.

(b) Integrity: the message received is identical to the one sent and no duplicate messages delivered.

The communication in a DS is termed reliable if validity and integrity can be ensured under *all possible* communication problems.

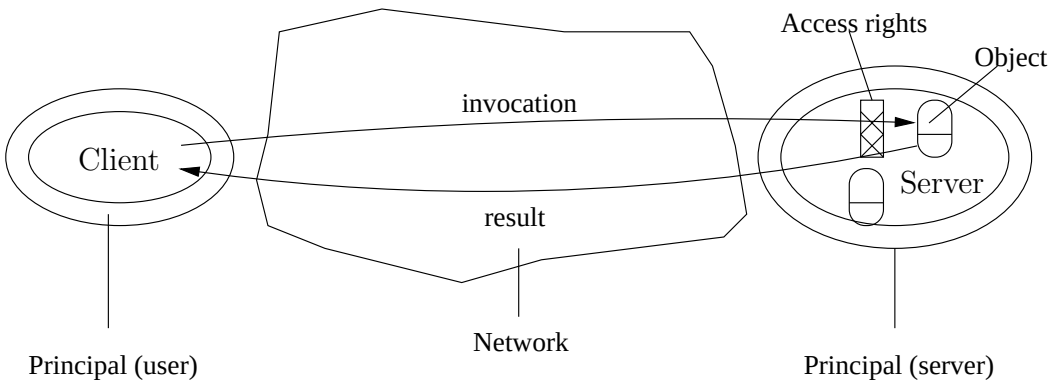


Figure 16: Objects and principals (Fig.2.17, p.72)

(4) Security model 重点

a. Security model: methods to

(a) secure distributed processes and communication channels; and

(b) protect objects accessed by these processes.

b. Protecting objects (Fig.2.17,p.72)

(a) *Access rights*: each object is associated with a list of access rights – who is allowed and what is allowed for each operation (method).

(b) *Principal*: a principal is an *authority* who issues an access request/invocation (from a client) or result (from a server). A principal can be a user or a process (service).

(c) A server is responsible for verifying the identity of the principal behind each invocation and checking that each such invocation has sufficient access rights.

c. Securing processes and their interactions against threats. Interactive processes have to communicate messages with one another.

(a) Traveling messages are subject to attacks.

(b) Processes exported interfaces for supported services, hence make themselves subject to external threats too.

Therefore both messages contents and processes must be secure.

d. Security threats

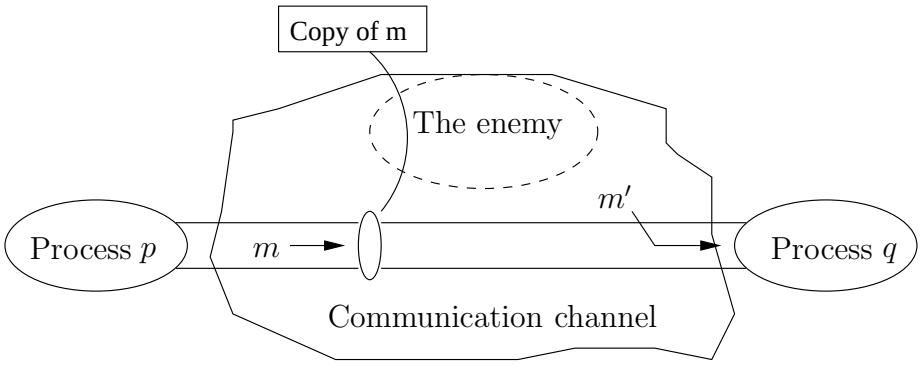


Figure 17: The enemy (Fig.2.18, p.73)

(a) *The enemy*. Under the proposed security model, an enemy is an object that is capable of sending any messages to any process and reading or copying any messages between a pair of processes (Fig.2.18, p.73)

(b) *threats to processes*.

i. A process may receive messages from another process whose identity may not be determined by the former. Eg.: a process P may intercept the IP address of a process P_1 which is communicating with a third process P_2. P then may use P_1's IP address to communicate with P_2.

ii. The described problem is caused by lack of knowledge of the source of a message by both client and server processes:

- For a server, it may serve many clients. It's difficult for it to check the identity of the principal behind a particular request message. As explained in above example, Even if a principal's identity is required, an enemy could still fake such an identity.
- Similarly, a client may have difficulty to determine the identity of principal of a reply message.

(c) *threats to communication channels*.

i. An enemy may copy, alter, or inject false messages onto a comm. channel.

ii. An enemy can also copy and save of a message, then send that message at a later time (e.g. making multiple deposits into a banking account). How to do that?

e. Defeating security threats – security channels

(a) Processes can be protected and communication channel threats can be eliminated if comm. channels are made *secure*, i.e. not subject to all threats.

(b) *cryptography* and *encryption*:

· *encryption*: the process of scrambling a message in a way to hide its contents.

· *cryptography*: the science of keeping messages secure. Cryptography is based on encryption algorithms that use secret keys, which are difficult to guess and usually a large number obtained by using mathematical theory.

(c) *Authentication* of messages: verify the the authenticity of messages by including in a message an encrypted portion that contains enough info to guarantee its authenticity.
Example: in a message requesting read permission of a file, the identity of the file, the date and time of request, the principal's identity, may be encrypted with a secret key shared between the file server and the requesting process.

(d) Security channels. A security channel is a comm. channel connecting a pair of processes, each of which acts on behalf of a principal (Fig.2.19,p.75). A security channel has the following properties:

i. Each process knows reliably the identity of the principal on whose behalf the other process is executing (client knows it is requesting/invoking the the identity of the principal of the server that will server its request; the server knows the identity of the principal behind the invocations and can check their access rights before operations are granted).

ii. A security channel ensures privacy and integrity (protecting against tampering) of the data transmitted across it (encryption is used). 窜改

iii. Each message includes a time stamp (physical or logical) to prevent messages from being replayed.

(e) Example: VPN

f. Other possible threats - *denial of service attacks*.

(a) *Deniel of service*.

· Making a service out of service by exploring software bugs in a server's code.

· Congesting comm. links to a server site or exhausting a server site's computation power by making excessive requests.

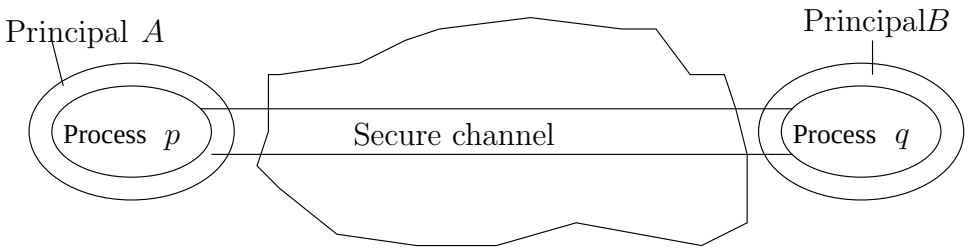


Figure 18: Secure channels (Fig.2.19, p.75)

(b) *Mobile code*. Emails that carry attachment are a good example of potential threats of mobile code. There are many other methods that are used to explore mobile code. Example: in a UNIX based computer, explore software bug in the code of a process with root privilege.

(5) Use of security models

a. The discussed security model can be used as the basis in dealing with the security aspect of a secure DS.

b. Secure channels have to be used against security threats.

c. Employment and management of security channels may incur substantial operation overhead.

d. Due to their enormous complexity, security threats in DSs may emerge from many sources in systems network environment, physical environment, and human environment.

e. Many different security models may be available in design of a DS.

(a) During the design phase, *threat models* are constructed to list all possible threats the their consequences.

(b) Strategies and techniques to prevent each such threats will be evaluated in terms of their effectiveness and cost.

(c) Example: in a UNIX system, do we use classical password protection scheme for user login authentication, or do we use *kerberos*?

Lecture 4
Distributed Time

(07-14,15-2020. Chapter 14)

Lecture Outline

```
1. Time
2. Clocks, events, and process states
3. Synchronizing physical clocks
4. Logical time and clocks
5. Global states
```

1. Time

(1) Time is important in DSs for many reasons.

a. Accounting/auditing. ???

b. Security and authentication (timestamps)

c. Maintenance of consistency of shared data.

(2) Consequences and conclusions drawn from Einstein's *Special Theory of Relativity*:

a. The speed of light is constant for all observers regardless of their relative velocity (still being studied and challenged).

b. Two events being judged to be simultaneous in one frame of reference are not necessarily simultaneous to observers in another frame of reference that are moving relative to it.

(a) An observer on the Earth and an observer on a spaceship flying away from Earth will disagree on the time interval between events.

(b) As speed of the spaceship increases, the disagreement also increases. If the spaceship could travel at the speed of time, the observer could *see/observe* events that were happening in the past (called *time channel*.

c. The relative order of two events can even be reversed for two different observers. Example: A traveler in a spaceship traveling at speed of light could observe events in the past that *are happening*, while to a static observer on the Earth, those past events had happened long time ago.

d. Non-existence of *absolute physical time*:
The classical notion of *absolute physical time* from Newton has been proven non-existing: there is no special physical clock in the universe to which we can appeal when we want to measure intervals of time.

(3) Difficulty in timestamping events in DSs.

a. As every where in the universe, there is no absolute, global time that we can appeal to.

b. Physical clocks at individual computers will drift and lack of absolute time makes it impossible to accurately synchronize them.

c. Even attempts of synchronizing physical clocks to be accurate to certain degree are difficult due to the need of message transfers and various of kinds of possible faults.

2. Clocks, events, and process states

(1) DSs: a DS consists of a set P of N processes $p_i, 1 \leq i \leq N$.

a. Each process executes on a single processor and processors do not share memory

b. At any given instance, a process p_i is in *local state* s_i. The process will transform to its next state s_{i+1} after the occurrence of the next *event*.

c. The local state of process p_i in p includes values of all the variables in it and values of any objects/resources in its local OS environment that p_i can access/modify. 什么意思

d. An *action* that a process can take is either a communication action that sends or receives a message, or an internal action which affects its local environment.

e. An *event* is the occurrence of an action that a process takes.

(2) *Event ordering* within a single process and *history* of a process.

a. The sequence of events within a single process p_i can be placed in a single straight time line, which result in a *total ordering* (what is this?) denoted by $\rightarrow_i$: 这是什么

(a) $e \rightarrow_i e'$ iff event e occurs before event e' in p_i.

(b) The assumption of a single processor makes this ordering well-defined, hence a total ordering (even in the presence of multi-threading of the OS).

b. The *history* of process p_i is a series of events that take place in it ordered by the total ordering relation $\rightarrow_i$:

$$history(p_i) = h_i =< e_i^0, e_i^1, e_i^2, \cdots >$$

(3) *Clocks*

a. Each event in a process is assigned a timestamp – a value recording the date and time of day of occurrence of the event.

(a) This timestamp can be either a hardware clock value $H_i(t)$ (generated through hardware interrupts) or a software clock value $C_i(t) = \alpha H_i(t) + \beta$, where parameters α and β are used to adjust software clock values (so that they can be more accurate).

(b) Since hardware clock cannot be completely accurate, the software clock values $C_i(t)$ will differ from the physical time t.

b. Successive events will correspond to different timestamps only if the *clock resolution* (the period between updates of the clock value) is smaller than the time interval between successive events. 什么意思?

c. *Clock skew* and *clock drift*, Fig.4.1, p.598 不懂

(a) *Clock skew*: the instantaneous difference between the readings of any two clocks is called their *skew*.

(b) *Clock drift*: physical clocks are usually crystal based. Manufactureship and interferences tend to make them operate at slightly different frequencies, termed as *clock drift*.

(c) *Clock drift rate*: is the change in the offset between the clock reading and a 名义的 nominal perfect reference clock per unit of time measured by the reference clock. Usually in the range 10^{-6} or smaller. 1 second every 1,000,000 seconds, or 11.6 days

(4) *Coordinated universal time* (UTC).

a. *International atomic time*: a standard second of IAT is defined as 9,192,631,770 periods of transitions between the two hyperfine levels of the ground state of Caesium-133 (Cs^{133}). 超精细的

b. Why is UTC used as the acronym for Coordinated Universal Time instead of CUT?

The following is from (last verified available 06/04/2008)

http://www.boulder.nist.gov/timefreq/general/misc.htm

```
In 1970 the Coordinated Universal Time system was devised by an
international advisory group of technical experts within the
International Telecommunication Union (ITU). The ITU felt it
was best to designate a single abbreviation for  use in all
languages in order to minimize confusion.   Since unanimous
agreement could not be achieved on using either the English
word order, CUT, or the French word order, TUC, the acronym
UTC was chosen as a compromise.
```

首字母缩略词

天文学的

c. *Astronomical time*: defined according to the rotation of the Earth on its axis and its rotation around the Sun.

d. The IAT and astronomical time (AT) have the tendency to get out of sync due to the instability of AT.

e. *Coordinated universal time* based on AT, but with the adjustment of inserting or deleting leap second to keep it synchronized with IAT.

(a) UTC signals are synchronized and broadcast globally and regularly from ground stations and satellites. GPS (global positioning system) is one source that broadcasts UTC.

(b) Signals received by individual computers with attached receivers have an accuracy in the order of 0.1 - 10 ms.

3. Synchronizing physical clocks

(1) The physical clock synchronization problem: how to synchronize your PC's clock? You may ask: synchronize with whom? This depends on your need:

a. Do you want to synchronize your clock as close as possible in reference to a standard clock source? Or

b. Do you want to synchronize your clock as close as possible with clocks in a group of other computers?

The former is called *external synchronization*, while the latter is called *internal synchronization*.

(2) *External synchronization* vs. *internal synchronization* (p.599). Assume I is a set of time intervals that will be taken by a UTC source. These two types of synchronizations are formally defined as follows.

"任意": ∀; "存在": ∃

a. *External synchronization*: For a synchronization bound $D > 0$ and for a source S of UTC time, $\forall t \in I |S(t) - C_i(t)| < D$, for $1 \leq i \leq N$.

b. *Internal synchronization*: For a synchronization bound $D > 0$ and for a source S of UTC time, $\forall t \in I |C_i(t) - C_j(t)| < D$, for $1 \leq i, j \leq N$.

c. Clocks that are externally synchronized are also internally synchronized and the reverse may not be true.

d. *Correctness* of clocks.

(a) A physical clock H is *correct* if its drift rate is within a known bound $\rho > 0$ (eg. $\rho = 10^{-6}$).

(b) Formally, a clock H is correct with respect to a drift rate ρ if for any two real time t and t':

这是针对任意时间 t

T是real time，而H(t)是实验对象的时间

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

禁止

The correctness notion thus defined forbids bumps in the value of hardware clocks (hence software clocks too).

不允许一下快一下慢

?

e. *Monotonicity* of clocks. A software clock C meets the *monotonicity* condition if it always advances:

$$t' > t \Rightarrow C(t') > C(t)$$

Clocks that do not satisfy the correctness condition can be adjusted to meet the monotonicity condition (to be seen shortly in Lamport's algorithm).

f. *Hybrid correctness*. A clock obeys the monotonicity condition and its drift is bounded *between synchronization points*, but is allowed to have bumpy drift ahead of synchronization points.

允许一下快一下慢

g. *Faulty* clocks.

(a) Clocks that are neither correct nor monotonic are called *faulty*.

(b) A clock that stops ticking completely is said suffering a *crash failure*.

(c) Any other clock failure is an *arbitrary failure*.

(3) Illustration of the problems with clock synchronization problem.

a. open an web page at the

URL http://www.time.gov/timezone.cgi?Central/d/-6/java

A clock showing the current CDT time will be displayed.

b. A clock skew value is also being displayed (such as *Accurate within 0.3 seconds*).

c. The JAVA program (actually a Java script) that displays the clock maintains a *virtual clock*.

* Observation 1: the Java script has to contact its external time source to re-synchronize with that source (this can be observed by disconnecting your PC from your ISP after displaying the URL). This involves *physical clock synchronization*.
* Observation 2: Displaying the URL page from different computers and different times may yield different *accuracy*. This indicates that accuracy is affected by communication *delays* between your browser and the external time server.
* Question: how does that Java script know the "accuracy" of current displayed clock?
* Answer: the Java script must be using certain algorithm to synchronize its clock with the external clock and to calculate "accuracy". The algorithm must take into accounts the comm. delays during synchronizations.

(4) Synchronization in a synchronous system

a. First attempt. Two processes are trying to synchronize their clocks internally.

(a) One process sends time t of its local clock to the other process.

(b) Assuming that it takes T_{trans} to deliver the synchronization message. The receiving process could set its clock value to $t + T_{trans}$.

b. Problem: the value of T_{trans} is difficult to obtain and estimate.

c. A minimum transmission time min can be estimated. This estimation can be done relatively accurately if the traffic between the two processes is very light.

d. In a synchronous system, there is an upper bound max on the time taken to deliver a message.

(a) Knowing the minimum possible transmission time min and maximum possible transmission time max, the *uncertainty* u in message transmission is

$$u = (max - min)$$

(b) If a receiving process sets its clock to be $t + min$ or $t + max$, then the clock skew is at most u in both cases. If set to $t + (max + min)/2$, then the clock skew is at most $u/2$.

(c) In general, in a system with N processes, the optimum bound on the clock skew is

$$u(1 - 1/N)$$

e. Most DSs are asynchronous.

(a) For such DSs, the value of max cannot be determined.

(b) For such DSs, the best we can say that $T_{trans} = min + x$ for some $x \geq 0$, which cannot be measured in theory.

(5) Cristian's synchronization algorithm (Cristian, 1989)

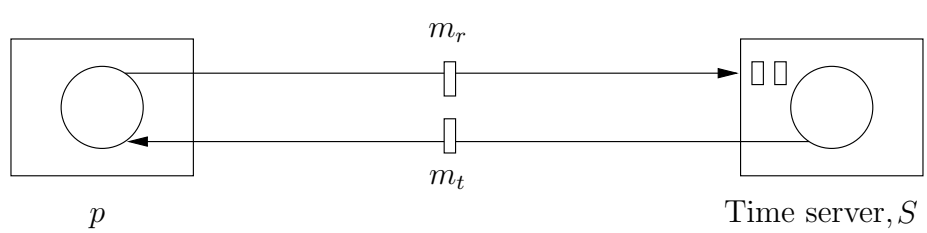


Figure 19: Clock synchronization using a time server (Cristian's algorithm)

a. Basic idea (Fig.14.2, p.601): the algorithm is a slight extension of the preliminary ideas of synchronization in synchronous systems. A time server connected to a device that receives UTC signals is configured. A client will send requests to the time server periodically which will send replies according to its clock values.

b. Basic observation and assumption:

   (a) Basic observation: even though there is no upper bound on message transmission delays in an asynchronous DS, the round-trip times between a pair of processes are often reasonably short, usually a fraction of a second.

   (b) Basic assumption: the observed round-trip times between a client and the time server are sufficiently short compared with the required accuracy.

c. The algorithm

   (a) A process p requests the time in a message m_r, and receives the time value t in a reply message m_t. To maximize accuracy, the server will insert the value t into m_t at the last possible instance before replying.

   (b) Process p records the total round-trip time T_{round} taken to send the request m_r and receive the reply m_t.

      · This round-trip time can be measured with reasonable accuracy if the clock drift rate is small.

      · Example: round-trip times on a typical LAN is in the order of 1-10 ms.什么意思 A clock with drift rate of 10^{-6} (seconds/second) varies by at most 10^{-5} milliseconds. 在说什么

   (c) The client process p will set its clock to $t + T_{round}/2$.

      · This calculation is quite accurate if the client and the server are on the same LAN segment.

      · If the client and server are on different networks, then the time taken to send m_r to server and the time taken to send reply m_t to client could be quite different. That would cause inaccuracy on the client clock.

d. Accuracy estimation. Assume that the minimum transmission time is known as min.

   (a) Observations:

      · The earliest instance at which S could have placed the time in m_t was min after p dispatched m_r; 就是信息第一帧

      · The latest instance at which S could have placed the time in m_t was min before m_t arrives at p. 信息最后一帧

      · The width between the above two instances is $T_{round} - 2min$.

   (b) Based on above, the accuracy is

$$\pm(T_{round}/2 - min)$$

   (c) Accuracy can be improved by making several requests to S and taking the minimum (not average!) value of T_{round}.

(d) The greater is the accuracy required (hence the DS is busy and demanding), the smaller is probability of achieving it. This is so because the most accurate results are those in which both messages are transmitted in a time close to *min* – an unlikely event in a busy network.为什么busy就不能接近min

e. Discussions

(a) Failure of the time server S may defeat the whole scheme. It was suggested that a group of time servers are used to improve availability of time servers. A client process can multicast its requests to all time servers and will only use the first reply received.

假的 江湖骗子

(b) A faulty time server that replied with spurious clock values, or an imposter time server that replied with deliberately incorrect times could corrupt the whole scheme. Again, a group of time servers can be used to resolve this problem. The issue is related to the distributed consensus problem to be covered in Chapter 12.

(6) The Berkeley algorithm (Gusella and Zatti, 1989)

a. Developed for systems of computers running Berkeley UNIX.

b. The algorithm:

(a) A single computer is designated as *master*. The rest of computers whose clocks are to be synchronized are called *slaves*.

(b) The master computer periodically *polls* slaves, which respond by sending back their own clock values.

(c) The master estimates their local clock times by observing the round-trip times (using techniques similar to in Cristian's algorithm). It averages values obtained (clock value plus half of round-trip time from each computer), including its own clock value.

(d) The master sends to each slave the amount by which the slave's clock has to be adjusted. This amount can be positive (the slave's clock is slower), or negative (the slave's clock is faster).

(e) Accuracy of the algorithm is improved by two special actions of the master:

名义上的

· The master assumes a nominal maximum round-trip time between the master and slaves. With this, the master will eliminate any occasional readings larger than the nominal maximum round-trip time.

· The master also has a pre-defined specified amount by which a slave's clock can drift. The master takes so called *fault-tolerant average* – the average calculated from slave's responses that do not differ from one another by that specified amount. This eliminates the adverse effect of slaves' clocks that may shift significantly from time to time.

c. Experiments: on a system with 15 computers, maximum round-trip time taken to be 10 ms, the local clocks' drift rate was less than 2×10^{-5}, the algorithm synchronized the clocks within about 20-25 ms.

d. Failure of the master of course will defeat the whole algorithm. An election algorithm can be used to elect new master.

(7) The NTP (network time protocol)

a. NTP is an external syn algo intended to provide accurate time services over Internet. Main features and goals of NTP are:

(a) *Providing a service enabling clients across the Internet to be synchronized accurately to UTC*. NTP does so in the presence of variable and large delays of message transmissions. It employs statistical techniques to filter timing data and discriminates between the quality of timing data from different servers. 什么意思?

(b) *Providing a reliable service that can survive lengthy losses of connections*: Redundant servers and redundant paths between servers are configured, reconfiguration will be performed if necessary, to provide high available services.

(c) *Enabling clients to re-synchronize sufficiently frequently to offset rates of drifts found in most computers*: NTP can scale gracefully to serve a large number of clients and servers.

(d) *Providing protection against interference with the time service, whether malicious or accidental*: NTP clients and servers authenticate the sources of timing data. They also verify the addresses of messages received. (Hence messages from impostors can be detected).

什么是 timing data

b. Basic structures and ideas

(a) The servers in NTP are connected in a logical hierarchy. All servers connected in a logical hierarchy form a *synchronization subnet* (Fig.14.3, p.604). The levels of servers in a logical hierarchy is called *strata*.

(b) In a synchronization subnet:

- *Primary servers*, located at stratum 1, are directly connected to UTC sources.
- *Secondary servers*, located at stratum 2, receive timing data from (are synchronized with) primary serves.
- Similarly, servers at stratum 3 are synchronized with secondary servers, and so on.
- Those leaf servers execute at individual users' workstations.

Note: there are no pure clients in NTP. Secondary servers are the clients of primary servers; servers at stratum 3 are clients of secondary servers, and so on.

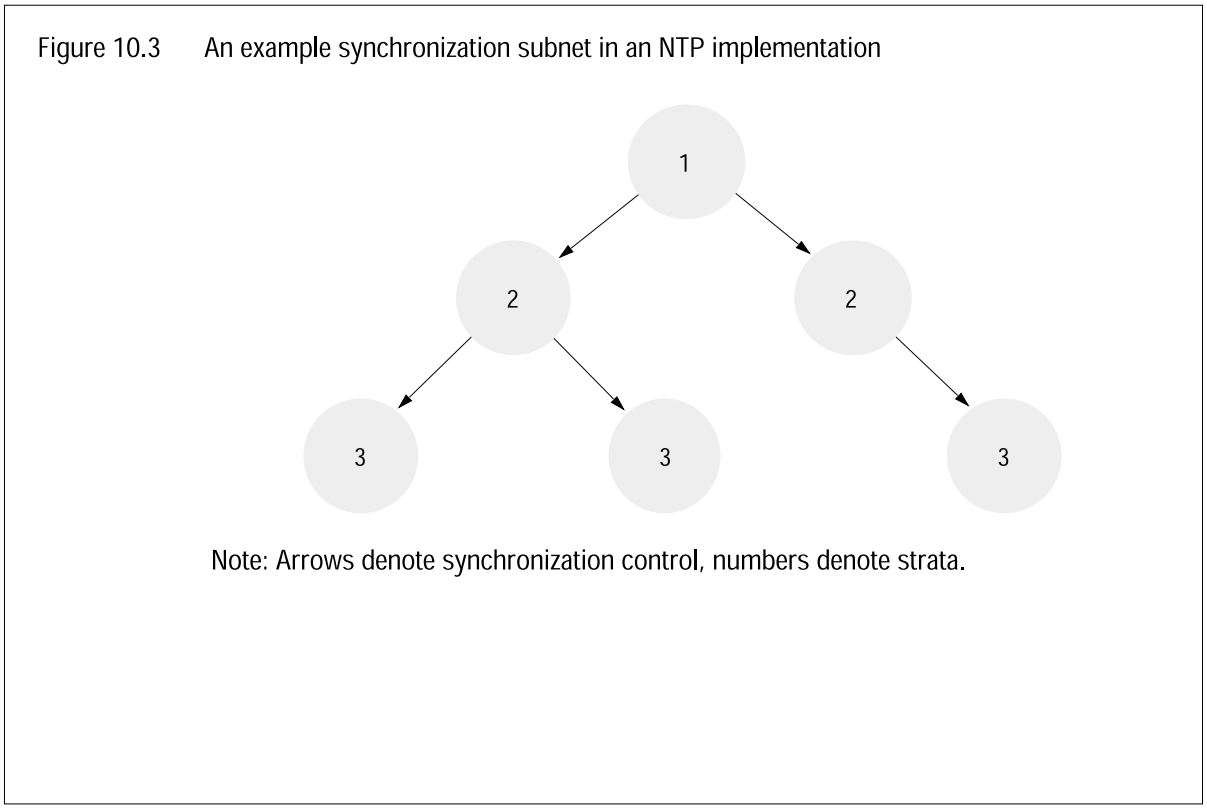


Figure 20: An example synchronization subnet in an NTP implementation (Fig.14.3,p.604)

(c) Due to the hierarchical structure, clocks in servers who are farther away to primary servers (namely who have a larger stratum numbers) are less accurate than those who are closer to the primary servers (errors accumulate at each level). NTP also uses the total round-trip delays to the root in accessing the quality of timekeeping data held by a particular server. 什么意思?

(d) A synchronization subnet can dynamically reconfigure as failures make timing sources unreachable or unavailable. Examples:

- If a lower level (with smaller stratum) server becomes unavailable, a connected higher level server (with larger stratum) may try to synchronize with another server (at a lower stratum or higher stratum).
- If the UTC source of a primary becomes unavailable, it may become a secondary server. 这是什么?

c. Operation modes: NTP servers synchronize with one another in one of three modes:

(a) *multicast* mode. Intended for use on high speed LANs, which have small delays and on which multicast/broadcast can be easily performed.

- One or more servers multicasts its clock value on a LAN, which will be

picked up by other servers attached to the LAN.

- This mode only achieves relatively low accuracies at a very low overhead. 经费

(b) *procedure-call.*

- This mode is similar to Cristian's algorithm. One server accepts clock inquiries from other servers and replies with its current clock reading.
- This mode is used for applications where multicast mode cannot work, or higher accuracies are needed.

(c) *symmetric.*

这是什么

- This mode can provide the highest accuracies. It can be used in multicast LANs or by higher levels of a synchronization subset to synchronize with one another (in order to achieve better accuracy. recall servers at higher levels have less accuracies). 什么意思
- Under this mode, a pair of servers exchange timing data. This exchange can last a period of time and exchanged time data are retained in order to improve accuracy of their clocks.

UDP is the underlying transport protocol used by all three modes. Unreliably

d. Details of *procedure-call* and symmetric modes.

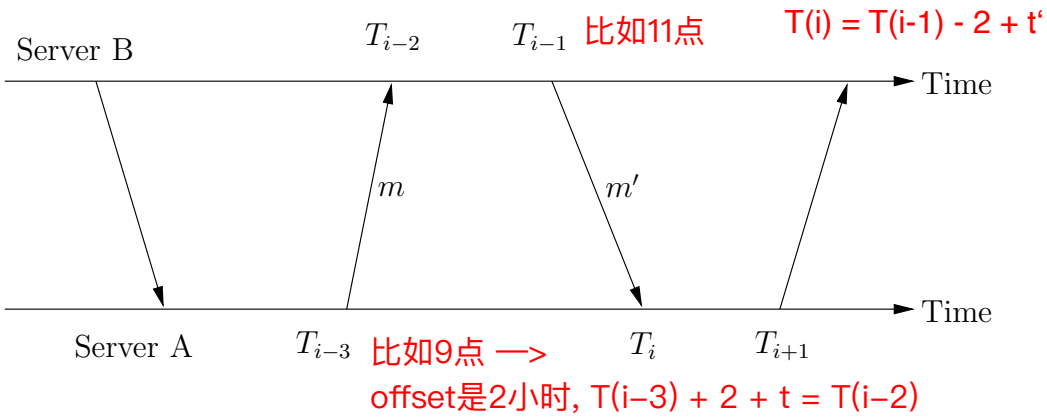


Figure 21: Messages exchanged between a pair of NTP peers (Fig.14.4,p.605)

(a) Processes (servers, or clients/servers) exchange pair of messages. Each such message bears three timestamps of recent message events (Fig.14.4, p.605). For example, when B sends m' to A, it includes T_{i-3}, T_{i-2}, and T_{i-1}. 这是什么意思?

- T_{i-3}: the local time when the previous message was sent;
- T_{i-2}: the local time when the previous message was received;
- T_{i-1}: the local time when the current message was sent;
- T_i: the local time when the current message was received;

因为是UDP, 所以要加上这三个时间点，以防发送失败可以重发（至少可以追溯问题）

Next time when A sends a message to B, it will include T_{i-1}, T_i, and T_{i+1}. 这是什么意思?

(b) Two more notes:

比如我发送失败100次，第101次发送的时候，我还是要attach最初的发送时间，T(i–1)可以变成最新发送时间，但是之前对方的发送时间要如实相告

- For symmetric mode, the interval between the arrival of one message and the dispatch of the next can be non-negligible.
- Messages can be lost. After a loss, a re-transmitted message will bear a new T_{i-1}, but same T_{i-3} and T_{i-2}.

(c) For each pair of messages exchanged between two servers, the NTP calculates *an offset* o_i, which is an estimate of the actual offset between the two clocks; and 偏移o-i 是对两个时钟的偏移估计

真实的offset无法获取，只能估计

- *a delay* d_i, which is the total transmission time for the two messages. 延迟d–i是两个消息传输的整个时间
- Let o be the true offset of the clock at B relative to that at A (remember we assume A starts the exchange).
- Let t and t' be the actual transmission times for messages m and m', respectively. Note: t and t' are *actually transmission times* for m and m' and cannot be actually measured.

(d) The following relationship holds:

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

which can be better understood from:

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

我们没法单独获取t或者t’ 但是合并这两个等式，可以发现o 不见了。

and be careful that:

$$t \neq T_{i-2} - T_{i-3} \quad \text{and} \quad t' \neq T_i - T_{i-1}$$

The above leads to:

延迟

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

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

and we therefore have:

估算offset

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

Using the fact that $t, t' \geq 0$, and

$$t' = d_i - t \quad \text{and} \quad t = d_i - t'$$

$$o = o_i + ((d_i - t) - t)/2 \quad \text{and} \quad o = o_i + (t' - (d_i - t'))/2$$

i.e.

$$o = o_i + d_i/2 - t \quad \text{and} \quad o = o_i - d_i/2 + t'$$

we can have:

$$o_i - d_i/2 \leq o_i - d_i/2 + t' = o = o_i + d_i/2 - t \leq o_i + d_i/2$$

i.e.

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

This shows that o_i is an estimate of o and d_i is a measure of the accuracy of this estimate. The following four examples demonstrate that o_i is a good estimate of o if t and t' stay close.

(e) Example 1: Assume $o = 0.1$ i.e. B's clock is 0.1 second faster than A's. Also assume that *actual time* t used to send m is the same as that of sending m' with value 0.5 seconds, i.e. $t = t' = 0.5$ seconds.

process A	process B	Notes
send(m) $T_{i-3} = 12.0$		At this point, B's clock has value 12.1
	receive(m) $T_{i-2} = 12.6$	At this point, A's clock has value 12.5
	send(m') $T_{i-1} = 13.2$	At this point, A's clock has value 13.1
receive(m') $T_i = 13.6$		At this point, B's clock has value 13.7

From above table, we have:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 = (12.6 - 12.0 + 13.2 - 13.6)/2 = 0.1$$

By the conclusion of NTP algorithm, it should have:

$$o = o_i + (t' - t)/2 = 0.1 + (0.5 - 0.5)/2 = 0.1$$

This shows that the estimated value is accurate and confirms the assumption that B's clock is 0.1 seconds faster than A's.

(f) Example 2: Here assume $o = -0.2$ (i.e. B's clock is 0.2 second slower than A's). Also that *actual time* t used to send m is the same as that of sending m' with value 0.5 seconds, i.e. $t = t' = 0.5$ seconds.

process A	process B	Notes
send(m) $T_{i-3} = 12.0$		At this point, B's clock has value 11.8
	receive(m) $T_{i-2} = 12.3$	At this point, A's clock has value 12.5
	send(m') $T_{i-1} = 13.4$	At this point, A's clock has value 13.6
receive(m') $T_i = 14.1$		At this point, B's clock has value 13.9

From above table, we have:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 = (12.3 - 12.0 + 13.4 - 14.1)/2 = -0.2$$

By the conclusion of NTP algorithm, it should have:

$$o = o_i + (t' - t)/2 = -0.2 + (0.5 - 0.5)/2 = -0.2$$

This again confirms the NTP's conclusion is correct.

(g) Example 3: Again assume $o = 0.1$ (i.e. B's clock is 0.1 second faster than A's). But we assume that *actual time* used to send m is 0.5 seconds and the *actual time* of sending m' is 0.7 seconds.

这里用o来得出T(i-2)和T(i-1)的？？？？

process A	process B	Notes
send(m) $T_{i-3} = 12.0$		At this point, B's clock has value 12.1
?	receive(m) $T_{i-2} = 12.6$	At this point, A's clock has value 12.5
	send(m') $T_{i-1} = 13.2$	At this point, A's clock has value 13.1
receive(m') $T_i = 13.8$		At this point, B's clock has value 13.9

From above table, we have:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 = (12.6 - 12.0 + 13.2 - 13.8)/2 = 0$$

$$o = o_i + (t' - t)/2 = 0 + (0.7 - 0.5)/2 = 0.1$$

This again confirms that NTP's conclusion is correct.

