Principles of Object-Oriented Middleware

Outline

- Computer Networks
- Types of Middleware
  - Transaction-Oriented Middleware
  - Message-Oriented Middleware
  - Remote Procedure Calls
- Object-Oriented Middleware
- Developing with Object-Oriented Middleware

Computer Networks
**Transport Layer**

- **Level 4 of ISO/OSI reference model.**
- **Concerned with the transport of information through a network.**
- **Two facets in UNIX/Windows networks:**
  - TCP and
  - UDP.

**Transmission Control Protocol (TCP)**

- Provides bi-directional stream of bytes between two distributed components.
- UNIX rsh, rcp and rlogin are based on TCP.
- Reliable but slow protocol.
- Buffering at both sides de-couples computation speeds.
User Datagram Protocol (UDP)

- Enables a component to pass a message containing a sequence of bytes to another component.
- Other component is identified within message.
- Unreliable but very fast protocol.
- Restricted message length.
- Queuing at receiver.
- UNIX rwho command is UDP based.

UDP for Request Implementation
Types of Middleware

Direct Use of Network Protocols implies

- Manual mapping of complex request parameters to byte streams
- Manual resolution of data heterogeneity
- Manual identification of components
- Manual implementation of component activation
- No guarantees for type safety
- Manual synchronization of interaction between distributed components
- No quality of service guarantees

Middleware

- Layered between Application and OS/Network
- Makes distribution transparent
- Resolves heterogeneity of
  - Hardware
  - Operating Systems
  - Networks
  - Programming Languages
- Provides development and run-time environment for distributed systems.
**Forms of Middleware**

- **Transaction-Oriented**
  - IBM CICS
  - BEA Tuxedo
  - Encina

- **Message-Oriented**
  - IBM MQSeries
  - DEC Message Queue
  - NCR TopEnd

- **RPC Systems**
  - ANSA
  - Sun ONC
  - OSF/DCE

- **Object-Oriented**
  - OMG/CORBA
  - DCOM
  - Java/RMI

- **First look at RPCs to understand origin of object-oriented middleware**

---

**Remote Procedure Calls**

- Enable procedure calls across host boundaries
- Call interfaces are defined using an Interface Definition Language (IDL)
- RPC compiler generates presentation and session layer implementation from IDL

---

**IDL Example (Unix RPCs)**

```c
const NL=64;
struct Player {
  struct DoB {int day; int month; int year;}  
  string name<NL>;
};

program PLAYERPROG {
  version PLAYERVERSION {
    void PRINT(Player)=0;
    int STORE(Player)=1;
    Player LOAD(int)=2;
    } = 0;
  } = 105040;
```
**ISO/OSI Presentation Layer**

Resolution of data heterogeneity

- Common data representation
- Transmission of data declaration

Marshalling and Unmarshalling

- **Marshalling**: Disassemble data structures into transmittable form
- **Unmarshalling**: Reassemble the complex data structure.

```c
char * marshal() {
    char * msg;
    msg = new char[4*(sizeof(int)+1) + strlen(name)+1];
    sprintf(msg,"%d %d %d %d %s",
            dob.day,dob.month,dob.year,
            strlen(name),name);
    return(msg);
}

void unmarshal(char * msg) {
    int name_len;
    sscanf(msg,"%d %d %d %d ",
           &dob.day,&dob.month,
           &dob.year,&name_len);
    name = new char[name_len+1];
    sscanf(msg,"%d %d %d %d %s",
           &dob.day,&dob.month,
           &dob.year,&name_len,name);
}
```

**Method Call vs. Object Request**

- **Caller**
- **Called**
- **Stub**

Transport Layer (e.g. TCP or UDP)
Stubs

- Creating code for marshalling and unmarshalling is tedious and error-prone.
- Code can be generated fully automatically from interface definition.
- Code is embedded in stubs for client and server.
- Client stub represents server for client, Server stub represents client for server.
- Stubs achieve type safety.
- Stubs also perform synchronization.

Synchronization

- Goal: achieve similar synchronization to local method invocation
- Achieved by stubs:
  - Client stub sends request and waits until server finishes
  - Server stub waits for requests and calls server when request arrives

Type Safety

- How can we make sure that
  - servers are able to perform operations requested by clients?
  - actual parameters provided by clients match the expected parameters of the server?
  - results provided by the server match the expectations of client?
- Middleware acts as mediator between client and server to ensure type safety.
- Achieved by interface definition in an agreed language.
### Facilitating Type Safety

- Client
- Server
- Request
- Reply
- Interface Definition

### Session Layer

- **Implements**
  - Identification of RPC servers
  - Activation of RPC servers
  - Dispatch of operations

<table>
<thead>
<tr>
<th>Application</th>
<th>Presentation</th>
<th>Session</th>
<th>Transport</th>
<th>Network</th>
<th>Data link</th>
<th>Physical</th>
</tr>
</thead>
</table>

### Example: RPC Server Identification

```c
print_person(char * host, Player * pers) {
    CLIENT *clnt;
    clnt = clnt_create(host, 105040, 0, "udp");
    if (clnt == (CLIENT *) NULL) exit(1);
    if (print_0(pers, clnt) == NULL)
        clnt_perror(clnt, "call failed");
    clnt_destroy(clnt);
}
```
Object-Oriented Middleware

Interface Definition Language

- Every object-oriented middleware has an interface definition language (IDL)
- Beyond RPCs, object-oriented IDLs support object types as parameters, failure handling and inheritance
- Object-oriented middleware provide IDL compilers that create client and server stubs to implement session and presentation layer

IDL Example

```c
interface Player : Object {
    typedef struct _Date {
        short day; short month; short year;
    } Date;
    attribute string name;
    readonly attribute Date DoB;
};
interface PlayerStore : Object {
    exception IDNotFound{};
    short save (in Player p);
    Player load(in short id) raises(IDNotFound);
    void print (in Player p);
};
```
Presentation Layer Implementation

- In addition to RPC presentation layer implementation, object-oriented middleware needs to
  - define a transport representation for object references
  - deal with exceptions
  - need to marshal inherited attributes

Session Layer Implementation

[Diagram showing Object References, Hosts, Processes, Objects]

Developing with Object-Oriented Middleware
### Development Steps

- **Design**
- **Interface Definition**
- **Server Stub Generation**
- **Client Stub Generation**
- **Server Coding**
- **Client Coding**
- **Server Registration**

### Facilitating Access Transparency

- **Client stubs have the same operations as server objects**
- **Hence, clients can**
  - make local call to client stub
  - or local call to server object without changing the call.
- **Middleware can accelerate communication if objects are local by not using the stub.**

### Facilitating Location Transparency

- **Object identity**
- **Object references**
- **Client requests operation from server object identified by object reference**
- **No information about physical location of server necessary**
- **How to obtain object references?**
**Server Registration**

- Object adapters need to be able to locate and start servers
- Server objects are registered in some form of implementation repository
- Registration processes is middleware and product-specific
- Object adapter performs implementation repository lookup prior to activation

**Key Points**

- Middleware builds on the transport layer
- There are several forms of middleware
- Object-oriented middleware provide IDLs
- Object-oriented middleware implements session and presentation layer
- Presentation layer implementation in client/server stubs is derived from IDL
- Session layer is implemented in object adapters