Communication

Topics to be covered

PART 1 (last week)
Layered Protocols
Remote Procedure Call (RPC)
Remote Method Invocation (RMI)

PART 2
Overview of last week and more details
Message-Oriented Middleware (MOM)
Streams

Overview

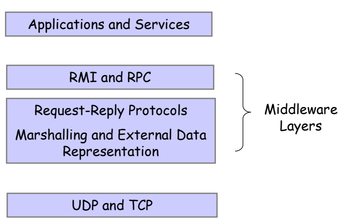
Internet protocols provide two alternatives building blocks: UDP (simple message-passing with omission failures) and TCP (guarantees message delivery but higher overhead)

Java and Unix Interface (sockets) API to UDP and TCP

Remote Method Invocation (RMI): allows an object to invoke a method in an object in a remote process

Examples are CORBA and Java RMI

Remote Procedure Calls (RPC): allows a client to invoke a procedure in a remote server



More on Part 1

Outline

OSI and Internet Protocol Stack

API to Internet Protocols

RPC and RMI (overview)

Java RMI

Layered Protocols

Processes define and adhere to rules (protocols) to communicate

Protocols are structured into layers – each layer deals with a specific aspect of communication

Each layer uses the services of the layer below it – an interface specifies the services provided by the lower layer to the upper layers

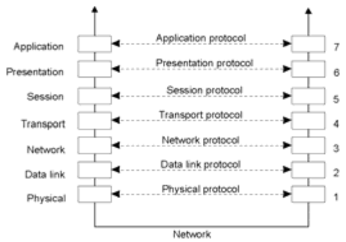
Two general type of protocols:

- Connection-oriented: before exchanging data, the sender and the receiver must *establish a connection* (e.g., telephone), possibly negotiate the protocols to be used, release the connection when done
- Connectionless: no setup in advance (e.g., sending an email)

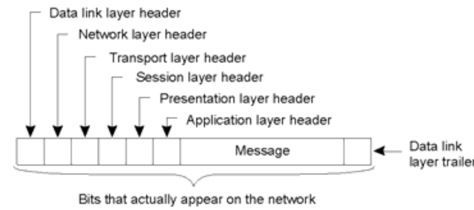
The OSI Model

The ISO OSI or the OSI model
Designed to allow *open* systems to communicate

- Each layer provides an interface to the one above
- Message send (downwards) Message received (upwards) *example*
- Each layer adds a header



The OSI Model



- The information in the layer n header is used for the layer n protocol
- Independence among layers
- Reference model (not an actual implementation)

OSI protocols not so popular, instead Internet protocols (e.g., TCP and IP)

Low-level Layers

Physical Layer: specification and implementation of bits, transmission between sender and receiver

Data Link Layer: groups bits into frames, error and flow control

Network Layer: routes packets

For many distributed systems, the lowest level interface is that of the network layer.

Transport Layer

Standard (transport-layer) Internet protocols:

- Transmission Control Protocol (TCP): connection-oriented, reliable, stream-oriented communication (TCP/IP)
  acknowledgement

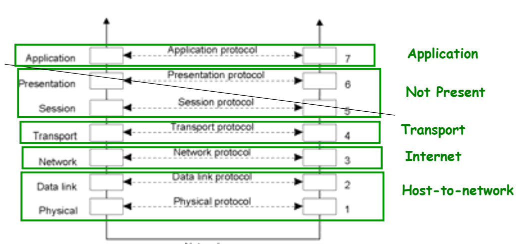
- Universal Datagram Protocol (UDP): connectionless, unreliable (best-effort) datagram communication (just IP with minor additions)
  omission failures, no ordering, non-blocking send, blocking receives

TCP vs UDP

Works reliably over any network
Considerable overhead

use UDP + additional error and flow control for a specific application

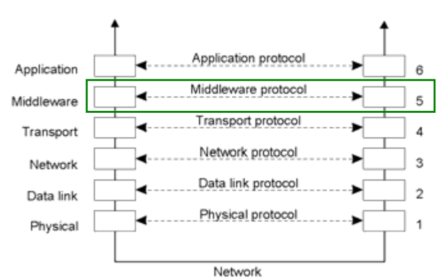
OSI vs TCP/IP Model



OSI reference model

TCP/IP not an official reference model (many details regarding the interfaces left open to implementation) – *but the de facto Internet communication protocol*

Middleware Protocols



An adapted reference model for networked communication.

