Communication in Distributed Systems

Yao Liu
Outline

• Fundamentals and Types of Communication
• RPC/RMI
• Message-oriented Transient Communication
• Message-oriented Persistent Communication
• Data Streaming
• Multicast
Fundamental issues

• Layered Protocols
• Types of communication
  • Synchronous vs. Asynchronous
  • Transient vs. Persistent
Layered protocols (1)

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

Message

Data link layer header
Network layer header
Transport layer header
Session layer header
Presentation layer header
Application layer header

Bits that actually appear on the network

Data link layer trailer
Middleware Protocols

Application protocol

Middleware protocol

Transport protocol

Network protocol

Data link protocol

Physical protocol

Network
Types of communication

- **Transient vs. persistent** communication
- **Asynchronous vs. synchronous** communication
Persistence and synchronicity in communication (1)

- (a) Persistent asynchronous communication
- (b) Persistent synchronous communication

[Diagrams showing communication processes (a) asynchronous and (b) synchronous]
Persistence and synchronicity in communication (2)

• (c) Transient asynchronous communication
• (d) Receipt-based transient synchronous communication
Persistence and synchronicity in communication (3)

- (e) Delivery-based transient synchronous communication at message delivery
- (f) Response-based transient synchronous communication
Communication models

• Remote Procedure Call (RPC)
• Message Oriented Middleware (MOM)
• Data Streaming
• Multicast
Transient vs. persistent communication

- RPC and RMI (by default) support communication between two processes that are executing at the same time
  - transient communication
- Typically, the client is blocked until the RPC/RMI returns
  - synchronous communication
- Not suitable for middleware that integrates applications in widely dispersed and large-scale distributed systems
  - Message-oriented middleware
Outline

• Fundamentals and Types of Communication
• RPC/RMI
• Message-oriented Transient Communication
• Message-oriented Persistent Communication
• Data Streaming
• Multicast
**RPC/RMI - Motivation**

- **Sockets API** $\equiv$ send & recv calls $\equiv$ I/O
- **Remote Procedure Calls (RPC)**
  - Goal: to provide a procedural interface for distributed (i.e., remote) services
  - To make distributed nature of service transparent to the programmer
- **Remote Method Invocation (RMI)**
  - RPC + Object Orientation
  - Allows objects living in one process to invoke methods of an object living in another process
Middleware layers

- Applications, services
- RMI and RPC
  - request-reply protocol
  - marshalling and external data representation
- UDP and TCP
Remote Procedure Call

- Principle of RPC between a client and server program.

Client:
- Call remote procedure
- Wait for result
- Request
- Reply

Server:
- Call local procedure and return results
- Return from call

Time
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
Passing value parameters (1)

1. Client call to procedure
2. Stub builds message
3. Message is sent across the network
4. Server OS hands message to server stub
5. Stub unpacks message
6. Stub makes local call to "add"
Passing value parameters (2)

- Data representation: not a problem when processes are on the same machine
- Different machines have different representations
- e.g., different computers may store bytes of an integer in different order in memory (e.g., little endian, big endian), called “host byte order”

The little numbers in boxes indicate the address of each byte.

Original message on Intel 486

Message after receipt on SPARC

Message after being inverted

The little numbers in boxes indicate the address of each byte.
Passing reference parameters

- Call-by-reference not possible: the client and server don’t share an address space. That is, addresses referenced by the server correspond to data residing in the client's address space.

- One approach is to simulate call-by-reference using copy-restore. In copy-restore, call-by-reference parameters are handled by sending a copy of the referenced data structure to the server, and on return replacing the client's copy with that modified by the server.

- However, copy-restore doesn’t work in all cases. For instance, if the same argument is passed twice, two copies will be made, and references through one parameter only changes one of the copies.
Parameter specification and stub generation

- Once a standard has been agreed upon for representing each of the basic data types, given a parameter list and a message, it is possible to deduce which bytes belong to which parameter, and thus to solve the problem.

