Module 1 - Distributed System Architectures & Models

System Architecture

- Defines the structure of the system
  - components identified
  - functions of each component defined
  - interrelationships and interactions between components defined
Standardization

Reference Model

- A conceptual framework whose purpose is to divide standardization work into manageable pieces and to show at a general level how these pieces are related to one another.

Approaches

- **Component-based**
  - Components of the system are defined together with the interrelationships between components.
  - Good for design and implementation of the system.

- **Function-based**
  - Classes of users are identified together with the functionality that the system will provide for each class.
  - The objectives of the system are clearly identified. But how do you achieve these objectives?
  - Example: ISO/OSI Model

Software Layers
Layers

- **Platform**
  - Fundamental communication and resource management services
  - We won’t be worried about these

- **Middleware**
  - Provides a service layer that hides the details and heterogeneity of the underlying platform
  - Provides an “easier” API for the applications and services
  - Can be as simple as RPC or as complex as OMA
    - RPC (Remote Procedure Call): simple procedure call across remote machine boundaries
    - OMA (Object Management Architecture): an object-oriented platform for building distributed applications

- **Applications**
  - Distributed applications, services
  - Examples: e-mail, ftp, etc

---

Hardware Organization

![Hardware Organization Diagram](image-url)
Software Organization

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
<th>Main Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOS</td>
<td>Tightly-coupled operating system for multi-processors and homogeneous multicomputers</td>
<td>Hide and manage hardware resources</td>
</tr>
<tr>
<td>NOS</td>
<td>Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)</td>
<td>Offer local services to remote clients</td>
</tr>
<tr>
<td>Middleware</td>
<td>Additional layer atop of NOS implementing general-purpose services</td>
<td>Provide distribution transparency</td>
</tr>
</tbody>
</table>

System Architectures

- Client-server
  - Multiple-client/single-server
  - Multiple-client/multiple-servers
  - Mobile clients
  - Thin clients
- Multitier systems
- Peer-to-peer systems
Multiple-Client/Single Server

- User interface
- Programmatic interface
- other application support environments

Advantages of Client/Server Computing

- More efficient division of labor
- Horizontal and vertical scaling of resources
- Better price/performance on client machines
- Ability to use familiar tools on client machines
- Client access to remote data (via standards)
- Full DBMS functionality provided to client workstations
- Overall better system price/performance
Client-Server Communication

Client

Request (invocation)

Process

Result

Computer

Request (invocation)

Client

Result

Server

Client-Server Timing

- 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 { /* sender's identity */
    long source; /* receiver's identity */
    long dest; /* requested operation */
    long opcode; /* number of bytes to transfer */
    long count; /* position in file to start I/O */
    long offset; /* 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)

- A sample server.

```c
#include <header.h>

void main(void) {
    struct message m1, m2; /* incoming and outgoing messages */
    int r; /* result code */

    while(TRUE) { /* server runs forever */
        receive(FILE_SERVER, &m1); /* block waiting for a message */
        switch(opcode) { /* dispatch on type of request */
            case CREATE: r = do_create(&m1, &m2); break;
            case READ: r = do_read(&m1, &m2); break;
            case WRITE: r = do_write(&m1, &m2); break;
            case DELETE: r = do_delete(&m1, &m2); break;
            default: r = E_BAD_OPCODE;
        }
        m2.result = r; /* return result to client */
        send(m1.source, &m2); /* send reply */
    }
}
```
An Example Client and Server (3)

- A client using the server to copy a file.

```c
#include <header.h>
int copy(char *src, char *dst)

struct message ml;
long position;
long client = 110;

initialize();
position = 0;
do {
   mlOpcode = READ;
   mlOffset = position;
   mlCount = BUF_SIZE;
   strcpy(ml.name, src);
   send(FILESERVER, &ml);
   receive(client, &ml);
   /* Write the data just received to destination file.
   mlOpcode = WRITE;
   mlOffset = position;
   mlCount = ml.result;
   strcpy(ml.name, dst);
   position += ml.result;
   while (ml.result > 0) {
      ml.result = number of bytes written
      send(FILE_SERVER, &ml);
      receive(client, &ml);
      block waiting for reply
      block waiting for reply
      return(ml.result > 0 ? OK : ml.result);
   }
}
```

Problems With Multiple-Client/Single Server

- Server forms bottleneck
- Server forms single point of failure
- System scaling difficult
Multiple Clients/Multiple Servers