The API for Internet Protocols

Message passing supported by two message communication operations

send and receive

One process (*sending process*) sends a message (a sequence of bytes) to a destination and another process (*receiving process*) at the destination receives the message

Synchronization of the two processes

A queue (buffer) is associated with each message destination

Synchronous and Asynchronous Communication

Synchronous

Both send and receive are *blocking* operations

Whenever a send is issued, the sending process is blocked until the corresponding receive is issued

Whenever a receive is issued, the process is blocked, until a message arrives

Asynchronous

Send is non-blocking

Receive either blocking or non-blocking

Non blocking-variant of receive: the receiving process proceeds after issuing a receive which provides a *buffer* to be filled in the background – must *receive notifications* when its buffer has been filled by polling or interrupt

not generally provided (why?)

Message Destinations

In the Internet protocols, messages are sent to:

(Internet Address, Local Port)

pairs.

A local port is a message destination within a computer specified by an integer (large number (2^{16}) possible port numbers)

- A port has exactly one receiver but may have many senders
- Processes may use multiple ports form which to receive messages
- Any process that knows the number of a port can send a message
- Servers publish their ports for use by clients

- Why not sending messages directly to processes?

Ports allows for several point of entries to a receiving process

Message Destinations

Location transparency

If we use a fixed Internet address, services must run on the same computer.

- Client programs refer to a service by name and use a name server or binder to translate their names into several locations at run time (no migration transparency however)

- The operating system (Mach does this) provides location-independent identifiers for message destinations

Reliability

(validity) a point-to-point message service is reliable if messages are guaranteed to be delivered despite a "reasonable" number of packets being dropped or lost – in contrast a point-to-point message service is unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost

(integrity) messages must arrive uncorrupted and without duplication

Ordering

The API for Internet Protocols

Both forms of communication (UDP and TCP) use the socket abstraction which provides an *endpoint* for communication between processes

Communication by transmitting a message between a socket in one process and a socket in another

More on Unix sockets (later)

Outline

OSI and Internet Protocol Stack

API to Internet Protocols

→ RPC and RMI (overview)

Java RMI

RMI and RPC

RMI and RPC are **programming models** for distributed applications

- The client that calls a procedure cannot tell whether it runs in the same or in a different process; nor does it need to know the location of the server
- The object making the invocation cannot tell whether the object it invokes is local or not; nor does it need to know its location

The protocols are independent of the underlying transport protocols

Use marshalling and unmarshalling to hide differences due to hardware architectures

Independent of the operating system

External Data Representation and Marshalling

- Data structures must be flattened
- Same representation for primitive values
- Use the same code to represent characters (e.g., ASCII or Unicode)

Either:

- The values are converted to an agreed external format
- The values are transmitted in the sender's format together with an indication of the format used

External data representation: an agreed standard for the representation of data structures and primitive values

Marshalling: the process of taking a collection of data items and assembling them into a form suitable for transmission in a message

Unmarshalling is the process of disassembling them on arrival to produce an equivalent collection of data items as the destination

Remote Procedure Call (RPC)

Basic idea:

Allow programs to call procedures located on other machines

Some issues:

Calling and called procedures in different address spaces
Parameter passing
Crash of each machine

Client and Server Stubs

RPC supports location transparency (the calling procedure does not know that the called procedure is remote)

Client stub:

- local version of the called procedure
- called using the "stack" sequence
- it packs the parameters into a message and requests this message to be sent to the server (calls send)
- it calls receive and blocks till the reply comes back

When the message arrives, the server OS passes it to the server stub

Server Stub:

- typically waits on receive
- it transforms the request into a *local* procedure call
- after the call is completed, it packs the results, calls send
- it calls receive again and blocks

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

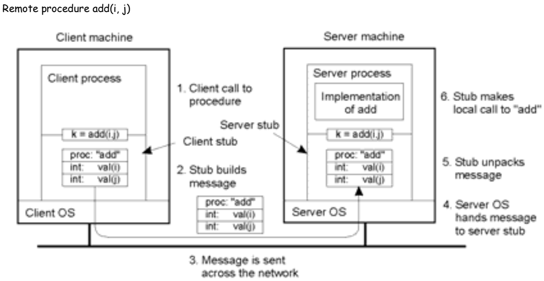
4. Remote OS gives message to server stub

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

Parameter Passing

Remote procedure add(i, j)



A server stub may handle more than one remote procedure

Two issues with parameter passing:

- Marshalling
- Reference Parameters

Distributed Objects

Expand the idea of RPCs to invocations on remote objects

- Data (state) and operations on those data encapsulated into an **object**
- Objects can be accessed by object **references**
- Operations are implemented as **methods**
- **Interfaces** provide a definition of the signatures of a set of methods (that is, the types of their arguments, return values and exceptions)
  - An object provides an interface if its class contains code that implements the methods of that interface
  - An object offers only its interface to clients.
    An object may implement many interfaces;
    Given an interface definition, there may be several objects that offer an implementation for it