- Given these rules, the requesting process knows that it must use this format, and the receiving process knows that incoming message will have this format.

- Having the type information of parameters makes it possible to make necessary conversions.

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

---

<table>
<thead>
<tr>
<th>foobar</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>z[0]</td>
</tr>
<tr>
<td></td>
<td>z[1]</td>
</tr>
<tr>
<td></td>
<td>z[2]</td>
</tr>
<tr>
<td></td>
<td>z[3]</td>
</tr>
<tr>
<td></td>
<td>z[4]</td>
</tr>
</tbody>
</table>
Request-reply communication

Client
- doOperation
  - (wait)
  - (continuation)

Server
- getRequest
- select procedure
- Execute procedure
- sendReply

Request message
Reply message
### Style of request-reply protocols

<table>
<thead>
<tr>
<th>Name</th>
<th>Messages sent by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Client</td>
</tr>
<tr>
<td>R</td>
<td>Request</td>
</tr>
<tr>
<td>RR</td>
<td>Request</td>
</tr>
<tr>
<td>RRA</td>
<td>Request</td>
</tr>
</tbody>
</table>
Operations of the request-reply protocol

```java
public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
    sends a request message to the remote server and returns the reply.
    The arguments specify the remote server, the operation to be invoked and the
    arguments of that operation.

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.
```
# Request-reply message structure

<table>
<thead>
<tr>
<th>messageType</th>
<th>int  ( (0=\text{Request}, 1=\text{Reply}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestId</td>
<td>int</td>
</tr>
<tr>
<td>remoteReference</td>
<td>( \text{RemotetRef} )</td>
</tr>
<tr>
<td>operationId</td>
<td>int or ( \text{Operation} )</td>
</tr>
<tr>
<td>arguments</td>
<td>array of bytes</td>
</tr>
</tbody>
</table>
Writing a client and a server

- The steps in writing a client and a server in DCE RPC (SUN RPC is similar)
Writing a client and a server (2)

- Three files output by the IDL compiler:
  - A header file (e.g., interface.h, in C terms).
  - The client stub.
  - The server stub.
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};

struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
    version VERSION {
        void WRITE(writeargs) = 1;
        Data READ(readargs) = 2;
    } = 2;
} = 9999;
Binding a client to a server (1)

- Client-to-server binding in DCE.
Binding a client to a server (2)

- Registration of a server makes it possible for a client to locate the server and bind to it.

- Server location is done in two steps:
  1. Locate the server’s machine.
  2. Locate the server on that machine.
RPC in presence of failures

• Five different classes of failures can occur in RPC systems
  • The client is unable to locate the server
  • The request message from the client to the server is lost
  • The reply message from the server to the client is lost
  • The server crashes after receiving a request
  • The client crashes after sending a request
Client cannot locate the server

• Examples:
  • Server might be down
  • Server might moved
  • Server evolves (new version of the interface installed and new stubs generated) while the client is compiled with an older version of the client stub

• Possible solutions:
  • Use special code, such as “-1”, as the return value of the procedure to indicate failure. In Unix, add a new error type and assign the corresponding value to the global variable errno.
    • -1 can be a legal value to be returned, e.g., sum(7, -8)
  • Have the error raise an exception or a signal.
    • Writing an exception/signal handler destroys the transparency…
Lost request message

• Client starts a timer when sending the message:
  • If timer expires before reply or ACK comes back: Client retransmits
  • If message truly lost: server will not differentiate between original and retransmission \(\rightarrow\) everything will work fine
  • If many requests are lost: client gives up and falsely concludes that the server is down \(\rightarrow\) we are back to “Cannot locate server”
Lost reply message

- Client starts a timer when sending the message:
  - If timer expires before reply comes back: Retransmits the request
    - Problem: Not sure why no reply (reply/request lost or server slow)?
  - If server is just slow: The procedure will be executed several times
    - Problem: What if the request is not idempotent, e.g., money transfer

- Server choices:
  - Re-execute procedure → service should be idempotent so that it can be repeated safely
  - Filter duplicates → server should hold on to results until acknowledged
    - Client assigns sequence numbers to requests to allow server to differentiate retransmissions from original
Problem: client cannot differentiate between (b) and (c)

- Wait until the server reboots and try the operation again. Guarantees that RPC has been executed at least one time (at-least-once semantics).
- Give up immediately and report back failure. Guarantees that RPC has been carried out at most one time (at-most-once semantics).
- Client gets no help about what happened. Guarantees nothing (RPC may have been carried out anywhere from 0 to a large number). Easy to implement.

