Outline

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

Computer Networks

- The communication between processes on different machines
  - At the high levels:
    - a connection to another process on another machine to simply send each message as a high level logical unit
    - Simple strings or arrays of bytes (binary data).
  - At the underlying layers:
    - Split message into smaller units for physical transfer called PDU’s (protocol data units).
    - A wrapper with Address of sender/receiver, Error detection codes, Protocol control information

Outline

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

Computer Networks

- The communication between processes on different machines
  - At the high levels:
    - a connection to another process on another machine to simply send each message as a high level logical unit
    - Simple strings or arrays of bytes (binary data).
  - At the underlying layers:
    - Split message into smaller units for physical transfer called PDU’s (protocol data units).
    - A wrapper with Address of sender/receiver, Error detection codes, Protocol control information
Transport Layer

- Concerned with the transport of information through a network.
- Two facets in UNIX/Windows networks:
  - TCP
  - UDP

Protocols

- An agreement between entities on the transmission of data.
- Different layers of network software have different concerns.
  - At lower levels: determining the amount of data to be sent, the size of packets and the speed of transmission.
    - Connection Control/Data transfer
    - Flow control/Error control
    - Synchronization/Addressing
  - At higher levels: cover the syntax and sequencing of logical messages.

Transmission Control Protocol (TCP)

- Provides bi-directional stream of bytes between two distributed components by establishing a virtual connection
  - Steps: Connection establishment, Data transfer, Connection termination.
  - Reliable: send logical messages without having to explicitly deal with these other issues (sequencing and re-transmission of lost packets).
  - Slow: as more time is required to set up and terminate the link and buffering at both sides decouples computation speeds.
  - UNIX rsh, rcp and rlogin are based on TCP.
TCP for Request Implementation

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>Presentation</td>
<td>Presentation</td>
</tr>
<tr>
<td>Session</td>
<td>Session</td>
</tr>
<tr>
<td>Requests</td>
<td>Input Stream</td>
</tr>
<tr>
<td>Transport</td>
<td>Transport</td>
</tr>
<tr>
<td>Output Stream</td>
<td></td>
</tr>
</tbody>
</table>

Connection-oriented Communication

User Datagram Protocol (UDP)

- Enables a component to pass a message containing a sequence of bytes (packet) to another component:
  - A program may send information to another at an indeterminate time with no explicit prior co-ordination
  - the packet may get lost or packets may arrive in a different order to that in which they were sent.
  - reorganize the packets and make requests for new packets when problems occur.
  - dynamically routed; split into a number of PDUs that take different routes to their destination depending on network congestion.

- Unreliable but very fast protocol: restricted message length, queuing at receiver, e.g. UNIX rwho command.
UDP for Request Implementation

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>Presentation</td>
<td>Presentation</td>
</tr>
<tr>
<td>Session</td>
<td>Session</td>
</tr>
<tr>
<td>Transport</td>
<td>Transport</td>
</tr>
</tbody>
</table>

Request Datagrams

Result Datagrams

Datagram Communication

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 TopFind
- RPC Systems
  - ANSA
  - Sun ONC
  - OSF/DCE
- Object-Oriented
  - OMG/CORBA
  - DCOM
  - JavaRMI
- 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)

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

- static
- dynamic

Marshalling and Unmarshalling

- Marshalling: Disassemble data structures into transmittable form

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

- Unmarshalling: Reassemble the complex data structure

```
void unmarshal(char * msg) {
    int name_len;
    sscanf(msg,"%d %d %d %d %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);
    return(msg);
}
```
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

Session Layer

- Implements
  - identification of RPC servers
  - activation of RPC servers
  - dispatch of operations
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);
}
```

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

Why do we use an IDL?
IDL Example (OO Middleware)

```idl
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
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?

Stub Generation

The CORBA IDL is translated into stubs and skeletons (server stubs). IDL-compiler generates four files (two for client stubs and two for server stubs).

Client and Server Implementation

The client implementation (Client.cc) includes Teamcl.hh so as to enable the compiler to check consistency between the declaration and use of client stub implementation. When translated to an executable, Client.cc is linked with the object code of the client stub (Teamcl.cc). Similar to Server.
Type Safe Server Object Implement.

Inheritance is used in classes Play_Dispatch and Player_Impl, which are contained in the server stub for object type Player.

An Interface-based implementation with the relationships of interfaces and classes involved on the server side.

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 provides 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