Communication

Introduction

Interprocess communication is at the heart of all distributed systems

Based on low-level message passing offered by the underlying network

Protocols: rules for communicating processes structured in layers

Four widely-used models:
- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)
- Message-Oriented Middleware (MOM)
- streams

Topics to be covered

Layered Protocols
Remote Procedure Call (RPC)
Remote Method Invocation (RMI)
Message-Oriented Middleware (MOM)
Streams

Layered Protocols

Low-Level
Transport
Application
Middleware

Layered Protocols

A wants to communicate with *B*

A builds a message in its own address space

A executes a call to the OS to send the message

Need to agree on the meaning of the bits being sent

The ISO OSI or the OSI model

The OSI Model

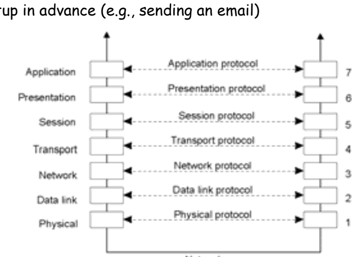
Designed to allow open systems to communicate

Two general type of protocols:

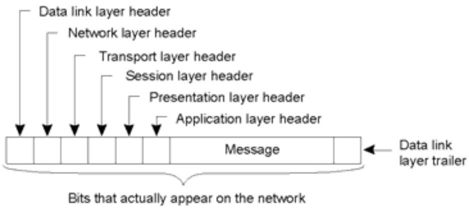
Connection-oriented: before exchanging data, the sender and the receiver must establish a connection (e.g., telephone)

Connectionless: no setup in advance (e.g., sending an email)

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



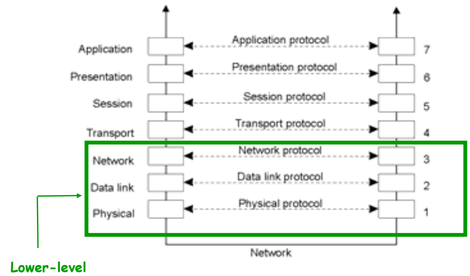
The OSI Model



- The information in the layer n header is used for the layer n protocol
- Independence among layers
- Protocol suite or protocol stack: collection of protocols used in a particular system
- OSI protocols not so popular, instead Internet protocols (e.g., TCP and IP)
- reference model

Low-level Layers

These layers implement the basic functions of a computer network



Low-level Layers

Physical layer:
Concerns with transmitting 0 and 1s
Standardizing the electrical, mechanical and signaling interfaces so that when A sends a 0 bit, it is receives as a 0
Example standard: RS-232-C for serial communication lines

the specification and implementation of bits, and their transmission between sender and receiver

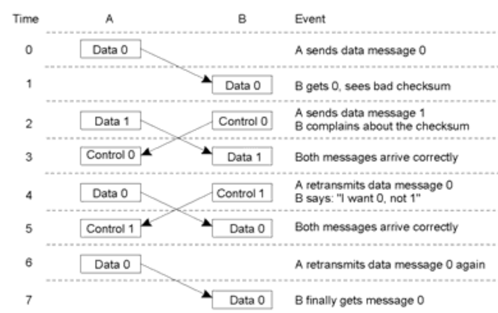
Data link layer:
Group bits into frames and sees that each frame is correctly received

Puts a special bit pattern at the start and end of each frame (to mark them) as well as a checksum

Frames are assigned sequence numbers

prescribes the transmission of a series of bits into a frame to allow for error and flow control

Low-level Layers: The Data Link Layer



Discussion between a receiver and a sender in the data link layer.
A tries to sends two messages, 0 and 1, to B

Low-level Layers

Network layer:

Routing: choose the best ("delay-wise") path

Example protocol at this layer: connectionless IP (part of the Internet protocol suite)

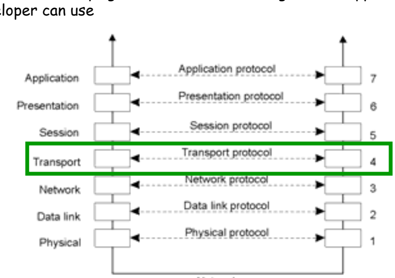
IP packets: each one is routed to its destination independent of all others. No internal path is selected or remembered

describes how packets in a network of computers are to be routed.