Exactly once impossible to achieve.
Client crashes

- Client sends a request and crashes before the server replies: a computation is active and no parent is waiting for result (orphan)
  - Orphans waste CPU cycles and can lock files or tie up valuable resources
  - Orphans can cause confusion (client reboots and does RPC again, but the reply from the orphan comes back immediately afterwards)

- Possible solutions
  - **Extermination:**
    - Before a client stub sends an RPC, it makes a log entry (in safe storage) telling what it is about to do. After a reboot, the log is checked and the orphan explicitly killed off.
    - Expense of writing a disk record for every RPC; orphans may do RPCs, thus creating grand-orphans impossible to locate; impossibility to kill orphans if the network is partitioned due to failure.
Client crashes (2)

- Possible solutions (cont’d)
  - **Reincarnation:**
    - Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring the start of a new epoch. When broadcast comes, all remote computations are killed. Solve problem without the need to write disk records
    - If network is partitioned, some orphans may survive. But, when they report back, they are easily detected given their obsolete epoch number
  - **Expiration:**
    - Each RPC is given a standard amount of time, \( T \), to do the job. If it cannot finish, it must explicitly ask for another quantum.
    - Choosing a reasonable value of \( T \) in the face of RPCs with wildly differing requirements is difficult
## Invocation semantics

<table>
<thead>
<tr>
<th>Retransmit request message</th>
<th>Duplicate filtering</th>
<th>Re-execute procedure or retransmit reply</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Maybe</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Re-execute procedure</td>
<td>At-least-once</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Retransmit reply</td>
<td>At-most-once</td>
</tr>
</tbody>
</table>

- **Maybe**
  - Invoker knows nothing about whether operation is executed
  - No fault tolerance mechanism used
Invocation semantics

<table>
<thead>
<tr>
<th>Retransmit request message</th>
<th>Duplicate filtering</th>
<th>Re-execute procedure or retransmit reply</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Maybe</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Re-execute procedure</td>
<td>At-least-once</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Retransmit reply</td>
<td>At-most-once</td>
</tr>
</tbody>
</table>

• At-least-once
  • Invoker receives result, or exception, if no result was received
  • If result received, invoker knows that operation was at least executed once, but knows nothing if exception occurs
  • Achieved through retransmissions of requests and re-executions
  • Can suffer from the following types of failures
    • Crash failure of server containing remote object
    • Arbitrary failures caused through re-execution of non-idempotent operation on server
## Invocation semantics

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request message</td>
<td>Duplicate filtering</td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- **At-most-once**
  - Invoker receives:
    - result, in which case s/he knows that operation was executed exactly once, or
    - exception, informing that no result was received, in which case the operation was either performed once, or not at all
  - Requires usage of all fault tolerance measures
RPC recap

• Logical extension of the procedure call interface
  • Easy for the programmer to understand and use
• Requires an Interface Definition Language (IDL) to specify data types and procedure interfaces
  • Provides language independence
• IDL compiler generates client and server stubs
  • Stubs handle the marshalling and unmarshalling of arguments and builds the message
  • Messages must be represented in a machine independent data representation
  • Must serialize complex types
• Calling and called procedures reside in different address space
  • Cannot send pointers or system specific data (locks, file descriptors, pipes, sockets, etc.)
  • Parameter passing can therefore be very expensive
RPC recap (2)