Multiple-Client/Multiple-Server Communication

Service Across Multiple Servers


Mobile Computing
Example Mobile Computing Environment

These types of environments are commonly called “spontaneous systems” or “pervasive computing”

“Thin” Clients

- Thin Clients and Compute Servers
  - Executing graphical user interface on local computer while application executes on compute server
  - Example: X11 server (run on the application client side)
  - In reality: Palm Pilots, Mobile phones
Multitier Systems

- Servers are clients of other servers
- Example:
  - Web proxy servers

Multitier Systems (2)

- Example: Internet Search Engines
Multitier System Alternatives

Communication in Multitier Systems
Peer-to-Peer Systems

Example Client/Server Middleware

- Remote Procedure Call (RPC)
  - Uses the well-known procedure call semantics.
  - The caller makes a procedure call and then waits. If it is a local procedure call, then it is handled normally; if it is a remote procedure, then it is handled as a remote procedure call.
  - Caller semantics is blocked send; callee semantics is blocked receive to get the parameters and a nonblocked send at the end to transmit results.
Example Client/Server Middleware

- Object Management Architecture (OMA)
  - Object-oriented

OMA Modules

- **Object Request Broker**: directs requests and answers between objects
- **Object Services**: basic functions for object management (e.g., a name service)
- **Common Facilities**: generic object-oriented tools for various applications (e.g., a class browser)
- **Application Objects**: classes specific to an application domain (e.g., a CASE tool)
OMG Object Model

- Generalized object model including objects, values (object names and handles), operations, signatures, types and classes
- An object is an abstraction with a state and a set of operations
- A request is an operation call with one or more parameters, any of which may identify an object (multi-targeting); Arguments and results are passed by value

CORBA Features

- Communications substrate
- A specific programmer interface (no implementation)
- Multi-vendor ORBs to interoperate (CORBA-2)
- Layered on top of other communication substrates (RPC, byte streams, IPC,…)
- Language mapping (C, C++, Ada, Smalltalk, Java, Cobol)
- Object adapters
- ORB interface
- Interface repository (IR)
OMA’s Client-Server Structure

ORB Functions

- Request dispatch to determine the identity of a method to be called
- Parameter encoding to convey local representations of parameter values
- Delivery of request and result messages to the proper nodes
- Synchronization of the requesters with the responses of the requests
- Activation and deactivation of persistent objects, e.g., Using the ODBMS servers
- Exception handling to report various failures to requesters and servers
- Security mechanisms to assure the secure conveyance of messages among objects
ORB Structure

CORBA Method Invocations

- Static
  - Faster
    - Everything at compile time
  - Easier to program
    - Invoke a method on an identified object
  - Static type checking
  - Code is self-documenting

- Dynamic
  - Flexible
    - Code genericity
  - Run-time addition of classes
    - Needed for tools (e.g., Browsers)
Design Requirements of Distributed Systems

- **Performance**
  - Metrics
  - Load balancing
  - Scalability
  - Use of caching and replication

- **Quality of Service**
  - Functional as well as performance

- **Dependability**
  - Reliability
  - Fault tolerance

- **Security**

---

**Performance**

- **Performance metrics**
  - Response time
    - How long it takes from a user’s request until the result is returned to the user
    - Average response time is usually meaningful
  - Throughput
    - Number of jobs per unit time
  - System utilization
  - Amount of network capacity consumed

- **Performance Problem**
  - Exchange of messages is typically slow (about 1 msec return trip time on a LAN, mainly due to protocol handling)
Changing Network Characteristics

Messaging Overhead

<table>
<thead>
<tr>
<th>Hardware Overhead</th>
<th>Software Overhead</th>
<th>New Protocols (e.g., Infiniband, FiberChannel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Mbps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 Mbps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Gbps</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalability

- Distributed systems operate at many different scales
  - Two workstations and a file server
  - The CS computers…
- Most current distributed systems designed to work with few hundred CPUs
- Future systems will be orders of magnitude larger