Distributed Objects

To invoke the methods of a remote object must have acess to its remote object reference

Every remote object has a remote interface that specifies which of its methods can be invoked remotely

Remote Object References

A remote object reference is an identifier for a remote object that is valid throughout a distributed system

- It is passed in the invocation to specify which object is to be invoked
- May also be passed as arguments or returned as results of RMIs
- Each remote object has a single remote object reference

Must be **unique**

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

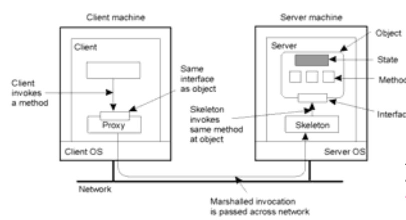
But the above is not location transparent

Distributed Objects

A client binds to a distributed object: an implementation of the object's *interface*, called a proxy, is loaded into the client's address space

Proxy (analog to a client stub): provides an implementation of the interface which: Marshals method invocations into messages, sends it to the target, waits & un-marshals reply messages

Actual object at a server machine: offers the same interface



Skeleton (analog to server stub)

Un-marshals requests to proper method invocations at the object's interface at the server

Note: the object itself is not distributed, aka remote object

May have both a dispatcher and a skeleton, the dispatcher selects the appropriate method in the skeleton

Basic RMI

Assume client stub and server skeleton are in place

- Client invokes method at stub
- Stub marshals request and send it to server
- Server ensures referenced object is active
  - Create separate process to hold object
  - Load the object into server process
- Request is unmarshalled by object's skeleton, and referenced object is invoked
- *If request contained an object reference, invocation is applied recursively*
- Result is marshalled and passed back to client
- Client stub unmarshals reply and passes result to client application

External Data Representation and Marshalling

- CORBA's common data representation (CDR)
  - an external representation for structured and primitive types that can be passed as arguments and results of remote method invocations in CORBA)
  - can be used by a variety of programming languages
- Java's object serialization
  - flattening and external data representation of any single object or tree of objects to be transmitted in a message or stored on a disk
  - can be used only by Java
- No involvement of the application programmer is needed

Java RMI

```
Public class Person implement Serializable {
        Private String name;

// followed by methods for accessing the instance
variables

}
```

Serialization: flattening an object or a connected set of objects into a serial form for storing on disk or transmitting

Deserialization

Java objects can contain **references to other objects**

- When an object is serialized, all the objects that it references are serialized together with it
- References are serialized as handles (in this case, a reference in the object within the serialized form)

To serialize: its class information (version numbers are used), types and names of instance variables (and their classes recursively); the content of instance variables

Java RMI

Server program

Classes for the dispatcher and skeleton + implementation of all the remote objects that it supports (aka servant classes)

An initialization section (e.g., main method) that creates and initializes at least one remote object

Register some of its remote objects with a binder

Binder: maps textual names to remote object references (in Java, RMIregistry)

Client program contains the classes of the proxies, can use a binder to look up remote object references

Java RMI

Remote interfaces are defined by extending an interface called Remote in the java.rmi package

Methods must throw RemoteException

Must implement the serializable interface

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;           1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;  2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```


Java RMI

- The parameters of a method are assumed to be input parameters
- The result of a method is a single output parameter

- When the type of a parameter or result values is defined as *remote interface*, the corresponding argument or result is passed as a remote object reference
- All *serializable non-remote objects* are copied and passed by value

- If the recipient does not possess the class of an object passed by value, its code is downloaded automatically
- Similarly, if the recipient of a remote object reference does not possess the class of a proxy, its code is downloaded automatically

Java RMI: RMIregistry

RMIregistry is the binder for Java RMI

An instance must run on every server computer that hosts remote objects

Table mapping textual, URL-style names to references to remote objects

//computerName:port/objectName

void rebind (String name, Remote obj)
: This method is used by a server to register the identifier of a remote object by name

void bind (String name, Remote obj)
: This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)
: This method removes a binding.

Remote lookup(String name)
: This method is used by clients to look up a remote object by name. A remote object reference is returned.

String [] list()
: This method returns an array of Strings containing the names bound in the registry.