(h) Example 4: Assume $o = -0.3$ (i.e. B's clock is -0.3 second faster than A's). But we assume that *actual time* used to send m is 0.9 seconds and the *actual time* of sending m' is 0.5 seconds.

process A	process B	Notes
send(m) $T_{i-3} = 12.0$		At this point, B's clock has value 11.7
	receive(m) $T_{i-2} = 12.6$	At this point, A's clock has value 12.9
	send(m') $T_{i-1} = 15.3$	At this point, A's clock has value 15.6
receive(m') $T_i = 16.1$		At this point, B's clock has value 15.8

NTP 服务器对于连续的 $<o_i, d_i>$ 对应用数据过滤算法，用于估计偏移 o 并计算这个估计的质量（采用称为过滤离中趋势（filter dispersion）的统计量形式）。若过滤离中趋势较高，则表示数据相对而言不可靠。保留 8 个最近的 $<o_i, d_i>$ 对。与 Cristian 算法一样，选择对应于最小 d_i 值的 o_j 的值用于估计 o。

NTP servers apply a data filtering algorithm to successive pairs $<o_i, d_i>$, which estimates the offset o and calculates the quality of this estimate as a statistical quantity called the *filter dispersion*. A relatively high filter dispersion represents relatively unreliable data. The eight most recent pairs $<o_i, d_i>$ are retained. As with Cristian's algorithm, the value of o_j that corresponds to the minimum value d_j is chosen to estimate o.

From above table, we have:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 = (12.6 - 12.0 + 15.3 - 16.1)/2 = -0.1$$

$$o = o_i + (t' - t)/2 = -0.1 + (0.5 - 0.9)/2 = -0.3$$

(i) Summary of the four examples:

- NTP's conclusion is verified by the four different scenarios. 这是什么意思?
- The transmission delays of sending m and m' affect the value o_i obtained by A.

(j) NTP employs a data filtering algo to eight successive pairs $< o_i, d_i >$. 这是什么意思?

这是什么意思?

- The algo calculates the quality of estimate of o_i as a statistical quantity called *filter dispersion*. 什么意思?
- High filter dispersion means unreliable data.
- The value of o_j that corresponds to the minimum value d_j is chosen as o_i.

(k) Multiple sources may be used by a server to improve accuracy.

- NTP engages synchronization actions with several possible peers.
- Besides the filter algo, NTP also applies a peer-selection algo. Generally, peer servers that exhibit unreliable timing data as determined by the peer-selection algo will be eliminated. 大概意思就是，最差的时间会被重新同步
- When the output of the peer-selection algorithm dictates change of a peer, NTP will try to select a peer that is at a lower stratum (they are closer to the primary server and hence are more accurate). 什么意思?
- In addition, peers with the lowest *synchronization dispersion* are favored. Syn dispersion is the sum of filter dispersions measured between the server and root of the synchronization subnet. Peers exchange synchronization dispersions in messages regularly.

这是啥意思?

e. A phase lock loop model is used to modify the local clock's update frequencies in terms of observations of its drift rate as determined from above discussion.

(a) This model reduces a clock's frequency if the clock is found faster and increases a clock's frequency otherwise.

(b) The model has a synchronization accuracy of one millisecond on LANs and an order of tens of milliseconds over Internet paths. Page 606

4. Logical time and clocks

(1) Introduction

a. Physical clock synchronization techniques are limited in accuracies. Therefore, in critical applications we cannot rely on physical time to find out the order of event executions in a DS.

b. In many applications, we are interested in the *relative order* of certain events, and we are not interested in the absolute physical timing of those events. Example: we want to make sure that a file is modified after a valid modification request has been issued.

(2) Lamport's method

a. The assumptions:

(a) Each individual process p_i is sequential, i.e. the events in a p_i are totally ordered according to their physical orders. This order is denoted as $\rightarrow_i$.

(b) The event of "sending" a message always occurs before the event of "receiving" the same message.

b. The *happened-before* relation: all events in a DS can be ordered with a simple scheme generalized from above assumptions. The resulted relation is called *happened-before* (or *causal ordering*, or *potential causal ordering*) relation, as defined formally:

HB1: if $\exists$ process $p_i : e \rightarrow_i e'$, then $e \rightarrow e'$.

HB2: For any message m, $send(m) \rightarrow receive(m)$, where $send(m)$ is the event of sending the message and $receive(m)$ is the event of receiving the same message.

HB3: Transitivity: if e, e', e'' are events and $e \rightarrow e'$ and $e' \rightarrow e''$ both hold, then $e \rightarrow e''$ also holds.

An example: Fig.14.5, p.607. Three processes, six events, and two message transmissions. We have $a \rightarrow f$, for instance.

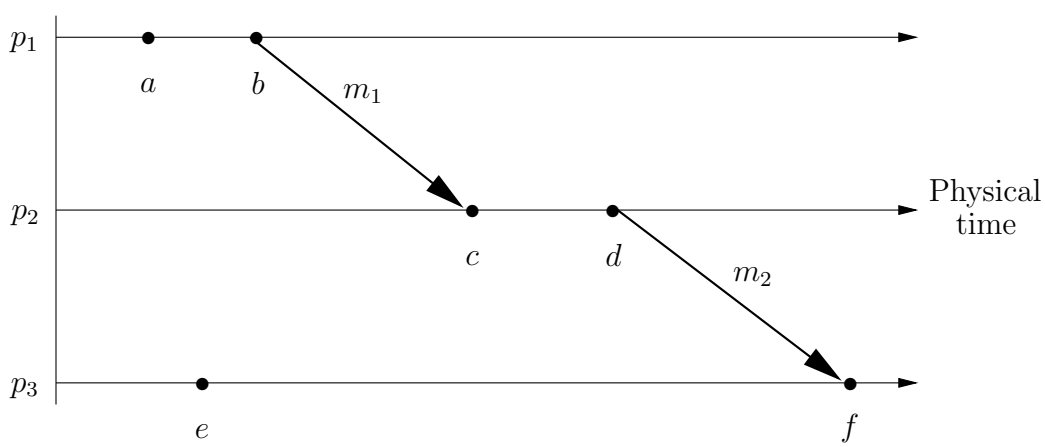


Figure 22: Events occurring at three processes (Fig.14.5,p.607)

c. Concurrent events and discussions

(a) Not all events in a DS are related by the happened-before relation $\rightarrow$.

- Example: Event a and e are not related, neither are c and e.
- Two events a and e that are not ordered by the relation $\rightarrow$ are said *concurrent* are denoted by $a \| e$.

(b) The relation $\rightarrow$ captures a flow of data based on sequential ordering of events within a process and on ordering of message passing. However, data can flow in ways other than message passing, which cannot be expressed by $\rightarrow$. Page 608

(c) If the relation $\rightarrow$ holds between two events, the first might or might not actually cause the second (A process could issue a new message after receiving a message. But that new message should be sent any way). 比如A—> B，只是代表A可以发送到B，但不是必须要发送

d. Logical clocks: introduced to capture and maintain the happened-before relation $\rightarrow$.

(a) Each process p_i maintains a monotonically increasing software variable L_i, called the logical clock for p_i. L_i is used to apply *Lamport timestamps* to events.

(b) Notation: for any event e at p_i, we use $L_i(e)$ to denote its timestamp. $L(e)$ is used to denote the timestamp of event e regardless process to which e belongs.

(c) The system of logical clocks $L_i, 1 \leq i \leq N$, are maintained according to the following rules:

LC1: L_i is incremented before each event is issued at p_i: $L_i = L_i + 1$

LC2: When a process p_i sends a message m, it attaches the value $t = L_i$ to m.

LC3: On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$, and then applies LC1 before timestamping the event $receive(m)$. 难道Lj会比t小吗？？？？

(d) Example and notes:

- Fig.14.6, p.609 shows the assignment of logical clock values to each event at three processes.
- The amount of increment at each process can be any positive value $\beta > 0$, not necessarily 1.
- By induction on the length of any sequences of events relating two events e and e', we can show that

$$e \rightarrow e' \Rightarrow L(e) < L(e')$$

(e) The reverse of above conclusion is not true: from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$.
Example: For the two events b and e in Fig.10.6 the relation $L(e) < L(b)$, but $b \| e$.

e. Totally ordered logical clocks

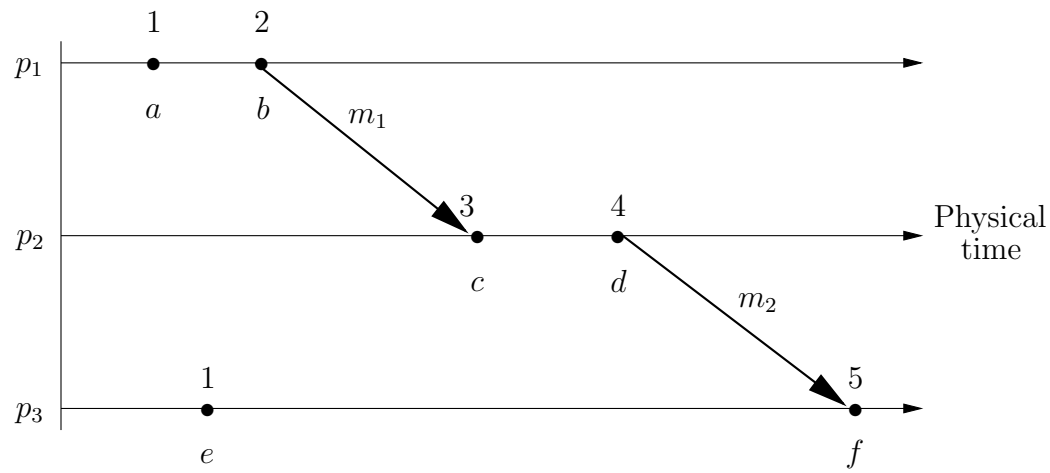


Figure 23: Lamport timestamps for the events shown in Fig.14.5 (Fig.14.6,p.609)

(a) It's possible for two events (from two different processes) to have same logical timestamp values.

(b) An artificial ordering of those pair of events can be created by including the identifier i of process p_i as part of the logical timestamps. Such timestamps are called *global logical timestamps*: (T_i, i). For two such timestamps, (T_i, i) and (T_j, j), $(T_i, i) < (T_j, j)$ if and only if either (i) $T_i < T_j$ or (ii) $T_i = T_j$ and $i < j$. 这是什么意思? 为何这里可以 i < j ??? 他们是时间吗?

不懂

Page 359

f. Vector clocks (Mattern [1989], Fidge[1991])

(a) Shortcoming of Lamport timestamps: from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$.

(b) The notion of *vector clocks* introduced can overcome this problem.

这里N是进程数量

(c) Each process p_i now keeps a vector of N-component as a logical clock variable V_i. Each event e is associated with an N-component vector as its timestamp $V_i(e) = (i_1, i_2, \cdots, i_N)$:

VC1: Initially, $V_i[j] = 0$, for $1 \leq i, j \leq N$.

VC2: Just before p_i timestamps an event, it sets $V_i[i] = V_i[i] + 1$.

VC3: p_i includes the value $t = V_i$ in every message it sends.

VC4: When p_i receives a message with timestamp t in it, it sets $V_i[j] := max(V_i[j], t[j]), 1 \leq j \leq N$. This is termed as a *merge* operation (taking component wise maximum of two or more vectors).

(d) Example: Fig.14.7, p.610.

(e) Notes:

i. Two N-component vectors V and V' can be compared by comparing their corresponding components:

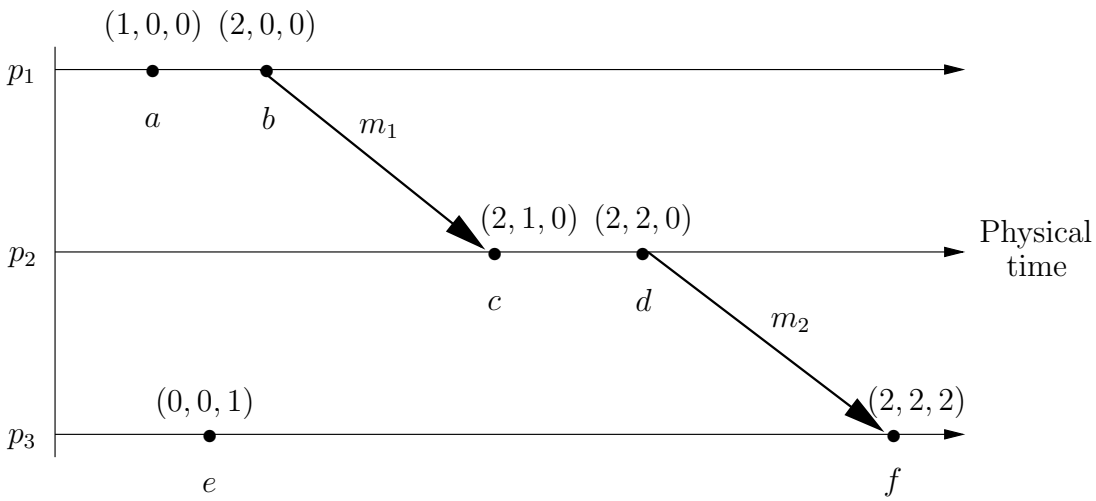


Figure 24: Vector timestamps for the events shown in Fig.14.5 (Fig.14.7,p.610)

$V = V'$ *iff* $V[j] = V'[j]$ for $j = 1, 2, \cdots, N$ iff = if and only if

$V \leq V'$ *iff* $V[j] \leq V'[j]$ for $j = 1, 2, \cdots, N$

$V < V'$ *iff* $V[j] \leq V'[j] \wedge V \neq V'$ 这个就是只要不完全相等，就是小于

ii. For any two events e and e', it is simple to see that

$$e \rightarrow e' \Rightarrow V(e) < V(e')$$

It is also can be shown that $V(e) < V(e')$ then $e \rightarrow e'$.

(f) Disadvantages of vector clocks: storage and message payload needed are proportional the size N of a DS.

g. Summary. Let P_1, P_2 and P_3 denote the following three properties:

$$P_1: \text{ if } \ e \rightarrow e' \text{ then } \ L(e) < L(e')$$

$$P_2: \text{ if } \ L(e) < L(e') \text{ then } \ e \rightarrow e'$$

$$P_3: \text{ if } \ e \neq e' \text{ then } \ L(e) \neq L(e')$$

For simplicity, only L is used in expressing P_1, P_2, and P_3. When discussing the vector timestamps, V should be used in place of L. 这里L就是V —> 当讨论

这两种method什么区别到底?

Method	P_1	P_2	P_3
Lamport's method without process ID	Always true	Not always true	Not always true
Lamport's total ordering method	Always true	Not always hold	Always true
Vector timestamps	Always true	Always ture	Always true

?

为啥?

5. Global states

(1) *Global states*. Given a set P of N processes $p_i, 1 \leq i \leq N$ that represent a DS,

a. informally speaking a global state is a collection of states $s_1, s_2, \cdots, s_N$, where each of $s_i, 1 \leq i \leq N$ is a local state in process p_i that occurred at the some time instance t_i.

b. another form of global state definition requires that all states $s_1, s_2, \cdots, s_N$, to occur at the same time instance. There are two problems with that definition:

* It is difficult (impossible) to know that two events occur at the same physical time instance.
* It also introduces the so called *state explosion* problem.

(2) Applications of knowledge of global states. With the availability of global states many important problems in DSs can be easily solved (Fig.14.8, p.611):

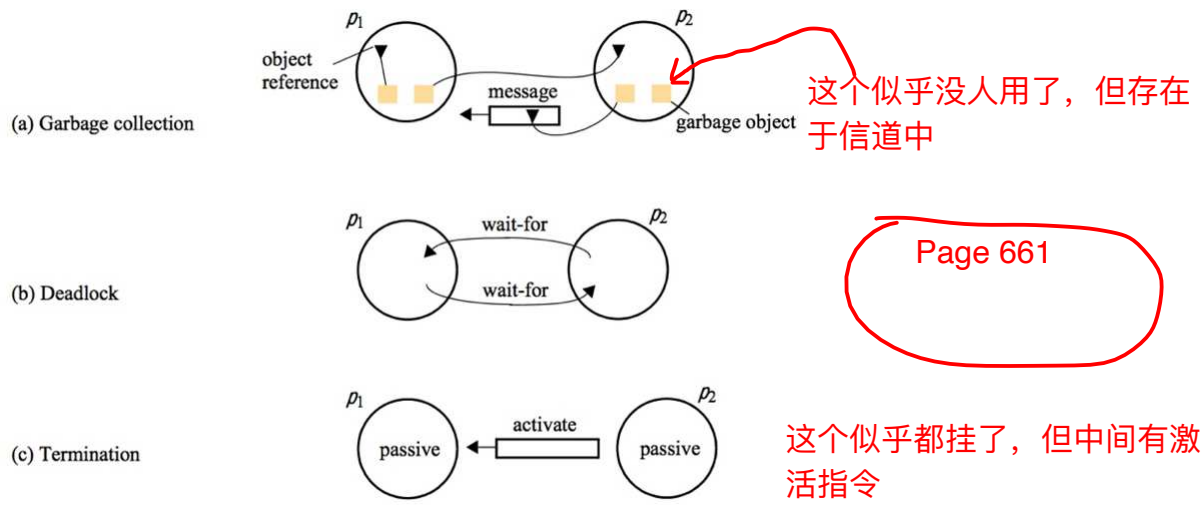


Figure 25: Detecting global properties (Fig.14.8, p.611)

a. *Distributed garbage collection*. A garbage is an object which is no longer being referenced by any processes in a DS. Hence the resources (such as memory) held by a garbage can be reclaimed by the system for reuse.

(a) Detecting whether an object is a garbage involves checking whether the object is still referenced by a process. However, in a DS an object can be referenced by a remote process (such as one of the two objects in p_1 in Fig.11.8).

(b) There is also the possibility that a process is sending a message to request access to an object which is not referenced at that given instance. Therefore the status of comm. channels should also be considered.

With global states available, we can just inspect global states from time to time to find and reclaim garbage.

b. *Distributed deadlock detection.*

   (a) The definition for deadlocks in a DS is the same as that in a centralized OS, except that in a DS a process may be holding a resource (local or remote) and waiting for allocation of another remote resource.

   (b) Therefore a distributed deadlock may involve processes and resources in a number of computers. Detecting distributed deadlocks demands cooperation of individual computers in a DS.

   Availability of global states will make deadlock detection trivial – we just inspect a global state to verify the existence of waiting cycles in it. 你在说什么？？？？？

c. *Distributed termination detection.* This is a problem to detect whether a given distributed algorithm has terminates.

   (a) It involves verification of individual processes that collectively execute the distributed algorithm have terminated.

   (b) A process can be either in a passive, in which a process does not perform any activities, or active state. Because a passive process may be re-activated by a remote process through messages, detecting whether a process will no longer be active is not trivial.

   Again, if we can obtain global state information, then we can easily check whether an activation message is in transition at a given instance.

(3) Global states and consistent cuts

a. *History* (already defined at the beginning of this lecture): For a process p_i, the history of p_i is a complete sequence of events that occur in p_i:

$$history(p_i) = h_i =< {e_i}^0, {e_i}^1, {e_i}^2, \cdots >$$

b. *Finite prefix* h_i^k of history h_i:

$$h_i^k =< {e_i}^0, {e_i}^1, \cdots, {e_i}^k >$$

c. *Global history* of a DS:

   (a) For a process p_i, the notation s_i^k is used to denote the state of p_i immediately before the kth event occurs. Thus s_i^0 denotes the initial state of p_i, because it is the state that is immediately before the 1st event e_i^0.

   (b) The state of all open comm. channels should be considered as a factor in global state definition.

(c) A *global history* H of a given DS is the union of histories of individual processes in it:

$$H = h_1 \cup h_2 \cup \cdots \cup h_N$$

d. *Cuts* of histories :

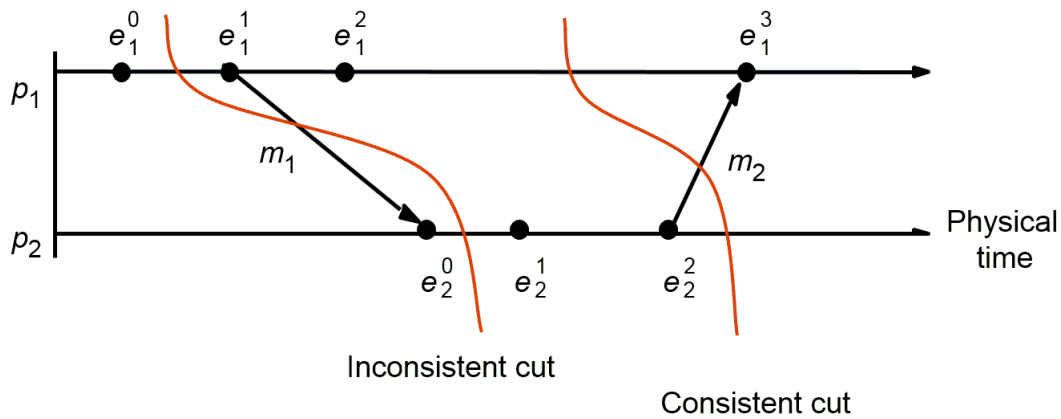


Figure 26: Cuts of a global state (Fig.14.9, p.613)

(a) A *cut* of a DS's execution is a subset of its global history. A cut must be a union of prefixes of process histories:

$$C = h_1^{C_1} \cup h_2^{C_2} \cup \cdots \cup h_N^{C_N}$$

(b) With definitions available so far, intuitively we would like to define a global state S to be a collection of states that correspond to the last states in a cut from each process.

在讲什么?

The state s_i in the global state S corresponding to a cut C is a state from process p_i that is immediately after the last event $e_i^{C_i}$ performed by p_i in the cut, $1 \leq i \leq N$. The set of events $\{e_i^{C_i} : 1 \leq i \leq N\}$ is called the *frontier* of the cut.

在讲什么?

(c) *Consistent cuts*. A cut C is *consistent* if for each event it contains it also contains all the events that *happened-before* that event:

$$\forall e \in C, f \rightarrow e \Rightarrow f \in C$$

Otherwise the cut C is said an *inconsistent cut*. 什么意思呢?

(d) *Consistent global states*: a consistent global state is one that corresponds to a consistent cut.

When we use the term *global states* we normally refer to consistent global states unless otherwise specified.

e. *Runs* and *consistent runs*:

什么区别?

(a) *Runs*. A *run* is a total ordering of all events in a global history H that is consistent with local history's ordering $\rightarrow_i$ of each process $p_i, 1 \leq i \leq N$. A run is also called a *trace* in literature. 不懂

(b) *Consistent runs* (or *linearization*). A *consistent run* is a run that is consistent with the happened-before relation $\rightarrow$ on H.

(c) Notes: Consistent runs pass through only consistent global states. Not all runs pass through consistent global states.

(d) *Reachable states*. A state S' is reachable from a state S if there is a linearization that passes S and then S'. 在说什么??

(4) Global state predicate, stability, safety, and liveness

a. A global state predicate is a function that maps from the set of global states to the set {True, False}.

* Example 1: predicate α_1: the global state is not a deadlock state. For some global states the predicate α_1 is true. For some others α_1 is false. For these states we can apply certain deadlock resolution strategies to resolve the deadlock.
* Example 2: predicate α_2: there are some garbage objects in a global state. For some global states, the predicate α_2 is true. Then we can identify the garbage objects and reclaim allocated resources. For other global states α_2 is false, i.e. in these global states there are no garbage objects.

感觉要考

b. *Stability*: a global state is stable with respect to a given predicate α, if once the system enters a global state in which the predicate α is true, it will remain in all possible future global states reachable from that global state in which α will be true.

c. *Safety*. Given an initial global state S_0, an *undesirable* property β which is expressed as a predicate of global states (e.g. β could be the property of being deadlocked). *Safety* with respect to β and S_0 is the assertion that β will never evaluate to true for all states S reachable S_0.

Safety describes some undesirable properties that will never occur.

d. *Liveness*. Liveness is a property complementing the safety property. Given a *desirable property* α (such a specific distributed algorithm will eventually terminate) and an initial global state S_0, *liveness* with respect to α and S_0 is the property that, for any linearization L starting in the state S_0, α will evaluate to true for some state S_L reachable from S_0.

Liveness describes some desirable properties that are bound to occur, sooner or later.

Lecture 5
Interprocess Communications
(July-21,22,27,2020. Chapter 4)

Lecture Outline

```
1. Introduction
2. TCP/IP API and Java UDP/TCP API
3. External data representation and marshalling
4. Remote object reference
5. Group communications
6. Network virtualization
7. Message passing example: MPI
```

1. Introduction

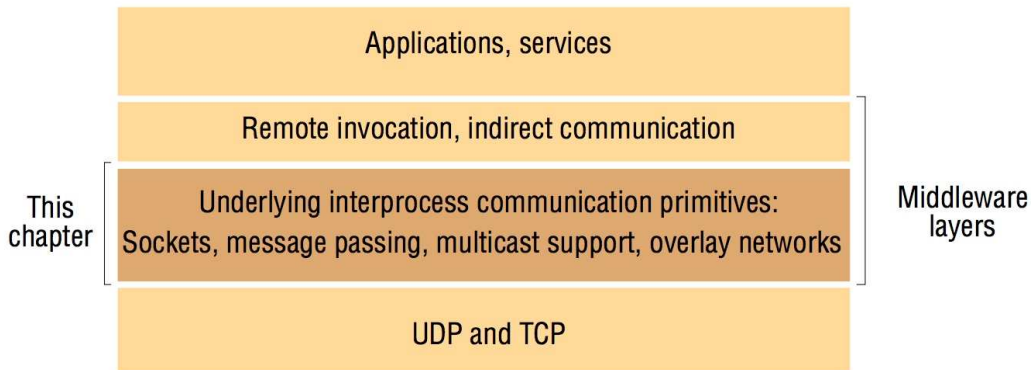


Figure 27: Middleware layers (Fig.4.1, p.146)

(1) Facilities for interprocess communications are essential for DSs.

(2) Facilities for interprocess communications can be classified in terms of abstraction levels:

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

a. Basic TCP/IP APIs: *Berkeley sockets* and *SVR4 TLIs*. (provided by AT&T UNIX System V)
https://en.wikipedia.org/wiki/Berkeley_sockets

b. RMI (remote method invocation) supported by certain object oriented programming languages or development tools such as $C++$, Java, and CORBA.什么是CORBA?

c. Remote procedure calls: SUN RPC, OSF DCE (distributed computing environment) RPC

(3) Characteristics of IPC

a. IPC involves message exchanges among processes

b. Messages

(a) A message consists of two parts: the data part, provided by the user, and the control part, attached by the system.

(b) Messages can be structured or unstructured

(c) Basic message passing primitives

标准输出

i. **send** *expression-list* **to** *destination-identifier*

ii. **receive** *variable-list* **from** *source-identifier*

c. Synchronous (blocking) and asynchronous (non-blocking) communications.

(a) Two queues, a *send queue* and a *receive queue* may be associated with each process.

(b) The action of sending a message causes the message to be transmitted and eventually to be queued to the receive message queue of destination process.

(c) A communication is *synchronous* if the the sender and receiver synchronize at every message. Both *send* and *receive* are block operations.

i. A *send* operation will block the issuing process until the corresponding *receive* operation is performed at the receiver.

ii. A *receive* operation will block the issuing process until the a message from the specified destination arrives.

(d) A communication is *asynchronous* if the *send* operation will not block the issuing process: the process can proceed as soon as the the message to be transmitted is copied to a local buffer. The *receive* operation can be either blocking or non-blocking.

i. The blocking receive operation behaves the same as in synchronous comm.

ii. The non-blocking receive, if a desired message hasn't arrived yet, after a receiving process issues a *receive* operation, the process will continue to proceed with its other operations. The system will fill a buffer (provided by the receiving process in the receive operation) later and notify the process after that.

(e) Discussions

i. Non-blocking communication is more flexible and efficient. However implementation of receiving processes is much harder. Buffers *may be* needed. 这里怎么理解Buffers

ii. Blocking communications are just the opposite.

iii. For the non-blocking receiving variant of the non-blocking communication, an extra notification is needed, which interrupts the running process. Few DSs adopts this variant.

iv. Parallelism may be lost in blocking communications 什么意思?

v. In systems where support of threads is available, *threads* can be used in synchronous communications to enhance performance. how?

(4) Quality of IPC

a. *Reliability*. An IPC facility that delivers messages un-corrupted and un-duplicated is *reliable*.

b. *Ordering*. Messages sent by a sending process may or may not arrive same as their sending order.

c. *Usability*. An IPC facility is *highly usable* it supports high degree of *location transparency*. 这里什么意思?

d. *Timeness and efficiency*. This demands quick and efficient delivery of messages.

e. Notes:

(a) Quality of IPC facilities is closely related to the quality of services at the transport layer.

(b) IPC facilities can improve their service qualities on top of services by the transport layer.

2. TCP/IP API and Java UDP/TCP API

(1) Identification of message destinations

a. Destinations of messages are identified by the *IP address* and *port number*: (*remote IP address, remote port*).

(a) In most cases a message also has to carry (*local IP address, local port*) to allow a remote destination to send replies.

(b) A process may use multiple ports to send receive messages simultaneously.

(c) Port numbers of services should be publicized by servers to clients.

b. Other means of destination identification. The use of a (*IP address, port*) pair voids the location transparency. Several other methods support location transparency:

(a) A *binder* may be supported in some DC application.

· A binder is a *service resolver*: given the name of service, it will return the server locations. https://en.wikipedia.org/wiki/Open_Network_Computing_Remote_Procedure_Call https://en.wikipedia.org/wiki/Portmap

· Sometimes the availability of a binder is essential. This is particularly true in a cloud. https://baike.baidu.com/item/portmap/8511094

· Example of binders: portmapper service; Java RMI RMIRegistry; DCE name services https://en.wikipedia.org/wiki/DARPA https://en.wikipedia.org/wiki/Distributed_Computing_Environment

(b) Some OSs such Mach provide location independent identifiers for message destinations, translating identifiers into lower-level addresses such as (IP address, port). https://en.wikipedia.org/wiki/Mach_(kernel)

原理
https://www.tutorialspoint.com/java_rmi/java_rmi_introduction.htm
https://www.mi1k7ea.com/2019/09/01/Java-RMI原理与使用/
https://developer.aliyun.com/article/66673

例子
https://www.yiibai.com/java_rmi/java_rmi_application.html
https://sites.cs.ucsb.edu/~cappello/lectures/rmi/helloworld.shtml

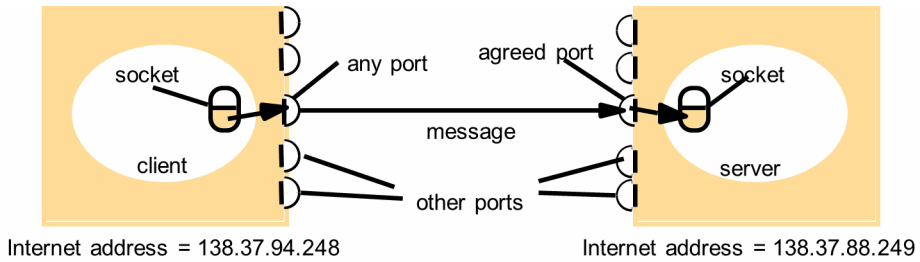


Figure 28: Sockets and ports (Fig.4.2, p.149)

(2) The Berkeley socket API

a. The *socket* API: a socket is an end point of a communication. *socket* API originated from Berkeley UNIX and are now available on Windows and Macintosh OSs.

b. Socket supports both reliable communications (connection oriented) through TCP API and un-reliable comm. (connectionless) through UDP API.

c. Illustration of *ports* and *sockets*: Fig.4.2, p.149.

d. UDP datagram communications

(a) Characteristics

i. UDP is a connectionless transport layer comm. protocol. It provides unreliable datagram delivery service.

ii. *Message sizes*: a message can be up to 2^{16} bytes (however, many implementations impose a smaller size 8 kbytes). A receiving process has to specify the maximum size of messages to be received.

iii. UDP employs *non-blocking send* and *blocking receive*.

iv. UDP relies IP to deliver datagrams. It does not handle timeouts by itself. Applications should handle timeout according to their own requirements.

v. UDP API provides several different primitives (functions) to send and receive messages. Several pairs of them allow *receive from any* semantics: they do not specify an origin of messages to be received through a socket. Any messages can be delivered and the system will fill the (IP address, port) pair of the sending process.

(b) Major UDP API functions:

i. *socket, bind* – for comm. management. 有什么区别

ii. *sendto, recvfrom, send, recv, read, write, sendv, writev, sendmsg, recvmsg* – for sending and receiving messages.

(c) Failure model:

i. Cf. Lecture 3 for failure model: it defines the ways in which failures may occur. Two properties of reliable communications: *validity* (messages are eventually delivered) and *integrity* (corrupted messages and duplicate messages can be detected).

ii. UDP provides an optional checksum that lets a receiving application verify if the contents of a datagram is corrupted. Notice checksum may not guarantee all possible corruptions to be found.

iii. *Omission failures*: messages (i.e. UDP datagrams) may not arrive at their destinations, validity is not met.

iv. *Ordering*: as individual datagrams may be delayed, they could arrive out of sender order. Since in general UDP does not do automatic retransmission, duplicate datagrams will not appear.

UDP applications that require reliable communications must provide their own acknowledgments, timeouts, and sequence numbers.

e. TCP stream communications.

(a) Characteristics

i. TCP is a connection-oriented transport layer comm. protocol. It provides reliable message delivery service.

ii. *Message sizes*: applications can decide the number of bytes to write to a stream or read from it. The underlying implementation maintains local write buffers and read buffers. The buffer sizes determines how much data to collect from a application before actually transmitting it, or how much data to accumulate from an incoming stream before delivering to an application. Applications can force data to be transmitted immediately.

https://youtu.be/LnbvhoxHn8M

iii. *Lost messages*: TCP employs the well-known *slide window* protocol to handle message transmissions. Messages are acknowledged and lost messages are retransmitted. In terms of slide window protocol concept, each byte is counted as a TCP packet. For example, when a client declares to a server window size $8k$, that means the server can send $8k$ bytes continuously before having to wait for acks from the client.

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

数据包

什么意思?

iv. *Flow control*: TCP attempts to *synchronize* the transmission speeds of a client and a server involved in a comm. in the sense that if one of them is slow, TCP will try to inform the other to slow down.

v. *Message duplication and ordering*: use of slide window protocol by TCP ensures that duplicated messages will be detected and discarded. In addition, messages delivering order will be the same as they are sent.

(b) Major TCP API functions:

i. *socket, bind, listen, connect, accept, close* – for comm. management.

ii. *read, write, readv, sendv* – for sending and receiving messages.

什么是comm. management

(c) Failure model:

i. TCP employs checksums, timeouts on individual messages, and retransmissions, in attempt to provide reliable communications.

ii. TCP also uses an overall timeout for a connection. If TCP cannot communicate with its peer for a prolonged period, it will assume that connection is broken. Hence, TCP *does not* provide reliable communications under our failure model, because it cannot guarantee delivery of messages in the faces of all possible problems. 为什么?

iii. When a TCP connection is broken, the sender process does not know if its messages have been delivered; and both the sender and receiver processes do not know whether disconnection is caused by a network failure or failure of its peer.

(3) Java UDP and TCP API

a. The package that provides TCP/IP API in Java is *java.net*.

(a) Java TCP/IP API (called networking API in java) is implemented using sockets. It can be regarded as on a little higher level than the original socket API. For instance, programmers do not have to know the socket structures and applications do not have to explicitly initialize individual fields of such structures.