<table>
<thead>
<tr>
<th>Computers in the Internet</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Computers</td>
<td>Web servers</td>
<td>Percentage</td>
</tr>
<tr>
<td>1979, Dec.</td>
<td>188</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1989, July</td>
<td>130,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999, July</td>
<td>56,218,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computers vs. Web servers in the Internet</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Computers</td>
<td>Web servers</td>
<td>Percentage</td>
</tr>
<tr>
<td>1993, July</td>
<td>1,776,000</td>
<td>130</td>
<td>0.008</td>
</tr>
<tr>
<td>1995, July</td>
<td>6,642,000</td>
<td>23,500</td>
<td>0.4</td>
</tr>
<tr>
<td>1997, July</td>
<td>19,540,000</td>
<td>1,203,096</td>
<td>6</td>
</tr>
<tr>
<td>1999, July</td>
<td>56,218,000</td>
<td>6,598,697</td>
<td>12</td>
</tr>
</tbody>
</table>
Guiding Performance Principles

- Distribution provides opportunities for parallelism that can improve performance
  - Inter-process vs intra-process parallelism
  - Pay considerable attention to the size of computations
    - Fine-grained parallelism: large number of small computations, interact highly with one another ⇒ potential trouble
    - Coarse-grained parallelism: large computations, low interaction rates, little data ⇒ maybe better
- Often the more important question is not can you scale, but can you scale well
  - Solutions for 200 machines will fail miserably for 2,000,000
- Designers should try to avoid the potential bottlenecks:
  - Centralized components (e.g., A single mail server for all users)
  - Centralized tables (e.g., A single Directory Name Server)
  - Centralized algorithms (e.g., Routing based on complete information)
- Difference between bandwidth and latency
- Use caching and data replication for performance reasons (replication can be used for dependability reasons as well)

Quality of Service (QoS)

- How well the system can meet the service requirements specified for it
- Functional
  - Whether the system can perform the specified functions
- Non-functional
  - How well the system can perform these functions with respect to
    - Performance
    - Reliability
    - Availability
- Examples
  - Frame rates, jitter in video transmission
  - Real-time constraints
  - Database access time requirements
  - 24/7 requirements
Dependability

- **Reliability**
  - A measure of success with which a system conforms to some authoritative specification of its behavior.
  - Probability that the system has not experienced any failures within a given time period.
  - Typically used to describe systems that cannot be repaired or where the continuous operation of the system is critical.

- **Availability**
  - The fraction of the time that a system meets its specification.
  - The probability that the system is operational at a given time $t$.

---

How to Improve Dependability

- **Hardware and software redundancy - masking**
  - Take a distributed system with 4 file servers, each with a 0.95 chance of being up at any instant
  - The probability of all 4 being down simultaneously is $0.05^4 = 0.000006$
  - So the probability of at least one being available (i.e., the reliability of the full system) is $0.999994$, far better than 0.95
  - If there are 2 servers, then the reliability of the system is $(1-0.05^2) = 0.9975$

- A design that does not require simultaneous functioning of a substantial number of critical components
Hardware Redundancy

- Two computers are employed for a single application, one acting as a standby
  - Very costly, but often very effective solution
- Redundancy can be planned at a finer grain
  - Individual servers can be replicated
  - Redundant hardware can be used for non-critical activities when no faults are present
  - Redundant routes in network

Software Redundancy

- Distributed Systems are foremost highly complex software systems
  - Nortel Networks DMS-100 switch: 25-30 million lines of code, 3000 software developers, 20 years life cycle to date.
  - Motorola: 20% of engineers produce hardware, 80% produce software
  - Subject to all kinds of software engineering problems
- Software redundancy requires
  - Process redundancy
    - When one process fails, another one can take over
  - Data redundancy
    - The state of permanent data can be recovered or “rolled back” when a fault is detected ⇒ transaction processing
    - Dilemma: Data replication improves availability, but increases the chances that they will be inconsistent, especially if updates are frequent
      - Replication protocols
Security

- Security is a big issue in computing in general, but even more so in distributed computing
  - Communication
  - Distributed resources
  - Infrastructure attacks
- Files/resources must be protected from unauthorized usage
- Problems
  - Threats to processes
  - Threats to communication channels
  - Denial of service attacks
- Some issues
  - Privacy
  - Authentication
  - Availability
  - Integrity

Summary

- Distributed systems consist of autonomous CPUs that work together to make the complete system look like a single computer
- System Models
  - Client-Server: Most widely used
    - Variants of Client-Server Model
      - Service provided by multiple servers, proxy servers, mobile code, mobile agents, network computers, thin clients, “spontaneous” (or “pervasive”) networking
  - Peer-to-peer Model
- Key issues when designing distributed systems:
  - Transparency, reliability, performance, scalability, …