Java RMI: Server

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();          1
                Naming.rebind("Shape List", aShapeList );           2
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Java RMI: Servant Classes

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
        private Vector theList;                          // contains the list of Shapes          1
        private int version;
       public ShapeListServant()throws RemoteException{...}
       public Shape newShape(GraphicalObject g) throws RemoteException {        2
              version++;
              Shape s = new ShapeServant( g, version);                                          3
                     theList.addElement(s);
                     return s;
       }
       public  Vector allShapes()throws RemoteException{...}
       public int getVersion() throws RemoteException { ... }
}
```

UnicastRemoteObject provides remote objects that live only as long as the process in which they are created

Java RMI: Client

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
               aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList")             ;              1
               Vector sList = aShapeList.allShapes();                                                  2
        } catch(RemoteException e) {System.out.println(e.getMessage());
        }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

Note: must know the host address, not a system-wide registry (but per host)

RMI

- Each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely
- Local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once
- Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers.

Invocation semantics

- Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed
- Duplicate filtering: when retransmission is used, whether to filter out duplicate messages
- Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operation at the server

Invocation semantics

- Local method invocation exactly-once (each method is executed exactly once)
- May-be: the invokes cannot tell whether a remote method has been executed once or not at all
- At least-once: the invoker receives either a result (in which case it knows that he method was executed at least once) or an exception informing it that no result was received
- At most-once: the invoker receives either a result (in which case it knows that he method was executed exactly once) of an exception informing it that no result was received, in which case the method will have been executed either once or not at all

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

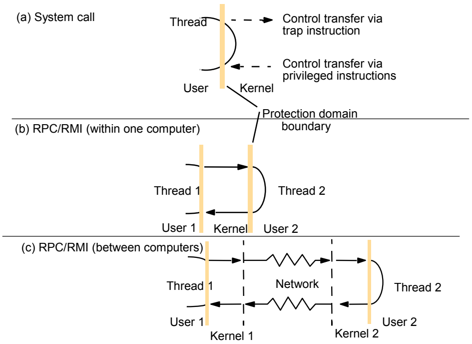
RMI and RPC

Few operating system kernels provide relative high-level communication primitives

(+) efficiency (saving in system call overhead)

(-) sockets provide portability and interoperability

Invocations between address spaces



RMI and RPC (performance)

The performance of RPC and RMI mechanisms is critical for effective distributed systems.

Typical times for "**null procedure call**":

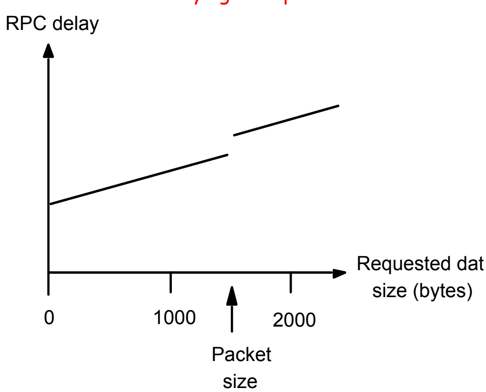
Local procedure call	< 1 microseconds
Remote procedure call	~ 10 milliseconds

'network time' (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only .01 millisecond; the remaining delays must be in OS and middleware - *latency*, not communication time.

Factors affecting RPC/RMI performance

- marshalling/unmarshalling + operation dispatch at the server
- data copying:- application -> kernel space -> communication buffers
- thread scheduling and context switching:- including kernel entry
- protocol processing:- for each protocol layer
- network access delays:- connection setup, network latency

RPC delay against parameter size



RMI and RPC