(b) For both UDP and TCP, Java TCP/IP API provides a sequence of methods corresponding to those in original socket API.

b. Java UDP API

(a) A Java UDP datagram packet: consisting four fields (p.151):

array of bytes containing messages	length of message	Internet address	port number

(b) An Java UDP example (Fig.4.3, p.152, Fig.4.4, p.153): a client sends a char string to a server, which sends it back as reply. The client is invoked with syntax

java UDPClient char_string server_IP_name

(c) The UDP client code (Fig.4.3, p.152)

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
    // args give message contents and server hostname
      DatagramSocket aSocket = null;
      try {
         aSocket = new DatagramSocket();
         byte [] m = args[0].getBytes();
         InetAddress aHost = InetAddress.getByName(args[1]);
         int serverPort = 6789;
         DatagramPacket request = new
              DatagramPacket(m, args[0].length(), aHost, serverPort);
         aSocket.send(request);
         byte[] buffer = new byte[1000];
         DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
         aSocket.receive(reply);
         System.out.println("Reply: " + new String(reply.getData()));
      }catch (SocketException e)
                {System.out.println("Socket: " + e.getMessage()); }
      }catch (IOException e)
                {System.out.println("IO: " + e.getMessage());}
      }finally { if (aSocket != null) aSocket.close();}
   }
}
```

(d) Major Java UDP API classes and methods:

i. Class *DatagramSocket*: represents a datagram socket. There are three constructors for this class:

(i) *DatagramSocket()*;

(ii) *DatagramSocket(int port)*;

(iii) *DatagramSocket(int port, InetAddress laddr)*.

ii. Major methods in class *DatagramSocket*:

(i) *void close()* closes the datagram socket.

(ii) *void send(DatagramPacket p)*: sends a datagram packet through this socket.

(iii) *void receive(DatagramPacket p)*: receives a datagram packet from this socket;

(iv) *void setSoTimeout(int timeout p)*: enable/disable SO_TIMEOUT with the specified timeout in units of milliseconds. A zero value means infinite timeout (disabled). When a timeout occurs, a *java.io.InterruptedIOException* is thrown;

iii. class *DatagramPacket*: represents a datagram packet. There are four constructors for this class:

(i) *DatagramPacket(byte[] buf, int length)*: constructs a DatagramPacket for receiving packets of length *length*.

(ii) *DatagramPacket(byte[] buf, int length, InetAddress address, int port)*: constructs a DatagramPacket for sending to the specified port number on the specified host.

(iii) *DatagramPacket(byte[] buf, int offset, int length)*: constructs a DatagramPacket for receiving packets of length *length*, specifying an offset into the buffer.

这句话什么意思? specifying an offset into the buffer

(iv) *DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)*: this method combines method 2 and 3 above.

iv. Major methods in class *DatagramPacket*:

(i) *byte[] getData()*: returns the data in bytes from the packet.

(ii) *int getLength()*: returns the length of the data to be sent or length of data received.

The data and length fields are illustrated by the structure of an Java UDP packet on p.151.

(e) Other components in the class.

i. The *getByName* method in class *InetAddress* takes an IPv4 address or name as argument and return the IPv4 address as return value (a NULL argument will return the local computer's IP address).

ii. A *try* block encloses a section of code that will be monitored for exceptions that the corresponding *catch* clauses will try to catch.

iii. An *IOException* object is thrown by both the client and server programs when there is an error caused by failed or interrupted I/O operations.

iv. Similarly a *SocketException* object is thrown when there is any error in the underlying protocol of the corresponding socket. *IOException* class is a superclass of *SocketException* class. Therefore if a program does not catch a SocketException, an IOException will be raised.

(f) The UDP server code (Fig.4.4,p.153)

```
import java.net.*;
import java.io.*;
public class UDPServer{
   public static void main(String args[]){
      DatagramSocket aSocket = null;
      try{
         aSocket = new DatagramSocket(6789);
         // create socket at agreed port
         byte[] buffer = new byte[1000];
         while(true){
            DatagramPacket request = new DatagramPacket(buffer,
                  buffer.length);
            aSocket.receive(request);
            DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(),
                    request.getPort());
            aSocket.send(reply);
         }
      }catch (SocketException e)
                {System.out.println("Socket: " + e.getMessage());
      }catch (IOException e)
                {System.out.println("IO: " + e.getMessage());}
      }finally { if (aSocket != null) aSocket.close();}
  }
}
```

c. Java TCP API

   (a) Unlike Java UDP, Java TCP will not create datagram packets for sending or preparing a datagram packet buffer for receiving. Java TCP creates *DataInputStream* and *DataOutputStream* for receiving and sending stream of bytes.

   (b) An Java TCP example (Fig.4.5, p.156, Fig.4.6, p.157):

      i. The client

         (i) It is invoked through a command line like:

            java TCPClient char_string server_IP_name

         (ii) The client creates a *communication socket* by calling a constructor of the *Socket* class.

         (iii) The client opens a *DataInputStream* for reading and a *DataOutputStream* for writing.

         (iv) The client sends the first command-line parameter to the server by calling the *writeUTF* method of the DataOutputStream. It attempts

to receive reply from the server by calling the *readUTF* method of the DataInputStream.

ii. The server

(i) The server calls *ServerSocket* class to create a server listen socket instance. It then enters *listen* mode and calls the *accept* method of the listen socket.

(ii) When a new connection request arrives, the server creates a new thread by calling the application defined *Connection* class, passing a client socket obtained from return of the accept method.

(iii) The constructor of the Connection class creates two data streams using the socket passed by the main process. It then calls *this.start()* method to put the thread in the *ready* state (when a thread is created, it is initially in *born* state).

(iv) The thread just reads a line of input from the client and sends it back to the client. It then terminates by calling the *close* method of the sockets it back to the client. It then terminates by calling the *close* method of the socket.

(c) The TCP client code (Fig.4.5, p.156)

```
import java.net.*;
import java.io.*;
public class TCPClient {
   public static void main (String args[]) {
   // arguments supply message and hostname
      Socket s = null;
      try{
         int serverPort = 7896;
         s = new Socket(args[1], serverPort);
         DataInputStream in = new
                    DataInputStream(s.getInputStream());
         DataOutputStream out = new
                    DataOutputStream(s.getOutputStream());
         out.writeUTF(args[0]);  // UTF is a string encoding (cf.4.3)
         String data = in.readUTF();
         System.out.println("Received: "+ data);
      }catch (UnknownHostException e){
                 System.out.println("Socket:"+e.getMessage());
      }catch (EOFException e)
                 {System.out.println("EOF:"+e.getMessage());
      }catch (IOException e)
                 {System.out.println("readline:"+e.getMessage());}
      } finally {if (s!=null) try {
```

```
            s.close();} catch (IOException e){/*close failed */}}
   }
```

(d) The TCP server code (Fig.4.6, p.157)

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
      try{
         int serverPort = 7896; // the server port
         ServerSocket listenSocket = new ServerSocket(serverPort);
         while(true) {
            Socket clientSocket = listenSocket.accept();
            Connection c = new Connection(clientSocket);
         }
      } catch(IOException e)
             {System.out.println("Listen:"+e.getMessage());}
   }
}

class Connection extends Thread {
      DataInputStream in;
      DataOutputStream out;
      Socket clientSocket;
      public Connection (Socket aClientSocket) {
         try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream());
            out = new DataOutputStream(clientSocket.getOutputStream());
            this.start();
          } catch(IOException e)
                {System.out.println("Connection:"+e.getMessage());}
      }
      public void run(){
         try {       // an echo server
            String data = in.readUTF(); // read a line of data from the stream
            out.writeUTF(data);
         }catch (EOFException e)
                  {System.out.println("EOF:"+e.getMessage());
         }catch (IOException e)
                  {System.out.println("IO:"+e.getMessage());}
         } finally { try { clientSocket.close();}
                         catch (IOException e) {/* close failed */}}
      }
```

```
}
```

3. External data representation and marshalling

https://en.wikipedia.org/wiki/Marshalling_(computer_science)

(1) The need of external data representation

a. The need of EDR

(a) Different programming languages support different primitive data types and data structures. Even for the same data type, such as floating point numbers, two different languages on two different platform may implement them differently.

(b) The set of codes used to present characters can be different according to platforms. UNIX systems use ASCII coding and take one byte per character. UTF-8 (universal transfer format) code takes two bytes per character. For integers, there are also the so called *big-endian* (most significant byte first) verse *little-endian* issue.

(c) These different primitive data types and data structures, plus potential different set of codes, have to be communicated through messages.

b. Two approaches to handle the above diversities

(a) The values are converted to an agreed external format before transmission and converted to the local format upon receipt.

(b) The values are transmitted in the sender's format together with an indication of the format used, and the recipient converts the values if necessary.

The first approach is commonly used.

c. *External data representation*: an agreed standard for the representation of primitive data types and other data structures.

d. *Marshalling* and *unmarshalling*:

(a) *Marshalling* is the action of taking a collection of data items to be used in a communication and transform them into items in a form suitable for transmission in a message.

(b) *Unmarshalling* is the opposite action of *marshalling*. It is the action that takes a received message, extract individual data items from the message, and produce equivalent data items that can be used by the receiving process.

Clearly marshalling and unmarshalling are based on EDR.

e. EDRs and marshalling discussed here:

(a) SUN RPC XDR

(b) DCE RPC interface

(c) CORBA's common data representation

(d) JAVA RMI

(e) JAVA IDL

(2) SUN RPC

a. SUN RPC's XDR (External Data Representation) provides a uniform type conversion mechanism. SUN RPC can handle arbitrary data types/structures, regardless of different machines' byte order and structure layout conventions, by always converting them to a network standard called *External Data Representation* (XDR) before sending them over the network.

b. *Serializing and deserializing*: In SUN RPC's terms, the action of converting from a particular machine representation to XDR format is called *serializing*. The reverse action is called *deserializing*.

c. SUN RPC provides three level interfaces to applications. XDR is used at the intermediate and low levels.

d. *Marshalling* (i.e. *serializing*) in SUN RPC:

(a) Each data type that is allowed to be transmitted in an RPC call must have a corresponding function (called an *XDR filter*) defined for it. The XDR filter handles the serializing and deserializing of that data type.

(b) XDR filters are written in XDR language, which is similar to C language in syntax and semantics. The XDR compiler *rpcgen* compiles an XDR file (each ends with suffix *xdr*) into C files, which can be compiled and linked with clients and servers.

(c) XDR filters can be built in a structured fashion. SUN RPC provides 11 built-in type XDR filters: xdr_int(), xdr_u_int(), xdr_long(), ... Users can define an XDR filter based on these built-in XDR filters or any existing XDR filters.

e. Example: For an application defined structure:

```
struct simple {
   int a;
   short b;
} simple;
```

then you would call callrpc() as

callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple, &simple ...);

where xdr_simple() is written as:

```
#include <rpc/rpc.h>
```

```
xdr_simple(xdrsp, simplep)

XDR *xdrsp; struct simple *simplep; {

   if (!xdr_int(xdrsp, &simplep->a))
      return (0);
   if (!xdr_short(xdrsp, &simplep->b))
      return (0);
   return (1);
}
```

(3) DCE RPC

https://baike.baidu.com/item/外部数据表示法

a. DCE RPC applications must first define their interfaces which are written in the IDL (interface definition language) language. An IDL file ends with suffix .idl and has syntax similar to the C language.

b. An IDL interface definition defines all data types that can be used by both client and server and all remote functions/procedures available as services. Each remote function/procedure declared in the interface contains complete specification of their arguments, their orderings, and return values.

c. All servers that support the interface must implement the remote function/procedures using the same function/procedure names, data types, arguments, and ordering of arguments. All clients must call a remote function/procedure consistent with its declaration in the interface.

d. *Marshalling* in DCE RPC:

   (a) Compilation of an interface file produces a C header file, a client stub file in C, and a server stub file in C. The C header file produced contains for each remote function/procedure declared in the IDL file its corresponding C declaration.

   (b) The client stub file performs all marshalling/unmarshalling actions for the client. It should be compiled and linked with other client files (such as the file that contains the client's main function) to produce an executable client program.

   (c) Similar to the server stub file.

   The DCE IDL compiler is supported by the IDL library in producing client and server stubs. Thus IDL compiler provides transparency in EDR and EDR marshalling. 什么意思??

(4) CORBA's common data representation (CDR)

a. CORBA CDR is an EDR.

(a) It defines 15 primitive data types (such as *short* (16-bit), *long* (32-b), *unsigned short*, *unsigned long*, *float* (32-b), *double* (64-b), *char*, *boolean*, *octet* (8-b), and *any* which can represent any basic or constructed data type.

(b) It also defines 6 composite data types (Fig.4.7, p.160) that are commonly used in programming languages:

Type	Representation
sequence	length(unsigned long) followed by elements in order
string	length(unsigned long) followed by characters in order (can also have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declarations of the components
enumerated	unsigned long(the values are specified by the order declared)
union	type tag followed by the selected member

Figure 4.7 CORBA CDR for constructed types

https://blog.csdn.net/wyzxg/article/details/5349896

(c) For primitive data types, CDR supports both big-endian and little-endian orderings. The specific ordering of each value is attached in the containing message.

(d) A value of the composite type will be represented as a sequence of bytes according to the ordering as specified by Fig.4.7, and the ordering of its underline primitive data types.

b. *Marshalling* in CDR.

(1) Like in DCE RPC, CORBA supports its own IDL. Interfaces written in CORBA IDL can be compiled by its IDL compiler to produce client and server stubs that will perform marshalling and unmarshalling automatically with no need of interferences from users.

(2) For a structure defined by

```
struct Person {
   string name;
   string place;
   long year;
};
```

CORBA IDL will flatten a Person struct with value: {'Smith', 'London', 1984} to a sequence of $4 \times 7 = 28$ bytes (Fig.4.8, p.161):

index in sequence of bytes	← 4 bytes →	*notes on representation*
0-3	5	*length of string*
4-7	"Smit"	'Smith'
8-11	"h___"	
12-15	6	*length of string*
16-19	"Lond"	'London'
20-23	"on_"	
24-27	1984	*unsigned long*

Figure 4.8 CORBA CDR message

(5) Java RMI (remote method invocation, since JDK1.1)

a. Basics

(a) Java RMI is Java's implementation of RPC for Java-object-to-Java-object distributed communications.

(b) A method of a Java object can be registered through RMI and will become remotely accessible by other Java methods thereafter.

i. The syntax of invoking a remote method is identical to that of invoking a method in the same program. 在说什么？？？？？

ii. Parameter marshalling/unmarshalling is done by RMI.

(c) The *java.rmi* package contains classes that support RMI.

b. *Marshalling* in Java RMI

(a) Marshalling in RMI is based on Java's concept of serialization and *deserialization*.

i. *Serialization* refers to the activity of *flattening* an object into certain form of serial bytes so that it can be transmitted as an argument or result of RMI invocation.

ii. As the recipient of a serialized object does not know the original object, additional information has to be attached in the serialized form of an object to help the deserialization operation.

iii. For a class, the information is the name of the class and its version number. The version number can be supplied by a programmer, or can be calculated through class name hash.

iv. Any object referenced by an object will be serialized when the latter is serialized. Referenced objects are serialized as *handles*, which is a reference to an object within the serialized form.

(b) Example. For a structure

Person p = new Person("Smith", "London", 1984)

The corresponding serialized form is shown in Fig.4.9, p.163.

Serialized values				Explanation
Person	8-byte version number		h0	*class name,version number*
3	int year	java.lang.String name:	java.lang.String place:	*number,type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h_0 and h_1 are handles

Figure 4.9 Indication of Java serialized form

(c) Serialization and deserialization actions on arguments and results of remote invocations are performed by Java RMI support (which is part of the middleware).

(d) Programmers may write special methods to fulfill special serialization requirements if desired.

i. The *writeObject* method in an instance of the *ObjectOutputStream* class can be invoked to serialize an object (such as the *Person* structure) that is passed to it.

ii. Similarly, the *readObject* method in an instance of the *ObjectInputStream* class can be invoked to deserialize (restore) an object.

(6) XML (Extensible markup language)

a. XML vs HTML

(a) HTML: hypertext markup language. It is a tagged language – elements in it are tagged by an open tag and a corresponding end tag. The elements in HTML contain infromation as how a web browser should display the text contents.

(b) XML is also a tagged language. However, XML elements contain information both for displaying the data and describing the logical structure of the data.

(c) Both XML and HTML belongs to class of "markup" languages that are defined by applying SGML (standard Generalized Markup Language). Each application of SGML is a markup language.

b. Main usage of XML:

取代

   (a) XML supercedes HTML as a web markup language. It can be used for web services to define their interfaces and other properties.
   (b) XML can be used for clients to communicate with web servers.
   (c) XML can be used for information archival and retrieval. Portable across platforms.
   (d) XML can be used for interface and services specifications of any applications.

c. Extensible XML

   (a) XML can define its own tags. HTML cannot.
   (b) SOAP (Simple Object Access Protocol) is a protocol for foundation layer of a web services protocol stack. XML is the standard language for SOAP. Tags 有例子吗? for SOAP are published for use by web clients and servers.

d. Self-describing languages: those that can describe their own features such the meaning of a field.

   (a) CORBA CDR is an example that is not self-describing. Prior knowledge or convention is necessary for the meaning of each field of Fig.4.8 example.
   (b) XML is a self-describing language – it's intended to be used in many applications. Tags and XML namespace make self-describing possible. Tags can also allow selective processing. 这是什么?

e. XML documents are readable by both humans and computer software.

f. XML elements and attributes: Fig.4.10, p.165 shows the XML definition for the *Person* structure used in CORBA CDR and Java RMI.

```
<person id="123456789"/>
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!- -a comment- ->
```

g. Parsing and well-formedness of XML documents

   (a) Well-formedness: the tags are paired and properly nested: they form a string in a context-free language.
   (b) Special characters or data.
   Example: the "<" character can be expressed as "&lt".
   (c) CDATA: special data section that should not be parsed by XML processor.
   https://en.wikipedia.org/wiki/CDATA
   (d) For a string: *King's Cross*:
   method 1: *<place>King&apos Cross</place>*
   method 2: *<place><![CDATA[King's Cross]]></place>*

h. XML namespaces: Extending the URL to within XML

(a) The *xmlns* attribute, with value of URL that refers to the file containing the name space definitions:
xmlns:pers = "http://www.cdk5.net/person"

(b) Then the *person* structure can be changed to:

<person id="123456789" xmlns:pers="http://www.cdk5.net/person">
<name>Smith</name>
<place>London</place>
<year>1984</year>
</person>

The name *pers* after *xmlns* can be used as prefix to refer to the elements in particular namespace.

i. XML schemas

(a) XML schemas like data types. An XML schema defines the names of the names and attributes of a XML definition, together with the nested structure of the elements within.

字写错了...

(b) A single XML schema can be shared by many differebt documents. XML documents that are defined to conform to a particular schema can be verified.

(c) Example: Fig.4.12: An XML schema for the *Person* structure.

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions  >
    <xsd:element name= "person" type ="personType" />
        <xsd:complexType name="personType">
            <xsd:sequence>
               <xsd:element name = "name"  type="xs:string"/>
               <xsd:element name = "place"  type="xs:string"/>
               <xsd:element name = "year"  type="xs:positiveInteger"/>
              </xsd:sequence>
        <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

Fig.4.12 An XML schema for the Person structure

j. API for processing XML: available software written in Java, Python, and other popular languages. As different object languages different class/object differently, users should choose the software according to their needs.

(7) Java IDL (since JDK1.2)

a. Java IDL is intended for Java applications and applets to communicate with objects written in any languages that support CORBA. RMI on the other hand is restricted to Java-to-Java communications.

b. Like DCE IDL, Java IDL is an interface definition language. Java IDL supports a compiler (idlj on UNIX, idlj.exe on Windows) to compile Java IDL files into Java files. The use of a standard interface definition language provides transparency to marshalling/unmarshalling operations. 什么意思?

为何要compile? 黑人问号????

c. *Marshalling* in Java IDL applications. Similar to DCE IDL, the first step of writing a Java IDL application is developing a Java IDL interface. For an interface named *Xyz* and an IDL file named *Xyz.idl*, the IDL compiler idlj.exe will compile it and produce the following files:

(a) A *server skeleton* file _XyzImplBase.java , which provides basic CORBA functionality for the server. It implements the interface. This is the server stub file.

(b) A *client stub* file _XyzStub.java, which provides basic CORBA functionality for the client.

(c) A file Xyz.java which is the Java version of the IDL interface.

(d) A file XyzHelper.java which provides auxiliary functionality required to cast CORBA object references to their proper types.

(e) XyzHolder.java which holds a public instance member of type Xyz/. It provides operations for *out* and *inout* arguments, which CORBA has but which do not map easily to Java's semantics.

A sample Java IDL file *Hello.idl*:

```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

4. Remote object references

(1) In RMI, there is a need of unique way to identify a particular remote object that implements the invoked method. 什么意思?

(2) Remote object reference is an identifier that is unique in the whole distributed/cloud environment and refers to unique remote object.

(3) Such a reference hence has to include the IP number of the computing hosting the remote object, a port number that refers a particular service. The use of IP number also allows relocation of remote objects (at the expense of location transparency). 这是什么意思?

(4) In addition, the following should be part of a remote object reference as well (Fig.4.13, p.169):

看不懂～

a. Time: the time when the remote object created;

b. Local object number: the remote object is a local object on the hosting computer;

c. interface of remote object.

* This could be interface name
* The information is useful to any process that receives a remote object reference as an argument, or as the result of a remote invocation, as the process needs to know the method offered by the remote object (Cf.Sect.5.4.2).

5. Group communications

(1) Multicasting: sending or receiving messages to or from a targeted group of processes. Major issues in group communications:

a. Group membership managements.

b. Message delivery mechanisms.

c. Reliability and validity issues.

d. APIs.

(2) Example applications of multicast protocols. Group communication is another important way of message exchanges in DSs.

a. *Fault tolerance based on replicated services*: A service may be replicated and provided by multiple servers in a DS. A client may send a request (such as modify a data file) to all servers. Reply from any of them will satisfy the client. As long as one of the servers is still available, the service will be available to all clients.

b. *Better performance through replicated data*: for a set of replicated data, multicast is used to send a update request to all processes that manage a copy of the replicated data.

c. *Finding the discovery servers in spontaneous networking*: a server of a RPC service or a client may locate available discovery services in order to register its interface or look up interfaces of certain services in a spontaneous networking environment.

这里在做什么?

d. *Propagation of event notifications*: certain events such as unavailability or re-availability of a comm. link can be sent to a group of routers through multicasting.

(3) IP multicast

a. IP multicast is based on the support of IP.

   (a) IP normally supports *unicast* communications (from one computer to another computer). However, in IPv4 an application can specify a *class D* IPv4 address (an address with prefix 1110) in an IP packet to indicate that the packet should be delivered to a group of computers.

   (b) Each class D IPv4 address can be used to represent a *multicast group*.

b. Multicast API.

   (a) The API to programmers is UDP. A process can perform the following steps to let the hosting computer become a member of a multicast group (Note: multicast membership is with respect to computers, not individual processes on a computer):

      i. Obtain a UDP *socket* descriptor and *bind* the descriptor to a *port number*.

      ii. Uses *setsockopt* function to set the *IP_ADD_MEMBERSHIP* option with the value of a class D IPv4 address.

   If successful, after above operations the hosting computer becomes a member of the multicast group represented by the class D IPv4 address. The computer will be able to send multicast packets to and receive multicast packets from other members in the group.

   (b) A process that initiated the membership of a multicast group can ask the hosting computer to leave a multicast group by calling *setsockopt* function to set the *IP_DROP_MEMBERSHIP* option.

   (c) Multiple processes on a computer can initiate membership joining operations. The host will remain a member of a specific multicast group as long as at least one process who initiated a membership joining operation before hasn't initiated the corresponding membership dropping operation.

   (d) Each multicast packet sent by a multicast process will also be copied to the hosting computer itself as an multicast packet input, although this can be explicitly disabled.

   (e) A process can send a multicast packet to a multicast group even if it hasn't initiated an appropriate membership joining operation. However, such a process is not able to receive multicast packets to that multicast group.

c. Multicast on LAN and WAN. Members of a multicast group can be anywhere on a WAN.

   (a) Management of multicast groups is done through *multicast routing protocols*.

   (b) When a computer joins a multicast group, the computer sends a special control message (a IGMP message) to some attached multicast routers, which in turn will exchange this info with other multicast routers about the change.

(c) Similar actions will be taken when a computer leaves a multicast group.

(d) Informally, the multicast routers for a specific multicast group form a tree. This tree can grow when new members join and be pruned when some member leaves.

(e) On the datalink level (such as Ethernet), when a computer joins a multicast group, the datalink interface of that computer will be instructed to receive datalink packets destined to a special datalink address. For Ethernet, that address is 01:00:5e:00:01:01, which is the Ethernet multicast address. Ethernet packets with this address will be picked up by the interface. The IP layer will then examine the multicast IP address inside.

这段没看懂?

d. Failure model of IP multicasting.

(a) As IP multicasting uses UDP and is based on IP, it is *unreliable*.

i. Any multicast packet may not be guaranteed to be delivered to all members of the multicast group.

ii. Multicast packets may be delivered not in FOFI order.

(b) IP multicast is a basic multicast. Reliable multicast can be implemented by enhancing a basic multicast (Ch.12).

(4) Consequences of IP multicast failure model on different applications.

a. *Fault tolerance based on replicated services* and *Better performance through replicated data*: The applicability of IP multicast depends on the nature of the replicated data and replicated services. Newsgroups data files and services do not have to be consistent all the time. A user may succeed in posting an article to one news server while failing to post the same article to another news server. But that is acceptable in most cases. On the other hand, a banking account file and service have to be consistent.

b. *Finding the discovery servers in spontaneous networking*: IP multicast is still usable here.

c. *Propagation of event notifications*: This is again application dependent. Failing to notify certain routers about the current condition a comm. link is acceptable, while failing to notify a consensus to some processes in a distributed consensus algorithm (Ch.11) is not acceptable.

什么意思?

(5) Example application of Java IP multicast API (Fig. 4.20, p.166)

a. After compilation, the program can be run by the command:

java MulticastPeer char_string multicast_IP_number

where *char_string* is a char string to be multicast to group members. Note: Java program takes the first command-line parameter as *args[0]*, the second command-line parmeter as *args[1]*, and so on. Different from C/C++.

b. The constructor of the *MulticastSocket* class creates a multicast socket (a UDP socket) and binds the socket to port 6789.

c. The program joins the IPv4 multicast group with group IPv4 number 228.5.6.7 through the *joinGroup* method of object *s* of class *MulticastSocket*.

d. The After joining the group, it multicasts a message (a char string, which is the first command-line argument) and then iterates in a *for* loop three times to receive three multicast messages.

e The process then leaves the group by invoking the *leaveGroup* method of object *s*.

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
   public static void main(String args[]){
   // args give message contents and destination multicast
   //               group (e.g. "228.5.6.7")
   MulticastSocket s = null;
   try {
      InetAddress group = InetAddress.getByName(args[1]);
      s = new MulticastSocket(6789);
      s.joinGroup(group);
      byte [] m = args[0].getBytes();
      DatagramPacket messageOut = new DatagramPacket(m,m.length,group,6789);
      s.send(messageOut);
      // get messages from others in group
      byte[] buffer = new byte[1000];
      for(int i=0; i< 3;i++) {
         DatagramPacket messageIn = new DatagramPacket(buffer,buffer.length);
         s.receive(messageIn);
         System.out.println("Received:"+ new String(messageIn.getData()));
      }
      s.leaveGroup(group);
   }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
   }catch (IOException e){System.out.println("IO: " + e.getMessage());}
   }finally { if (s!=null) s.close();}
}
```

6. Network virtualization: overlay networks

(1) The need of network virtualization

a. Internet comm. protocols have to support all different types of applications

b. Different applications desire different preferred support of comm. protocols

c. It's impossible to alter comm. protocols to support all these different protocols

(2) Network virtualization: contruction of various *virtual networks* that are supported by the underlying existing networks (such as Internet).

a. Each virtual network is designed with the goal of supporting a particular distributed/cloud application or classes of such applications. Examples:

* Multimedia streaming virtual network that supports applications such as BBC iPlayer, Boxee TV (boxee.tv), or Hulu (hulu.com).

b. Each virtual network has its own addressing schemes, protocols, and routing schemes. They are tailored to facilitate the targeted applications

Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].

Figure 29: Type of overlay (Part 1, Fig.4.15, p.176)

(3) Overlay networks (section 4.5.1)

a. An overlay network is a virtual network consisting of nodes and virtual links. Such a network sits on top of an underlying network (such as IP network) and offers features that are not provided by the underlying network (Fig.4.15, p.176):

* A service that is tailored towards the need of class of applications or a particular higher-level service. Examples: multimedia content distribution, stricter authentication.
* More efficient operation in a given networked environment, such as routing in an ad hoc mobile network

Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

Figure 30: Type of overlay (Part 2, Fig.4.15, p.176)

- Other additional features such as more efficient multicast or secure communication.

b. Main advantages of overlay networks
   (a) Providing additional services without need of changing underlying network protocols, which is difficult to change due to the need of being generic and standardization.
   (b) Facilitating experiments and deployment of new network services and applications and customization of services to particular classes of applications.
   (c) Providing a more open and extensible network architeture. Multiple overlays (both vertical and horizontal) can be defined and can coexist.

c. Problems of overlay networks
   (a) Extra level of indirection and added complexity of network services.
   (b) Extra overhead

d. Network layers vs overlay networks
   (a) Existing network layers are generic and covers the core of standard parts of networks.
   (b) Overlays are on top of existing network layers (outside the standard network layer architecture. Hence they can exploit the resultant degree of freedom.
   (c) Overlay designers and developers have freedom of proposing core parts of a network as desired.

(d) Example: Distributed hash tables introduce a new style of addressing based on a keyspace. It results in a network topology in such a way that each node in such a network either owns the key or connects through a link to a node that is closer to the owner. This often leads to ring topology suitable and efficient for certain applications.

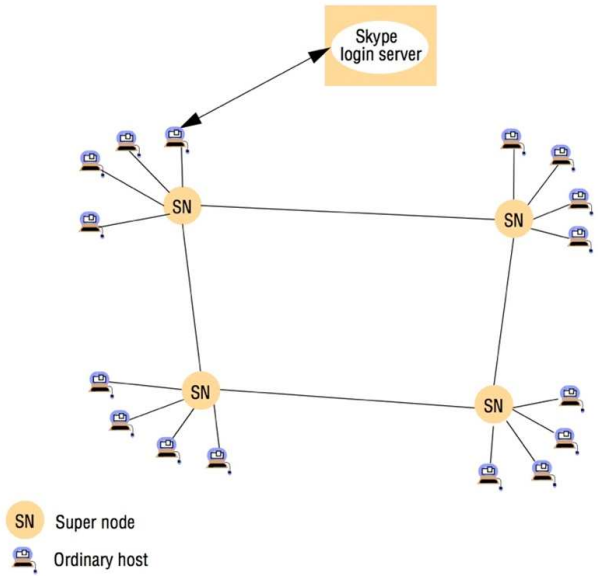


Figure 31: Skype overlay architecture (Fig.4.16, p.177)

(4) Skype: example of an overlay network

a. A peer-to-peer application offers VoIP (voice over IP) service, 2003. Other services:

(a) Instant messaging

(b) Video conferencing

(c) Interfacing to standard telephony service (through features of SkypeIn and SkypeOut)

370 million users at end of 2009.

b. Architecture and operation: Fig.4.16, p.177

(a) Peer-to-peer application

(b) SN (super node). Super nodes are ordinary hosts, selected to carry out enhanced activities/duties. Selection is based on multiple parameters such as bandwidth availability, reachability, and availability. Other nodes are ordinary hosts (pure clients).

(c) User connection: user login is authenticated through a well-known login server, in cooperation with a seleted super node. Each client maintains a cache of super node identifiers (IP address and port numbers) and the cache is updated periodically.

(d) Search for users: this is another extremely critical function. This is done through cooperation of super nodes. Normally on average eight super nodes are involved to complete a search.

(e) Voice connection: after a user is discovered, TCP is used to signalling call requests and termination. Voice signals are carried using either TCP or UDP. UDP is preferred (due to flexibility and voice communication nature).

7. Message passing example: MPI

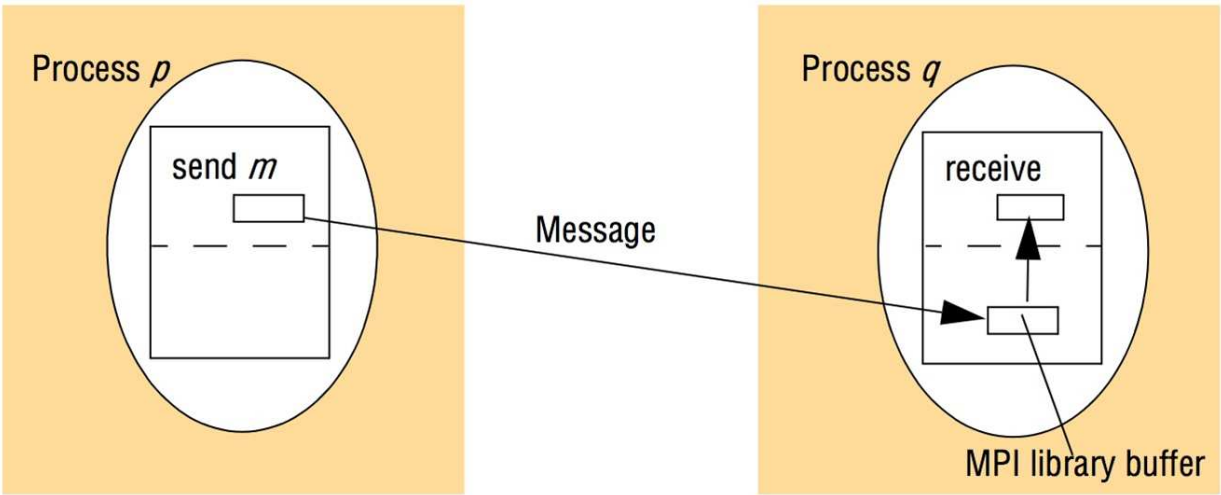


Figure 32: An overview of point-to-point communications in MPI (Fig.4.17, p.179)

(1) MPI (Message Passing Interface, 1994): an effort to provide uniform, efficient message passing mechanism for high performance computing, independent of proprietary implementations of various high performance programming software/language (such as C/C++, Fortran).

(2) Overview of MPI message passing: Fig.4.17, p.179.

   a. A send message m is sent and copied to the MPI library buffer at receiver's end.

   b. The receiver can execute a corresponding receive operation to copy the message.

(3) Selected send operations in MPI: Fig.4.18, p.180

   a. *Generic*: *MPI_Send*

b. *Synchronous*: *MPI_Ssend*

c. *Buffered*: *MPI_Bsend*

d. *Ready*: *MPI_Rsend*

Send operations	*Blocking*	*Non-blocking*
Generic	*MPI_Send*: the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend*: the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via *MPI_Wait* or *MPI_Test.*
Synchronous	*MPI_Ssend*: the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	*MPI_Issend*: as with *MPI_Isend*, but with *MPI_Wait* and *MPI_Test* indicating whether the message has been delivered at the receive end.
Buffered	*MPI_Bsend*: the sender explicitly allocates an MPI buffer library (using a separate *MPI_Buffer_attach* call) and the call returns when the data is successfully copied into this buffer.	*MPI_Ibsend*: as with *MPI_Isend* but with *MPI_Wait* and *MPI_Test* indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
Ready	*MPI_Rsend*: the call returns when the sender's application buffer can be reused (as with *MPI_Send*), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	*MPI_Irsend*: the effect is as with *MPI_Isend*, but as with *MPI_Rsend*, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),

Figure 33: Selected send operations in MPI (Fig.4.18, p.180)

Lecture 6
From RPC to RMI (Remote Method Invocation)

(June-23,27-2020. Chapter 5)

Lecture Outline

```
1. Introduction
2. Request-reply protocols
3. Remote procedure calls
4. Distributed objects and RMI (remote method invocation)
```

1. Introduction

   (1) Client-server communications are typical in distributed/cloud computing

      a. Client-server communications usually employ the *request-reply* protocol.

      b. Request-reply communications can be synchronous or asynchronous. But most of time it is synchronous – a client normally suspends its operations while waiting for reply from server.

      c. Request-reply communications are independent of underlying transport layer protocols. Both UDP and TCP can be used.

   (2) From RPC to RMI

      a. RPC

      b. Object oriented paradigm

      c. Distributed objects and RMI

2. Request-reply protocols

   (1) Typical to support the roles and message exchanges in typical client-server interactions

      a. Illustration: Fig.5.2, p.187

         (a) The presented protocol is generic. However most RPC and RMI implementations support similar request-reply protocols.

