Distributed Systems
Introduction

Chapter 1
A distributed system is:

A collection of independent computers that appears to its users as a single coherent system.
A distributed system organized as middleware.
Note that the middleware layer extends over multiple machines.
### Transparency in a Distributed System

<table>
<thead>
<tr>
<th>Transparency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and how a resource is accessed</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where a resource is located</td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that a resource may move to another location</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that a resource may be moved to another location while in use</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that a resource may be shared by several competitive users</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that a resource may be shared by several competitive users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide the failure and recovery of a resource</td>
</tr>
<tr>
<td>Persistence</td>
<td>Hide whether a (software) resource is in memory or on disk</td>
</tr>
</tbody>
</table>

Different forms of transparency in a distributed system.
## Scalability Problems

<table>
<thead>
<tr>
<th>Concept</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized services</td>
<td>A single server for all users</td>
</tr>
<tr>
<td>Centralized data</td>
<td>A single on-line telephone book</td>
</tr>
<tr>
<td>Centralized algorithms</td>
<td>Doing routing based on complete information</td>
</tr>
</tbody>
</table>

Examples of scalability limitations.
Scaling Techniques (1)

The difference between letting:

b) a server or
c) a client check forms as they are being filled
An example of dividing the DNS name space into zones.
General structure of a distributed system as middleware.
In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.
## Comparison between Systems

<table>
<thead>
<tr>
<th>Item</th>
<th>Distributed OS</th>
<th>Network OS</th>
<th>Middleware-based OS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multiproc.</td>
<td>Multicomp.</td>
<td></td>
</tr>
<tr>
<td>Degree of transparency</td>
<td>Very High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Same OS on all nodes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Number of copies of OS</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Basis for communication</td>
<td>Shared memory</td>
<td>Messages</td>
<td>Files</td>
</tr>
<tr>
<td>Resource management</td>
<td>Global, central</td>
<td>Global, distributed</td>
<td>Per node</td>
</tr>
<tr>
<td>Scalability</td>
<td>No</td>
<td>Moderately</td>
<td>Yes</td>
</tr>
<tr>
<td>Openness</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
</tr>
</tbody>
</table>

A comparison between multiprocessor operating systems, multicomputer operating systems, network operating systems, and middleware based distributed systems.
Clients and Servers

General interaction between a client and a server.
An Example Client and Server (1)

The `header.h` file used by the client and server.

```c
/* Definitions needed by clients and servers. */
#define TRUE 1
#define MAX_PATH 255 /* maximum length of file name */
#define BUF_SIZE 1024 /* how much data to transfer at once */
#define FILE_SERVER 243 /* file server's network address */

/* Definitions of the allowed operations */
#define CREATE 1 /* create a new file */
#define READ 2 /* read data from a file and return it */
#define WRITE 3 /* write data to a file */
#define DELETE 4 /* delete an existing file */

/* Error codes. */
#define OK 0 /* operation performed correctly */
#define E_BAD_OPCODE -1 /* unknown operation requested */
#define E_BAD_PARAM -2 /* error in a parameter */
#define E_IO -3 /* disk error or other I/O error */

/* Definition of the message format. */
struct message {
    long source; /* sender's identity */
    long dest; /* receiver's identity */
    long opcode; /* requested operation */
    long count; /* number of bytes to transfer */
    long offset; /* position in file to start I/O */
    long result; /* result of the operation */
    char name[MAX_PATH]; /* name of file being operated on */
    char data[BUF_SIZE]; /* data to be read or written */
};
```
An Example Client and Server (2)

#include <header.h>
void main(void) {
    struct message ml, m2; /* incoming and outgoing messages */
    int r; /* result code */

    while(TRUE) { /* server runs forever */
        receive(FILE_SERVER, &ml); /* block waiting for a message */
        switch(ml.opcode) { /* dispatch on type of request */
            case CREATE: r = do_create(&ml, &m2); break;
            case READ: r = do_read(&ml, &m2); break;
            case WRITE: r = do_write(&ml, &m2); break;
            case DELETE: r = do_delete(&ml, &m2); break;
            default: r = E_BAD_OPCODE;
        }
        m2.result = r; /* return result to client */
        send(ml.source, &m2); /* send reply */
    }
}
A client using the server to copy a file.
The general organization of an Internet search engine into three different layers:
Multitiered Architectures (1)

Alternative client-server organizations (a) – (e).

Alternative client-server organizations (a) – (e).
An example of a server acting as a client.
Modern Architectures

An example of horizontal distribution of a Web service.
Communication Methods

Chapter 2
Layered Protocols (1)

Layers, interfaces, and protocols in the OSI model.
Layered Protocols (2)

A typical message as it appears on the network.
Discussion between a receiver and a sender in the data link layer.
Client-Server TCP

(a) Normal operation of TCP.
(b) Transactional TCP.
An adapted reference model for networked communication.
Conventional Procedure Call

(a) Parameter passing in a local procedure call: the stack before the call to read

(b) The stack while the called procedure is active
Client and Server Stubs

Principle of RPC between a client and server program.
Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client
Steps involved in doing remote computation through RPC
Passing Value Parameters (2)

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>J</td>
<td>5</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>I</td>
<td>J</td>
<td></td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>J</td>
<td>I</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>I</td>
<td>J</td>
<td></td>
</tr>
</tbody>
</table>