Most invocation middleware (Corba, Java RMI, HTTP) is implemented over TCP

- For universal availability, unlimited message size and reliable transfer
- Sun RPC (used in NFS) is implemented over both UDP and TCP and generally works faster over UDP

Research-based systems have implemented much more efficient invocation protocols, E.g.

- Firefly RPC (see www.cdk3.net/oss)
- Amoeba's doOperation, getRequest, sendReply primitives in the operating system (www.cdk3.net/oss)
- LRPC [Bershad et. al. 1990] (Uses shared memory for interprocess communication, while maintaining protection of the two processes; arguments copied only once (versus four times for conventional RPC); Client threads can execute server code via protected entry points only (uses capabilities); Up to 3 x faster for local invocations

Concurrent and asynchronous invocations

- middleware or application doesn't block waiting for reply to each invocation

Message-Oriented Communication

Persistence and Synchronicity
Message-Oriented Transient (sockets, RMI)
Message-Oriented Persistent/Message Queuing

Communication Alternatives

RPC and RMI hide communication and thus achieve access transparency

Client/Server computing is generally based on a model of **synchronous communication**:

- Client and server have to be ***active*** at the time of communication
- Client issues request and *blocks* until it receives reply
- Server essentially *waits* only for incoming requests, and subsequently processes them

Drawbacks synchronous communication:

- Client cannot do any other work while waiting for reply
- Failures have to be dealt with immediately (the client is waiting)
- In many cases, the model is simply not appropriate (mail, news)

Asynchronous Communication Middleware

Message-oriented middleware: Aims at high-level **asynchronous** communication:

Processes send each other messages, which are queued

Asynchronous communication: Sender need *not wait* for immediate reply, but can do other things

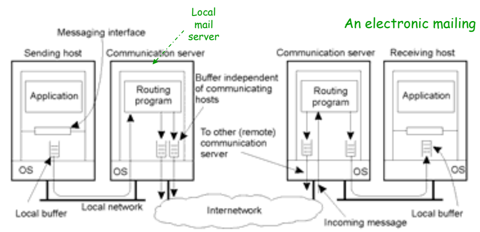
Synchronous communication: Sender blocks until the message arrives at the receiving host or is actually delivered and processed by the receiver

Middleware often ensures *fault tolerance*

Example Communication System

- Applications execute on *hosts*
- *Communication servers* are responsible for passing (and routing) messages between hosts
- Each host offers an interface to the communication system through which messages can be submitted for transmission
- *Buffers* at the hosts and at the communication servers

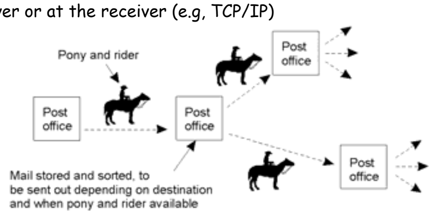
An electronic mailing system



Persistent vs Transient Communication

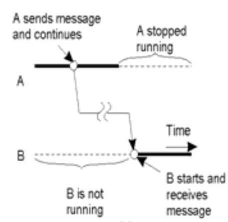
Persistent communication: A message is stored at a communication server as long as it takes to deliver it at the receiver (e.g., email)

Transient communication: A message is discarded by a communication server as soon as it cannot be delivered at the next server or at the receiver (e.g, TCP/IP)

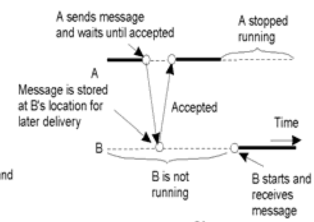


Typically, all transport-level communication services offer only transient, a communication server corresponds to a store-and-forward router

Messaging Combinations



(a)



(b)

Persistent asynchronous

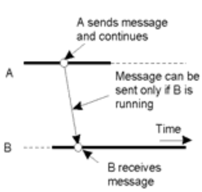
Message stored persistently at the sending host or at the first communication server

e.g., electronic mail systems

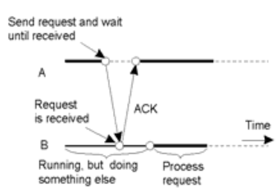
Persistent synchronous

Message stored persistently at the receiving host or the connected communication server (weaker)

Messaging Combinations



(c)



(d)

Transient asynchronous

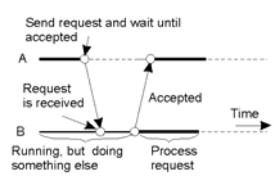
Transport-level datagram services (such as UDP)

One-way RPC

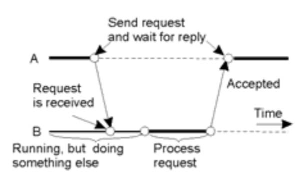