         (b) Three comm. primitives (Fig.5.3, p.188):

            i. *doOperation* method: performs the action of invoking a remote method (i.e. sending a request).

            ii. *getRequest* method: is used by a remote server process to receive service requests.

            iii. *sendReply* method: sends the reply message to the client.

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments);

`sends a request message to the remote object and returns the reply. The arguments specify the remote object, the method to be invoked, and the arguments of that method.`

public byte[] getRequest ();

`acquires a client request via the server port.`

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

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

https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/network/sctp_intro.html

b. Can be both synchonous and asynchronous

c. Transport layer can be either UDP or TCP (or SCTP)

d. UDP implemented RR protocol: less comm. overhead

   (a) No explicit ack is needed, as replies can serve as acks.

   (b) TCP connection and termination involves seven message exchanges.

   (c) There is no need of flow control (available in TCP) as most remote invocations are short in duration.

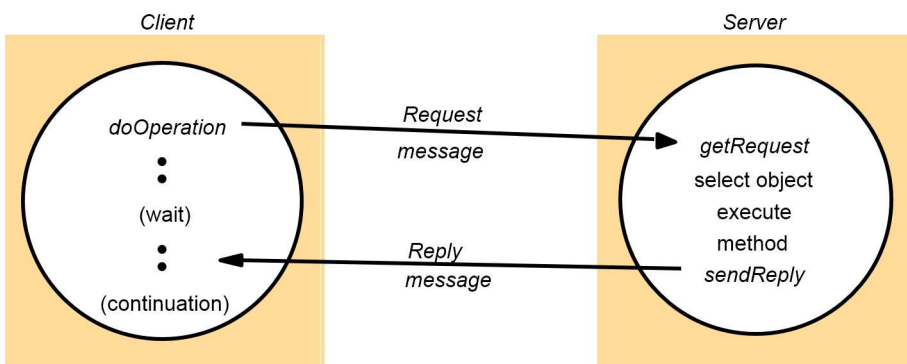


Figure 34: Request-reply communications (Fig.5.2, p.187)

(2) Structure of request-reply messages: Fig.5.4, p.188

messageType	*int (0=Request, 1=Reply)*
requestID	*int*
remoteReference	*RemoteRef*
operationID	*int or Operation*
arguments	*//array of bytes*

Figure 5.4 Request-reply message structure

a. *requestId*: (Note: the book showed a *requestId* in Fig.5.4. It then said that a *message identifier* contains a *requestId*) it consists of two parts: a *requestId* and an identifier of the sender process (its IP address and port number). *requestId* is taken from an increasing sequences of integers. A retransmitted request will carry the same requestId as in the original request.

b. *remoteReference*: this is remote object reference. It may consist of remote server's IP, port number, together with remote object ID.

c. Possible errors. For each request, the following events might happen:

   (a) Request message is not received by the server:

      i. The request message is lost during comm.

      ii. The server process is not running.

      iii. The computer hosting the server process is not running.

   (b) The reply message does not reach the client

      i. The reply message is lost during comm.

   (c) The server process crashes and restarts

   (d) The client process crashes and restarts

   (e) The request message is delayed

   (f) The request message is corrupted

   (g) The reply message is corrupted

   * If (a), or (c), or (f) occurs, the request has not been served.
   * If (b) or (g) occurs, the request has been served.
   * If (c) or (d), the request may or may not have been served.

   The critical observation is that, from a client's perspective, it has no way to know what exactly has happened that has caused it not to be able to receive the reply.

d. *Idempotent operations* and semantics of request-reply protocols

   (a) An operation is *idempotent* if the effect of performing the operation repeatedly is same as that of performing it exactly once. Example idempotent operations:

      i. Access a WEB page;

      ii. The operation of adding an element to a set. The property of sets guarantee idempotency.

   (b) *At-least-once, at-most-once, maybe*, and *exactly-once* semantics of request-reply protocol

i. *At-lease-once*: a request is guaranteed to be served by a server *at least once*. Example application: client may request a remote email server to send a email message to a specific user to notify approval of the user's application. This semantics is suitable for idempotent operations.

ii. *At-most-once*: a request is guaranteed to be served by a server at most once. But it may not be served at all. Not very common.

iii. *Maybe*: a request may or may not be served. Not very common. It arises when no any fault tolerance mechanism is taken.

iv. *Exactly-once*: a request is guaranteed to be served by a server *exactly once*. Too many applications require this semantics.

(c) Comments:

i. Exactly-once semantics is from local procedure calls. Combination of at-least-once and at-most-once semantics provides exactly-once semantics.

why??? ii. Most IPC implementations do not support exactly-once semantics due to its overhead. It's left to applications to apply additional fault tolerance mechanisms to achieve exactly-once semantics.

This topic will be re-examined in Section 6 "Distributed objects and RMI" in this lecture.

e. Failure model of the request-reply protocol.

(a) A request-reply protocol based on UDP that does not add any mechanism to enhance reliability by itself suffers from two well-know failures:

i. Omission failures;

ii. Non-delivery of messages (violation of *validity* in our failure model).

Such a protocol can only support *at-most-once* or *maybe* semantics. It cannot provide *at-least-once* or *exactly-once* semantics.

不懂 (b) *Timeout* and *retransmission*.

i. Timeout and retransmission are essential for a request-reply protocol to provide at-least-once or exactly-once semantics (although for the latter, additional mechanism is needed). The *doOperation* uses timeout to detect loss of requests or replies.

ii. For at-most-once and maybe semantics, timeout alone is sufficient.

(c) *Detecting and discarding duplicate request messages*.

i. For non-idempotent operations, a server in a client-server comm. must be able to detect a duplicate request message and discard it. Otherwise inconsistency may arise.

??? ii. A server should also re-transmit the reply when it detects a duplicate request message. This will keep the client from performing pair of time-out/retransmit operations repeatedly.

(d) *History*.

i. In order for a server to able to detect duplicate requests and retransmit replies, it must maintain, a record of executions of requests. Such a record is called *history*.

ii. When a request from a specific client arrives, the server will check the newly arrived request against the history. If that request is new, it will be served. If that request is a duplicate, a stored copy of reply will be fetched from the history and retransmitted to the client.

??? iii. If the client-server comm is synchronous, the server only has to keep one copy of completed request for each client. A new request from a client signals that that client has received reply to its previous request. Hence the server can purge the history to delete reply for that client.

(3) RPC exchange protocols (Fig.5.5, p.191)

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

Figure 5.5 RPC exchange protocols

a. The three RPC exchange protocols are:

(a) The request (R) protocol

(b) The request/reply (RR) protocol

(c) The request/reply/acknowledge (RRA) protocol

b. The three RPC exchange protocols were special request-reply protocols originally designed for RPC. But its semantics also applies to request-reply protocols in general.

c. **R** protocol. Used when no value has to be returned in response to a request and the client does not require/care confirmation from the server. It implements the *maybe* and *at-most-once* semantics. This protocol is the least reliable and least expensive.

d. **RR** protocol. Used in most RPC systems. This protocols implements the *exactly-once* semantics. why? what is it?

(a) In order to filter out duplicate requests a server must keep three separate histories (also called *pools*) containing information of all outstanding requests:

i. *Completion pool*: which contains those requests, together with their replies, that have been *recently* completed;

ii. *Execution pool*: which contains those requests that are currently being served;

iii. *Waiting pool*: which contains those requests that were received but are waiting their turn to be served.

(b) A newly arrived RPC request is checked against the three pools as follows:

i. If there is a request in the waiting pool or execution pool with the same *messageId* as the newly arrived request, the newly arrived request message is simply discarded. This is a duplicate request whose original request has been received but has not been completed.

ii. If the messageId of the newly received request matches that of a request in the completion pool, this newly received request is also a duplicate, hence discarded. However, the reply for that request in the completion pool will be fetched and retransmitted to the client. This is a duplicate request whose original request has been completed.

iii. If the newly arrived request came from some process with some *SourceAddress* $= x$ such that the *messageId* is not in the completion pool, then the newly arrived request is either put into the waiting pool or given the CPU for execution. In addition, entry in the completion pool with the *SourceAddress* equal to x is purged, because the client must have received the reply to its previous request.

???

e. **RRA** protocol.

(a) Problems with RR protocols: the waiting pool may grow very large, consuming system resources (a completed request is kept in the completion pool until the initiating client makes a new request. However, a client may never send new requests after initial interactions). RRA is a protocol that can remedy this problem.

why??

(b) The RRA protocol:

- An acknowledgment message is required from the client to which a reply message has been sent (by the server). The ack message contains messageId of the original request message.
- After receiving the ack message the server can free all memory associated with that particular client's request.

(c) Alternatively, the server can adopt the strategy of holding completed requests in its histories for a certain period τ of time and deletes them after τ time. Deciding the value of τ is critical to the correctness and performance of a

request-reply protocol. Too short: may not ensure at-most-once semantics. Too long: large waiting pool.

(4) Request-reply protocols implementation using TCP streams

a. Problems of datagram communications: (besides unreliability) the buffers sizes at the sender and receiver are difficult to decide.

   (a) Certain applications require large buffers that are difficult to support by clients or servers.

   (b) If a client supports small buffers, an application request that is significantly larger than size of the buffers will result in multiple datagram packets. An UDP based request-reply protocol has to deal with the issue of disassembling and reassembling these scrambled packets.

b. Benefits of TCP based request-reply protocols

   (a) TCP is a stream protocol, hence buffer sizes no longer present a problem.

   (b) TCP is reliable. Therefore retransmission is not an issue here.

   (c) TCP significantly simplifies the implementation of request-reply protocols. If a client intends to communicate with a server for a relatively long time period, the overhead of TCP connection establishment and termination is negligible.

c. The request-reply protocol in Java RMI is TCP based. A server *extends* the *UnicastRemoteObject* class, which provides support for point-to-point active object references (invocations, parameters, and results) using TCP stream.

(5) Example: the HTTP (hypertext transfer protocol) protocol

a. A Web server manages different resources in different ways:

   (a) as data such as texts and images of an HTML page.

   (b) as program such *cgi* scripts/programs and *servlets*, which run on the web server.

b. HTTP is a request-reply protocol. It specifies formats of messages involved in a request-reply exchange, the supported methods on Web servers, arguments and results and rules for representing (i.e. marshalling) them in messages.

   (a) HTTP supports a fixed set of methods that are applicable to all resources managed by a Web server. This is different from other protocols where different resources (objects) have different methods.

   (b) HTTP supports content negotiations (what data format/representation a client can accept) and password based authentication mechanism.

   (c) HTTP is based on TCP.

i. HTTP 1.0 would open/close a TCP session for every pair of request/reply exchange between a client and server (high overhead).

ii. HTTP 1.1 (and newer) uses *persistent connections* which allows a connection to be up through a sequence of request/reply exchanges. A persistent connection can be explicitly closed by either a client or server.

c. *Marshalling* in HTTP

(a) Requests and replies are marshalled into ASCII text strings unless specified otherwise.

(b) Requests and replies can also be communicated as byte sequences and compressed.

??https://en.wikipedia.org/wiki/MIME

(c) Multimedia data are sent and received as MIME formed structures, which has a prefix about its MIME data type and subtype (such as *text/html, image/gif*). This information allows the receiving process to use appropriate applications to handle the data.

d. HTTP *messages*

i. Request messages (Fig.5.6, p.193)

method	*URL*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmul.ac.uk/index.html	HTTP/1.1		

Figure 5.6 HTTP *request* message

ii. Reply messages (Fig.5.7, p.195)

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		

Figure 5.7 HTTP *reply* message

iii. *header* field: contains request modifiers and client information: for reload, a condition (date) of retrieve a web page; for password based authentication, a encrypted password or other credentials.

iv. *message body* field: itself may also be structured and contain its own headers so that the client may invoke different applications to handle different types of data in it.

e. HTTP *methods*

(a) *GET*: requests all resources specified by the given URL. An HTML page located by an URL may refer to programs (CGI or Servlet), which will be invoked and results of the invocation will be returned.

(b) *HEAD*: Similar to GET. But it will not retrieve message body. It can be used by a proxy server or cache server to check freshness of a resource.

(c) *POST*: Places a document to a Web server. The URL that will be used to process the document is specified. Example applications that uses this method include:

i. Posting a message to a discussion group or message board.

ii. Submitting an online application, an online electronic payment, or an online order request.

A POST request normally will invoke a CGI script or Servlet on the Web server.

(d) *PUT*: specifies that the data supplied in the request should be stored with the given URL as identifier.

(e) *DELETE*: requests the remote Web server to delete the resource identified by the specified URL. Potentially insecure and disabled on many Web server.

(f) *OPTIONS*: mainly used by a server to declare to a client the list of methods that are allowed to be applied to the resources with the specified URL.

(g) *TRACE*: the server will echo the request sent by a client. For debugging purpose.

3. Remote procedure calls

(1) Motivations

a. A process that wants to access to a service hosted on a remote computer should be able to make a service request that specifies the *service name* and possibly some needed parameters, just like making a local procedure call (LPC). The calling process may not and does not want to know the physical location of the service.

b. Advantages of RPC.

The concept of RPC and its implementation are important steps toward *access transparency*. RPC relieves applications from the details of locations of available services, communications, transmission errors, and various failures.

(2) Syntax and semantics of local procedure calls

a. Before a procedure is actually invoked, a new environment is created, which includes the actual parameters passed, local variables, and variables in the program enclosing the called procedure.

b. The caller is suspended upon the call.

c. When the called procedure completes, it returns the results to the calling procedure and terminates.

d. The calling procedure resumes its execution.

(3) Syntax and semantics of RPCs:

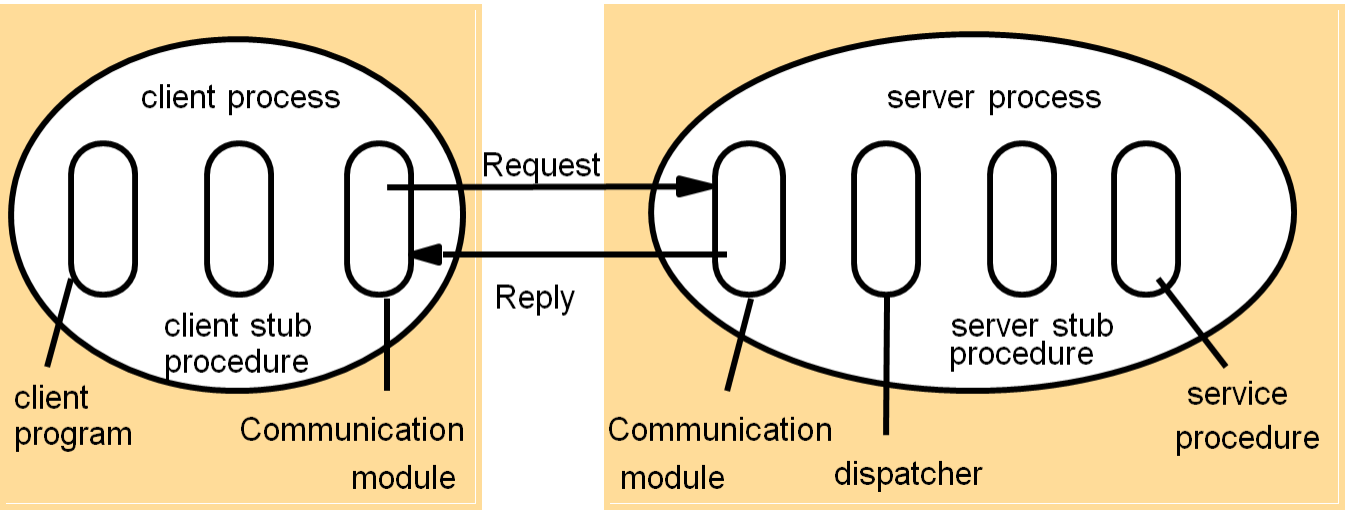


Figure 35: Role of client and server stub procedures in RPC (Fig.5.10, p.201)

a. The local process makes a RPC, providing the name of a remote procedure, and possibly a list of parameters.

b. The RPC will be caught by a communication daemon, which calls the *client stub* procedure. The client stub prepares a *request message* and sends it to the remote host.

c. A corresponding server stub procedure at the remote host accepts the request message, and prepares a local procedure call that implements the requested service. It then initiates the local procedure. The client and server stubs are illustrated in Fig.5.8, p.198.

d. The LPC at the remote host completes, and the results are given to the server stub at the remote host.

e. The server stub at the remote host prepares a *reply message* and sends it back to the client stub at the local host.

f. Upon receiving the reply message, the client stub will unpack the message and send the results back to the local process.

(4) Major issues of RPCs

a. *Uniform call semantics* – transparency problem. Should communication failures and long delays be hidden?

b. *Type checking* – strong type checking available in local procedure call implementation should also be used for RPCs.这是什么?

c. *Full parameter functionality* – all primitive data types should be allowed. Supports to structured and application defined data types must also be considered.

d. *Concurrency control and exception handling* – not fundamental, but should be considered and implemented if possible.

e. *Binding* – a remote procedure name called by a client will be eventually associated with a local procedure at a remote host. When will this association occur?

(a) *Static binding*: bind it at compilation time. when a client program is compiled, every remote procedure name is bound for each RPC.

i. Advantages: simple and easy to implement.

ii. Problems: programs need to be recompiled when the service locations change.

(b) *Dynamic binding*: delay the binding until run-time

i. Advantages: location transparency is maintained because no recompilation is necessary. 什么是location transparency???

ii. Problems: each site must maintain a table that maps every remote procedure name to a list of possible servers. The table must consulted for each RPC at run-time and updated frequently to reflect the real configuration.

(c) Binding based on name servers

i. A server name of a remote service must be registered

ii. Remote procedure name is resolved through name servers 什么意思?

iii. Flexible. Could be dynamic or static.

(5) Interfaces in distributed computing

a. Interface provides abstraction and supports transparency

b. Interface in RPC: an interface must be defined for a RPC system. Each remote procedure must be assigned a unique procedure id and have an entry in the interface which must contain the following information:

(a) The types of parameters and results, for type checking. This is just like in LPC.

(b) Different languages have different notions of data types. The interface must contain adequate information so that stubs can perform appropriate type conversions if necessary (real in Pascal, float and double in C).

(c) CORBA IDL interface: Fig. 5.8, p.197

```
// In file Person.idl
struct Person {
      string name;
      string place;
      long year;
} ;
interface PersonList {
      readonly attribute string listname;
      void addPerson(in Person p) ;
      void getPerson(in string name, out Person p);
      long number();
};
```

Fig. 5.8 CORBA IDL example

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

Figure 36: Call semantic s(Fig.5.9, p.198)

(6) RPC call semantics: Fig.5.9, p.198

a. Retry request semantics: whether to retransmit the request message (and how many tries?)

b. Duplicate filtering: whether to filter out duplicates (in the case of retry semantics)

c. Retransmission of results: whether to keep track of reply results

(7) Transparency

a. RPC is a high level client-server IPC facility. As such it faces all possible errors a request-reply protocol has to deal with (cf. RPC protocols in Section 4 of this lecture)

b. Many factors can affect transparency support in RPC:

(a) Comm. failures, which cause omission failures (loss of request and result messages).

(b) Crash failures of server hosts.

(c) Delay of message delivery.

Supporting high degree of transparency is challenging and is impossible in most cases.

(8) Concurrency

a. Blocking calls and their effects

(a) By definition, a client is blocked after a RPC request until a reply message is received.

(b) Problems: degree of parallelism is reduced and hence system performance may degrade. A process may be both a client and server. Blocking a client may prevent it from accepting incoming service requests.

b. *Threads*: a collection of processes created by a single parent that share the same address space as the parent. Benefits of using threads:

(a) Unlike conventional processes, threads have little context switching overhead. 什么意思并且， 为什么?

(b) A client may create a collection of threads. Some them are responsible to accept incoming requests and others are responsible for making outgoing requests to other servers.

(c) Similarly, a server may also create multiple threads to serve requests concurrently.

(9) Implementation of RPC: Fig. 5.10, p.201

a. External data representation

b. Methods of passing parameters and results in RPCs

a. *Input* parameters are passed by value

b. *Output* parameters are passed by reference

c. Marshalling. The client and server stub processes will perform marshalling/unmarshalling for applications.

d. Interface declaration

f. Stubs hide the remote nature of RPC. A stub has the following responsibilities:

(a) For each RPC creates an appropriate message that contains the arguments, message id, client id, message type, request id, and procedure id. A typical message is of the form. Note: each retransmitted RPC request message will carry the same messageId but different requestId fields.

```
type message record
        MessageType: {request, reply};
        MessageId: integer; (* one per message request*)
        RequestId: integer; (* one per RPC request *)
        SourceAddress: port; (* network address of client process *)
        ProcedureId: integer; (* unique server id in interface *)
        Argument: flattenedList;
    end;
```

(b) Perform type checking and conversions according to the interface specification and declaration.
Example. The remote procedure has as its parameters an array of integers. The client calls the stub with an array name. Then the stub must get the values of the array and pass them as individual arguments.

(c) Initiate a message exchange request. Usually the message exchange will be done by a system *message module* on behalf of the stub.

(d) At the server end, the server stub will unpack the messages. Each host has a *dispatcher*, which, for each incoming request message, will look at the message for a *procedureId*, and *dispatches* the request to the intended server.

(e) When the remote procedure completes, the results are submitted to the server stub. The results are marshalled and a reply message is prepared by the server stub. It then sends the reply message to the client host through the message module at the remote host.

(f) The client stub at the client host receives and unpack the reply message and delivers the results to the client.

Stub hides and encapsulates all the details of type conversions and communications, making RPCs just like LPCs.

g. Special data structures: pointers, arrays, linked lists and etc.

(a) Prohibited by many RPC systems.

(b) Even if supported by some RPC systems, their semantics are much more restrictive than in LPCs.

什么意思?

Example. Linked lists: copy it and pass the whole; created at remote node; send back as results.

4. Distributed objects and RMI (remote method invocation)

(1) The object model

a. **Objects**. An object contains a collection of of data items and a set of exported (also called public) methods for accessing and manipulating the data items.

(a) Data items in an object can be internal (private) or external (public). Private data items can only be accessed through exported methods. Therefore private data items and ways to access them are *protected*.

(b) Similarly methods can be private or public and public objects can be invoked by applications.

(c) Arguments and return values of each exported method is clearly defined.

(d) The name, argument types and their positions, and type of return value of a method together is termed as *signature* of that method. *Method overloading* allows the same method name to be associated with several different signatures to provide maximum convenience.

(e) The selection of data structures for data items and implementation of methods are also encapsulated. They can be changed if necessary without affecting object users as long as declarations of exported methods haven't been changed.

b. **Object references**. Objects are accessed through object references.

(a) Informally speaking, an object reference (which is given as the name of a class instance variable in C++ and Java) is a pointer that points to a storage location that contains specification of the object, values of each data item in the object, and code for each method (private or public).

(b) An object reference is internally represented an *object number* within each process (like file descriptors).

(c) To invoke a method in a given object, an application supplies an object reference to the object and the method name (plus appropriate arguments as specified in the method declaration). An object whose method is being invoked is called the *target* or *receiver*.

(d) Object references are *first-class* values – they may be assigned to variables, passed as arguments and as return values.

c. **Interfaces**.

(a) An interface contains definitions of signatures of a set of methods and possibly some data types that are used by arguments and return values of some methods . An interface does not contain implementation code for each method in it.

(b) The information in an interface is adequate for programmers to write applications that will invoke methods contained in the interface.

(c) Some OO programming languages like Java allow a class to have multiple interfaces, one for each specific type of applications.

(d) Interfaces do not contain constructors.

d. **Actions**. An action is an invocation of a method of an object by another object.

(a) Besides object reference, an invocation contains all the information in the signature of the invoked method.

(b) Effects of an action.

i. The target will execute the method and send return values back to the invoker.

ii. The execution of the specified method may cause changes in the state of the storage associated with the invoked object. Other objects may be invoked in the course of execution.

e. **Exceptions**. Exceptions are unexpected conditions that are raised during object invoking.

(a) The concept of exceptions is similar to the concept of *signals* in UNIX systems. For each exception, an implementation has its default ways to handle it. Applications can specify their own methods, which override the default ways, to handle exceptions.

(b) Use of exceptions enhances debugging and free programmers from details of dealing with each possible error.

f. **Garbage collection**. When an object is not referenced by any process, it becomes garbage. Garbage should be collected by reclaiming all physical resources (storage) allocated to it. Some OO languages like Java can automatically detect and collect garbages, while others like C++ do not have this support. In the latter case, applications have to free resources explicitly.

(2) Distributed objects and remote interfaces

a. Besides all the characteristics and issues of normal objects as discussed above (object interfaces and object references), distributed objects have its own unique properties and issues.

(a) Distributed objects are physically distributed on different computers and managed by different processes. This further enhances encapsulation.

(b) An invocation of a method of a distributed object involves an IPC message exchange.

b. Distributed processes can access distributed objects through client-server models using request-reply protocols.

(3) The distributed object model

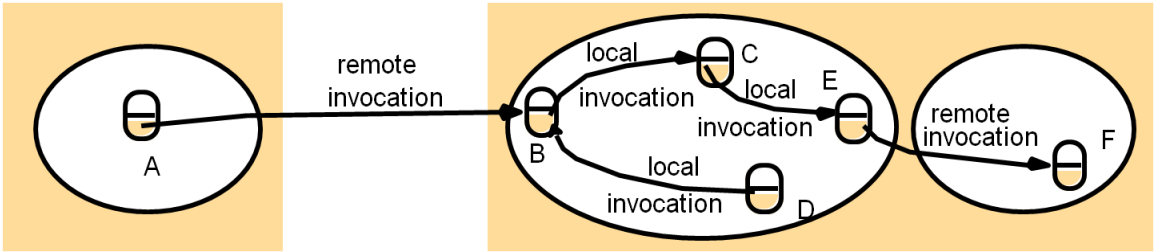


Figure 37: Remote and local method invocations (Fig.5.12, p.207)

a. The model (Fig.5.12, p.207)

   (a) A DS consists of a collection processes, each of which manages a collection of objects. Some of these objects, termed as *remote objects*, are remotely invokable, while the rest can only receive local invocations.

   (b) Method invocations between objects in different processes are called *remote method invocations* (RMI).

   (c) Two fundamental concepts of distributed object model: *remote object reference* and *remote interface*.

b. Remote object references: Fig.4.13,p.169

   (a) A remote object reference is an extension of local object reference. It is an identifier that can be used throughout a DS to refer and identify a particular unique remote object.

   (b) Fig.4.13 illustrates elements of a remote object reference.

      i. The *IP address* determines the remote host.

      ii. The *port number* and *time* components uniquely identify a process on the remote host.

      iii. A process will assign an *object number* (unique within the process) for every object created in the process.

      iv. The *interface of remote object* is provided so that any process, which receives a remote object reference as an argument or result of a remote invocation, can use the specified remote object interface to find out the methods exported by the remote object. 这是什么

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Figure 4.13 Representation of a remote object reference

c. Remote interfaces: Fig.5.13, p.208

(a) The concept of remote interfaces extends that of normal interfaces.

(b) Each remote object supports two groups of methods: remote invocable methods and those that can only be invoked locally. A remote interface specify all those remote invocable methods.

(c) In Java RMI, a remote interface is declared the same as a normal interface except that it must *Extends* the interface java.rmi.Remote to *acquire* remote invocable ability for all methods listed in the interface.

(d) CORBA IDL, Java IDL, and DCE IDL all support interface definitions. All objects (or procedures in DCE) can be invoked remotely.

d. Actions in distributed object system: Fig.5.14, p.209

(a) A method invocation may cause cascading remote object invocations by different processes hosted on different computers.

(b) The objects involved in such a chain of RMI may reside on different processes (elements).

(c) When an object is *instantiated* by a RMI, the new object usually resides on within the element where the instantiation is requested.

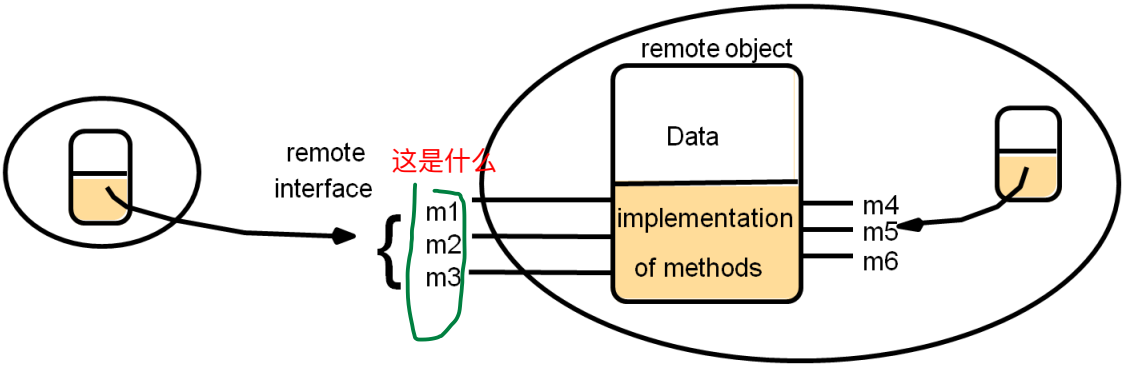


Figure 38: A remote object and its remote interface (Fig.5.13, p.208)

(4) Design issues of RMI

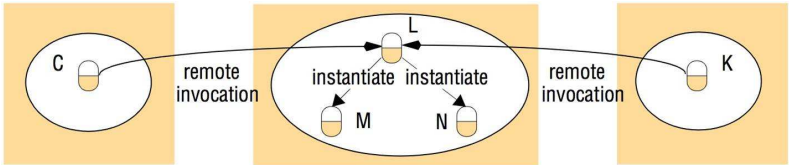


Figure 39: Instantiation of remote objects (Fig.5.14, p.209)

a. RMI invocation semantics. RMI can be viewed as a special form of client-server communications. Therefore the *at-least-once, at-most-once, maybe* and *exactly* semantics of request-reply protocols are also applicable to RMI (Cf. Section 4 of this lecture). Fig.5.9, p.198 summarizes RMI invocation semantics.

   (a) *Maybe invocation semantics*. The method in an RMI request may or may have not been executed. The invoker cannot be sure either way.

      i. This is the least reliable semantics. An RMI design that does not take any fault tolerance mechanisms (timeout, retransmissions, histories) will lead to this semantics. The server does not keep track if a RMI has been successfully completed and the client has no way to know one way or another.

      ii. It suffers: (i) omission failures if the invocation or result message is lost; (ii) crash failures when the server hosting the remote object fails. A crash failure may occur either before or after the method is executed.

      iii. Although this semantics is the least expensive to implement, it also has the least applicability. An example application using this semantics is sending a fun email message to a group of friends.

   (b) *At-least-once invocation semantics*.

      i. With this semantics, after an RMI the invoker will either receive a result, or it will receive an exception (such as a timeout) informing it that no result is received. In the former case, the invoker knows that the method has been executed at least once (possibly more than once!), and in the latter case, it knows that the method has been either executed once or not at all. In the latter case, retransmission will be used until a result is received.

      网页就是一个好例子

      ii. This semantics is suitable for *idempotent* RMI methods – methods whose multiple executions have the same effects as a single execution

      iii. It suffers: (ii) crash failures when the server hosting the remote object fails; (ii) arbitrary fails – multiple executions of non-idempotent RMI methods will cause inconsistency.

   (c) *At-most-once invocation semantics*. With this semantics, after an RMI the invoker will either receive a result, or it will receive an exception (such as

a timeout) informing it that no result is received. In the former case, the invoker knows that the method has been executed *exactly once*, and in the latter case, it knows that the method has been either executed once or not at all. It is up to the application to decide what to do next in the second case.

b. Transparency. This issue is very similar to that in RPC (cf. Section 5 of this lecture)

   (a) A RMI design supporting IDL can allow programmers to have a choice of calling semantics (at-least-once and etc.) so that a suitable semantics can be selected according to requirements of applications.

   (b) RMI designs normally allow applications to place different exceptions handlers to handle different failures.

   (c) The consensus is that an RMI should provide transparent syntax, but should specify the remote nature in interfaces. Java RMI uses the Remote interface to achieve this.

(5) Implementation issues of RMI

a. The main objects and modules in a RMI implementations are shown in Fig.5.15, p.210.

b. **Communication module**. This is the same module as in RPC (Fig.5.10). This module is responsible for actual receiving and sending of RMI request and reply messages, whose structure is shown in Fig.5.4.

   (a) The comm. module only interprets the first three fields in an RMI message. The rest are left for RMI software.

   (b) The comm. modules collectively implement a specified invocation semantics such as at-most-once.

   (c) The comm. module in a server selects the dispatcher for the class of object to be invoked.

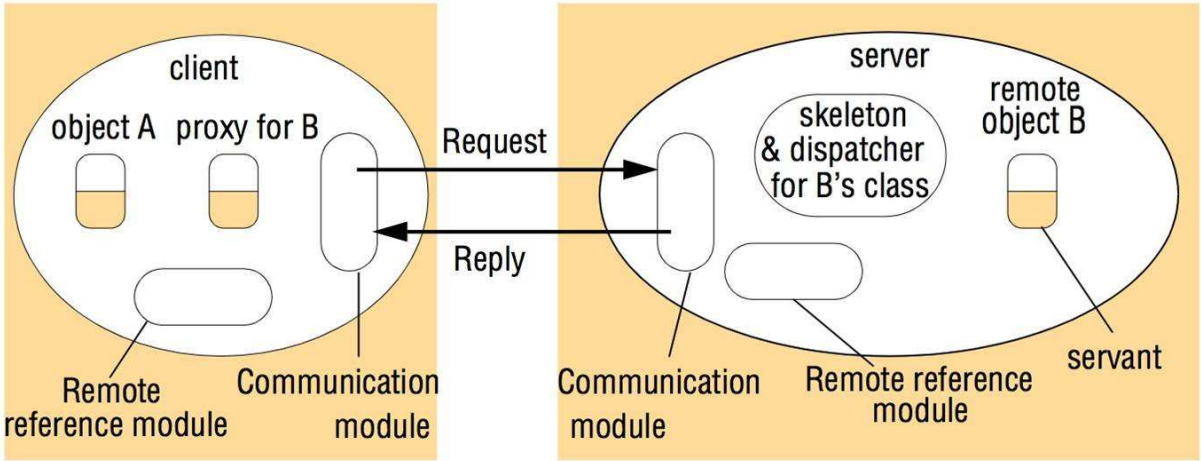
c. **Remote reference module** (RRM). Responsible for: a) translating between local and remote object references; b) creating remote object references.

   (b) The RRM in each process has a *remote object table* that records the correspondence between local object references in that process and remote object references (system wide). That is, the table maps between local object references to remote object references. The table contains:

      i. An entry for all the remote objects held by the process. That is one entry for each remote object managed by the process.

      ii. An entry for each local proxy.

   (b) This module is called components of RMI software when they marshall or unmarshall remote object references. When a request message arrives, the



这是什么?

Figure 40: The role of proxy and skeleton in remote method invocation (Fig.5.15, p.210)

table is consulted to find out which local object is to be invoked. When a request message is to be sent out, the table will record the remote object reference and its local object reference.

(c) The actions of the RRM:

i. When a remote object is to be passed as argument or result for the first time, the RRM is asked to create a remote object reference that will be added to the table.

ii. When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object. If the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.

d. **The RMI software**. A layer of software between the application objects and the communication and remote reference modules. It plays a role similar to stub procedures in RPC. It consists a *proxy*, a *dispatcher*, and a *skeleton*.

(a) *Proxy*. Proxies are *client stubs* (in terms of RPC terminologies) and hence are only in client processes. There is one proxy for each remote object for which a process holds a remote object reference.

A proxy for a remote object acts as a client's local representative or proxy (hence the name) for the remote object. The caller invokes a method on the local proxy which is responsible for carrying out the method call on the remote object. In RMI, a proxy for a remote object implements the same set of remote interfaces that a remote object implements.