• New failure models:
  • Cannot locate server (throw exception)
  • Lost request message (resend request)
  • Lost reply message
    • Retransmit request without incrementing the request sequence number
    • Requires that the server retain old replies for some set time period
  • Server crashes before sending reply
    • RPC resends request (at least once semantics)
    • Give up and report failure (at most once semantics)
      • Or server saves all replies on persistent storage and re-send previous replies
      • Client gets no help, semantics determined by the client
  • Client crashes
RPC example: Apache Thrift

- You will be using this for your projects
- Developed at Facebook and open-sourced in 2007
- Specifies its own IDL

```java
enum Operation {
    ADD = 1,
    SUBTRACT = 2,
    MULTIPLY = 3,
    DIVIDE = 4
}

struct Work {
    1: i32 num1 = 0,
    2: i32 num2,
    3: Operation op,
    4: optional string comment,
}

service Calculator {
    void ping(),
    i32 add(1:i32 num1, 2:i32 num2),
    i32 calculate(1:i32 logid, 2:Work w) throws (1:InvalidOperation ouch),
}

//IDL compiler generates client/server stubs

// Server has to implement the handler for the service

class CalculatorHandler:
    def __init__(self):
        #Your initialization goes here
    def ping(self):
        #Your implementation goes here
    def add(self, n1, n2):
        #Your implementation goes here
    def calculate(self, logid, work):
        #Your implementation goes here

class CalculatorHandler implements Calculator.Iface {
    public CalculatorHandler() {
        // Your initialization goes here
    }
    public void ping() {
        // Your implementation goes here
    }
    public int add(int n1, int n2) {
    }
    public int calculate(int logid, Work work) throws InvalidOperation {
    }
```
RPC example: Apache Thrift

• Supports
  • C++, C#, Cocoa, D, Delphi, Erlang, Haskell, Java, OCaml, Perl, PHP, Python, Ruby, Smalltalk

• Different data encoding types
  • Binary, JSON

• IDL provides relatively simple datatypes
  • Primitives
  • Simple structures (structs, lists, maps)

• At most once semantics
• Does not provide any solution for orphans
RPC example: Apache Thrift

- Client and server consist of three main layers:
  - written code
  - generated code
  - the Apache Thrift libraries
- The written code allows to choose
  - marshalling mechanisms (TProtocol)
  - I/O mechanisms are used (TTransport)
RPC example: Apache Thrift

- Thrift Transport defines:
  - How is transmitted? In terms of I/O operations

- Thrift transports (implementations of TTransport interface)
  - TSocket
    - Using blocking I/O for transport
  - TNonblockingSocket
    - non-blocking I/O Operations
  - TFileTransport
    - Abstraction of an on-disk-file to a data stream.
  - TMemoryTransport
    - Allows directly reading from the heap or stack memory owned by the process
  - ...
RPC example: Apache Thrift

- Thrift Protocol defines:
  - What is transmitted?
  - i.e., how message is constructed, encoding, decoding, (marshalling)
- Thrift protocols (implementations of TProtocol interface)
  - TBinaryProtocol:
    - Numeric values are encoded binary instead of text
  - TCompactProtocol:
    - Efficient and dense encoding of data
  - TJSONProtocol:
    - User JSON for encoding of data
  - …
RPC example: Apache Thrift

- The real service procedures (e.g., CalculatorHandler) are encapsulated in a TProcessor object in the generated code.

```python
def main():
    handler = CalculatorHandler()
    processor = Calculator.Processor(handler)
    transport = TSocket.TServerSocket(port=9090)
    tfactory = TTransport.TBufferedTransportFactory()
    pfactory = TBinaryProtocol.TBinaryProtocolFactory()
    server = TServer.TSimpleServer(processor, transport, tfactory, pfactory)
```

```python
def main():
    handler = CalculatorHandler()
    processor = Calculator.Processor(handler)
    transport = TSocket.TServerSocket(port=9090)
    tfactory = TTransport.TBufferedTransportFactory()
    pfactory = TBinaryProtocol.TBinaryProtocolFactory()
    server = TServer.TSimpleServer(processor, transport, tfactory, pfactory)
```
Asynchronous RPC

- (b) The interaction using asynchronous RPC.