Receipt-based transient synchronous

Sender blocks until the message is stored in a local buffer at the receiving host

Messaging Combinations



(e)



(f)

Delivery-based transient synchronous

Sender blocks until the message is delivered to the receiver for further processing

Asynchronous RPC

Response-based transient synchronous

Strongest form

Sender blocks until it receives a reply message

RPC and RMI

Communication Alternatives

Need for *persistent communication services* in particular when there is large geographical distribution

(cannot assume that all processes are simultaneously executing)

Outline

Message-Oriented Transient Communication

Transport-level sockets

Message-Passing Interface (MPI)

Message-Oriented Persistent Communication

Message Queuing Model

General Architecture

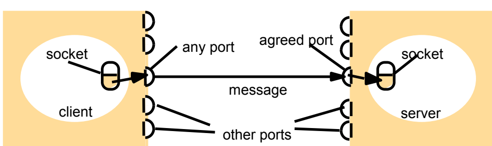
Example (IBM MQSeries: check the textbook)

Berkeley Sockets

Socket: a communication *endpoint* to which an application can write data to be sent out over the network and from which incoming data may be read

a message is transmitted between a socket in one process and a socket in another

each socket is associated with a particular protocol, either UDP or TCP



Internet address = 138.37.94.248 Internet address = 138.37.88.249

Sockets

UDP (connectionless)

A message passing abstraction that enables a sending process to send a **single** message to a receiving process

The independent packets containing the message are called *datagrams*

The sender specifies the destination using a socket

TCP (connection-oriented)

A two-way stream abstraction that enables the communication of a **stream** of data items with no message boundaries

Data are queued on arrival

Sender blocks when no data are available

Berkeley Sockets

A process to **receive** a message, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs

server

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

Socket primitives for TCP/IP.

Sockets

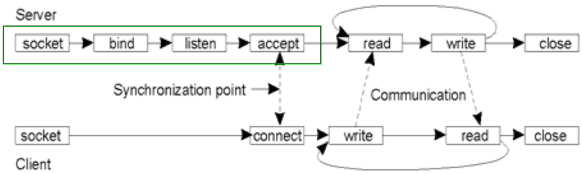
To send or receive messages first

Create a socket

Bind it to an (Internet address, local port)

Receive returns the address of the sender

Berkeley Sockets



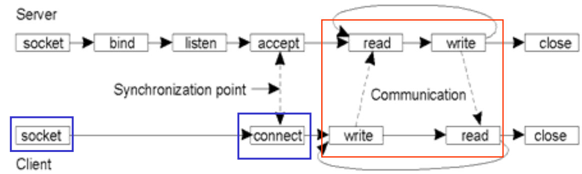
socket: *creates* a new communication endpoint for a specific transport protocol (the local OS reserves resources to accommodate sending and receiving messages for the specified protocol)

bind: *associates a local address* (e.g., the IP address of the machine + a port number) with the newly created socket

listen: (only in the case of connection-oriented communication) non-blocking call; allows the OS to reserve enough buffers for a specified max number of connections

accept: blocks the server until a connection request arrives. When a request arrives, the OS creates a new socket and returns it to the caller. Then , the server can fork off a process that will subsequently handle the actual communication through the new connection.

Berkeley Sockets



socket: (client)

connect: attempt to establish a connection; specifies the transport-level address to which a connection request is to be sent

write/**read**: send/receive data

close: called by both the client and the server

Sockets used for datagrams

Sending a message

Socket descriptor
Communication domain
type

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

To get a reply

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

Must be made known to the sender

ServerAddress and ClientAddress are socket addresses

Sockets used for streams

- A pair of sockets must be connected
- The receiver listens for a request for a connection, the sender asks for a connection
- Any available data is read immediately in the same order as it was written, no indication of messages boundaries