When a proxy's method is invoked, it does the following:

· Initiates a connection with the remote JVM containing the remote object;

- Marshals (converts, writes and transmits) the parameters to the remote JVM;
- Waits for the result of the method invocation;
- Unmarshall (reads and converts) the return value or exception returned, and returns the value to the caller.

(b) *Dispatcher*. A dispatcher and a skeleton collectively function as a *server stub*, hence they only reside on a server process for a remote object. A server has one dispatcher and one skeleton for each class representing a remote object. A dispatcher receives *request* messages from the communication module. For each request message, it uses *methodId* to select appropriate method in the skeleton, passing on the *request* message to the skeleton.

(c) *Skeleton*. A skeleton *implements* the methods in the remote interface. However, this implementation is not actual implementation of remote methods themselves. Rather, a skeleton implements how remote methods in the remote interface will be invoked on the server.

When a skeleton receives an incoming method invocation it does the following:

- Unmarshall (reads and converts) the parameters for the remote method;
- Invokes the method on the actual remote object implementation; and
- Marshals (converts, writes and transmits) the result (return value or exception) to the caller.

Comment: Java 1.2 no longer requires a skeleton class. Only a stub class is required and generated by rmic.

e. **Generation of the classes for proxies, dispatchers, and skeletons**. The classes for the RMI software are generated automatically by an interface compiler. In Java RMI, the set of methods available in a remote object is defined as a Java interface that is implemented within the class of the remote object. The Java RMI compiler (rmic.exe in Windows platform) generates the code for the proxy, dispatcher, and skeleton classes from the class of the remote object.

f. **Server and client programs**.

(a) The server program contains the classes for the dispatcher and skeleton, together with the codes for classes (called *servant classes*) of all the remote objects it supports.

(b) Because remote interfaces cannot include constructors, remote objects are created in either an *initialization section*, or in methods in a remote interface designed for that purpose. Such methods are called *factory methods*. Any remote object that intends to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose.

(c) An *initialization section* is responsible for creating and initializing at least one of the remote objects to be hosted by the server. Additional remote objects

may be created in response to requests from clients. The initialization section may also register some of its remote objects with a *binder*. Normally it will register just one of the remote objects, which can be used to access the rest.

为啥连接一个就可以包括剩余的了

(d) The client program consists of classes for proxies for all of the remote objects that it will invoke. A binder can be used to look up remote object references.

g. **The binder**. Before invoking any remote method, a client has to obtain a reference to a remote object that hosts the remote method.

(a) A *binder* in a DS is a separate service that maintains a table containing mappings from textual names of remote objects to remote object references.

(b) A server can use a binder to register its remote objects by name. A client can use a binder to obtain remote object references.

(c) Java RMI supports a binder RMIregistry. DCE RPC supports DCE name services. CORBA has its own Naming Service.

h. **Object location**.

(a) In a distributed object system, if remote objects are static, i.e. they stay under management of the same process for the rest of their life after their creation, the remote object references as defined in Section 4.4 of the 1st textbook are adequate for clients to find remote object servers.

(b) If remote objects can migrate from process to process and from computer to computer, then the previously presented representation of remote object references is not sufficient for clients to locate remote object servers.

- In this case a so called *location service* is needed. Such a service will map remote object references to their probable current locations.
- A client will then have to present both a remote object reference and an a current location of the remote object for each RMI.

i. **Server threads**.

(a) An RMI executes a remote object in a server. During the execution, other local and remote objects may be invoked. Hence an RMI may take an arbitrary amount of time to return. In a single thread server, the server will be blocked before the return.

(b) A server can allocate a separate thread to executing each remote invocation. This demands the designers of a remote object to deal with potential problems with concurrent threads. (cf. the *Thread* class in Section 2 about Java TCP API of this lecture).

j. **Activation of remote objects**.

(a) Most servers that provide RMI services are not running as physical processes until explicitly invoked by a remote invocation. This is similar to an FTP

or TELNET server, which is not actually *forked* as a separate server process (by the so called *Internet super daemon* process *inetd*) until a client request an FTP or TELNET session.

(b) *Active* and *passive* remote objects.

i. A remote object is termed as *active* when it is available for invocation within a running process. It is called *passive* if it is not currently active but can be made active.

ii. A passive object contains two ports: (i) the implementation of the methods; and (ii) its state in the marshalled form.

(c) *Activators*. A process that starts server processes (more precisely it activates passive processes) to host remote objects are called a activator. An activator is responsible for:

- Registering passive objects that are available for activation, which involves recording the names of servers against the URLs or file names of the corresponding passive objects.
- Starting named server processes and activating remote objects in them.
- Keeping track of the locations of the servers for remote objects that it has already activated.

In Java RMI each host must run an activator as a system daemon process. The activator is responsible for activating objects on that host.

k. **Persistent object stores**.

(a) *Persistent objects*. An object that lives between activations is called persistent object (like *static* variables can survive between procedure calls).

(b) Persistent objects are generally managed by persistent object stores in a marshalled form on disk. They will be activated when methods in them are invoked by other object.

(c) The *Activatable* class Java and *Entity beans* in JavaBeans are examples of persistent objects.

Lecture 7
Case Studies of RPC and RMI
(07-28,29-2020)

Lecture Outline

```
I.   SUN RPC networks
   1. Introduction and history
   2. The RPC programming guide
   3. rpcgen: a compiler for RPC
   4. RPC protocol specification
II. DCE RPC
   1. Overview
   2. Using DCE RPC interface
   3. Writing clients
   4. Pointers and arrays
   5. Writing servers
III.  Java RMI (Chapter 5.4)
   1. Introduction
   2. Tasks in writing an RMI client-server distributed
      application
   3. Example 2: the ‘‘ShapeList" RMI application
   4. Example 3: the ‘‘Weather information server"
```

I. SUN RPC

1. Introduction and history

(1) First commercial implementation of RPC

(2) Illustration of RPC semantics (Page 1, RPC Prog. Guide)

2. The RPC Programming Guide

(1) Levels of RPC – three levels

a. The higher level: totally transparent about OS, machine, and networks. System calls for RPC are provided to users as ordinary system calls

b. The intermediate level: a collection of system routines for defining and writing RPC application programs are provided. Among them are *registerrpc(), callrpc()*, and *svc_run*(). *registerrpc*() obtains a unique system-wide identification number, while *callrpc*() actually calls a RPC.
Not flexible, for instance cannot choose transport protocol, time-out intervals.

c. Lower level: allows control of many lower-level details, such as transport protocol, time-out intervals.

(2) Higher Level (p.3, RPC Prog. Guide)

a. **Example**. writing a program that counts the number of users that are logged in on a remote machine. This can be done using the RPC service library routine *rnusers*.

b. The library function *rnusers*() is called within the client program.

c. The program should be compiled with -lrpcsvc flag. The library librpcsvc.a contains several other functions: *rusers*() (return information about users on a remote machine), *havedisk*() (determine if a remote machine has disk). *rstats*() (get performance data from remote kernel), *rwall*() (write to specified remote machines).

d. Some of RPC services such as *ether*(), *rquota*(), and *spray*(), cannot be used as library routines. They can be used from the *callprc*() function (to be introduced next).

(3) The Intermediate Level (p.4-9, RPC Prog. Guide)

a. The example client expressed using intermediate level calls: p.4

(a) The *callrpc*() call takes eight parameters.

· The first is remote host name.

· The 2nd, 3rd, and 4th are program number (RUSERSPROG in the eg.), program version number (RUSERSVER), and procedure number (RUSERSPROC_NUM) respectively.

· The 5th and 6th are a pair. The 5th is an XDR filter (type handler) for the input parameter, which is the 6th parameter.

· The 7th and 8th are another pair. The 7th is an XDR filter for the output parameter, which is the 8th parameter. In the example, the 8th parameter is a pointer type because it carries return value.

(b) If more than one value is needed for input, a built-in or user-defined type with corresponding XDR filters can be used.

(c) XDR (External Data Representation) provides a uniform type conversion mechanism (more to be covered next): **RPC can handle arbitrary data types/structures, regardless of different machines' byte order and structure layout conventions, by always converting them to a network standard called *External Data Representation* (XDR) before sending them over the network**.

(d) Return values of *callrpc*(): 0 if successful, non-zero otherwise. Detailed error values are in <rpc/clnt.h>.

(e) The Transport protocol used by *callrpc*() is UDP. The timeout interval is un-adjustable. To be able to adjust the timeout and number of retries, lower level primitives are needed. 为什么?

b. The example server: p.5

(a) The remote procedure: the code for computing the number of users is left unfinished.

(b) The main program calls two RPC routines: *registerprc*() and *svc_run*().

(c) The *registerprc*() routine registers a C procedure as corresponding to a given RPC procedure number. It six parameters. The first three are the program number. program version number, and procedure number. The 4th is the name of the local procedure (*nuser* eg.) that implements the remote procedure (*nusers* eg.). The 5th is the XDR filter of the input parameter (xdr_void eg.), and the 6th is the XDR filter of the output parameter (xdr_u_long eg.).

(d) Only UDP transport mechanism can use *registerrpc*(). So this call can be safely used with the *callrpc*() call.

(e) UDP can only deals with arguments and results less than 8k bytes in length.

(f) The *svc_run*() routine, which is the RPC library's remote procedure dispatcher. This function will call the remote procedure in response to RPC call messages. The dispatcher also takes care of decoding the remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

c. Program number assignments. Program numbers are assigned in group of 0x20000000 according to the chart shown on p.6:

(a) 0x0 to 0x1fffffff defined by SUN: this group is administered by SUN for well-known and well-defined popular applications. Customers can apply for program numbers of their new programs by contacting SUN, sending a compilable rpcgen ".x" file describing the user's protocol for each new program.

(b) 0x20000000 to 0x3fffffff defined by user: reserved for specific user applications, primarily used for debugging new programs.

(c) 0x40000000 to 0x5fffffff defined transient: is reserved for applications that generated program numbers dynamically.

(d) 0x60000000 to 0xffffffff reserved for future use/development.

(e) Program numbers and protocol specifications for standard SUN RPC services are described in the include files in /usr/include/rpcsvc dir.

(f) A complete list of all supported RPC services: Table 3-2, p.6-7.

d. Passing Arbitrary Data Types

(a) *Serializing and deserializing*: The action of converting from a particular machine representation to XDR format is called *serializing*. The reverse action

is called *deserializing*.

(b) Built-in type routines (XDR filters): 11 of them: xdr_int(), xdr_u_int(), xdr_long(), ... 区别?

(c) Notice: any xdr filter that is used with the *callrpc*() and *registerrpc*() calls can only have two parameters (they only pass two parameters to their XDR routines). Therefore although the routine *xdr_string*() exists, which takes three parameters, it cannot be used with *callprc*() and *registerrpc*(). *xdr_wrapstring*() 区别 区别 has only two parameters, and thus is fine. In fact, *xdr_wrapstring*() calls *xdr_string*().

(d) User defined types and XDR routines: example 1: the type *simple*() structure p.7-8.

- The structure has two components: an integer and a short.
- The corresponding XDR filter is shown on page 8.
- Notice that *xdr_simple*() calls *xdr_int*() and *xdr_short*() respectively to convert the two components.

(e) Other prefabricated XDR filters (page 8): *xdr_array, xdr_vector, xdr_string, xdr_bytes, xdr_union, xdr_opaque, xdr_reference, xdr_pointer.*

(f) Sending variable length int arrays: utilize the prefabricated *xdr_array*() XDR filters.

- The type *varintarr*(): two components: the array and the length of the array (integer).
- The XDR filter *xdr_varintarr*(): it calls *xdr_array*(), which takes six parameters (XDR handle, the pointer to the 1st element, the pointer to the length, the maximum length constant, the size of each array element, and the xdr filter for individual element (xdr_int in this case)).

(g) Sending fixed length int array: utilize *xdr_vector* filter:

- No need to have a structure any more.
- The XDR filter *xdr_intarr*() calls *xdr_vector*(), which takes five parameters.

(h) Dealing with 32-bit unit problem: XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the example above involved characters instead of integers, each char would occupy 32 bits.

- The XDR filter *xdr_bytes*() is similar to *xdr_array*(), but it packs characters. This filter has four parameters, similar to the first four parameters of *xdr_array*() (An example is given in the section about XDR memory allocation).
- Dealing with null-terminated strings: the *xdr_string*() filter, which is the same as *xdr_bytes*(), except it only has the first three parameters (not have the 4th, which is the length parameter). On serializing, it calls the

C function *strlen*() to get length of a string. On deserializing, it creates a null-terminated char string.

- The final example: it calls the *xdr_simple*() filter previously defined, and built-in functions *xdr_string*() and *xdr_reference*(), which takes four parameters. The *xdr_reference*() filter can be easily replaced with the *xdr_simple*() in this example.

(4) The Lower Level

a. Problems with higher and intermediate levels:

(a) At the higher level, only a number of fixed functions are available.

(b) At the intermediate level, only UDP can be used as the transport protocol. UDP only allows 8k bytes of data.

(c) There is no support at higher or intermediate levels for allocating and freeing memory explicitly. Memory allocations and freeing are done implicitly (to be discussed next).

(d) There is no support for authentication on either the client or server side that allows verifications of credentials.

b. The server side of the *nusers*() program written with lower level calls: page 10.

(a) The *svcudp_create* call returns a UDP transport handle, which is used for receiving and replying to RPC messages (the *registerrpc* call in the intermediate level uses *svcudp_create* to get a UDP handle). The *svctcp_create* returns a TCP handle.

(b) The *svcupd_create* call takes one parameter, which should be socket descriptor. If you specify your own socket descriptor, it can be bound or unbound. If it is bound to a port by the user, the port numbers of *svcudp_create* and the corresponding call *clntupd_create* at client side must match.

(c) The special constant *RPC_ANYSOCK* asks the RPC library to create and open a socket on which to receive and reply RPC messages. The routine *svcudp_create*() and *clntudp_create*() will cause the library routines to *bind*() their socket if it is not bound yet.

(d) The call *pmap_unset*() will clean up the port mapper's tables for any entries left for the *nusers* program.

(e) The call *svc_register* registers the service's port number with the local portmapper service.

- This call takes five parameters: the transport handle, the program number, the version number, the program name, and the protocol type.
- A client can discover the server's port number by consulting the portmapper on the server's machine. This is done by giving a zero port number in *clntudp_create*() and *clnttcp_create*().

- Notes: The registration is done at the program level, not at the procedure level; Also, there is no XDR routines involved during registration process.

(f) The user routine *nuser*() must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by *nuser*() and *registerrpc*() automatically:

- The procedure *NULLPROC* returns with no results. This can be used as a simple test for detecting if the remote program is running;
- There is a check for in valid procedure numbers. The routine *svcerr_noproc*() will be called in response to invalid procedure number error.

(g) The remote procedure *nuser*() serializes the results and returns them to RPC caller via the routine *svc_sendreply*(), which takes three parameters. The first is the transport handle, the 2nd is the XDR routine, and the 3rd is a pointer to the data to be returned.

c. Handling input parameters in the server: the example procedure *RUSERPROC_BOOL* (page 12).

(a) This procedure passes a parameter, whose value is the number of users on a remote machine guessed by the caller. The caller responds TRUE or FALSE depending upon if the actual number of users is equal to the guessed value.

(b) The procedure will be another branch in the body of the *nuser*() program.

(c) The input parameter *nuserquery* is handled by the routine *svc_getargs*(), which takes three parameters: a transport handle, an XDR filter, and a pointer to the input value.

(d) The routine *svc_sendreply*() is still used to send a boolean value back to the caller.

d. Memory allocation with XDR

(a) XDR routines not only do input and output, they also do memory allocation.

- That is why the second parameter of *xdr_array* (it has 6 parameters) is a pointer to an array, rather than the array itself.
- If that parameter is NULL, then *xdr_array*() will allocate space to the array and return a pointer to it. It will also put the size of the array in the third parameter.

(b) An example: the *xdr_chararr1*() routine that deals with a fixed array of bytes with length *SIZE* (page 12).

- If the space has been allocated, the *xdr_chararr1*() can be used with the syntax:

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

- If the user wants XDR to allocate memory, the routine *xdr_chararr1* has to be rewritten (renamed to *xdr_chararr2*() shown on page 13).
- Notice the difference between *xdr_chararr1*() and *xdr_chararr2*(): The input parameter *chararrp* in *xdr_chararr2*() is now a pointer to a pointer of char string, while *chararr* in *xdr_chararr1* is a pointer to a char string.
- The RPC call now initializes the input parameter to have a NULL value.
- The routine *svc_freeargs*() free (deallocate) the memory pointed by its last parameter. This routine is smart and will not attempt free memory of the pointer parameter has NULL value.

(c) Summary of memory handling:

- Each XDR routine is responsible for serializing, de-serializing, allocating (if needed), and deallocating (freeing) memory;
- Serializing: when *callrpc*() is called;
- De-serializing: when *svc_getargs*() is called;
- Deallocating: when *svc_freeargs*() is called.

e. The calling side (client) of the *nusers* example:

(a) The code: page 14.

(b) Highlights:

- The socket structure *sockaddr_in* data type;
- The *gethostbyname* function;
- Initialization of the *server_addr* structure;
- The routine *clntudp_create*: will create a client handle for the client to send and receive messages. It takes five parameters: the socket structure pointer, program number, version number, a predefined timeout interval, and a pointer to a socket descriptor (it has *RPC_ANYSOCK* in this example).
- The actual remote procedure is called via the routine *clnt_call*(), which takes seven parameters: the client handle, the remote procedure number, the input XDR filter, the input parameter, the output XDR filter, the output parameter, and a timeout structure.
- Finally, the routine *clnt_destroy* is called with the client handle as the single parameter. That call deallocates the memory associated with the client handle. It closes the socket associated with the CLIENT handle only when the RPC library previously opened it. The socket will be left open if it was opened outside this function.
- For TCP client, the routine *clnttcp_create*() can be used. This call does not take timeout parameter. Instead it has an input buffer size and an output buffer size parameters.

- For TCP the server side uses the *svctcp_create*() call, which takes three parameters: the first is the socket descriptor. The last two are the send and receive sizes respectively. "0" for these two values that the system chooses a default value.

3. *rpcgen*: a compiler for RPC

(1) Introduction: what is and why a RPC compiler?

a. rpcgen is a compiler that automatically converts procedure arguments and return-results into their network formats and vice versa.

b. The details of programming applications to use RPC can be overwhelming. The most challenging task is writing the various XDR routines to convert parameters and results to their network format and vice versa.

c. rpcgen is a compiler and is intended as a tool that do most of the dirty work.

d. Input: remote program interface definition written in a language (named RPC language, similar to C).

e. Output: client stubs, skeleton of servers, XDR filter routines for both parameters and results, head files containing common definitions,

f. The client stubs interface with the RPC library and effectively the network from their callers. Similarly the server stub effectively hide the network from the server procedures that are to be called by remote clients. The XDR filter routines are generated automatically from the interface definition.

g. The rpcgen output files can be compiled and linked same as C programs.

* A server developer writes server procedures - in any language that observe SUN calling conventions - and links them with the server skeleton produced by rpcgen to produce an executable server program.
* To call a remote program, a user writes an ordinary main program that make local procedure calls to the client stubs produced by rpcgen. Linking this program with the rpcgen's stubs creates an executable client program. (At present, the main program can only be in C).

h. rpcgen options can be used to suppress stub generation and to specify the transport (UDP/TCP) to be used by the server stub.

i. Code generated by rpcgen may be less efficient than hand-written code. But one can mix these two types of code. In addition, code generated by rpcgen can be starting point to write good code.

(2) Converting local procedures to remote procedures: a local program that prints a message on the console of the local machine.

a. The code consists of the main program and the procedure *printmessage*() on page 2 of the rpcgen Prog. Guide.

b. How to convert such a local procedure to a remote a procedure so that one can call it to print a message on the console of a remote machine?

   (a) Analyze the program: the procedure is *printmessage*(), which takes a string as input, and returns an integer as output.

   (b) Knowing this info, we can write a protocol specification in RPC language that describes the remote version of *printmessage*() (page 3 the file *msg.x*).

   (c) The file *msg.x* actually declared a remote program (a remote procedure is part of a remote program). That remote program only contains a single procedure *PRINTMESSAGE*. This procedure is declared to be in version 1 of the remote program.

   (d) No null procedure (procedure 0) is necessary because rpcgen generates it automatically. Also notice that everything is capital case – this is not required but is a good convention to follow.

   (e) The argument is type *string*, not *char* $*$, because the latter is used in C.

c. Two more things to write to complete the conversion: The remote procedure itself, and the client program that will call the remote procedure.

   (a) The remote procedure itself: page 3, *msg_proc.c*. The remote procedure *printmessage_1*() differs from its local version in three ways:

      i. It takes a pointer to a string instead of the string itself as the single parameter. This is true for all remote procedures: they always take pointers to their arguments instead of arguments themselves.

      ii. It returns a pointer to an integer instead of an integer itself. This is also true for all remote procedures.

      iii. It has a "_1" suffix appended to its name. In general all remote procedures called by rpcgen are named by the following rule: the name in the program definition (*PRINTMESSAGE* here) is converted to all lower-case letters, an underscore letter "_" is appended to it, and finally the version number is appended (here is 1).

   (b) The main client program that calls the remote procedure: page 5-6: *rprintmsg.c*.

      i. The call *clnt_create*() from the RPC library routine creates a client "handle", which will be passed to the client stub routines which calls the remote procedure.

      ii. The remote procedure *printmessage_1* is called exactly the same way as it is declared in *msg_proc.c*, except for the inserted client handle as the first parameter.

d. How to compile the server and the client programs: two main programs are compiled.

(a) % rpcgen msg.x

(b) % cc rprintmsg.c msg_clnt.c -o rprintmsg

(c) % cc msg_proc.c msg_svc.c -o msg_server

e. *rpcgen* did the following to the input file *msg.x*:

(a) Created a header file *msg.h* that contains #define's for *MESSAGEPROG*, *MESSAGEVERS*, and *PRINTMESSAGE* for use in other modules;

(b) Created client "stub" routines in the *msg_clnt.c* file. In this case there is only one, *printmessage_1*. Names for client stubs always has the form FOO_clnt.c for an input file FOO.x.

(c) Created server program that calls *printmessage_1*() in *msg_clnt.c*. The server program is named as *msg_svc*. Names for server output files always has the form FOO_svc.c for an input file FOO.x.

f. Running the server and client

(3) Generating XDR routines: rpcgen can also generate the XDR routines, besides the client and server RPC code.

a. An example: a remote dir listing service. Will use rpcgen to generate the server/client and the XDR routines.

b. The protocol specification file *dir.x* on page 7.

– Running rpcgen on dir.x produces four files: The header file, client stub routines, and server skeleton. In addition, the fourth, named *dir_xdr.c*, are the XDR routines that are needed for converting the data types.

c. The implementation of the *READDIR* procedure *readdir_1*(): page 8.

d. The implementation of the client side program *rls.c* procedure: page 8.

e. Note: The client program and and the server procedure can be tested together on a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls will be executed as ordinary local procedure calls and the program can be debugged with debuggers such dbx.

(4) The C-Preprocessor

a. The C-preprocessor is run all input files before they are compiled by rpcgen. So all the preprocessor directives are legal within the ".x" file

b. Four symbols may be defined, depending upon which output file is getting generated: RPC_HDR for header file output; RPC_XDR for XDR routine output; RPC_SVC for server skeleton output; and RPC_CLNT for client stub output

c. Any line beginning with a percentage sign % is ignored. (not a recommended practice).

4. RPC protocol specification

(1) Introduction

a. The SUN RPC protocol

b. Transport and semantics: the RPC protocol is independent of the underlying transport protocol, i.e. it does not care how messages are passed from one process to another process.

(2) RPC Protocol Requirements

a. Three fundamental requirements:

* Unique specification of a procedure to be called;
* Provisions for matching response messages to request messages;
* Provisions for authenticating the caller to service and vice-versa.

b. Other supported features:

* RPC protocol mismatch checking;
* Remote program protocol version mismatch checking;
* Protocol errors detection;
* Notification of reasons of authentication failures;
* Notification of any other reasons why a desired procedure was not called.

c. Programs and procedures

* A RPC call message carries three unsigned fields: remote program number, remote program version number, and the remote procedure number. These three quantities uniquely identify the procedure to be called.
* The RPC protocol version number: the currently version is 2. This protocol version number may also evolve with time.
* The reply message contains enough info to distinguish the following five error conditions:
  - RPC protocol version mismatch;
  - The remote program is not available on the remote system;
  - Remote program version number mismatch;
  - The requested procedure number not available;
  - The parameters passed cannot be understood by the remote server.

d. Authentication

* A call message has two authentication fields: the credentials and the verifier. The reply message has one authentication field: the response verifier. The RPC protocol specification defines the all three fields to be the following opaque type (p.3).

* Any *opaque_auth* structure is an *auth_flavor* enumeration followed by bytes which are opaque to the RPC protocol implementation.
* The interpretation and semantics of the data contained within the authentication field is specified by individual, independent of authentication protocol specification.
* If authentication parameters were rejected, the response message contains info stating why they were not accepted.

e. Program number assignment (already covered in RPC PROG. GUIDE)

(3) RPC Message Protocol (p.5-8).

II. OSF DCE RPC

1. Overview

   (0) DCE application programming interface (p.15 & 16 of the the book *Understanding DCE*)

   (1) The client-server model (Fig.1-1)

   (2) The RPC mechanism (Fig.1-2)

   (3) Three phases required to develop a distributed application (Fig.1-3):

      a. Interface development

      b. Server development

      c. Client development

   (4) The arithmetic example

      a. An one-client/one-server RPC application. The remote server system has special hardware (e.g. processor array) for performing arithmetic operations;

      b. Outline: Clients make RPC call to the server to perform arithmetic operations; Server computes and returns results; Results are displayed on client's screen.

      c. A simple interface for the arithmetic example (Fig.1-4).

         (a) Interface definition language (IDL). E.g.: the rpcgen language and XDR files.

         (b) IDL resembles C in syntax and semantics

      d. Universal unique identifier (UUID)

         (a) UUID: a number, uniquely identifies an interface.

         (b) *uuidgen*: a utility that generates UUIDs.

```
% uuidgen -i
```

      e. The interface definition: p.7, Example 1-1.

         (a) *uuid* attribute specifies the interface UUID

         (b) The *interface* keyword, followed by the name of the interface *arithmetic*.

         (c) Section for type and constant definitions

         (d) Section for declaring procedures: The *in* and *out* parameters.

      f. Generating stub and header files using IDL compiler

         (a) A header file that contains the definitions needed by the stubs and the application code. Can be included anywhere appropriate.

(b) A client stub file, to be linked with the client portion of the application code.

(c) A server stub file, to be linked with the server portion of the application code.

(d) Invoking the IDL compiler:

% idl arithmetic.idl

g. A simple client

(a) Produce a client (Fig.1-5);

(b) A simple client code: Example 1-2, p.9

i. Header file;

ii. RPC call.

h. A minimal server: two parts: the actual remote procedure – sometimes called the *manager*; And the code that initializes the server.

(a) Files and utilities needed to produce a server (Fig.1-7, p.13, not Fig.1-6).

(b) Remote procedure implementation (Example 1-3, p.11-12).

i. header file "arithmetic.h" included

ii. The procedure definition matches its corresponding def. in interface def.

iii. The body of the remote procedure

i. A distributed application environment (Fig.1-7,p.13).

(a) *Binding*: relate the RPC call on a local machine to the remote procedure implemented on a remote machine.

(b) *Binding information*, which includes:

i. Protocol sequence: an RPC-specific name containing a combination of comm. protocols that describe the network comm. used between a client and a server. Example: *ncacn_ip_tcp* stands for a Network Computing Architecture connection-oriented protocol, over a network with the Internet Protocol and the TCP.

ii. Server host: the name or network address of the host on which the server resides

iii. Endpoint: a number representing and identifying the specific server process on the server host – typically a port number.

(c) *Name service* on DCE: a database service that provide binding information.

i. A server can store binding info that a client on another system can retrieve later;

ii. Called CDS (Cell Directory Service)

iii. *Name service independent*(NSI) routines: provides as RPC runtime library routines. They communicate with CDS to put info into the database

when the server called them.
NSI routines are a level of abstraction above the particular name service on a system.

iv. Name service mechanism, although not required, is preferred in distributed application environments.

j. Server initializing: Fig.1-9, p.15

(a) A server first registers the interface with the RPC runtime library (Step 1), which creates binding information to identify this service (Step 2).

(b) The server places binding info in appropriate databases, and places comm. and host info in the name service database (Step 3)

(c) The server places process information (called *endpoint*, i.e. port number) in a special database called *local endpoint map* on the server system (not CDS!).

(d) The serve waits and listens for incoming RPC requests (Step 5).

k. Client finding a server: Fig.1-10, p.16

(a) An RPC is made (Step 6). The client stub looks for the server system info from the CDS (Step 7).

(b) The runtime library finds the server process endpoint by contacting the local endpoint map on the server system (Step 8).

(c) Now the binding info is complete and the client completes the binding

l. Complete a RPC: Fig.1-11, p.17

(a) The client stub puts parameters and other calling info into an internal RPC format, which is transmitted over the network (Step 10 & 11).

(b) The server runtime library receives the data and transfers it to the server stub (Step 12); The server stub converts the data back to a format appropriate for use (Step 13).

(c) Remote procedure is dispatched (Step 14).

(d) Now the reverse operations are performed: Step 15 (prepare output); Step 16 (transmit output); Step 17 (receive output); Step 18 (convert the output).

m. Server initialization code of the *arithmetic* example

(a) Register the interface.

· The routine *rpc_server_register_if* is used to register the interface. The variable *arithmetic_v0_0_s_ifspec* is called an *interface handle*, which is produced by the IDL compiler and refers to the info such as UUID that the application needs. The *CHECK_STATUS* macro is defined in the *check_status*.h header file for examples in this book.

(b) Create binding info.

- The routine *rpc_server_use_all_protseqs* requests that the clients can use all available protocols. During this call, the RPC runtime library gathers info about available protocols, your host, and endpoints to create binding info. The system allocates one buffer for each endpoint to hold incoming call info. The size of the buffers is decided by the *rpc_c_protseq_max_calls_default* argument.

(c) Obtain the binding info.

- When creating binding info, the RPC runtime library stores binding info for each protocol seq. A *binding handle* is a reference in application code to the info for one possible binding. *binding vector*: a set of binding handles. The binding vector can only be obtained through the *rpc_server_inq_bindings* routine to pass the info to other DCE services with other runtime routines.

(d) Advertise the server location.

i. The routine *rpc_ns_binding_export* places all the binding info to the name service database.

ii. Entry name: is a string obtained from an environment variable (*ARITHMETIC_SERVER_ENTRY* for this example) set by the user specifically for the application.

iii. The interface handle *arithmetic_v0_0_s_ifspec* associates interface info with the entry name in the name service database.

iv. The client later uses name service routines to obtain binding info by comparing the interface info in the name service database with info about its own interface.

v. The argument *rpc_c_ns_syntax_default* tells the routine how to interpret an entry name. At present DCE has only one syntax.

(e) Register the endpoints in the local endpoint map.

i. The RPC runtime library assigns endpoints to the server as part of creating binding info.

ii. The runtime routine *rpc_ep_register* lets the endpoint map on the local host know that the process running at these endpoints is associated with this interface.

(f) Free the set of binding handle.

i. The runtime routine *rpc_server_inq_bindings* allocates memory for binding handles.

ii. The runtime routine *rpc_binding_vector_free* frees this memory when the server finishes passing binding info to other parts of DCE.

(g) Listen for remote calls.

i. The runtime routine *rpc_server_listen* let the server waits for calls to arrive. The 1st argument *rpc_c_listen_max_calls_default* gives the maximum number of calls that a server can accept at one time.

ii. Note: the concept here is the same as the `listen` function in networking programming.

n. Producing the application: compile and link the server and client separately.

(a) DCE libraries: -ldce and -lcma libraries. The libcma.a (Concert Multi-threaded Architecture library) is required for DCE to handle threads internally.

(b) Compile and link the client code

(c) Compile and link the server code

o. Running the application

(a) The binding method used in *arithmetic* example: *automatic binding*, which requires that the client has to set the RPC-specific environment variable *RPC_DEFAULT_ENTRY* so that, so that the client has an entry name with which to begin looking for binding info.

(b) Other binding methods (implicit method and explicit method) exist and will be covered later (Ch.3).

(c) The example uses the simplistic approach by assigning *RPC_DEFAULT_ENTRY* the same entry name used in server initialization when exporting the binding info to the name service database.

(d) To run the distributed arithmetic application:

i. The server process must have read and write access to the name service database to export binding info to it.

ii. First set the environment variable *RPC_DEFAULT_ENRTRY* before the executing the server. This variable represents a name for the entry that this server uses when exporting the binding info to the name service database

iii. The usual convention for entry names is to concatenate the interface and host names.

iv. An environment variable is needed here because the name can vary depending on which host the server is to run.

v. If such a name is not valid, the binding info will not be placed in the name service database and the program will fail.

vi. The special prefix /.:// is required to represent the global portion of a name in the hierarchy of name service database.

```
moxie> setenv ARITHMETIC_SERVER_ENTRY /.://arithmetic_moxie
```

vii. Execute the server.

moxie> server

(e) On the client host:

moxie> setenv RPC_DEFAULT_ENTRY /.://arithmetic_moxie

and run the client:

moxie> client

(f) Using ctl-C to kill the server.

p. Complete development process of the arithmetic example: Fig.1-12, p24.

2. Using DCE RPC interface

(0) DCE RPC Interfaces

a. Interface definition is written in IDL and interface definition files end with suffix .idl

b. Compilation of an interface produces a C header file, the client stub file, the server stub file.

c. An interface definition, which describes the procedures offered by a server, is usually written by the server developer. Client developer has to read and interpret the definition.

d. All servers that support the interface must implement the remote procedures using the same procedure names, data types, and parameters. All clients must call the RP consistently

e. A *procedure definition* in an interface definition specifies the proc. name, the data type of the value it returns, and the number, order, and data types of its parameters.

(1) The Interface Definition Language (IDL)

a. IDL has syntax similar to C

b. *Attributes*: IF attributes are special keywords that offer info that helps describe an application. IF attributes are enclosed in square brackets in the IDL file.

(a) Some attributes distinguish one interface from another on a network. Eg.: the uuid attribute declares the UUID for the interface. These attributes guarantees that the client will find the server that implements the proper RP.

(b) Some attributes explicitly describe data transmitted over a network. Some aspects, which are taken for granted in normal applications, must be described explicitly for distributed applications. Eg.: *union*, in distributed applications, a separated variable indicating the exact data type in an application must be specified in IDL so that it can be transmitted with a union parameter.

(c) Some attributes make data transmission more efficient: input, output, or both.

Table A-1 to A-8 in Appendix A show all IDL attributes.

c. Structure of an interface definition

(a) The interface header

i. Interface header attributes

ii. Interface name

(b) The interface body

i. Import statements

ii. Constant definitions

iii. Data type definitions

iv. Procedure declarations

d. Interface header attributes: they specify RPC features that apply to the entire interface.

(a) uuid attribute; major version number; minor version number.

(b) Rules that determine whether a client can use an interface that a server supports:

i. The UUID of client and server must match;

ii. The major version number of the client and server must match;

iii. The minor version number of the client must be less than or equal to the minor version number of the server. Upward compatibility.

e. Another example: the *inventory* application

(a) A product database is store on the server system. A client makes inquiries based on a part number. (complete code in Appendix D).

(b) The inventory interface (Example 2-1, p.28).

i. Notice the square brackets;

ii. The *uuid* attribute contains the UUID of the IF;

iii. The *version* attribute is optional;

iv. The *pointer_default* attribute is optional and used to more efficiently transmit pointer data;

v. The keyword *interface* and a name are required to identify an IF. The IDL compiler uses this name to construct data structure names.

f. Type definitions, data attributes, and constants

(a) Concepts of IDL data types.

i. In C, a data type can map to different sizes on different systems. Eg.: long int

ii. IDL Rules: to ensure that the size of an IDL data type is same on all systems (so that DCE applications can exchange data), the IDL compiler generates C code data types that begin with *idl_*, and places them in the header file.

iii. Although the size of a particular IDL data type is always the same, it may map to different C data types different systems. The *idl_* data types are defined in a DCE RPC-supplied header file to map to the proper sized C data type for the system the compile took place.