(a) Call remote procedure → Request → Server → Call local procedure and return results → Time

(b) Call remote procedure → Request → Client → Accept request → Server → Call local procedure → Time
Two asynchronous RPCs

• A client and server interacting through two asynchronous RPCs.
Distributed object systems

• Most large enterprise applications are written in an OO language and utilize an OO paradigm to manage complexity.
• Naturally, in place of RPCs, we have distributed objects and Remote Method Invocation.
• Provides transparency for OO applications
RMI

- RMI = RPC + Object-orientation
  - Java RMI
  - CORBA
    - Middleware that is language-independent
  - Microsoft DCOM/COM+
  - SOAP
    - RMI on top of HTTP
Object model

• Object references
  • Objects accessed via object references
  • Object references can be assigned to variables, passed as arguments and returned as results

• Interfaces
  • Provide a signature of a set of methods (types of arguments, return values and exceptions) without specifying their implementations

• Actions (invocations)

• Exceptions

• Garbage Collection
Distributed objects

- Remote object references
  - An identifier that can be used throughout a distributed system to refer to a particular remote object

- Remote interfaces
  - specify the methods of an object that are available for invocation by objects in other processes and define the types of the input and output arguments of each of them
  - CORBA provides an interface definition language (IDL) for specifying a remote interface
  - JAVA RMI: Java interface that extends Remote interface

- Actions: remote invocations

- Remote Exceptions may arise for reasons such as partial failure or message loss

- Distributed Garbage Collection: cooperation between local garbage collectors needed
Object distribution model

- Adoption of client-server model
  - Servers maintain objects and allow remote objects to access the methods of those objects that are specified for remote use in the interface definition
  - Clients invoke the methods offered by servers using remote method invocation (RMI)

- RMI implemented using a request/reply protocol
  - Request carries remote method reference and remote object’s in parameters to server
  - Reply returns out parameters (results of RMI) to client
Object distribution issues

- Remote Object
  - object capable of receiving RMIs (in example at least B and F)
- Remote Object Reference
  - needed by an object performing a RMI
  - refers to server of RMI
  - typical format
    - Internet address: 32 bits
    - port number: 32 bits
    - time: 32 bits
    - object number: 32 bits
    - interface of remote object
  - can be passed as parameter or result
Remote interface

- Specifies which methods can be invoked remotely and defines the types of the input and output arguments of each of them
CORBA IDL example

```idl
// 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();
};
```
Request-reply communication

- RMI is usually implemented via request/reply protocol

![Request-reply protocol diagram]
Marshalling

- Pack method arguments and results into a flat array of bytes
- Use a canonical representation of data types, e.g., integers, characters, doubles
- Examples
  - CORBA CDR
  - Java serialization
## CORBA CDR for constructed types

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>length (unsigned long) followed by characters in order (can also</td>
</tr>
<tr>
<td></td>
<td>can have wide characters)</td>
</tr>
<tr>
<td>array</td>
<td>array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>in the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>
The flattened form represents a Person struct with value: {'Smith', 'London', 1934}
## Indication of Java serialized form

The true serialized form contains additional type markers; h0 and h1 are handles.

<table>
<thead>
<tr>
<th>Person</th>
<th>8-byte version number</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>int year</td>
<td>class name, version number</td>
</tr>
<tr>
<td></td>
<td>java.lang.String</td>
<td>number, type and name of instance variables</td>
</tr>
<tr>
<td>1934</td>
<td>5 Smith</td>
<td>values of instance variables</td>
</tr>
<tr>
<td></td>
<td>6 London</td>
<td></td>
</tr>
</tbody>
</table>

### Serialized values

<table>
<thead>
<tr>
<th>Person</th>
<th>8-byte version number</th>
<th>h0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>int year</td>
<td>java.lang.String name:</td>
</tr>
<tr>
<td>1934</td>
<td>5 Smith</td>
<td>java.lang.String place:</td>
</tr>
<tr>
<td></td>
<td>6 London</td>
<td>h1</td>
</tr>
</tbody>
</table>
Parameter Passing: local vs. remote objects