```
s = socket(AF_INET, SOCK_STREAM,0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Implicitly binds

Requesting a connection

Similar to reading/writing for files

```
s = socket(AF_INET, SOCK_STREAM,0)
•
bind(s, ServerAddress);
listen(s, 5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

Max number of connection requests

Listening and accepting a connection

ServerAddress and ClientAddress are socket addresses

Sockets

Processes may be use the same socket for both sending and receiving messages

A process may use multiple ports to receive messages, but it cannot share ports with any process in the same computer (except from processes using IP multicast)

Any number of processes may send messages to the same port

The Message-Passing Interface (MPI)

- Suitable for COWs and MPPs
- MPI designed for parallel applications and thus tailored to transient communication
- There are no communication servers
- Assumes communication within a known group of processes, a (group_ID, process_ID) uniquely identifies a source or destination of a message
- Provides a higher-level of abstraction than sockets

The Message-Passing Interface (MPI)

Some of the message-passing primitives of MPI

Primitive	Meaning
MPI_bsend	(transient-asynchronous) Append outgoing message to a local send buffer
MPI_send	(blocking send) Send a message and wait until copied to local or remote buffer
MPI_ssend	(delivery-based transient synchronous) Send a message and wait until receipt starts
MPI_sendrecv	(response-based transient synchronous, RPC) Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue (for local MPI)
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there are none
MPI_irecv	Check if there is an incoming message, but do not block

Supports diverse forms of buffering and synchronization (over 100 functions)

Outline

Message-Oriented Transient Communication

Transport-level sockets

Message-Passing Interface (MPI)

→ Message-Oriented Persistent Communication

Message Queuing Model

General Architecture

Example (IBM MQSeries: check the textbook)

Message-Oriented Middleware

Message-queuing systems or Message-Oriented Middleware (MOM)

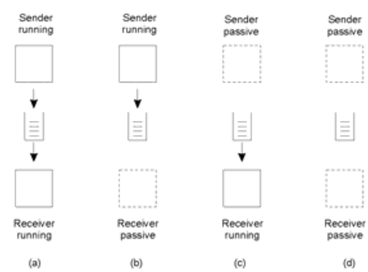
- Targeted to message transfers that take minutes instead of seconds or milliseconds

- In short: asynchronous persistent communication through support of *middleware-level* queues
Queues correspond to buffers at communication servers.

- Not aimed at supporting only end-users (as e.g., e-mail does). Enable persistent communication between ***any*** processes

Message-Queuing Model

Four combinations for loosely-coupled communications using queues.



- Message can contain any data
- Addressing by providing a system-wide unique name of the destination queue

Message-Queuing Model

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

Primitive	Meaning
Put	Call by the sender Append a message to a specified queue Non-blocking
Get	Block until the specified queue is nonempty, and remove the first message Variations allow searching for a specific message in the queue
Poll	Check a specified queue for messages, and remove the first. Never block.
Notify	Install a handler (as a callback function) to be automatically invoked when a message is put into the specified queue. Often implemented as a daemon on the receiver's side

General Architecture of a Message-Queuing System

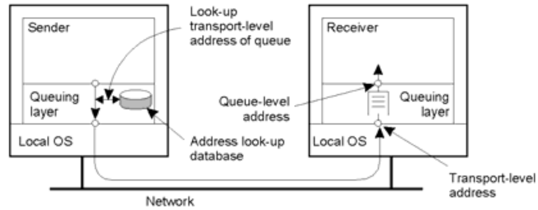
- Messages are put only into local to the sender queues - *source queues*
- Messages can be read only from local queues
- A message put into a queue contains the specification of a destination queue
- Message-queuing system: provides queues to senders and receivers; transfers messages from their source to their destination queues.

Queues are distributed across the network ⇒ need to map queues to network addresses

- A (possibly distributed) database of queue names to network locations

Queues are managed by queue managers

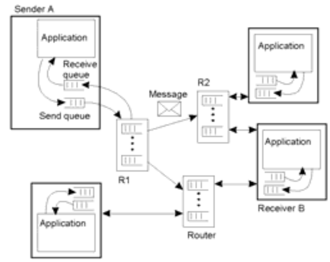
General Architecture of a Message-Queuing System



The relationship between queue-level addressing and network-level addressing

General Architecture of a Message-Queuing System

Relays: special queue managers that operate as routers and forward incoming messages to other queue managers
⇒ overlay network



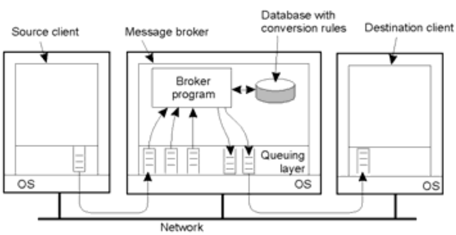
Why routers?

- Only the routers need to be updated when queues are added or removed
- Allow for secondary processing of messages (e.g., logging for fault tolerance)
- Used for multicasting purposes
- Act as message brokers

Message Brokers

Message broker: acts as an application-level gateway, coverts incoming messages to a format that can be understood by the destination application

Contains a database of conversion rules



IBM MQSeries

Check textbook

Simple Mail transfer protocol (SMTP)

SMTP the standard mail protocol used by e-mail servers to route e-mails

SMTP relies on TCP/IP and DNS for transport and destination server discovery

A client can access the closest mail server and receive mail using simple mail access protocols such as POP and IMAP

Network News Transport Protocol (NNTP)

NNTP servers propagate news articles using "flood fill"