(b) Basic IDL data types (Table 2-1, p.29,30).

i. Notes on basic IDL data types: Table 2-2, p.30-31.

ii. How do the IDL data types help to distribute and application? – the stubs perform the *marshalling and unmarshalling* operations. Mashalling converts data into a byte stream format and packages it for transmission using an NDR (Network Data Representation). NDR allows successful sharing between systems with different data formats. It handles differences like big- versus little- endian (byte order), ASCII chars versus EBCDID chars, and other incompatibilities.

(c) An example IDL type definition for the inventory example: Example 2-2, p.32

i. The keyword *const* defines a constant;

ii. The keyword *typedef* defines a new data type;

iii. A data type is defined by other data types, constants, and possibly some attributes.

(d) Pointers: pointers in distributed applications cannot provide the efficiency as in normal applications, as they involve extra copying and transmission. Two types of pointers are supported in IDL to balance convenience and efficiency:

那我们为何继续用pointer?

i. *Full pointers*: has all the capabilities that usually associate with pointers. Cause additional stub overhead. Notated by the keyword *ptr*.

ii. *Reference pointers*: a simpler pointer that refers to existing data. Less costly. Limited in capabilities. No new memory can be allocated for the client during RPC. Notated by the keyword *ref*.

(e) Arrays: subscript starts at 0, same as in C. To be more efficient in handling arrays, IDL provides three types of arrays:

i. *Fixed array*: like a standard C array

ii. *Varying array*: has a maximum size determined at compile time. But it also has subset bounds represented by variables. Only the portion of the array you need is transmitted.

iii. *Conformant array*: The size of a conformant array is represented by a dimension variable and the actual size is determined when the array is used.

(f) Strings: strings in IDL does not have to end with the null char. To require a string to end with a null char, the *string* attribute must be used. Example 2-3, p.34:

   i. The data type of a string array must be char or byte.
   ii. The example defines a *conformant string*.
   iii. The array attributes *size_is* and *max_is*

(g) Enumerated types: like in C. 1st element is mapped to zero. Example 2-4, p.35.

(h) Structures: defined with the same syntax as in C. Example 2-5, p.35.

(i) Discriminated unions

   i. Conventional discriminated union: a discriminator variable that is separate from the union structure is used to keep track of the data type stored in the union.
   ii. In IDL, a discriminated union includes a discriminator as part of the data structure itself.

   Example 2-6, p.36, Defining discriminated union in IDL

   iii. Keywords *typedef union* introduce the type definition;
   iv. The keyword *switch* specifies the the data type and name of the discriminator variable – can be boolean, char, or int.
   v. Name of the union data type (*total*.
   vi. Keyword *case*. The case value is of the same type as the discriminator variable.
   vii. The name of the new data type (*part_quantity*).

   The IDL compiler converts each of such definitions into a structure with two components: the 1st component is the discriminator variable and the 2nd component is a C union. Example 2-7: A discriminated union generated by the IDL compiler.

   Example 2-8, p.37: using discriminated union in application code:

   viii. The *part_quantity* dis. union is a member of the *part_record* structure (Example 2-5).
   ix. The *part.quantity* struct member is a dis. union.
   x. The *part.quantity.units* struct member is the discriminator for the union.
   xi. The *part.quantity.total* struct member is the union.

   The union name can be omitted and IDL compiler will generated *tagged_union* for the user.

(j) Pipes: pipes can make data transmissions much more efficient than regular RPCs in the following case:

   i. Large amount of data must be transmitted at once;

ii. The total amount of data is unknown until the application is running such as when processing files;

iii. The data is incrementally produced and consumed (Ch.11 has details).

g. Procedure declarations and parameter attributes: Example 2-9, p.39. Seven procedures declared.

* An IDL procedure can return a value, just like in C.

(a) Procedures defined with *void* type do not return a value.
Input and output parameters;
As in C, arrays and strings are implicitly passed by reference.

(b) Some procedures may return a data structure or a pointer to a data structure.

(c) Output pointers requires pointers to pointers when new memory is allocated.

(d) Parameters that are changed by the RP use both *in* and *out*.

(e) *Idempotent* procedures: those that can execute more than once with the same arguments and to produce identical results without any undesirable side-effects.

(2) Using the IDL Compiler

a. Utilities used and files produced during interface production: Fig.2-1, p.41.

(a) An ACF (Attribute Configuration File) file is optional. Such an file contains info that changes how IDL compiler interprets the IF definition.

(b) The files produced depend on the compiler options used.

i. Client stub file names: contain *_cstub* suffix;

ii. Server stub file names: contain *_sstub* suffix;

iii. Header file: end with .h;

iv. Object stub files: .o for both client and server. These object files are generated by C compiler called by IDL compiler.

v. Auxiliary files: *_caux* for client and *_saux* suffix for server. AUX files are generated when certain features are used. They contain special routines required for certain data types to prepare for data transmission. Features requiring AUX files include self-referencing pointers and pipes. Those routines are placed in AUX files rather than in the stub files so that the data types can be used in other IF definitions without linking the entire stub.

b. Generating client files: p.42

(a) -v option: verbose mode;

(b) -server none option: suppresses the generation of stub and auxiliary files for the server;

(c) -Iexplicit option: search the dir *explicit* for additional file for inclusion;

(d) -out explicit option: places the output file in the chosen directory *explicit*.

c. Generating server files

(a) -v option: verbose mode;

(b) -client none option: suppresses the generation of stub and auxiliary files for the client.

(3) Using an ACF to Customize Interface Usage

a. ACF files allows a user to control some aspects of RPC on the client side without affecting the server – These aspects are not in the IF definition because we do not want to force them on all clients and servers.

(a) A client or server developer can use an optional ACF file to modify the IDL compiler creates stubs, without changing the way the stubs interact across the network.

(b) Most significant effect: add additional parameters to remote procedures not declared in the IF definition: binding method selection (as 1st parameter of RP; status parameters to the end of parameter list.

(c) One does not specify ACF files when compiling an IF file. IDF compiler will automatically uses an ACF file if one is available in the search dir.

b. Selecting a binding method:

(a) The *auto_handle* attribute selects the automatic binding method. Additional advantage of this method is error recovery: a client stub sometimes can find another server transparently if the original server is unreachable.

(b) The *implicit_handle* attribute selects the implicit binding method. This allows a user to select a specific server for all the RPC calls

(c) The *explicit_handle* attribute selects the explicit binding method. This allows a user to select a specific server for each RPC.

(d) An example ACF file: Example 2-10, p.44.

i. The *implicit_handle* attribute applies to the entire IF. A global binding handle of type *handle_t* is established in the client stub to refer to binding info the client uses to find a server;

ii. The IF name *inventory* must match the IF name in the corresponding IDF file.

c. Controlling Errors and Exceptions

(a) DCE causes comm. and server errors to be raised as exceptions;

(b) The normal action in the face of an exception is that the server will exit;

(c) An ACF can allow you to provide extra code for exception handling: the attributes *comm_status* and *fault_status* can be applied to procedure parameters or return values. The presence of these attributes will allow comm. or server errors to be comm. to the client as values in the named parameters, rather than raised as an exception (Ch.3 has more coverage).

d. Excluding Unused Procedures

(a) The *code* and *nocode* ACF attributes allow users to define which procedures a client stub supports.

(b) However, all the procedures declared in the IF must be implemented by the server.

什么是IF

e. Controlling marshalling and unmarshalling

(a) The *out_of_line* ACF attribute causes constructed data types such as unions, pipes, and large structures to be marshalled or unmarshalled by auxiliary routines, thus reducing the stub size: the marshalling and unmarshalling code now is put into auxiliary files, rather than in the stub directly.

(b) The *in_line* ACF attribute causes data types to be marshalled or unmarshalled as fast as possible. The marshalling and unmarshalling code is put into the stub directly. This is the default action.

(c) These two attributes only affect the stub code.

3. Writing clients

(1) Binding

a. Implementing a binding method

(a) Automatic method: the client stub automatically manages binding handles after the application calls a RP. The client stub obtains binding info from a name service database and passes the binding handle to the RPC runtime library. If the connection is disrupted, new binding info can sometimes be automatically obtained and the call is tried again.

(b) Implicit method: a binding handle is held in a global area of the client stub. After the application calls a RP, the stub passes the binding handle to the RPC runtime library. One writes application code to obtain the binding info and set the global binding handle with RPC runtime routine calls.

(c) Explicit method: an individual RPC in the application passes a binding handle explicitly as its 1st parameter. One writes application code to obtain the binding info and set the binding handle with RPC runtime routine calls.

(d) Comparisons of binding methods: Fig.3-1,p.49

i. Top half: the client application code; bottom half: the client stub code generated by IDL compiler; Shadings represents the portion of the client where binding handles are managed.

ii. Different binding methods can be employed for different RPCs in a given client instance.

iii. Automatic binding is convenient for learning and test purposes. But is not useful for most production applications because one cannot use security.

iv. The auto and implicit methods apply to an entire interface. However, in these cases, one can still use the explicit method for some or all RPCs to that interface – the explicit method takes precedence over the other two methods because the binding handle is present as the 1st parameter in the procedure.

v. A client cannot use both auto and implicit methods simultaneously for RPCs to a single interface.

vi. The imp and exp methods requires that the application code obtain binding info and manage the binding handles. Binding handles need to be obtained and managed in the client app code under the following three circumstances:
i. The client uses a specific server;
ii. The client needs to set authentication and authorization info for specific binding handles;
iii. The server has more than one implementation of the same remote procedure (UUIDs are used to differentiate between difference RP implementations).

(e) One can use an ACF file to establish a binding method with attributes *auto_handle*, *implicit_handle*, or *explicit_handle*.

(f) *Context handle*: is a special RP parameter defined in an IF definition with the *context_handle* attribute. Applications use a context handle in a seq of RPCs to refer to a context (state) on a specific server. Context handles carry with them binding info and thus can act as binding handles in RPCs – when a context handle is active, it carries with it the binding info necessary to find the same server as it did before.

b. Deciding on binding methods

(a) When an IF definitions compiled, the auto binding method is the default unless:

i. The 1st parameter of a procedure declaration is a binding handle;

ii. The proc declaration has an input context handle;

· An ACF establish a different binding method.

(b) If the 1st parameter is a binding handle, that procedure must use the explicit method.

(c) For those procedures whose 1st parameters are not binding handles, one must decides whether to use the auto or implicit binding methods to all of them.

(d) The auto binding method only works well within a small network. However you have no control over which server you get, so applications that use servers scattered over a wide area may be inefficient.

(e) If most of the RPCs need to use a specific server chosen in one's application code, the implicit method is appropriate.

(f) The explicit binding method allows one to have more control over which server to use.

(g) The explicit method is necessary for clients that make multithreaded RPCs. Eg. a commodity trade application may request a commodity price with RPCs to many locations at the same time. Load is branched over the whole system in this case.

c. Automatic binding management

(a) Simplest.

i. One does not deal with the binding handle in the IF definition, ACF, or application code.

ii. The binding handle is hidden in the client stub. One only has to set up the environment variable for entry name and the rest is done entirely by the client stub and the runtime library.

iii. Rebinding is possible if a server connection is lost.

iv. Different servers implementing the same RP may be chosen at different call instances.

(b) Interface development for automatic binding: there are no special requirements in the interface for auto binding. If *auto_binding* attribute in an ACF can be used for documentation purpose.

(c) Client development for automatic binding: the client requires the user to:

i. Include the header file in the client application code;

ii. Link the client application object code with the client stub, client stub auxiliary file (if present), and DCE libraries.

iii. Set the environment variable *RPC_DEFAULT_ENTRY* to a valid name service entry so that the client stub can automatically begin a name service database search.

The RPC is just like a LPC.

(d) Server development for automatic binding

i. A server must advertise binding info to a name service entry with the *rpc_ns_binding_export* runtime routine in the server initialization code.

ii. Alternatively, an administrator can insert the info from the shell using DCE RPC control program (*rpccp*).

d. Implicit binding management

(a) It gives a user the control of binding management in the client application code without a visible binding handle parameter in the RPC.

i. Use this method for applications that need the same server for all or most RPCs of an IF.

ii. An ACF defines the binding handle and the IDL compiler generates it as a client-global variable in the client stub.

iii. The client application code sets the binding handle before any RPCs.

iv. During a RPC, the client stub uses the global binding handle to complete the call to the RPC runtime library.

(b) Interface development for implicit binding:

i. Use the *implicit_handle* attribute in an ACF to declare a global binding handle for the client: Example 3-1,p.54.

ii. When the IF definition is compiled with the ACF present, a global binding handle is defined in the client stub, which uses this handle every time the client calls a RP for this interface.

(c) Client development for implicit binding

i. Include the header file.

ii. The client must obtain the binding info and assign its handle to the global binding handle. The binding info can be from the name service database, or composed from strings of binding info. The *do_import_binding* routine is developed in later of this ch.

(d) Server development for implicit binding

i. No special requirements in server development

ii. A server must export to a name service database if the clients use a name service to find servers. Server binding info can also be exported by *rpccp* control program.

e. Explicit binding management

(a) This binding method manages each RPC separately. The 1st parameter of the RP is a binding handle.

i. Use this method if the clients need to make RPCs to more than one server. The application code has total binding info control.

ii. If a procedure declared in an IF has a binding handle as its 1st parameter, one must use the explicit method for this RP.

iii. If a procedure does not have a binding handle as its 1st parameter, one still can use an ACF add one.

(b) Interface development for explicit binding

i. An IF definition or an ACF uses the *handle_t* data type to define binding handle parameter;

ii. Application code uses the *rpc_binding_handle_t* data type to represent and manipulate binding info. These two data types are in fact equivalent. The former exists for compatibility with previous RPC versions.

iii. An example that uses an ACF file that adds a binding handle parameter: Example 3-3, p.56.
When the IDL compiler uses that ACF file, all procedure declarations in the header file have a binding handle of type *handle_t*, named *explicit_handle* added as the first parameter.
The header file generated by the IDL compiler contains a code (shown on p.56).
The *explicit_handle* attribute can also be added to a specific procedure in the ACF file to add a binding handle as the first parameter. Example: p.57

iv. Adding a binding handle in the IF definitions: Example 3-4, p.57.
This approach is more restrictive than a binding handle defined with a ACF file.
However, RPs can be used the same way either case

(c) Client development for explicit binding

i. Before making a RPC, the client must obtain binding info and set the binding handle. The methods of obtaining binding handles are almost the same as for implicit method.
For the explicit method, one uses a specific binding handle instead of assigning the binding info to the implicit global binding handle.

ii. Example 3-5, p.57.
Header file included
Binding handle variable declared
Obtaining binding handle
Notice the 1st parameter is a binding handle in the RPC

(d) Server development for explicit binding

i. If the IF definition does not have a binding handle parameter for the RP, and one wants the RP to obtain client binding handle (such as for authentication and authorization), one must use an ACF with *explicit_handle* attribute to create the binding handle.

ii. If clients use a name service to find servers, the server must export to a

name service database. Can also be done through control program *rpccp*.

(2) Steps in Finding Servers

a. Finding a protocol sequence: two possible methods:

(a) The preferred method: using a name service database to import or look up both host address and protocol sequence at the same time. Use RPC runtime routines that begin with *rpc_ns_binding_import_* or *rpc_ns_binding_lookup_*.

(b) Use a protocol sequence string obtained from your application or from a call to the *rpc_network_inq_protseq* routine. The two runtime routines *rpc_string_binding_compose* and *rpc_binding_from_string_binding* to prepare and set the binding handle.

b. Finding a server host: two different ways:

(a) Use a name service database to import or look up a host address and at the same time get a protocol sequence.

i. The RPC runtime routines that begin with *rpc_ns_binding_import_* or *rpc_ns_binding_lookup_* to set the binding handle.

ii. For automatic binding method, the client stub does this for you.

(b) Use a host name or host network address string obtained from your application. Use the RPC runtime routines *rpc_string_binding_compose* or *rpc_binding_from_string_binding* to the binding handle.

Partially bound binding handle: is one that contains a protocol sequence and server host, but not an endpoint.

(c) That is what one gets from CDS.

(d) When a partial bound binding handle is passed to the RPC runtime library, and endpoint is automatically obtained for you from the interface or the endpoint map of the server's system.

c. Finding an endpoint

(a) Several terms:

i. *Fully bound binding handle*: is one that contains info of a partial bound binding handle together with the endpoint info.

ii. *Well-known endpoint*: is a preassigned system address that a server process uses every time it runs. Usually a well-known endpoint is assigned by the authority responsible for a transport protocol.

iii. *Dynamic endpoint*: is a system address of a server process that is requested and assigned by the RPC runtime library when a server is initiated. Most applications should use dynamic endpoints to avoid the network management needed for well-known endpoints.

(b) Endpoints can be obtained through one's application code. But it is best left for the runtime library to obtain it. Four ways to obtain an endpoint:

   i. If the binding info obtained during an import or lookup of the protocol seq and host in the name service database includes an endpoint, the binding handle is fully bound in one step. However, a name service database only stores well-known endpoints, not dynamic endponts.

   ii. A well-known endpoint is found that was established in the interface definition in the *endpoint* attribute. The RPC runtime library or the application finds the endpoint from an interface specific structure.

   iii. An endpoint is found from the endpoint map on the server system. They can be well-known or dynamic. The runtime library first looks for an endpoint from the interface spec. If it cannot find it there, one has to use the routine *rpc_ep_resolve_binding* or routines beginning with *rpc_mgmt_ep_elt_inq_* in the application to find the endpoint.

   iv. One can use a string from the application that represents an endpoint, and the use the runtime routine *rpc_string_binding_compose* and *rpc_binding__from_string_binding* to set the binding handle. These endpoints can be well-known or dynamic.

d. Interpreting binding info: what goes on in the *do_import_binding* procedure shown earlier?

   (a) When using implicit or explicit binding, one needs to interpret the binding info. Example: one needs to use a server on a particular host, that means that one has to extract the host from the binding handles one gets from the CDS.

   (b) One may interpret the binding info of a binding handle to control which server a RPC will use or which binding handles a server will offer. binding handles offer the following binding info:

      i. Object UUID

      ii. Protocol seq

      iii. Network address or host name

      iv. Endpoint

      v. Network options (are specific to a protocol seq and not relevant to the connection-oriented or datagram protocol sequences).

   (c) Example 3-6, p.63: how to use RPC runtime routines to interpret binding info. These routines can be used in either clients or servers. The *do_interpret_binding* procedure is called in the *do_import_binding* procedure (next example).

      i. The example interprets binding info and extract and return protocol seq

      ii. The *rpc_binding_to_string_handle* routine converts binding info to its char string representation.

iii. The *rpc_string_binding_parse* routine obtains binding info items as separate allocated strings. The components include all five pieces of info in a binding handle.

iv. The *rpc_string_free* free strings allocated by other RPC runtime routines.

e. Finding a server from a name service database

(a) The usual way for a client to find binding info is from a name service database using the name service RPC runtime routines (routines beginning with *rpc_ns*). The name service database contains entries of info, some of which contain binding info about specific servers; some of them about a set of servers; some of them a search list.

(b) Importing a binding handle: Example 3-7, p.65, the *do_import_binding* routine

i. The *rpc_ns_binding_import_begin* routine establishes the beginning of a a search for binding info in a name service database. An entry name syntax of *rpc_c_ns_syntax_default* uses the syntax in the RPC-specific environment variable *RPC_DEFAULT_SYNTAX*.

ii. The starting search entry is */.:/inventory_group*, which is passed as a parameter. If a null string is passed as entry name, the search begins with the name in the environment variable *RPC_DEFAULT_ENTRY*.

iii. The IF handle *inventory_v1_0_c_ifspec* refers to the IF specification. It is generated by the IDL compiler and defined in the header file *inventory.h*.

iv. The *rpc_ns_binding_import_next* routine obtains binding info that supports the IF, if any exists. The routine only accesses the database and does not communicate with the server. The import handle, established with the call *rpc_ns_binding_import_begin*, controls the search for compatible binding handles.

v. Once binding info is obtained, the application code can perform any action it intends. In this example, the routine *do_interpret_binding*, which returns the protocol seq of a given binding handle, is called.

vi. Each call to *rpc_ns_binding_import_next* requires a corresponding call to the *rpc_binding_free* routine, which frees memory containing the binding info and sets the binding handle to null.

vii. The *rpc_ns_binding_import_done* routine signifies that a client has finished looking for a compatible server in the name service database. This routine frees the memory of the import context created by a call to *rpc_ns_binding_import_begin*. These two calls must be used in a pair.

(c) Looking up a set of binding handles

i. Runtime routines that begin with *rpc_ns_binding_lookup_* obtain a set of binding handles from the name service database.

ii. These set of binding handles can then be selected with the routine *rpc_ns_binding_select* or one's own selection criteria.

iii. Lookup routines give users a little more control than import routines because *rpc_ns_binding_import_next* returns a random binding handle from a list of compatible binding handles.

f. Finding a server from strings of binding data

(a) Without using the name service database, one has to to construct one's own binding info and binding handles. Binding info may be represented with strings. Binding handles may be composed from appropriate strings of binding info.

(b) Minimum info for an application to obtain a binding handle is:

- A protocol seq of comm. protocols
- A server network address or host name

Endpoints can be obtained by RPC runtime library. One can set a binding handle by calling RPC runtime routines with binding info.

(c) Example 3-8, p.67-69. The procedure *do_string_binding* sets binding handle from strings of binding info. This procedure us used by the *remote_file* application.

- The network address or host name of the the server system is required. Here it is passed as a char string parameter from the caller.
- The routine *rpc_network_inq_protseqs* creates a list of valid protocol sequences. Each of these protocol sequences is tried until a valid binding handle is created.
- The routine *rpc_string_binding_compose* creates a string of binding info in the argument *string_binding* from all the necessary binding info components.
- The *rpc_binding_from_string_binding* routine obtains a binding handle from the string of binding info, which can come from either the routine *rpc_string_binding_com* or *rpc_binding_to_string_binding*.
- When binding handle is finished its usage, the routine *rpc_binding_free* is called to set the binding handle to null, hence free its associated memory.
- The *rpc_string_free* routine frees strings allocated by other RPC runtime routines.
- The *rpc_protseq_vector_free* routine is called to free the list of protocol sequences. This call has to be used with the call *rpc_network_inq_protseqs* in pair.

(3) Customizing a Binding Handle

a. Several terms:

   (a) *Primitive binding handles*(PBH): those binding handles that we used so far: with five components;

   (b) *Customized binding handles*(CBH): handles obtained by adding to the PBH extra info that one's application wants to pass between client and server.

   CBHs can only be used with implicit or explicit binding method.

b. Why CBHs?

   (a) A CBH can be used when application-specific data is appropriate to use for finding a server, and the data also is needed as a procedure parameter.

   (b) Example: in the *transfer_data* application, a structure contains a host name and a remote file name. The application creates the necessary binding info from the host name, and the filename is passed with the binding info so the server knows what data to use.

c. How a customized binding handle works: Fig.3-2,p.70

   (a) A CBH is defined with the *handle* attribute applied in a type def in an IF def.

   (b) A CBH can be used the same way as a PBH. But one has to write special *bind* and *unbind* procedures. These two procedures will be called from client stubs, not the application code. These two are used by the client stub to obtain the PBH and do application cleanup when finished with the binding handle.

   (c) A CBH must be the 1st parameter in a RPC (or the global handle for the implicit method) to act as the binding handle for the call. A CBH no present as the 1st parameter of a RPC acts as a standard parameter.

d. The *transfer_data* example: Example 3-9, p.71-72

   (a) The *handle* attribute in the IF def is used to associate the CBH with a data type;

   (b) The *file_spec* data type is a structure whose members are file specifications. This is application-specific info used by the bind procedure to obtain server binding info.

   (c) The CBH is the first parameter of a procedure declaration because explicit binding method is used.

e. Implement the *bind* and *unbind* procedures used in *transfer_data*: Example 3-10, p.72.

   (a) The bind and unbind procedure names are constructed from the data type name *file_spec*, with suffix *_bind* and *_unbind* respectively.

(b) The bind procedure takes an input parameter of the customized handle data type, and returns a primitive binding handle. It calls the application-specific procedure *do_string_binding* (defined in Example 3-8) to obtain a primitive binding handle.

(c) The unbind procedure *file_spec_unbind* takes input parameters of the customized handle data type and a primitive binding handle. It calls the RPC runtime routine *rpc_binding_free* to free the binding handle.

f. A client with a CBH: Example 3-11, p.73.

(a) The application allocates a CBH in the def.

(b) The CBH variable is initialized before the RPC is made.

(c) The RPC is called with the CBH as the 1st parameter.

(4) Error Parameters or Exceptions

a. DCE RPC clients and servers require special error handling techniques to deal with errors that may occur during a RPC.

(a) Server and comm. errors are raised to the client as exceptions during a RPC. RPC exceptions are equivalent to the RPC error status codes.

(b) Types of errors include:

i. Exceptions raised on the client system (Eg. client system is out of memory);

ii. Exceptions raised to the client application by the client stub due to comm. errors (Eg. crashes on the remote system);

iii. Exceptions raised by the client stub on behalf of the server. These errors can occur in server stub, in the RP, in pipe support routines, or in the server's RPC runtime routines.

b. Using exception handlers in clients or servers: One can handle exceptions by writing exception handler code in the application to recover from an error or gracefully exit the application. DCE threads supply macros as framework to handle exceptions in one's client or server code.

c. Using remote procedure parameters to handle errors

(a) One can add error parameters to remote procedures in the ACF to conveniently handle comm. and server errors. The RPC runtime library then stores error values in these parameters rather than raising exceptions. One can use a combination of exception handlers and error parameters.

(b) For the *is_part_available* procedure: p.75, a statement in an ACF file shown on p.75 establishes an error parameter for that procedure.

i. The ACF attributes *comm_status* and *fault_status* establish a parameter in which to report comm. and server errors.

ii. These attributes can be associated with one or more parameters.

(c) In a client, an RPC to this procedure now has an additional parameter – the last parameter.

(d) The error code returned can be interpreted by the RPC runtime routine *dce_error_inq_text*. The application-specific macro *CHECK_ERROR* (Example 3-12, p.76) uses this routine.

(5) Compiling and Linking Clients

a. Files and libraries required to produce an executable client: Fig.3-3, p.78.

(a) Client auxiliary file is generated only when certain features are present:

i. Self-referencing pointers in IF def.

ii. The *out_of_line* attribute in an ACF.

(b) Compile the client modules with C compiler

(c) Link the client application, client stub, and client auxiliary object files (if any) with the DCE libraries to produce the executable client

b. Local testing: one can compile a local version of the client to test and debug RPs without using RPCs.

(a) Compile the client object files and remote procedure implementations without the stub or auxiliary files.

(b) A compilation directive -DLOCAL is used in this textbook to distinguish a test compilation.

(c) The object files produced in test compilation should be deleted to prevent it from interfering with the executable version.

4. Pointers and arrays

(1) kind of Pointers

a. Pointers: A pointer is a variable containing the address of another data structure or variable.

b. Pointer declaration in IDL: the same as in C.

c. Two types of pointers:

(a) Full pointers: a full pointer has all the capabilities usually associated with pointers. IF def. requires full pointers for the following cases:

i. When a RPC allocates new memory for the client. The client stub usually allocates the memory;

ii. When the value of the pointer is NULL, as with an optional parameter;

iii. When two pointers refer to the same address, as in a doubly-linked list.

Full pointers are specified using the *ptr* attribute in an IF def. Full pointer incurs significant stub overhead.

(b) Reference pointers: a reference pointer is used to refer to existing data. A reference pointer has performance advantage over a full pointer because stub overhead is reduced.

i. For reference pointers, the data pointed by such a pointer can change, but the value of the pointer itself cannot.

ii. Reference pointers are specified using the *ref* attribute in an IF def.

(c) Reference pointers

d. Pointers as output parameters

e. Pointers as input parameters

f. Using pointers to pointers for new output

g. Pointers as procedure return values

h. Pointers summary: Table 4-1, p.88

(2) Kind of Arrays

a. Three types of arrays

(a) *Fixed arrays*:

(b) *Varying arrays*:

(c) *Conformant arrays*:

b. Selecting a portion of an array

c. Managing the size of a conformant array

d. Conformant arrays as procedure parameters

e. Dynamic memory allocation for conformant arrays

f. Memory allocation for conformant structures

5. Writing servers

(1) Some Background on Call Processing

a. Fig.1-9 shows how a server is initialized before it accepts and processes RP calls.

b. Fig.1-10 shows how a client finds a server using automatic binding method.

c. Fig.1-11 shows the basic steps during a remote procedure call after the client finds the server.

d. How the RPC runtime library handles an incoming call: Fig.5-1, p.98:

(a) A call request for the server comes in over the network.

i. The request is placed in a request queue for the endpoint;

ii. The server can select more than one protocol seq on which to listen for calls, and each protocol seq can have more than one endpoint associated with it;

iii. Request queues temporarily store all requests, thus allowing multiple requests on an endpoint;

iv. Requests are rejected if the request queue is full.

(b) The RPC runtime library dequeues requests on at a time from all request queues and places them in a single call queue.

i. Using threads, the server can process RPCs concurrently. The number of threads can be determined at server initialization.

ii. If all threads are in use, the call remains in the call queue until one thread is available.

(c) After a call is assigned to a thread, the IF spec of the client call is compared with the IF spec of the server.

i. An IF spec is an opaque data structure containing info (including the UUID and the version number) that identifies the interface. *Opaque* here simply means that the details are hidden from you.

ii. If the server supports the client's IF, processing goes to the stub code. Otherwise, the call is rejected.

(d) When the call reaches the stub, it unmarshalls the input data. During unmarshalling, memory is allocated if necessary, data is copied from the RPC runtime library, and data conversion is performed if needed.

(2) Initializing the Server

a. Steps in the initialization code:

(a) Register the IF with the RPC runtime library

(b) Create server binding info by selecting one or more protocol seqs for the RPC runtime library to use in one's network environment.

(c) Advertise the server location so the clients have a way to find it. This usually involves storing binding info in a name service database.

(d) Manage endpoints in a local endpoint map.

(e) Listen for RPCs.

b. Example of server initialization of the inventory application: Example 5-1, p.100:

(a) The header file is always included.

(b) A *unsigned32* type variable is needed to report errors that may occur when an RPC runtime routine is called.

(c) Some RPC runtime routines used data structures called a *vector*, which in RPC applications contains a list (array) of other data structures and a count of elements in the list.

c. Registering IFs

(a) Before a client makes a RPC, it checks its IF against the one advertised in the server's binding info. The server will also compare its IF against the IF in the client's call.

(b) The application code uses an IF handle to refer to the IF spec. An IF handle is a pointer defined in the C header file generated by IDL compiler and contains the following four pieces of info:

i. The IF name;

ii. Version number (both major and minor);

iii. The letter *s* or *c*, depending upon whether the handle is for the server or client portion of the application;

iv. The word *ifspec*.

(c) Example 5-2: Registering an Interface with Runtime Library, p.101-102.

i. The *rpc_server_register_if* routine is required to register each IF that the server supports. The IF handle refers the the IF specification;

ii. The *CHECK_STATUS* macro is defined previously.

(d) Multiple IFs may be registered from a single server by calling *rpc_server_register_if* with a different IF handle.

(e) The 2nd and 3rd parameters of *rpc_server_register_if* are used in complex applications to register more than one implementation for the set of remote procedures. A NULL value for these two parameters means only one implementation.

(f) In the event of a symbol name conflict between the RP names of an IF and other symbols in the server, one can use these parameters to assign different names to the server code's remote procedures.

d. Creating server binding info

(a) Server binding info is created when one selects protocol seqs during server initialization. RPC uses protocol sequences to identify the combinations of comm. protocols that RPC supports.

i. Most servers offer all available protocol sequences to facilitate clients' connections.

ii. One may want use only one specific protocol sequence for debugging.

iii. The server address info is furnished by the system kernel itself.

An endpoint must be selected after protocol seqs are determined.

(b) Using dynamic endpoints: most servers use this type of endpoints.

i. Flexible;

· Selected by the RPC runtime routine and vary from one invocation to another.

ii. Example 5-3: a portion of the inventory server initialization:

• The *rpc_server_use_protseq* routine is called with the chosen prot. seq as the 1st parameter. This call selects one prot seq on which the server listens for RPCs. The constant *rpc_c_protseq_max_calls_default* sets the request queue size. At this time, the size is system dependent and cannot be controlled by users.

• The *rpc_server_use_all_protseqs* routine selects all available prot seqs.

• The *rpc_server_inq_bindings* routine is required to obtain the set of binding handles referring to all of this server's binding info.

(c) Using well-know endpoints

i. A well-know endpoint is selected and assigned to the same server every time it runs. They are more restrictive. Users must register them to certain Internet authority.

ii. Well-known endpoints are often used for widely-used applications such as core DCE servers like *rpcd*, which servers to endpoints.

iii. RPC routines for creating binding info with well-known endpoints: Table 5-1,p.105.

• *rpc_server_use_protseq_ep* takes as parameters a specified prot seq and a well-known endpoint to establish server binding info.

• *rpc_server_use_protseq_if* takes as parameter a specified prot seq, but the well-known endpoint is given in the IF def with *endpoint* attribute. Both clients and servers know the well-known endpoint from the IF def.

• *rpc_server_use_protseq_if* use all supported prot seqs, but the well-known endpoint is given in the IF def with *endpoint* attribute.

e. Advertising the server

(a) Advertising: making the binding info available for clients to find this server.

i. Three methods to advertise a server:

(i) Export to a name server database (most commonly used); (ii) Store binding info in an application-specific database; (iii) Print or display binding info for clients.

Which one to use is application dependent.

ii. An RPC entry name includes the IF name and remote host name. This means that the client has to know the specific server entry name before making an RPC.

iii. A name service group associates a general group name with a set of server entries, so that a client does not have to know specific host names.

iv. The convention for naming RPC group entries includes the the IF name. The server name is added as a member of the group.

v. More sophisticated naming approach is discussed in Ch.6.

(b) Export to a name server database: most commonly used. Example 5-4, p.106-107.

i. The RPC runtime routine *rpc_ns_binding_export* exports the server binding info to a name service database.

ii. The constant *rpc_c_ns_syntax_default* establishes the syntax the RPC runtime library uses to interpret an entry name.

iii. The IF handle *inventory_v1_0_s_ifspec* is needed so that IF info is associated with the binding info in the name service database.

iv. The *binding_vector* parameter contains the list of binding handles that the binding info exported.

v. The RPC runtime routine *rpc_ns_group_mbr_add* adds the server entry exported with the *rpc_ns_binding_export* call as a member of the name service group /.:/inventory_group.

(c) Three ways to manipulate the name service database:

i. Use the *rpc_ns_binding_export* routine together with other RPC runtime routines in the server initialization code (previous example);

ii. Use the RPC control program *rpccp* to export binding info;

iii. Use the *rpc_ns_mgmt_entry_create* routine together with other RPC runtime routines in a separate management application (not covered in this book).

(d) The routine *rpc_ns_binding_export* exports well-known endpoints together with other binding info to the name service database. Dynamic endpoints are not exported by this routine.

i. They are short-living in nature;

ii. May degrade the performance of the name service database if many obsolete dynamic endpoints exist;

iii. May cause client applications to fail if obsolete endpoints are used.

f. Managing server endpoints

(a) *Endpoint map*: a database on each RPC server system that associated endpoints with other server binding info. Normally the endpoint map contains all the endpoints (well-known or dynamic) on the server system.

(b) The endpoint map on a server system is managed by the *rpcd* daemon process. Applications use RPC runtime routines. The *rpccp* program can also interact with *rpcd*.