(c)

a) Original message on the Pentium
b) The message after receipt on the SPARC
c) The message after being inverted. The little numbers in boxes indicate the address of each byte
Parameter Specification and Stub Generation

a) A procedure
b) The corresponding message.

foobar( char x; float y; int z[5] )
{
    ....
}

<table>
<thead>
<tr>
<th>foobar's local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>z[0]</td>
</tr>
<tr>
<td>z[1]</td>
</tr>
<tr>
<td>z[2]</td>
</tr>
<tr>
<td>z[3]</td>
</tr>
<tr>
<td>z[4]</td>
</tr>
</tbody>
</table>
The principle of using doors as IPC mechanism.
a) The interconnection between client and server in a traditional RPC

b) The interaction using asynchronous RPC
A client and server interacting through two asynchronous RPCs
Writing a Client and a Server

The steps in writing a client and a server in DCE RPC.

1. Uuidgen
2. Interface definition file
3. IDL compiler
4. Client code
5. Client stub
6. Header
7. Server stub
8. Server code
9. Include
10. C compiler
11. Client object file
12. Client stub object file
13. Client binary
14. C compiler
15. Server stub object file
16. Server binary
17. Include
18. C compiler
19. Server object file
20. Include
21. C compiler
22. Server object file
23. Linker
24. Runtime library
25. Linker
Binding a Client to a Server

Client-to-server binding in DCE.

Client machine

Directory machine

Server machine

Client

Directory server

DCE daemon

1. Register endpoint

2. Register service

3. Look up server

4. Ask for endpoint

5. Do RPC

Endpoint table

Client-to-server binding in DCE.
Distributed Objects

Common organization of a remote object with client-side proxy.
Binding a Client to an Object

Distr_object* obj_ref;
obj_ref = ...;
obj_ref-> do_something();

(a) An example with implicit binding using only global references

Distr_object objPref;
Local_object* obj_ptr;
obj_ref = ...;
obj_ptr = bind(obj_ref);
obj_ptr -> do_something();

(b) An example with explicit binding using global and local references
Parameter Passing

The situation when passing an object by reference or by value.
a) Distributed dynamic objects in DCE.
b) Distributed named objects
General organization of a communication system in which hosts are connected through a network
Persistence and Synchronicity in Communication (2)

Mail stored and sorted, to be sent out depending on destination and when pony and rider available.

Persistent communication of letters back in the days of the Pony Express.
a) Persistent asynchronous communication
b) Persistent synchronous communication
Persistence and Synchronicity in Communication (4)

a) Transient asynchronous communication
b) Receipt-based transient synchronous communication
a) Delivery-based transient synchronous communication at message delivery

b) Response-based transient synchronous communication
### Berkeley Sockets (1)

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

Socket primitives for TCP/IP.
Connection-oriented communication pattern using sockets.
The Message-Passing Interface (MPI)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_bsend</td>
<td>Append outgoing message to a local send buffer</td>
</tr>
<tr>
<td>MPI_send</td>
<td>Send a message and wait until copied to local or remote buffer</td>
</tr>
<tr>
<td>MPI_ssend</td>
<td>Send a message and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_sendrecv</td>
<td>Send a message and wait for reply</td>
</tr>
<tr>
<td>MPI_isend</td>
<td>Pass reference to outgoing message, and continue</td>
</tr>
<tr>
<td>MPI_isendrecv</td>
<td>Pass reference to outgoing message, and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_recv</td>
<td>Receive a message; block if there are none</td>
</tr>
<tr>
<td>MPI_irecv</td>
<td>Check if there is an incoming message, but do not block</td>
</tr>
</tbody>
</table>

Some of the most intuitive message-passing primitives of MPI.
Message-Queuing Model (1)

Four combinations for loosely-coupled communications using queues.

(a) Sender running, Receiver running
(b) Sender running, Receiver passive
(c) Sender passive, Receiver running
(d) Sender passive, Receiver passive
### Message-Queuing Model (2)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block.</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue.</td>
</tr>
</tbody>
</table>

Basic interface to a queue in a message-queuing system.
The relationship between queue-level addressing and network-level addressing.
The general organization of a message-queueing system with routers.
The general organization of a message broker in a message-queuing system.
Example: IBM MQSeries

General organization of IBM's MQSeries message-queuing system.
### Channels

Some attributes associated with message channel agents.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport type</td>
<td>Determines the transport protocol to be used</td>
</tr>
<tr>
<td>FIFO delivery</td>
<td>Indicates that messages are to be delivered in the order they are sent</td>
</tr>
<tr>
<td>Message length</td>
<td>Maximum length of a single message</td>
</tr>
<tr>
<td>Setup retry count</td>
<td>Specifies maximum number of retries to start up the remote MCA</td>
</tr>
<tr>
<td>Delivery retries</td>
<td>Maximum times MCA will try to put received message into queue</td>
</tr>
</tbody>
</table>
The general organization of an MQSeries queuing network using routing tables and aliases.
## Message Transfer (2)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQopen</td>
<td>Open a (possibly remote) queue</td>
</tr>
<tr>
<td>MQclose</td>
<td>Close a queue</td>
</tr>
<tr>
<td>MQput</td>
<td>Put a message into an opened queue</td>
</tr>
<tr>
<td>MQget</td>
<td>Get a message from a (local) queue</td>
</tr>
</tbody>
</table>

Primitives available in an IBM MQSeries MQI