- Remote object references are passed by reference while local object references are passed by value
Handling failures

• Types of failure
  • The client is unable to locate the server
  • The request message from the client to the server is lost
  • The reply message from the server to the client is lost
  • The server crashes after receiving a request
  • The client crashes after sending a request
## RMI invocation semantics

<table>
<thead>
<tr>
<th>Fault tolerance measures</th>
<th>Invocation semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retransmit request message</td>
<td>Duplicate filtering</td>
</tr>
<tr>
<td>No</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
RMI implementation

Client machine

Client

Proxy

Client invokes a method

Server machine

Server

Skeleton

Object

State

Method

Interface

Same interface as object

Skeleton invokes same method at object

Client OS

Server OS

Network

Marshalled invocation is passed across network
RMI implementation

• Communication module
  • carries out the request / reply protocol
  • uses message type, requestID and remote object reference
    • in server
      • select dispatcher for the class of the object to be invoked
      • obtains local reference from remote reference module by providing remote object identifier delivered with request message
      • passes on the local reference to dispatcher
RMI implementation

- Remote reference module
  - Translation between local and remote object references using remote object table
    - contains entries for all remote objects maintained by the server
    - and for all proxies (explained later)
  - Used when marshalling and unmarshalling remote object references
The RMI sublayer of the middleware architecture consists of:

- proxies (in client): local placeholder for remote objects
- dispatcher (in server): receives request and uses methodID to select appropriate method in skeleton
- skeleton (in server): implements methods in the remote interface
  - unmarshaling arguments
  - invokes corresponding method in remote object
  - waits for completion of invocation
  - marshals results, possibly exceptions, and returns them in a reply to invoking method of proxy
RMI programming

• RMI software
  • Generated by IDL compiler

• Proxy
  • Behaves like remote object to clients (invoker)
  • Marshals arguments, forwards message to remote object, unmarshals results, returns results to client

• Skeleton
  • Server side stub;
  • Unmarshals arguments, invokes method, marshals results and sends to sending proxy’s method

• Dispatcher
  • Receives the request message from communication module, passes on the message to the appropriate method in the skeleton

• Server and Client programs
RMI programming

• Binder
  • Client programs need a means of obtaining a remote object reference
  • Binder is a service that maintains a mapping from textual names to remote object references
  • Servers need to register their remote objects by name with the binder
  • Java RMIregistry, CORBA Naming service

• Server threads
  • Several choices: thread per object, thread per invocation
  • Remote method invocations must allow for concurrent execution

• Persistent object stores
  • An object that is guaranteed to live between activations of processes is called a persistent object
  • Stores the state of an object in a marshalled (serialized) form on disk

• Location service
  • Objects can migrate from one system to another during their lifetime
  • Maintains mapping between object references and the location of an object
Distributed garbage collection

• Java approach based on reference counting
  • Each server process maintains a list of remote processes that hold remote object references for its remote objects
  • When a client first acquires a remote reference to an object, it makes an addRef() invocation to server before creating a proxy
  • When a client’s local garbage collector notices that a proxy is no longer reachable, it makes a removeRef() invocation to the server before deleting the proxy
  • When the local garbage collector on the server notices that the list of client processes that have a remote reference to an object is empty, it will delete the object (unless there are any local objects that have a reference to the object)

• Other approaches
  • “Evictor” pattern
  • Leases
Outline

• Fundamentals and Types of Communication
• RPC/RMI
• Message-oriented Transient Communication
• Message-oriented Persistent Communication
• Data Streaming
• Multicast
Communication Middleware

• Transient
  • TCP/IP sockets API
  • MPI, PVM
  • RPC, RMI

• Persistent
  • Message-oriented Middleware (e.g., Java Message Service)
Message-oriented transient communication (1)

- Socket primitives for TCP/IP.