(c) With a partially bound binding handle, the client will do:

   i. If there is a well-known endpoint given in the IF def, that endpoint will be used;

   ii. Otherwise it will contact the server system *rpcd* daemon process to obtain an endpoint.

   iii. The daemon process will compare the info in the endpoint map with the client's IF specification and binding info.

   iv. When an valid endpoint is returned, the client has a fully bound binding handle to complete the connection.

(d) Example 5-5, p.109: Managing endpoints in an endpoint map: how to register its endpoints in the endpoint map.

   i. The *rpc_ep_register* routine registers the server endpoints in the local endpoint map. Use the same IF handle, binding vector, and object UUID as using *rpc_ns_binding_export* routine. Annotation is encouraged.

   ii. The *rpc_binding_vector_free* routine is used in pair with the *rpc_server_inq_bindings* routine (Example 5-3). This call should be made before put the server in listening mode.

(e) Some more notes:

   i. The *rpc_ep_register* routine is required if dynamic endpoints are established with the *rpc_server_use_protseq* or *rpc_server_use_all_progseqs* runtime routines.

   ii. If a well-known endpoint is established with the *rpc_server_use_protseq* routine, one also has to use the *rpc_ep_register* routine so that the client can find the value of the endpoint.

(f) Outdated endpoints are removed by *rpcd* daemon.

   i. An unpredictable amount of time exists before an outdated endpoint is removed from the endpoint map.

   ii. A server can use the routine *rpc_ep_unregister* to explicitly remove an endpoint before it terminates.

(g) Three ways to manage the endpoints in the endpoint map:

   i. Use the *rpc_ep_register* and *rpc_ep_unregister* routines;

   ii. Use the *rpccp* program;

   iii. Use the *rpc_mgmt_ep_elt_inq_begin* routine and other RPC runtime routines in a separate management application.

g. Listening for RPCs

(a) The *rpc_server_listen* routine puts the server in listen mode and will not return unless the server is requested to stop listening by another process, by one of its own RPs using the *rpc_mgmt_stop_server_listening* routine.

(b) When an error occurs during stub or remote procedure code execution, an exception is raised. The server will immediately terminate unless the user provides code to handle it.

(c) The macros *TRY, FINALLY*, and *ENDTRY* delineate code sections in which exceptions are controlled:

i. If an exception occurs in the *TRY* section, code in the *FINALLY* section is executed to handle any necessary error recovery or cleanup;

ii. The *FINALLY* section contains clean-up code that does things as remove outdated endpoints;

iii. The *TRY* + 1 and *FINALLY* sections both end with the *ENDTRY* macro.

(d) Example: 5-6: listening for RPCs, p.111

i. The *TRY* macro begins a section of code the one expects exceptions to occur. In this example, it contains only one call.

ii. The *rpc_server_listen* routine is a required. The first parameter sets the number of threads that the RPC runtime library to process RPCs.

iii. The *rpc_server_inq_bindings* routine obtain a set of binding handles referring to all of the server's binding info.

iv. The *rpc_ep_unregister* routine removes the server endpoints from the local endpoint map.

v. The *rpc_binding_vector_free* routine is called in pair with the routine *rpc_server_inq_bindings*.

(3) Writing Remote Procedures

a. Issues in writing RPs:

(a) Memory management

i. Is especially important because the client and the RP are not in the same address space. Memory requested to be allocated by the client cannot be freed by the client.

ii. Memory management for RPs is done through the special *stub support routines* in RPs. Stub support routines enable the server stub to free memory allocated in RPs after the RPs finish their execution.

(b) Threads

i. The number of threads that a server uses to process RPCs is decided by the 1st parameter to the routine *rpc_server_listen*.

ii. *Thread safe*: multiple threads of the same RP will not cause unexpected or inconsistent results.

iii. The implementation of a RP should be thread safe if more than one thread is allowed for the RP. Locks may be used in the implementation.

(c) Client binding handle: *Client binding info* is used in server code to inquire and authenticate the client. This info includes:

i. The object UUID requested by the client; Can be simply nil (more to be discussed in Ch.7);

ii. The RPC protocol sequence used by the client for the call;

iii. The network address of the client;

iv. The optional client authentication and authorization information.

Client binding info is stored in the *Client binding handle*. If the client binding handle is available, it is the 1st parameter of the RP. If one requires CBI, and the procedure declarations in the IF def. do not have a binding handle as the 1st parameter, one must generate the server stub and header file using an ACF with the *explicit_handle* attribute.

b. Managing memory in remote procedures

(a) Typical C applications use *malloc* and *free* to allocate and deallocate memory.

(b) In RPC servers, if a remote procedure returns a pointer to newly allocated memory to the client, use stub support routines to manage memory in the RP.

(c) Use the RPC routine *rpc_ss_allocate*, not the C routine *malloc*. The former allows the RPC runtime library to do bookkeeping about memory management. It also ensures that the allocated memory will be automatically freed after the remote procedure completes.

(d) For ref pointers, memory on the client side must already exists, and hence no memory management required for RPs whose output parameters are ref pointers:

i. After one makes the RPC, first the server stub automatically allocates necessary memory and copies the data for the ref pointers into the new memory;

ii. Then it calls the implementation of the RP.

iii. Finally, the RP completes, output data is transmitted back to the client stub and the server stub frees the memory it allocated.

(e) Full pointers need more complex memory management on both clients and servers.

i. If a RP allocates memory for an output parameter, the server stub copies and marshalls the data. After completion of the RPC, the stub frees the memory that was allocated in the RP.

ii. When the client receives the data, the client stub allocates memory and copies the data into the new memory. It is the client application's responsibility to free the memory allocated by the client stub.

(f) Example 5-7: Memory management in RPs, p.114

i. The additional char is needed to accommodate the null char;

ii. The RP calls the stub support routine *rpc_ss_allocate* to allocate memory in the RP.

iii. When the RP finishes, the server stub automatically frees memory allocated by *rpc_ss_allocate*.

iv. When the RP returns to the client, the client stub automatically allocates memory for the returned string.

v. The client application code must explicitly free the memory allocated by the client stub.

vi. Stub support routine *rpc_ss_free* is more sophisticated than *free*.

(g) Memory used by context handles will not be allocated or freed with *rpc_ss_allocate* or *rpc_ss_free*.

c. Allocating memory for conformant arrays

* The *whatare_subparts* procedure of the inventory application allocates memory for a conformant array in a structure, and returns a copy of the conformant structure to the client.
* That procedure is shown on p.115.

i. The output parameters are ref pointers, which must have memory allocated in the client prior to the call.

ii. Therefore we have to use a full pointer in order for new memory to be automatically allocated by client stub for the ∗∗*subparts* structure where *whatare_subparts* procedure returns.

iii. A pointer to pointer is required so that the ref pointer points to a full pointer, which in turn points the structure.

* Example 5-8: Conformant array allocation in a RP,p.116

i. The allocated memory includes the size of the conformant array plus enough memory for all the elements of the conformant array. The conformant structure generated by the IDL compiler already has an array of one element, so the new memory allocated for the array elements is one less than the number in the array;

ii. Use RPC stub support routine *rpc_ss_allocate* to allocate memory so bookkeeping is maintained for memory management and the server stub will free the memory automatically after the RPC completes.

iii. When the data for conformant structure returns to the client, the client stub allocates memory and copies the data into the new memory. The client application code uses the data and frees the memory (p.117, top).

(4) Compiling and linking servers

a. Files and libraries used to produce a server: Fig.5-2,p.117

b. Server auxiliary files: caused by the following features in an IF:

* Self-referencing pointers in the IF def;
* The *pipe* data type in the IF def;
* The *out_of_line* attribute in an ACF.

c. Compile all server application modules with the C compiler;

d. Link the server application and server stub object files with the DCE libraries to produce the executable server.

6. Using a name service

(1) Naming in DCE

(2) Environment Variables

(3) Server Entries

(4) Group Entries

(5) Profile Entries

III. Java RMI

1. Introduction

(1) Java RMI is an example implementation of the RMI concept presented in last lecture.

a. As an RMI, Java RMI extends the RPC mechanism to provide RMI for distributed objects.

b. Java supports basic TCP/IP networking API, as discussed in Lecture 5. Java RMI is a higher level API than Java TCP/IP API, just like SUN RPC and DCE RPC are higher level API than Berkeley socket TCP/IP API.

c. Java RMI (since JDK 1.1) was designed with the following main goals:

(a) Support seamless remote invocation on objects in different JVM.

(b) Support callbacks from servers to applets.

(c) Integrate the distributed object model into the Java programming language in a natural way while retaining most of the Java language's object semantics.

(d) Provide a transparent interface of distributed object model and make writing reliable distributed applications as simple as possible.

(e) Maintain the safe environment of the Java platform provided by security managers and class loaders.

d. Like SUN RPC which provides a compiler *rpcgen*, Java RMI provides a RMI compiler *rmic* that will generate *stub* files.

(2) Preliminaries of Java RMI

a. Overview of Java RMI interfaces and classes (Fig.5.21, p.224)

b. In Java RMI, a remote interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine. Java RMI's remote interface package is java.rmi.Remote.

(a) The special interface java.rmi.Remote is a so called *tagging interface*. It has no methods declared inside and it will *tag* any interface that extends it as *remote*, meaning that methods in that interface become remote methods.

(b) Each remote method must be defined in an *remote interface* that *extends* the *Remote* interface in java.rmi.∗.

(c) Example: the BankAccount interface.

```
import java.rmi.*

public interface BankAccount extends Remote {
   public void deposit(float amount)
```

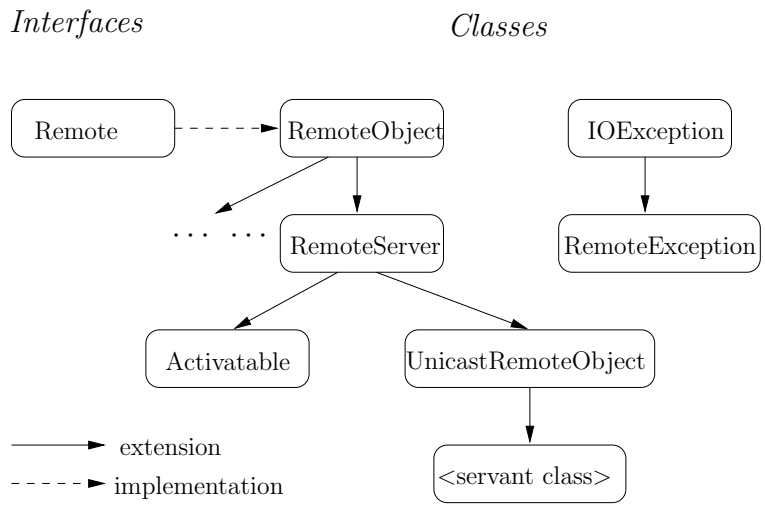


Figure 41: Classes supporting Java RMI

```
            throws RemoteException;
    public void withdraw(float amount)
            throws OverdrawnException, RemoteException;
    public float getBalance()
            throws RemoteException;
}
```

c. A remote interface only provides definitions of a set of methods that are remotely invocable. Each remote interface must be implemented by a remote object that either *extends* the class *UnicastRemoteObject* or the class *Activatable*:

   (a) The java.rmi.server.UnicastRemoteObject class defines a singleton (unicast) remote object whose references are valid only while the server process is alive.

   (b) The class java.rmi.activation.Activatable is an abstract class that defines an activatable remote object that starts executing when its remote methods are invoked and can shut itself down when necessary.

d. Each remote object may support several remote methods (servers).

   谁是remote object

   (a) Each remote method must be part of an interface that extends java.rmi.Remote

   (b) Each remote method in a remote interface must have a *throws* clause that indicates possibilities of a *RemoteException*.

   (c) The class of *RemoteException* is from the package java.IOException

e. Each RMI supporting computer runs a *RMIrgistrtry* (with executable name *rmiregistry*) daemon process at port 1099. This port number is the default (i.e. a well-known port number) for RMI service requests. Each remote server must *bind* its remote object name with its server object instance in RMIregistry.

f. Remote server object names in RMI are specified in the following format:

$$//\text{Hostname}[:\text{Port}]/\text{RemoteServer}$$

which is similar to the URL format.

(a) The *Hostname* component specifies the IP name of the computer hosting an remote server object. The *Port* component is optional.

(b) *RemoteServer* is the name of the remote server object.

(c) Both the remote server and an RMI client must specify the remote server object name in this format.

i. A remote server object declares this name using a construct like:

$$Naming.bind(\text{"//Hostname/RemoteServer", obj})$$ 如果已经绑定过了会报错

or

$$Naming.rebind(\text{"//Hostname/RemoteServer", obj})$$ 覆盖以前的绑定如果已经存在

to bind the remote object name to a remote object in rmiregistry.

ii. A remote client uses the *Naming.lookup* method to locate the remote server object:

$$Naming.lookup(\text{"//Hostname/RemoteServer"})$$

2. Tasks in writing an RMI client-server distributed application (from Java RMI manual):

(1) Three tasks in writing an RMI client-server distributed application (from Java RMI manual)

a. Define an RMI interface that lists functions (methods) of the remote class

b. Write the implementation and server classes

c. Write a client program that uses the remote services

(2) Define a RMI interface

a. In creating a client-server distributed application using RMI, the first step is to define a *remote interface* that describes the *remote methods* that the client can use to access *remote objects* managed by the remote methods in the remote server through RMI.

b. This is done by declaring the interface under design as **Remote**. Example 1:

```
package examples.hello;

import java.rmi.Remote;
```

```
import java.rmi.RemoteException;

public interface Hello extends Remote {
   String sayHello() throws RemoteException;
}
```

(3) Write the implementation and server classes. A remote object implementation should contain at least three parts:

a. Implementation of a remote interface previously defined

b. Definition of constructor(s) for the remote object

c. Implementation of each remote method

```
public class HelloImpl extends UnicastRemoteObject
   implements Hello {

   public HelloImpl() throws RemoteException {
      super();
   }

   public String sayHello() throws RemoteException {
      return "Hello World!";
   }

   public static void main (String args[]) throws Exception {

      System.err.println ("Initializing hello server, please wait ...");

      // create an instance of the remote object
      HelloImpl obj = new HelloImpl();

      // bind HelloImpl object to rmiregistry
      Naming.rebind("//myhost/HelloServer", obj);
   }
}
```

(4) Write a client program that uses the remote services

```
package examples.hello;
```

```
import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {

   String message = "blank";

   // "obj" is the identifier that we'll use to refer
   // to the remote object that implements the "Hello"
   // interface
   Hello obj = null;

   public void init() {
      try {
         obj = (Hello)Naming.lookup("//" +
                getCodeBase().getHost() + "/HelloServer");
         message = obj.sayHello();
      } catch (Exception e) {
         System.out.println("HelloApplet exception: " + e.getMessage());
         e.printStackTrace();
      }
   }

   public void paint(Graphics g) {
      g.drawString(message, 25, 50);
   }
}
```

(5) Compile/run the server and client: see the on-line manual for Java RMI for details and explanations as how to compile and run the program.

3. Example 2: the "ShapeList" RMI in Sect.5.5.1 of textbook

(1) The Java interface *Shape* and *ShapeList* (Fig.5.12, p.209)

```
import java.rmi.*;
import java.util.Vector;
```

```
public interface Shape extends Remote {
   int getVersion() throws RemoteException;
   GraphicalObject getAllState() throws RemoteException;
}

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

a. Remote interfaces are defined by extending an interface called *Remote* Provided by *java.rmi* package. Each remote method listed in a remote interface must throw *RemoteException*, and it can throw other application related exceptions as well.

b. The *Shape* interface

   (a) GraphicalObject is a class that holds the state of a graphical object (its type, position, enclosing rectangle, line color, fill color) and provides operations for accessing and updating its state.

   (b) GraphicalObject must implement the *Serializable* interface as it is used in RMI.

   (c) The method *getVersion* returns an integer version number of the interface, while the method *getAllState* returns an instance of the *GraphicalObject* class.

c. The *ShapeList* interface

   (a) The only new remote method in this interface is *newShape*, which defines a new shape of a graphical object. This method requires an instance of the *GraphicalObject* as argument and returns an object with a remote interface (ie a remote object).

d. Both ordinary and remote objects can appear as arguments or results in a remote interface. In the latter case (when a remote object is used as return value), the remote object return value is always denoted by their remote interface.

(2) The server Java class *ShapeListServer* of with main method (Fig.5.18, p.220).

```
import java.rmi.*;
public class ShapeListServer {
   public static void main(String args[]){
```

```
        System.setSecurityManager(new RMISecurityManager());
        System.out.println("Main OK");
        try{
           ShapeList aShapelist = new ShapeListServant();
           Naming.rebind("ShapeList", aShapelist);
           System.out.println("ShapeList server ready");
        }catch(Exception e) {
           System.out.println("ShapeList server main " + e.getMessage());}
     }
  }
```

(3) The *ShapeListServant* class that implements *ShapeList* interface (Fig.5.19, p.221).

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

   public ShapeListServant()throws RemoteException{
      theList = new Vector();
      version = 0;
   }

   public Shape newShape(GraphicalObject g) throws RemoteException{
      version++;
      Shape s = new ShapeServant(g, version);
      theList.addElement(s);
      return s;
   }

   public  Vector allShapes()throws RemoteException{
      return theList;
   }

   public int getVersion() throws RemoteException{
        return version;
   }
```

```
}
```

(4) The Java client class that invokes methods in interface *ShapeList* (Fig.5.20, p.222).

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

public class ShapeListClient{
   public static void main(String args[]){
      System.setSecurityManager(new RMISecurityManager());
      ShapeList aShapeList = null;
      try{
         aShapeList =(ShapeList)
                   Naming.lookup("//bruno.ShapeList");
         Vector sList = aShapeList.allShapes();
      } catch(RemoteException e)
         {System.out.println("allShapes: " + e.getMessage());
      } catch(Exception e)
         {System.out.println("Client: " + e.getMessage());}
   }
}
```

4. Example 3: the "Weather information server"

(1) Program description. Copyright notice: this example is from Chapter 13 of Prentice-Hall's book *Advanced Java 2 Platform: How to Program* by Deitel, Deitel, & Santry , ISBN 0-13-089560-1, 2002.
The example has been updated for Java 1.6.

a. The server, `WeatherServiceServer`, upon startup, will contact the HTTP server at

http://www.weather.gov/view/national.php?prodtype=tempprecip

to obtain the weather information.

b. A client will first contact the server obtain the weather information. It then displays the information in a window with icons for different types of weather conditions.

(2) Interface definition for interface *WeatherService* (Fig.13.1, p.793).

a. As discussed previously, the interface **WeatherService** has to extend the tagged interface **Remote**;

b. The interface declares the only remote method **getWeatherInformation**;

c. As discussed earlier, the interface does not provide the implementation of the remote method.

```
1  // WeatherService.java
2  // WeatherService interface declares a method for obtaining
3  // wether information.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.rmi.*;
8  import java.util.*;
9
10 public interface WeatherService extends Remote {
11
12    // obtain Vector of WeatherBean objects from server
13    public List getWeatherInformation() throws RemoteException;
14
15 }
```

(3) The server implementation class **WeatherServiceServer** that implements the remote interface (Fig.13.2, p.794-796)

a. The server implementation class *extends* **UnicastRemoteObject** (from package **java.rmi.server**). The class **UnicastRemoteObject** provides the basic functions required for all remote objects:

(a) The constructors of a class that extends **UnicastRemoteObject** *exports* the remote object to make it available to receive remote invocations;

(b) *Exporting* the object enables the remote object to wait for client requests on an anonymous port number (chosen internally by the kernel of the client computer). This enables the remote object to communicate with any client through *unicast communication* (i.e. point-to-point communication);

(c) The Java RMI implementation abstracts away all the above communication details from programmers.

· The constructor (Lines 19-23) invokes the default constructor for class **UnicastRemoteObject** at line 21.

· The class **UnicastRemoteObject** has overloaded constructors that allow programmers to specifiy additional information such as an explicit port number on which to export the remote object.

(d) All **UnicastRemoteObject** constructors must throw a **RemoteException** (line 19).

(e) constructor invokes the private method **updateWeatherConditions** (line 22) to obtain the weather conditions for the current running.

b. The service only contact the HTTP server once at start up: Line 118, the invocation of the class constructor, which in turn calls the method **updateWeatherConditions** at line 22.

c. The method **updateWeatherConditions** (lines 26-91) contacts the NWS web site, reads weather information, and stores the weather information in a **List** of **WeatherBean** objects:

(a) Lines 32-33 creates a **URL** object for the *Traveler's Forcast* web page.

(b) Lines 36-37 invoke the **openStream** method of the **URL** class to open a connection to the specified **URL** and wrap that connection in an object of class **BufferedReader**.

(c) The server depends on the structure of the HTML page to fetch the needed information: Lines 40-59, and the while loop at lines 70-85.

(d) Lines 65-85 read each city's weather information and place it in **WeatherBean** objects:

- Each **WeatherBean** object contains three pieces of information: the city name, the tempature, and the description of weather condition.
- Line 61 creates a **List** object for storing **WeatherBean** objects.
- Lines 76-79 construct a **WeatherBean** object for the current city.
- Line 82 adds the new **WeatherBean** object to the **List**.

d. The method **getWeatherInformation** (lines 107-110) implements the remote method listed in the remote interface. This method is invoked in response to every RMI request.

d. The **main** method (lines 113-127):

(a) It first creates the **WeatherServiceServer** object (line 118). The creation of this object exports the remote object.

(b) Line 121 creates a URL for the remote object:

rmi://localhost/WeatherService

which is the same as

rmi://127.0.0.1/WeatherService

(c) Line 124 invokes the **static** method **rebind** of class **naming** to bind the a remote object to the Java RMI registry.

The **rebind** method ensures that if the remote object has already been registered under the given name, the new remote object will replace the previoulsy registered name.

```
// WeatherServiceServer.java
// WeatherServiceServer implements the WeatherService remote
// interface to provide a WeatherService remote object.
// package com.deitel.advjhtp1.rmi.weather;

// Java core packages
1   import java.io.*;
2   import java.net.URL;
3   import java.rmi.*;
4   import java.rmi.server.*;
5   import java.util.*;

// JDK1.5
6   import java.rmi.registry.LocateRegistry;
7   import java.rmi.registry.Registry;

8   public class WeatherServiceServer implements WeatherService {

9   public WeatherServiceServer ()
10  {
11      super();
12      updateWeatherConditions();
13  };

14  private List weatherInformation;  // WeatherBean objects

    // get weather information from NWS
15  private void updateWeatherConditions()
16  {
17      try {
18         System.err.println( "Update weather information..." );

         // National Weather Service Travelers Forecast page
19         URL url = new URL(
            //"http://iwin.nws.noaa.gov/iwin/us/traveler.html" );
20            "http://www.weather.gov/view/prodsByState.php?state=TX&prodtype=hourly" );

         // set up text input stream to read Web page contents
21         BufferedReader in = new BufferedReader(
```

```
            new InputStreamReader( url.openStream() ) );

         // helps determine starting point of data on Web page
         //String separator = "TAV10";
         //String separator = "CENTRAL AND SOUTH CENTRAL TEXAS";
22         String separator = "TXZ083>090-098>101-113>116-127>130-139>142-154-155-030000-

23         System.err.println( "Update weather information...1" );
         // locate separator string in Web page
24         while ( !in.readLine().startsWith( separator ) )
25            ;    // do nothing

         // strings representing headers on Travelers Forecast
         // Web page for daytime and nighttime weather
26         String dayHeader =
            "CITY            WEA     HI/LO   WEA     HI/LO";
    //"CITY            WEA     LO/HI   WEA     LO/HI";
27         String nightHeader =
            "CITY            WEA     LO/HI   WEA     LO/HI";

 //nightHeader = "CITY           SKY/WX    TMP DP  RH WIND       PRES   REMARKS";
 //nightHeader = "OTHER LOCATIONS IN TEXAS";
28  nightHeader = "WESTERN NORTH TEXAS";

29       String inputLine = "";

30       System.err.println( "Update weather information...2" );
         // locate header that begins weather information
31       do {
32            inputLine = in.readLine();
33       } while ( !inputLine.equals( dayHeader ) &&
                   !inputLine.equals( nightHeader ) );

34       weatherInformation = new ArrayList(); // create List

         // create WeatherBeans containing weather data and
         // store in weatherInformation Vector
35       inputLine = in.readLine();  // skip an empty line
36       inputLine = in.readLine();  // skip the title of the section
37       inputLine = in.readLine();  // get first city's info

         // The portion of inputLine containing relevant data is
         // 45 characters long. If the line length is not at
```

```
         // least 40 characters long, done processing data.
38       System.err.println( "Update weather information...3" );
39       while ( !inputLine.equals("$$") ) {

40          System.err.println( "Update weather information...3.5"+inputLine);
41          System.err.println( "Update weather information...3.6"+inputLine.substring(0,
42          System.err.println( "1qqq");
43          System.err.println( "Update weather information...4"+inputLine.substring( 0,
44          WeatherBean weather = new WeatherBean(
45             inputLine.substring( 0, 14 ),
46             inputLine.substring( 15, 23 ),
47             inputLine.substring( 24, 27 ) );

            // add WeatherBean to List
48          weatherInformation.add( weather );
49          System.err.println( "Initializing WeatherService...5" );

50          inputLine = in.readLine();  // get next city's info
51       }

52       in.close();  // close connection to NWS Web server

53       System.err.println( "Weather information updated." );
54    }

      // process failure to connect to National Weather Service
55    catch( java.net.ConnectException connectException ) {
56       connectException.printStackTrace();
57       System.exit( 1 );
58    }

      // process other exceptions
59    catch( Exception exception ) {
60       exception.printStackTrace();
61       System.exit( 1 );
62    }
63 }

   // implementation for WeatherService interface method
64  public List getWeatherInformation() throws RemoteException
65  {
66      return weatherInformation;
67  }
```

```
   // launch WeatherService remote object
68  public static void main( String args[] )
69  {
70     try {
71              System.setSecurityManager (new RMISecurityManager());
72              System.err.println( "Initializing WeatherService..." );

73              WeatherServiceServer obj = new WeatherServiceServer();
74              WeatherService stub = (WeatherService) UnicastRemoteObject.exportObject(o
76              Registry registry = LocateRegistry.getRegistry();
77              registry.rebind("WeatherService", stub);

78              System.err.println( "WeatherService running." );

79         } catch (Exception e) {
80             System.err.println("Server exception: " + e.toString());
81             e.printStackTrace();
82         }
83      }
84  }
```

(4) The *WeatherBean* utility class (Fig.13.3,p.799,800): stores data the class **WeatherServiceServer** retrieves from the National Weather Service Web site.

a. The class stores the city, temprature, and weather description as **String**s.

b. Lines 25-45 load a property file that contains image names for displaying the weather information. The block (lines 25-45) is **static** to ensure that the image names are available as soon as the java VM loads the **WeatherBean** class into memory. Namely the image names will be available to any instance of **WeatherBean** class.

c. Lines 64-85 provide three *get* methods for each piece of information.

```
1  // WeatherBean.java
2  // WeatherBean maintains weather information for one city.
3  package com.deitel.advjhtp1.rmi.weather;
4
5  // Java core packages
6  import java.awt.*;
7  import java.io.*;
8  import java.net.*;
9  import java.util.*;
```

```
10
11 // Java extension packages
12 import javax.swing.*;
13
14 public class WeatherBean implements Serializable {
15
16    private String cityName;         // name of city
17    private String temperature;      // city's temperature
18    private String description;      // weather description
19    private ImageIcon image;         // weather image
20
21    private static Properties imageNames;
22
23    // initialize imageNames when class WeatherInfo
24    // is loaded into memory
25    static {
26       imageNames = new Properties(); // create properties table
27
28       // load weather descriptions and image names from
29       // properties file
30       try {
31
32          // obtain URL for properties file
33          URL url = WeatherBean.class.getResource(
34             "imagenames.properties" );
35
36          // load properties file contents
37          imageNames.load( new FileInputStream( url.getFile() ) );
38       }
39
40       // process exceptions from opening file
41       catch ( IOException ioException ) {
42          ioException.printStackTrace();
43       }
44
45    } // end static block
46
47    // WeatherBean constructor
48    public WeatherBean( String city, String weatherDescription,
49       String cityTemperature )
50    {
51       cityName = city;
52       temperature = cityTemperature;
```

```
53       description = weatherDescription.trim();
54
55       URL url = WeatherBean.class.getResource( "images/" +
56          imageNames.getProperty( description, "noinfo.jpg" ) );
57
58       // get weather image name or noinfo.jpg if weather
59       // description not found
60       image = new ImageIcon( url );
61    }
62
63    // get city name
64    public String getCityName()
65    {
66       return cityName;
67    }
68
69    // get temperature
70    public String getTemperature()
71    {
72       return temperature;
73    }
74
75    // get weather description
76    public String getDescription()
77    {
78       return description;
79    }
80
81    // get weather image
82    public ImageIcon getImage()
83    {
84       return image;
85    }
86 }
```

(5) The client class *WeatherServiceClient* (Fig.13.4,p.801,802):

a. This class defines the client application that retrieves weather information from the remote object **WeatherServiceServer**.

(a) The client invokes remote method **getWeatherInformation** (line 32) of the interface **WeatherService** to obtain weather information.

(b) The class uses a **JList** with a customerized **ListCellRenderer** to display weather info (lines 73-76).

b. The main method (lines 61-77):

   (a) It first checks whether there is a command-line parameter (the hostname of the remote method hosting computer): lines 67-70.

   (b) It then sets display properties through three method invocations: lines 73-75.

   (c) It invokes the method **setVisible** to display the weather information: line 76.

c. The constructor: lines 16-58.

   (a) It first invokes the default constructor of the class **JFrame** (the **super** method at line 18).

   (b) Line 24 creates a URL string for the remote object to be invoked.

   (c) Line 27,28 invokes the **static** method **lookup** of the class **Naming** (which is in the **java.util**— **package**). This method connects to the RMI registry and returns of a **Remote** reference to the remote object identified by the URL string. As this remote object is known to be **WeatherSerice** object, line 28 type casts the returned reference to type **WeatherService**.

      · Question: which RMI registry does the **lookup** method connect to? The local one, or the one at the remote method hosting computer?

      · The obtained remote object reference can then be used as if it referred a local object. In fact, this remote object reference refers to a *stub* object in the client.

   (d) Lines 31, 32 invoke the remote method **getWeatherInformation** on the remote reference.

      · The remote object invocation will result in a copy of the **List** of **WeatherBeans** objects.

      · Because parameters and returned objects of a remote method must be *marshalled* and *unmarshalled*, also termed *serialized* and *unserialized*, they must be declared as **Serializable**.

   (e) Lines 35, 36 creates a **WeatherListModel** object by calling the **WeatherListModel** method to facilitate the display of the weather info in a **JList** (line 39). Line 40 sets up a **ListCellRenderer** for the **JList**. Line 41 invokes the **add** method to actually display the **JList**.

```
1  // WeatherServiceClient.java
2  // WeatherServiceClient uses the WeatherService remote object
3  // to retrieve weather information.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.rmi.*;
```

```
8  import java.util.*;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class WeatherServiceClient extends JFrame
14 {
15    // WeatherServiceClient constructor
16    public WeatherServiceClient( String server )
17    {
18       super( "RMI WeatherService Client" );
19
20       // connect to server and get weather information
21       try {
22
23          // name of remote server object bound to rmi registry
24          String remoteName = "rmi://" + server + "/WeatherService";
25
26          // lookup WeatherServiceServer remote object
27          WeatherService weatherService =
28             ( WeatherService ) Naming.lookup( remoteName );
29
30          // get weather information from server
31          List weatherInformation = new ArrayList(
32             weatherService.getWeatherInformation() );
33
34          // create WeatherListModel for weather information
35          ListModel weatherListModel =
36             new WeatherListModel( weatherInformation );
37
38          // create JList, set ListCellRenderer and add to layout
39          JList weatherJList = new JList( weatherListModel );
40          weatherJList.setCellRenderer( new WeatherCellRenderer() );
41          getContentPane().add( new JScrollPane( weatherJList ) );
42
43       } // end try
44
45       // handle exception connecting to remote server
46       catch ( ConnectException connectionException ) {
47          System.err.println( "Connection to server failed. " +
48             "Server may be temporarily unavailable." );
49
50         connectionException.printStackTrace();
```

```
      }

      // handle exceptions communicating with remote object
      catch ( Exception exception ) {
         exception.printStackTrace();
      }

   } // end WeatherServiceClient constructor

   // execute WeatherServiceClient
   public static void main( String args[] )
   {
      WeatherServiceClient client = null;

      // if no sever IP address or host name specified,
      // use "localhost"; otherwise use specified host
      if ( args.length == 0 )
         client = new WeatherServiceClient( "localhost" );
      else
         client = new WeatherServiceClient( args[ 0 ] );

      // configure and display application window
      client.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      client.pack();
      client.setResizable( false );
      client.setVisible( true );
   }
}
```

(6) The classes **WeatherListModel** (Fig.13.5,p.803,804), *WeatherCellRenderer* (Fig.13.6, p.805) and *WeatherItem* (Fig.13.7, p.805,806)

a. The class *WeatherListModel* (Fig.13.5,p.803,804)

(a) This class is a **ListModel** implementation. An object of this class contains **WeatherBean**s to be displayed in a **JList**.

(b) This is a so called *adapter* class.

- In Java the interface **List** is not compatible with the interface **JList**. A **JList** can retrieve elements only from a **ListModel** object.
- This class adapts interface **List** to make it compatible with **JList**'s interface.

(c) Specifically, this class adapts as follows:

- When the **JList** invokes **WeatherListModel** method **getSize**, **WeatherListModel** in turn invokes the method **Size** of the interface **List**.
- When the **JList** invokes **WeatherListModel** method **getElementAt**, **WeatherListModel** in turn invokes the method **get** of the interface **List**.

and so on.

(d) The class also plays the role of the model in the Swing's delegate-model architecture (c.f. Fig.3.1 for illustration).

```
1  // WeatherListModel.java
2  // WeatherListModel extends AbstractListModel to provide a
3  // ListModel for storing a List of WeatherBeans.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.util.*;
8
9  // Java extension packages
10 import javax.swing.AbstractListModel;
11
12 public class WeatherListModel extends AbstractListModel {
13
14    // List of elements in ListModel
15    private List list;
16
17    // no-argument WeatherListModel constructor
18    public WeatherListModel()
19    {
20       // create new List for WeatherBeans
21       list = new ArrayList();
22    }
23
24    // WeatherListModel constructor
25    public WeatherListModel( List itemList )
26    {
27       list = itemList;
28    }
29
30    // get size of List
31    public int getSize()
32    {
33       return list.size();
34    }
```

```

   // get Object reference to element at given index
   public Object getElementAt( int index )
   {
      return list.get( index );
   }

   // add element to WeatherListModel
   public void add( Object element )
   {
      list.add( element );
      fireIntervalAdded( this, list.size(), list.size() );
   }

   // remove element from WeatherListModel
   public void remove( Object element )
   {
      int index = list.indexOf( element );

      if ( index != -1 ) {
         list.remove( element );
         fireIntervalRemoved( this, index, index );
      }

   } // end method remove

   // remove all elements from WeatherListModel
   public void clear()
   {
      // get original size of List
      int size = list.size();

      // clear all elements from List
      list.clear();

      // notify listeners that content changed
      fireContentsChanged( this, 0, size );
   }
}
```

b. The class **WeatherCellRenderer** (Fig.13.6, p.805):

   (a) The class **JList** uses the class **ListCellRenderer** to render each element in that **JList**'s **ListModel**.

(b) The class **WeatherCellRenderer** *extends* the **DefaultListCellRenderer** (i.e. it is a subclass of the latter) to render **WeatherBean**s in a **JList**.

(c) The only method **getListCellRendererComponent** creates and returns a **WeatherItem** object for the given **WeatherBean**.