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

Transport Protocols

Turns the underlying network into something that an application developer can use



Transport Layer

Reliable connection

- The transport layer provides the actual communication facilities for most distributed systems.
- Breaks a message received by the application layer into packets and assigns each one of them a sequence number and send them all
- Header: which packets have been sent, received, there is room for, need to be retransmitted
- Reliable connection-oriented *transport* connections built on top of connection-oriented (all packets arrive in the correct sequence, if they arrive at all) or connectionless *network* services

Transport Layer

Standard (transport-layer) Internet protocols:

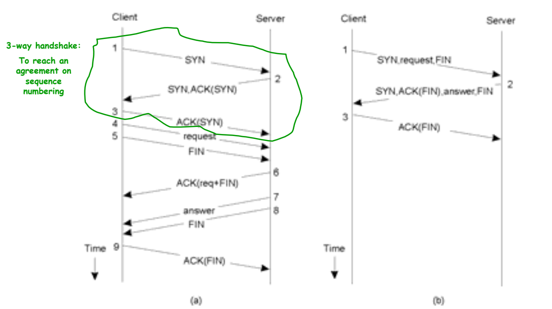
- Transmission Control Protocol (TCP): connection-oriented, reliable, stream-oriented communication (TCP/IP)
- Universal Datagram Protocol (UDP): connectionless, unreliable (best-effort) datagram communication (just IP with minor additions)

TCP vs UDP

Works reliably over any network
Considerable overhead

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

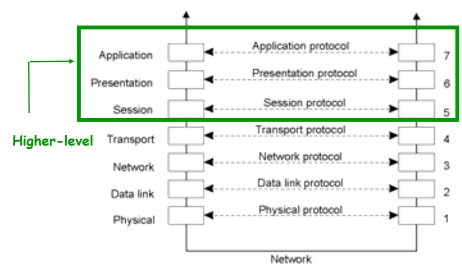
Transport Layer: Client-Server TCP



a) Normal operation of TCP.
b) Transactional TCP (T/TCP) enhancement

Higher-level Layers

In practice, only the application layer is used



Application Layer

Intended to contain a collection of standard network applications, such as those for email, file transfer, etc

From the OSI reference model, all distributed systems just applications

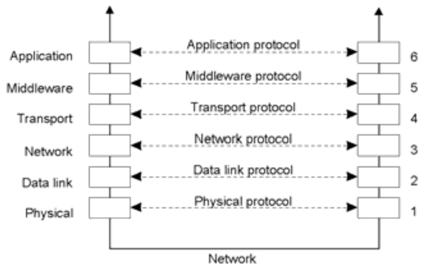
Many application protocols are directly implemented on top of transport protocols, doing a lot of application-independent work.

Middleware Layer

Middleware is invented to provide common services and protocols that can be used by many rich set of communication protocols, but which allow *different* applications to communicate

- Marshaling and unmarshaling of data, necessary for integrated systems
- Naming protocols, so that different applications can easily share resources
- Security protocols, to allow different applications to communicate in a secure way
- Scaling mechanisms, such as support for replication and caching
- Authentication protocols

Middleware Protocols



An adapted reference model for networked communication.