<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>
Connection-oriented communication pattern using sockets

Server

`socket` → `bind` → `listen` → `accept` → `read` → `write` → `close`

Client

`socket` → `connect` → `write` → `read` → `close`

Synchronization point

Communication
Message-oriented transient communication (2)

- The Message-Passing Interface (MPI)
- Used for developing message-passing parallel applications

<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_issend</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>
Outline

• Fundamentals and Types of Communication
• RPC/RMI
• Message-oriented Transient Communication
• Message-oriented Persistent Communication
• Data Streaming
• Multicast
Message-queuing model (1)

- Four combinations for loosely-coupled communications using queues.

(a) Sender running

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

Receiver running

(b) Sender running

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

Receiver passive

(c) Sender passive

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

Receiver running

(d) Sender passive

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

Receiver passive
Message-queuing model (2)

• Basic interface to a queue in a message-queuing system.

<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>
General architecture of a message-queuing system (1)

- The relationship between queue-level addressing and network-level addressing.
General architecture of a message-queuing system (2)

- The general organization of a message-queuing system with routers.
Message transfer
Message brokers

• The general organization of a message broker in a message-queuing system
Example: IBM WebSphere MQ

- General organization of IBM's WebSphere MQ message-queuing system (formerly MQSeries).

Diagram:
- Sending client
  - Program
  - MQ Interface
  - Stub
  - RPC (synchronous)

- Routing table
  - Queue manager
  - Server stub
  - MCA

- Send queue
  - Client's receive queue

- Receiving client
  - Program
  - Stub
  - Enterprise network
  - To other remote queue managers

Message passing (asynchronous)
Local network
Message-queuing systems vs. email systems

• The underlying architecture of message-queuing systems is very similar to that for email services.
• The difference is that email systems primarily provide support for end users.
• General message-queuing systems enable persistent communication between processes regardless of what the process is doing.
  • leads to different requirements, e.g., guaranteed message delivery, message priorities, logging facilities, load balancing, fault tolerance, etc.
Outline

• Fundamentals and Types of Communication
• RPC/RMI
• Message-oriented Transient Communication
• Message-oriented Persistent Communication
• Data Streaming
• Multicast
Data streaming

• A general architecture for streaming stored multimedia data over a network.
Streams and Quality of Service

Properties for Quality of Service:
- The required bit rate at which data should be transported.
- The maximum delay until a session has been set up.
- The maximum end-to-end delay.
- The maximum delay variance, or jitter.
- The maximum round-trip delay.
Enforcing QoS

• Using a buffer to reduce jitter.
Enforcing QoS

- How to reduce the effects of packet loss (when multiple samples are in a single packet?)

(a)

(b)
Stream synchronization

• Given a complex stream, how do you keep the different sub-streams in synchronization?
• Example: think of playing out two channels, that together form stereo sound. Difference should be less than 20–30 µsec
Synchronization mechanisms

- The principle of explicit synchronization on the level of data units of simple streams.

Diagram:
- Incoming stream
- Application
- Receiver's machine
- Procedure that reads two audio data units for each video data unit
- OS
- Network
Synchronization mechanisms

• Alternative:
  • Multiplex all sub-streams into a single stream, and demultiplex at the receiver. Synchronization is handled at multiplexing/demultiplexing point (MPEG).
Outline

• Fundamentals and Types of Communication
• RPC/RMI
• Message-oriented Transient Communication
• Message-oriented Persistent Communication
• Data Streaming
• Multicast
Multicast

• Network routing protocols provide support for multicast
  • Heavily studied topic but “native” multicast not widely used by network applications
  • Instead, “Application-level” multicasting has had more success

• Application-level multicasting
  • Nodes organize themselves into an overlay network
  • P2P applications
Overlay construction

- The relation between links in an overlay and actual network-level routes.
Reading

- Chapter 4, Sections 8.3.2, 10.1, 10.3 of Tbook
- Chapter 5, Cbook
- Textbook on computer networks