```
1  // WeatherCellRenderer.java
2  // WeatherCellRenderer is a custom ListCellRenderer for
3  // WeatherBeans in a JList.
4  package com.deitel.advjhtp1.rmi.weather;
5
6  // Java core packages
7  import java.awt.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class WeatherCellRenderer extends DefaultListCellRenderer {
13
14    // returns a WeatherItem object that displays city's weather
15    public Component getListCellRendererComponent( JList list,
16       Object value, int index, boolean isSelected, boolean focus )
17    {
18       return new WeatherItem( ( WeatherBean ) value );
19    }
20 }
```

c. The class **WeatherItem** (Fig.13.7,p.805,806):

(a) The class **WeatherItem** *extends* (i.e. it is a subclass of) **JPanel** class to display weather info in a **WeatherBean**. in a **JList**.

(b) This class is used by the **WeatherCellRenderer** (line 18 of that class) to display weather info in a **JList**.

(c) The **static** block at lines 22-29 loads the **ImageIcon** *backgroundImage* into memory when the Java virtual machine loads the **WeatherItem** class itself. This action ensures that **backgroundImage** will be available to all instances of class **WeatherItem**.

(d) The method **paintComponent** at lines 38-56 draws the **backgroundImage** (line 43), the city name (line 50), the temperature (line 51), and the **WeatherBean**'s ImageIcon (line 54).

All weather condition images are provided in a subdirectory in the class directory.

```
1  // WeatherItem.java
```

```
// WeatherItem displays a city's weather information in a JPanel.
package com.deitel.advjhtp1.rmi.weather;

// Java core packages
import java.awt.*;
import java.net.*;
import java.util.*;

// Java extension packages
import javax.swing.*;

public class WeatherItem extends JPanel {

   private WeatherBean weatherBean;  // weather information

   // background ImageIcon
   private static ImageIcon backgroundImage;

   // static initializer block loads background image when class
   // WeatherItem is loaded into memory
   static {

      // get URL for background image
      URL url = WeatherItem.class.getResource( "images/back.jpg" );

      // background image for each city's weather info
      backgroundImage = new ImageIcon( url );
   }

   // initialize a WeatherItem
   public WeatherItem( WeatherBean bean )
   {
      weatherBean = bean;
   }

   // display information for city's weather
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      // draw background
      backgroundImage.paintIcon( this, g, 0, 0 );

```

```
      // set font and drawing color,
      // then display city name and temperature
      Font font = new Font( "SansSerif", Font.BOLD, 12 );
      g.setFont( font );
      g.setColor( Color.white );
      g.drawString( weatherBean.getCityName(), 10, 19 );
      g.drawString( weatherBean.getTemperature(), 130, 19 );

      // display weather image
      weatherBean.getImage().paintIcon( this, g, 253, 1 );

   } // end method paintComponent

   // make WeatherItem's preferred size the width and height of
   // the background image
   public Dimension getPreferredSize()
   {
      return new Dimension( backgroundImage.getIconWidth(),
         backgroundImage.getIconHeight() );
   }
}
```

(7) Compile and run the program

a. Compile the program:

javac *.java

b. Security consideration. Java (1.6 and newer) requires a security policy file for each java server. The security file manages following security aspects:

* The IP names or numbers of computers that allowed the connect to the server or connections to external sources;
* The port numbers of allowed services;
* Type of allowed operations: connect, resolve, accept, and etc.

Examples:

```
grant {
  permission java.net.SocketPermission "www.weather.gov:80", "connect,resolve";
  permission java.net.SocketPermission "127.0.0.1:0-", "connect,resolve,accept";
  permission java.net.SocketPermission "keystone.cs.txstate.edu:0-", "connect,resol
};
```

You need to create such a file on both client and server sides. Name it, say, "policy1-file.txt".

c. Start the *rmiregistry* process:

rmiregistry

Notes: By default, rmiregistry will start at port 1099, which usually is non-privileged. However, only one process can use this port number. If you cannot start this program because of it being used by somebody else, then you have to start it at a different number (that is not be used), such as 9019:

rmiregistry 9019

d. Run the server:

java -Djava.security.policy=policy1-file.txt WeatherServiceServer

e. Run the client:

java WeatherServiceClient

If the server and client are not running on the same host, run the client with the server's IP number

java WeatherServiceClient -Djava.security.policy=policy1-file.txt server-IP

If the server runs at a port number other than 1099, then use the syntax (this will not work for JDK 1.8):

java WeatherServiceClient -Djava.security.policy=policy1-file.txt server-IP:server-port

Note: although both the server and client needs the policy file, the content of that file will be different.

5. Java IDL

(b) Implementing the Server (XyzServer.java)

(c) Implementing the Client (XyzClient.java)

(d) If necessary, implementing the Applet Client (XyzApplet.java)

Lecture 8
Distributed Coordination and Agreement
(08-3,4-2020, Chapter 15)

Lecture Outline

```
1. Introduction
2. Centralized synchronization algorithms
3. Distributed synchronization based on timestamps
4. Distributed synchronization based on token passing
5. Election algorithms
6. Distributed Consensus Problem
```

1. Introduction

(1) Problem description

a. Coordination

b. Agreement

Same in nature

(2) Variants of the problem

2. Centralized synchronization algorithms

- A centralized approach based on *central process*

- One of the node, called the*coordinator*, is designated to coordinate the entry to critical region.
- Each process that wants to enter its critical region sends a *request* message to the coordinator.
- Everytime the coordinator receives a request message, it checks if some other process is within the CS. If not, the coordinator informs its approval by sending a *reply* message. Otherwise the coordinator puts the request into a waiting queue.
- Once a process exits its critical region, it send a *release* message to the coordinator.
- Everytime the coordinator receives a release message, it checks if the waiting queue is empty. If not, it removes one of the waiting request from the queue and sends *reply* message to the requesting process. The above algorithm satisfies all the three requirements of CS solutions if the properly queue is properly maintained (say as a FIFO). Each entry to the CS requires three messages: *request, reply*, and *release*.
  One severe problem is what happen if the coordinator goes down. The whole system would halt. We must be able to detect the failure of the coordinator and elect a new coordinator.

3. Distributed synchronization based on timestamps

Distributed approach – Decision-making is distributed accross the whole system.

- When a process P_i wants to enter its CS, it generates a new timestamp, *TS*, and sends the message *request*(P_i, *TS*) to all other processes in the system (including itself).
- On receiving a *request* message, a process may reply immediately by sending a *reply* message to P_i, or it may delay sending a reply.
- A process that has received a *reply* message from all other processes in the system can enter its CS, queueing incoming requests and defering them. After exiting its CS, the process sends reply to all its defered requests.
- The decision whether a process P_j replies immediately to a request *request*(P_i,*TS*) message or defers its reply is based on three factors:
  a. If P_j is in its CS, then it has to defer the reply.
  b. If P_j does not want to enter its CS, then it sends a *reply* immediately to P_i.
  c. If P_j wants to enter its CS but has not entered yet, then it compares its own request timestamp with the timestamp *TS* of the incoming request from P_i. If its own request timestamp is greater than *TS*, then it sends *reply* immediately (P_i asked first). Otherwise, the reply is defered.
- The algorithm has the following desirable behaviour:
  a. Mutual exclusion is preserved and freedom from deadlocks is ensured.
  b. Freedom from starvation is ensured, because entry to CS is scheduled according to the timestamp ordering. The timestamp ordering ensures that the requests are served in a first-come-first-served order.
  c. The number of messages per CS entry is $2 \times (n - 1)$.
- **Example**. Three processes P_1, P_2, P_3. Assume P_1 and P_3 want to enter their CSs. P_1 sends a message *request*(P_1, $TS = 10$) to P_2 and P_3, while P_3 sends message *request*($P_3, TS = 4$) to P_1 and P_2. (The timestamp values 10 are 4 are obtained from the logical clocks). When P_2 receives the two requests, it sends its replies back immediately. P_1 receives the request from P_3, it also sends its reply immediately, because its own timestamp 10 is greater than the timestamp 4 of P_3's request. When P_3 receives request from P_1, it defers its reply. On receiving replies from both P_1 and P_2, P_3 enters CS. After exiting from its CS, P_3 sends reply to P_1, and P_1 can enter its CS.
- Three problems:

i. The processes must know the identities of all other processes in the system. When a new process joins the group of processes, the following actions need to be taken:

   a. The process must receive the identities of all other processes in the group.
   b. The name of the new process must be sent to all other processes.

   Not a trivial task.

ii. If one of the processes fails, the entire algorithm is defeated. We can monitor the states of all processes in the system, and notify all active processes once a process failure is detected.

iii. Processes that have not enter its CS must pause frequently to participate the decision making. Therefore this protocol is only suitable for a small, stable set of processes.

- Complexity: needs $3(n-1)$ messages in a system with n processes.

4. Distributed synchronization based on token passing

(1) General ideas:

A single token is circulating around the distributed system. Only the process that is holding the token is allowed to execute its C.S.

(2) The algorithm by Ricart and Agrawala (CACM 1979)

a. **Assumptions**:

   * The system is fully linked.
   * The transmission is error-free.
   * Transmission is variable.
   * Desequencing is possible – messages may be recieved in an order different from that in which they were sent.

b. **Principles**:

   (a) Like any token-based algorithm, a single token is circulating. Token holder can enter its C.S. without asking permission from other processes.
   (b) Initially the token is assigned arbitrarily to one of the n processes. A process P_i that wishes to enter its C.S. broadcasts to all other processes a request message m that contains a timestamp T_i.
   (c) The token is implemented as a vector of n components. The ith component of the token records the timestamp when the last time the token was assigned to process P_i.

(d) When a process, say P_j, which is holding the token, no longer needs the token, it will search the entries $j+1, j+2, \cdots, n, 1, 2, \cdots$ of the token for the first value l such the timestamp of P_l's last request for the token is greater than the value recorded in the token for the timestamp of P_l's last holding of the token. P_j then sends the token to P_l.

c. The algorithms. The following data structures are maintained in each process P_i:

clock:	0, 1, ..., **initialized** 0; (logical clock)
token_present:	**boolean**;
token_held:	**boolean initialized** F;
token:	**array** (1,2,..., n) **of** (0,1,...) **initialized** 0;
request:	**array** (1,2,..., n) **of** (0,1,...) **initialized** 0;

The boolean variable 'toke_present' is initialized to F in every process except one, which is holding the token at the start.
The operation **wait**(access, token) causes process to wait until a message of the type 'access' is received.

The algorithms has two parts. The first part deals with the use of C.S. and consists of a prelude section, followed by the C.S. and ending with a postlude section. The second part deals with the actions to be performed when messages are received.

```
{Prelude}
if ¬ token_present then begin clock ← clock+1;
                              broadcast(request, clock, i);
                              wait(access, token);
                              token_present ← T;
                        end;
endif;

token_held ← T;
    <critical section>

{Postlude}
token(i) ← clock;
token_held ← F;
for j from i + 1 to n, 1 to i − 1 do
        if request(j) > token(j) ∧ token_present then
          begin
            token_present ← F;
            send(access, token, j);
          end;
        endif;

when received(request, k, j) do
        request(j) ← max(request(j), k);
        if token_present ∧ ¬ token_held then
          <text of postlude>
        endif;
enddo
```

Notes:

* If the token is not present, execution of the prelude section will broadcast the request to all the other processes and wait until the token arrives.
* The postlude section first records in the token the time of its last holding by P_i, looks to see if any process has requested the token and if so transfers it appropriately.
* The receipt of a request from P_j has the effect of updating the local variable 'request(j)' which records the time of P_j's last request, followed by the transfer of the token if it is neither held nor being used by any other process.
* Correctness is guaranteed by showing that at any time instance, only exactly one variable token_present has value T. This can be proved by induction. Initially, it is true. The prelude for process P_i changes the variable token_present

from F to T when it receives the token. The process P_j that sent the token must have changed its variable token_present from T to F before sending it. This establishes the mutual exclusion property. (all variables token_present have value F when and only when the token is being transmitted).

* Fairness and freedom from deadlocks.
  - Freedom from deadlock can be proved by contradiction argument
  - Fairness is guaranteed by (1) all messages are delivered within a finite time of issue. The postlude requires that P_i transfer the token to the first process P_l, found in scanning the set of in the order $l = i+1, i+2, \cdots, n, 1, \cdots i-1$, whose request has reached P_i. If the transmission delays are finite (no messsge lost), all the processes will learn of the wish of some P_j to enter its C.S. and will agree on this when its turn comes.

5. Election algorithms

(1) Why do we need election algorithms and what is an election algorithms?

- Distributed algorithm with centralized controls: when the coordinator fails, the whole algorithm fails.
- Some node is providing some special service to the system. Failure of such node should be detectd.
- Normally the nodes are numbered and the node with the largest id will be the coordinator. If the coordinator fails the node with the largest id in the remaining active nodes will be elected.

(2) The Chang and Roberts Algorithm (CACM 1979)

a. Assumptions:
   * Applicable to a system connected by a unidirectional ring.
   * Each process has a unique id, and the process with the largest id is the coordinator.
   * Each process knows its own id but may not know others ids.

b. General ideas. Based on the principle called "selective extinction".
   * The election procedure can be initiated by any process P_i. First P_i marks itself and send an *election* message containing its id to its left neighbor.
   * Every time an unmarked process receives the election message, it will compare the id in the message with its own id, and sends out the larger of two. It will then mark itself.
   * When a marked process receives the election message with its own id in it, it knows that it is elected and will broadcast an *elected* message to all other processes.

c. The algorithm:

Each process P_i maintains the following local variables and constants:

```
constant   my_number:   value i;
varaibles  participant: boolean initialized F;
           co-ordinator: integer;

when decision (initiate_election) do
        participant ← T;
        sendL(election, my_number);
when received (election, j) do
        case j > my_number then begin
                sendL(election, j);
                participant ← T;
                end case
        case j < my_number and ¬ participant then begin
                        sendL(election, my_number);
                        participant ← T;
                end case
        case j = my_number then begin
                        sendL(elected, i);
                end case
when received (elected, j) do
                co-ordinator ← j;
                participant ← F;
                if j ≠ my_number then sendL(elected,j);
                end if
end do;
```

d. Time complexity: two extreme cases:

* Every process initiates an election at the same time. The largest number is found in one turn round the ring. Time needed is proportional to n.
* The process immediate after the largest process P_{max} initiates the election. It takes $n-1$ transmissions for P_{max} to receive the election message, another $n-1$ message for the largest number to come back to the largest process. All together, time needed is still $O(n)$.

e. Message complexity:

i. The most favorable situation is one the processes are arranged in increasing order of their processes id's and the election is initiated by the largest process P_{max}. Only $2n$ messages are needed.

ii. The worst situation is one the processes are arranged in decreasing order of their processes id's and all initiate the election at the same time. In this case the election message issued by process P_i can travel i hops. So the total number of messages

$$\Sigma_{i=1}^{n} i = \frac{1}{2}n(n+1)$$

So the message needed is $O(n^2)$.

iii. Average case. Let $P(i,k)$ be the probability that the election message issued by process P_i can travel $k-1$ hops, i.e. the $k-1$ processes after P_i has labels less than that of P_i and the kth process after P_i has label larger than that of P_i. Then

$$P(i,k) = \frac{C(i-1,k-1)}{C(n-1,k-1)} \times \frac{n-i}{n-k}$$

Where $C(i,k)$ is the number of *combinations* k items from a set of i items. The mean number of transfers from process P_i, $i < n$ is

$$E_i = \Sigma_{k=1}^{n-1} k P(i,k)$$

The mean number of total messages is

$$E = n + \Sigma_{i=1}^{n+1} E_i$$

Which can be shown is $O(n \log n)$.

(3) The Hirschberg and Sinclair Algorithm (CACM 1980)

a. **Assumptions**:

* The processes are organized around a bidirectional ring in arbitrary order.
* Each process has a unique numerical identifier. Each process may not know the identifies of its two neighbors.

b. **General Ideas**:

* The algorithm is based on the idea of constructing a sequence of elections on increasingly larger subsets of processes until all the processes have been included.
* The election procedure can be initiated by any process P_i. Process P_i initiates an election by declaring itself as a candidate and sending its id to its two neighbors, say P_j and P_k. P_j and P_k will compare its own id with the P_i's id and replace P_i as new candidates if their id's are larger than P_i's. Otherwise, P_i will remain to be the candidate, and P_i will test the legitimacy over a range of 4 processes (P_k, P_j, and neighbors of the two).
* Every time a round of test is successful the range of tests is doubled.

* If the test if not successful, process P_i will remain *passive* from that point on and pass incoming messages.

c. The algorithms.

The following communication primitives are used in the algorithm

sendLR send the same message to both the left-hand and right-hand neighbors.

pass pass the message received from right-hand neighbor to left-hand neighbor and vice versa.

respond send a response to the neighbor from which a message has just been received.

The following data structures are maintained in each process P_i:

constant	my_number	**value** i;
variables	state:	**position initialized** not_involved;
	lgmax:	**integer**;
	winner:	**integer**;
	nbresp:	0,1,2 **initialized** 0;
	respOK:	**boolean initialized** T;

The type **position** is defined as

type position = (not_involved,candidate,lost,elected)

Three types of messages (two for elections):

* **(candidate,number,lg,lgmax)**, where:

  number is the identifier of the process sending the candidacy message.

  lg is the distance (length) around the ring already traveled by this message.

  lgmax is the maximum distance the message can travel.

* **(response,bool,number)**, associated with **pass** and **response**.

  bool is T if the response is favorable ('respOK'), F otherwise.

  number is the identifier of the destination process (therefore is a means of addressing on the ring, since id's are unique).

* **(ended, number)**, used to indicate the election ends.

  number is the id of the elected process.

```
when decision (initiate_election) do
        state ← candidate;
        lgmax ← 1;
        while state = candidate do
            nbresp ← 0; respOk ← T;
            sendLR(candidate, my_number, 0, lgmax);
            wait nbresp = 2;
            if ¬respOk then state ← lost;
            endif;
            lgmax ← 2*lgmax;
        endwhile;
end do;
when received (response, bool, number) do
    if number = my_number then
            nbresp ← nbresp + 1;
            respOK ← respOK∧bool;
    else pass(response, bool, number);
    end if;
end do;
when received (candidate, number, lg, lgmax) do
    case number < my_number then
            respond(response, F, number);
            if state = not_involved then initiate_election;
            end if;
        end case;
    case number > my_number then
            state ← lost;
            lg ← lg + 1;
            if lg < lgmax then
                pass(candidate, number, lg, lgmax);
            else respond(response, T, number);
            end if;
        end case;
    case number = my_number then
            if state ≠ elected then
                state ← elected;
                winner ← my_number;
                pass(ended, my_number);
            end if;
        end case;
end do;
```

```
when received(ended, number) do
        if winner ≠ number then
                pass(ended, number);
                winner ← number;
                state ← not_involved;
        end if;
end do;
```

d. Time and message complexities: two extreme cases.
Assume that n is a power of 2, i.e. $n = 2^l$ for some $l \geq 0$.

* Only P_{max} initiates the election. It will take $l+1$ rounds of testing before P_{max} receives its own testing message. During the ith $(0 \leq i \leq l)$ round, a message issued will travel to the process 2^i processes away from P_{max}. Therefore 4.2^i messages will be transmitted. The total number of messages transmitted is

$$4(1 + 2 + 2^2 + \cdots 2^l) = 4(2^{l+1} - 1) = 8n - 4$$

The time needed is also bounded by $8n - 4$ because message transmissions might be serial. So both time and message complexities are $O(n)$ in the best case.

* More than one process initiates election. Consider the following facts:

Fact 1: A candidacy message issued by process P_j during the ith round can at most travel to a process that is 2^i processes away from P_j.

Fact 2: P_j can launch a test on a range of $22^i + 1$ processes if and only if P_j has successfully tested its candidacy on a range $22^{i-1} + 1$ processes.

Fact 3: In any set of $2^{i-1} + 1$ processes, only one can launch a candidacy testing on a path of length 2^i.

Although in the worst case all processes can launch their candidacy testing simultaneously, we have

- At most $n/2$ that are candidates with paths of length 2;
- At most $n/3$ that are candidates with paths of length 4;
- At most $n/(2^{i-1} + 1)$ that are candidates with paths of length 2^i.

The two candidacy testing messages issued by a process on a path of length 2^i will incur 42^i messages. The total number of messages is

$$4 \times (1.n + 2\frac{n}{2} + 4\frac{n}{3} + \cdots + 2^i\frac{n}{2^{i-1} + 1} + \cdots + 2^l\frac{n}{2^{l-1} + 1})$$

Each term in above summation is less than $2n$. Since we have $l+1 = 1+\log n$ terms, the number of total messages is bound above by $4.2n.(1 + \log n) = 8n(1 + \log n)$. So the number of messages is $O(n \log n)$. Time complexity is also $O(n \log n)$ because at most $8n(1 + \log n)$ time units are needed to transmit $8n(1 + \log n)$ messages.

6. Distributed Consensus Problem (Chapter 15.5.3)

   (1) Introduction

   (2) Failures in DS: they may cause components to behave unpredicably – sending conflicting information to different parts of the system.

   (3) Synchronization vs. broadcast in DS

      a. Broadcasting has been used by many DSs as a basic mechanism for sending an item of information from one process (the issuer) to a number of other processes (the receivers).

      b. Synchronization problems in DS:

         (a) The critical section problem (CSP)

         (b) The problem of managing transactions in DSs

         (c) The election problem

      c. Effects of failures: they complicate the above problem.

   (4) The distributed consensus problem.

      a. This problem is also a synchronization problem. However it is a more general synchronization problem than the CSP, transaction management, or, election problem, because processes in this problem can be *faulty* – they can send conflicting info in response to a synchronization request.

      b. Problem description: A network of n processes can communicate with each other only by means of message transmission over bidirectional comm. channels. It is assumed that reliable broadcasting of messages is available, i.e. when any process issues any item of information (i.e. message), all other processes will receive that item unchanged (i.e. have the same "perception" of the item). We say that a consensus has been reached if all the processes that are operating *reliably* reached the same conclusion.

      c. Note:

         (a) If the issuing process is reliable, then it will send (broadcast) the same message to every other processes. If the issuing process is faulty, however, it may send different messages to different processes, or even may not send a message to a process.

(b) A receiving process does not know if a sending process is reliable or not.

(c) This problem has also been called the *Byzantine Generals Problem*, or the *Unanimity Problem* in the literature.

(d) An analogue of this problem (the reason why it is also called the Byzantine Generals problem) is: In ancient time, a group of Byzantine generals surrended an enemy's city. The generals had to reach a consensus about the next step (attack time, withdraw time, and etc.). There were two types of generals: loyal generald and traitorous generals. Any of the generals could initiate a proposal of next step action and such a proposal would be transmitted among the generals using messengers. Messengers were reliable – they wouldn't change the message contents. A traitorous general on the other hand could change the message content before sending it to next general. The problem is the following: how could we let the loyal generals reach the same conclusion of the next step action, regardless the sabotage of the traitorous generals?

(5) Features of solutions to the problem

a. Due to the possibility that the issuing process may be faulty, and the requirements that the reliable processes must agree among themselves on the common decision, the receiving processes have to exchange values they received in order to achieve consensus.

b. We can assume that there is a function *majority* available to each reliable process, which will apply the function to messages received. This function will return same value as long as the parameters (i.e. messages) are the same. The ordering of the parameters is immaterial.

c. If the maximum number of simultaneously faulty processes is t, then any solution to the problem will need at least $t + 1$ stages of message exchanges in order to arrive at a consensus.

d. Measurements of Solutions:

(a) The number r of stages of exchanges necessary to arrive a consensus (mimimum is $t + 1$ where t is the number of faulty processes).

(b) The total number of messages that have to be exchanged.

(c) The maximal length m of messages exchanged.

(6) Impossibility result

Theorem There is no solution to the distributed consensus problem unless the maximum number t of faulty processes at any time is strictly less than one third of the total number n of processes in a network. Namely:

$$n \geq 3t + 1$$

Proof: Induction on t and n. For simility, we only consider the case $t = 1$ and $n = 3$ and show that no consensus is possible in this case.

Let P_0, P_1, and P_2 are three processes and P_0 be the issuing process.

Case 1 : P_2 is the faulty process. See Figure 42. Here P_1 receives (a, b) and its conclusion should be a (same as the other reliable process P_1).

Case 2 : P_0 is the faulty process. See Figure 43. Here P_1 receives (a, b) again and its conclusion should be a according Case 1. P_2 receives (a, b) also and concludes to a too.

Case 3 : P_1 is the faulty process. See Figure 44. Here P_2 receives (a, b) again and its conclusion should be consistent to a (Case 2). However, the other reliable process is P_0 and its conclusion is b (it issues b in the first place). Therefore a contradiction arises: P_2 has to conclude both a and b for the same messages sequence (a, b) received.

Note: Fig.15.18, p.666 shows similar argument for impossibility of three generals.

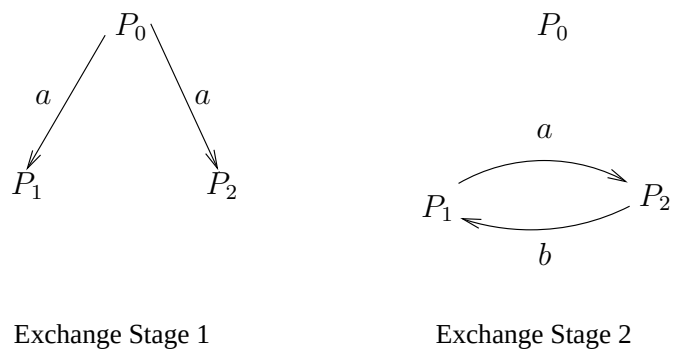


Figure 42: Case 1 of Impossibility Result Proof

(7) The Lamport, Shotstak, and Pease algorithm (ACM TOPLAS, July 1982)

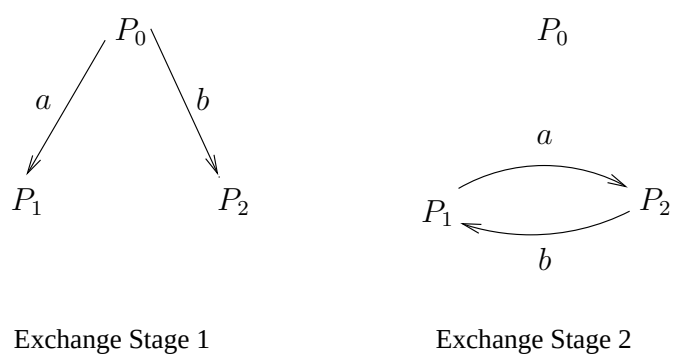


Figure 43: Case 2 of Impossibility Result Proof

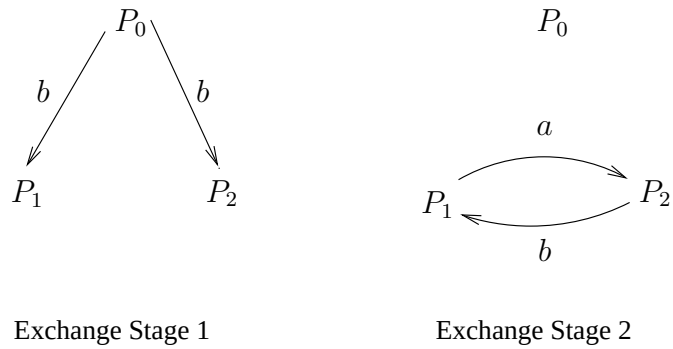


Figure 44: Case 3 of Impossibility Result Proof

a. Assumptions about the network
   (a) There are n processes, and t of them are *faulty*;
   (b) The comm. environment through which the processes exchange messages is reliable;
   (c) The comm. channels are bi-directional and the network topology is a complete graph (i.e. every process is directly linked with every other process);
   (d) Messages are transmitted in FIFO order.
b. Assumptions about the messages
   (a) A faulty process may modify the contents of an incoming message before forwarding;
   (b) The id of the issuer of a message is unknown to the receiver;
   (c) A faulty process may elect not to forward a message after receiving it. In this case, we assume there is a special message v_{def}, which a process P_i may assume is from P_j if P_j doesn't forward message to P_i in a specific stage of message exchanges.
c. Outline of the algorithm
   (a) The algorithm consists of a sequence of stages of message exchanges. This is true for all algorithms for solving this problem. Cf. the note in 1.(3).c of this lecture.
   (b) If upto t processes can be faulty, then at least $t+1$ such stages are necessary. This fact can be easily deduced from the impossibility result Theorem.
   (c) This is a recursive algorithm. For similicity of presentation and without loss of generality, we assume the issuer is always P_0.
   (d) The first stage consists of the issuer P_0 sending out the value v to every other processes. In the subsequent stages, these "other processes" exchange the values they receive. Every process attaches its id to every message it sends

out (so that the receiver processes will not exchange that same message with the original sender). Each message is of the form:

$$(v, P_0, P_{i_1}, P_{i_2}, \cdots, P_{i_{n-1}})$$

Notice that inculding P_0 as part of the message is not really necessary. The rule is that, once P_i receives a message, it will forward the message to every process listed in the message, excluding itself and the original sender.

(e) All reliable processes exchange messages unchanged. Those faulty processes may alter a message before exchanging it with others, or elect to withhold the message.

(f) Assume there is a function *majority* that will return a value from its given parameters. The order of parameters for this function is not important. For example, $majoriry(a, b, c)$ is equal to $majority\ (c, a, b)$. At the end of message exchange stages, each process will invoke this function with message values received from other processes as parameters. The value returned will be the conclusion reached by that process.

d. Algorithm $UM(n, t)$

begin

(a) Issuer process P_0 sends its value v to each other $n-1$ processes $P_1, P_2, \cdots, P_{n-1}$.

(b) Each receiver process notes the value received from P_0, or records the default value v_{def} if it has not received anything.

(c) **if** $t > 0$ **then**

begin **for** every process P_i do

i. Let v' be the value noted by P_i in Step b. P_i acts as the issuer, and sends v_i by calling $UM(n-1, t-1)$ to all other $n-2$ processes $P_1, \cdots, P_{i-1}, P_{i+1}, \cdots, P_{n-1}$. The message sent is of the form:

$$(v', P_i, P_1, \cdots, P_{i-1}, P_{i+1}, \cdots, P_{n-1})$$

ii. Let v_j $(i \neq j)$ be the value received by P_i from P_j at the end of Step b.(ii), or v_{def} if nothing is received. P_i records its conclusion as value

$$majority(v_0, v_1, \cdots, v_{i-1}, v_{i+1}, \cdots, v_{n-1})$$

end

endif

end

e. Correctness of the algorithm: can be proved by induction on n and t and is omitted.

f. Example. Consider the case, $n = 4$, $t = 1$. We invoke the function $UM(4, 1)$.

(a) Case 1. P_3 is faulty. Figure 45 shows the messages exchanged. At the end of Stage 1:

- At P_1, $v_0 = a$;
- At P_2, $v_0 = a$;
- At P_3, $v_0 = a$;

At the end of Stage 2:

- At P_1, $v_0 = a$, $v_2 = a$, $v_3 = b$;
- At P_2, $v_0 = a$, $v_1 = a$, $v_3 = c$;
- At P_3, $v_0 = a$, $v_1 = a$, $v_3 = a$.

At the end of the algorithm both P_1 and P_2 have the same conclusion (*majority*(a, a, b) and *majority*(a, a, c) respectively), which is also is P_0's conclusion a. Notice that the conclusion of P_3 is not important, because it is a faulty process anyway.

(b) Case 2. P_0 is faulty. It issues a to P_1, b to P_2, and does not send anything to P_3. Figure 46 shows the messages exchanged. At the end of algorithm, all three reliable processes P_1, P_2, and P_3 have the same conclusion *majority*(a, b, v_{def}).

Note: Fig.15.19, p.667 shows similar argument for $n = 4$ and $t = 1$ where the faulty process is p_3.

g. Complexity of the Algorithm

(a) Number of stages: $t+1$. This is optimal in light of of the impossibility result. It can tolerate t faulty processes when $n > 3t$ holds.

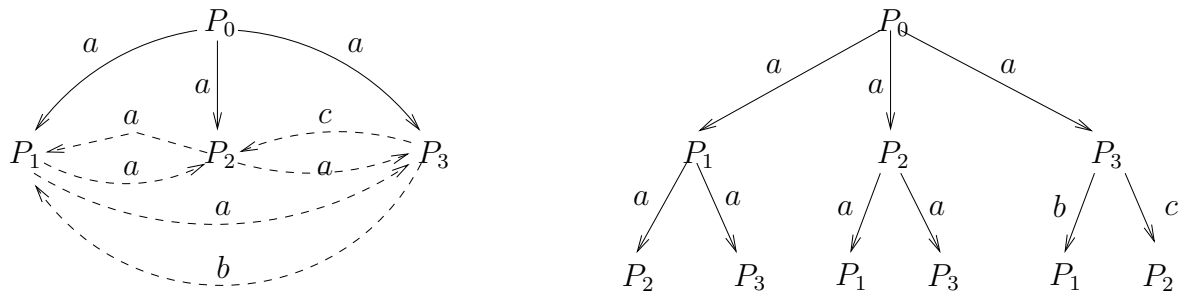
(b) Number messages

- Stage 1: $UM(n, t)$ will generate $n - 1$ messages;
- Stage 2: there are $n-1$ $UM(n-1, t-1)$ function calls, each will generate $n - 2$ messages, for a total $(n - 1)(n - 2)$ messages;
- Stage 3: there are $(n - 1)(n - 2)$ $UM(n - 2, t - 2)$ function calls, each will generate $n - 3$ messages, for a total $(n - 1)(n - 2)(n - 3)$ messages;
- ...
- Stage t will generate a total $(n-1)(n-2)\cdots(n-t)$ and Stage $t+1$ (the last stage) will generate

$$(n - 1)(n - 2)\cdots(n - t)(n - t - 1)$$

messages.

- Therefore the total number of messages is $(n - 1) + (n - 1)(n - 2) + (n - 1)(n - 2)(n - 3) + \cdots + [(n - 1)(n - 2)\cdots(n - t)(n - t - 1)] = O(n^{t+1})$.

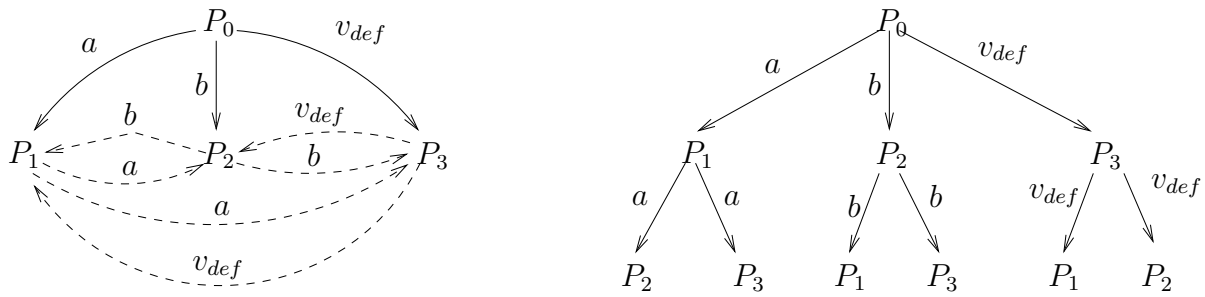


(a) Message Exchanges

(b) Another View of Message Exchanges

P_3 Faulty

Figure 45: Case 1 of Example: P_3 Faulty



(a) Message Exchanges

(b) Another View of Message Exchanges

P_0 Faulty

Figure 46: Case 2 of Example: P_0 Faulty

(c) Message sizes: at stage k, each message contains a sequence of $n-k$ process ids, therefore is proportional to k.

(8) Other algorithms

	# of Faulty Processes	# of Stages	Maximal Message Size
Optimal Case	$n \geq 3t+1$	$t+1$	1
LSP Algorithm	$n \geq 3t+1$	$t+1$	t
Berman & etl. Algo.(1989)	$n \geq 3t+1$	$t+1$	polynomial
Berman & etl. Algo.(1989)	$n \geq 3t+1$	$3t+3$	2
Moses & etl. Algo.(1988)	$n \geq 6t+1$	$t+1$	polynomial