Each server has one or more peer and each article from a peer server or a user is sent to all servers that haven't yet seen the article

Stream-Oriented Communication

Streams
Quality of Service
Synchronization

Support for Continuous Media

So far focus on transmitting discrete, that is time independent data

Discrete (representation) media: the *temporal relationships* between data items **not** fundamental to correctly interpreting what the data means

Example: text, still images, executable files

Continuous (representation) media: the *temporal relationships* between data items fundamental to correctly interpreting what the data means

Examples: audio, video, animation, sensor data

Example: motion represented by a series of images, in which successive images must be displayed at a uniform spacing T in time (30-40 msec per image)

Correct reproduction ⇒ *showing the stills in the correct order and at a constant frequency of 1/T images per sec*

A data stream is a sequence of data units

Transmission Modes

Different timing guarantees with respect to data transfer:

- Asynchronous transmission mode: data items are transmitted one after the other but no further timing constraints

  Discrete data streams, e.g., a file

  (order)

- Synchronous transmission mode: there is a maximum end-to-end delay for each unit in a data stream

  E.g., sensor data

  (order & max delay)

- Isochronous transmission mode: there is both a maximum and minimum end-to-end delay for each unit in a data stream (called bounded (delay) jitter)

  (order & max delay & min delay)

  E.g., multimedia systems (audio, video)

Continuous Data Stream: a connection oriented communication facility that supports isochronous data transmission

Stream Types

- Simple stream: only a single sequence of data

- Complex stream: several related simple streams (substreams)

  - Relation between the substreams is often also time dependent

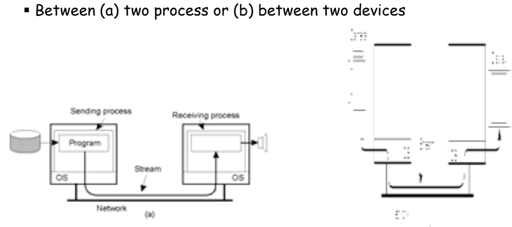
  - Example: stereo video transmitted using two substreams each for a single audio channel

    Data units from each substream to be communicated pairwise for the effect of stereo

  - Example: transmitting a movie: one stream for the video, two streams for the sound in stereo, one stream for subtitles

Data Streams

- Streams are **unidirectional**
- Considered as a virtual connection between a **source** and a **sink**
- Between (a) two process or (b) between two devices



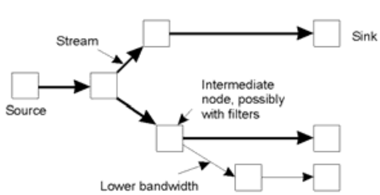
Data Streams

Multiparty communication: more than one source or sinks

Multiple sinks: the data streams is multicasted to several receivers

Problem when the receivers have different requirements with respect to the quality of the stream

Filters to adjust the quality of the incoming stream differently for outgoing streams



Quality of Service

Quality of Service (Qos) for continuous data streams: timeliness, volume and reliability

Difference between **specification** and **implementation** of QoS

Flow Specification of QoS

One way to specify QoS is to use flow specifications: specify both the service required and characteristics of the input

Input parameters can be guaranteed by traffic shaping

Who specifies the flow? (provide a set of predefined categories)

Characteristics of the Input	Service Required
• Maximum data unit size (bytes) • Token bucket rate (bytes/sec) • Token bucket size (bytes) • Maximum transmission rate (bytes/sec)	• Loss sensitivity (bytes) • Loss interval (μsec) Maximum acceptable loss rate • Burst loss sensitivity (data units) How many consecutive data units may be lost • Minimum delay noticed (μsec) How long can the network delay delivery of a data unit before the receiver notices the delay • Maximum delay variation (μsec) Maximum tolerated jitter • Quality of guarantee Indicates how firm are the guarantees

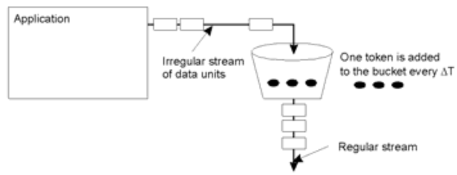
Flow Specification of QoS

token-bucket model to express QoS

Token: fixed number of bytes (say k) that an application is allowed to pass to the network

Basic idea: tokens are generated at a fixed rate

- Tokens are buffered in a **bucket** of limited capacity
- When the bucket is full, tokens are dropped
- To pass N bytes, drop N/k tokens



Implementing QoS

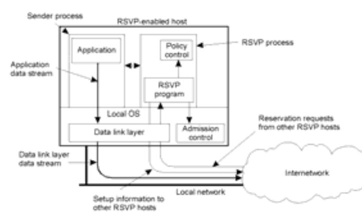
QoS specifications translate to resource reservations in the underlying communication system

Resources: bandwidth, buffers, processing capacity

There is no standard way of (1) QoS specs, (2) describing resources, (3) mapping specs to reservations.

Implementing QoS

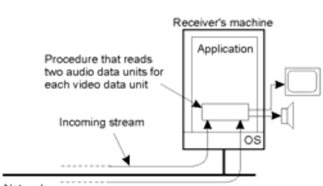
Resource reSerVation Protocol (RSVP) a transport-level control protocol for resource reservation in network routers



Stream Synchronization

Given a complex stream, how do you keep the different substreams in synch?
Two forms: (a) synchronization between a discrete and a continuous data stream and (b) synchronization between two continuous data streams

The principle of explicit synchronization on the level of data units.



A process that simply executes read and write operations on several simple streams ensuring that those operations adhere to specific timing and synchronization constraints

Stream Synchronization

The principle of synchronization as supported by high-level interfaces.