RPC

Basic RPC Model
Parameter Passing
Variations

Remote Procedure Call (RPC)

Basic idea:

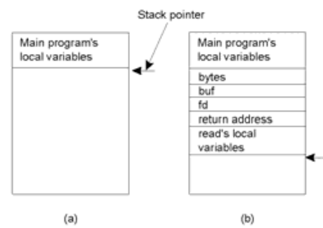
Allow programs to call procedures located on other machines

Some issues:

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

Conventional Procedure Call

Local procedure call: count = read(fd, buf, nbytes)
1: Caller: Push parameter values of the procedure on a stack + return address
2: Called procedure takes control
3: Called proc: Use stack for local variables, executes, pop local variables,save in cache return result, use return address
4: Caller: Pop results (in parameters)



Principle: "communication" with local procedure is handled by copying data **to/from the stack** (with a few exceptions)

Example: incr(i, i), initially i = 0

Call-by-Value, i = 0

Call-by-Reference, (push the address of the variable), i = 2

Call-by-Copy/Restore

The value is copied in the stack as in call-by-value, and then copied back by the called procedure, i = 1

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" procedure
- 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

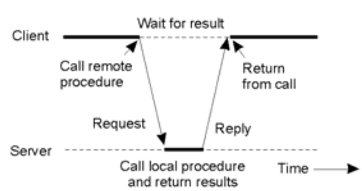
When the message arrives, the server OS passes it to the server stub

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

Client and Server Stubs

Call to procedure *x* -> call client stub for *x* -> client stub calls send and blocks -> upon receipt, the server stub gets control -> the server stub calls the local procedure *x* -> after the procedure x ends, control returns to the server stub that calls send, and then receive again and blocks -> the client OS, passes it to the client stub, copies it to the caller and returns



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

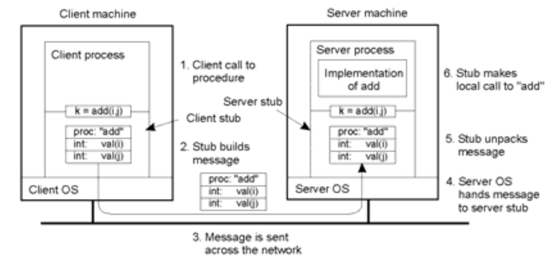
4. Remote OS gives message to server stub

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

Parameter Passing

Parameter marshaling: There's more than just wrapping parameters into a message:

- Client and server *machines* may have different data *representations* (think of byte ordering)
- Wrapping a parameter means transforming a value into a sequence of bytes
- Client and server have to agree on the same encoding:
  - How are basic data values represented (integers, floats, characters)
  - How are complex data values represented (arrays, unions)
- Client and server need to properly interpret messages, transforming them into machine-dependent representations.

Passing Value Parameters

An integer (one 32-bit word), and a four-character string (one 32-bit word)

Example, integer 5 and string JILL

a) Original message on the Pentium (right-to-left)
b) The message after receipt on the SPARC (left-to-right)
c) The message after being inverted. The little numbers in boxes indicate the address of each byte

Passing Reference Parameters

Pointer refers to the address space of the process it is being used

Solutions:

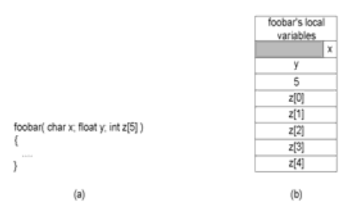
- Forbid pointers and reference parameters in general
- Use **copy in/copy out** semantics: while procedure is executed, nothing can be assumed about parameter values (only Ada supports this model).

RPC assumes *all* data that is to be operated on is passed by parameters. Excludes passing **references** to (global) data.

One optimization, if the stubs know which are parameters are input and output parameters -> eliminate copying

What about pointers to complex (arbitrary) data structures?

Parameter Specification and Stub Generation



Need to agree on:

Encoding rules

Actual exchange of messages (e.g., TCP/IP)

Implement the stubs!

Stubs for the same protocol and different procedures differ only in their interfaces to the applications

Interface Definition Language (IDL)

Extensions

- Calls to local procedures
- Asynchronous RPC

Doors

Try to use the RPC mechanism as the only mechanism for interprocess communication (IPC).

Doors are RPCs implemented for processes on the same machine

A single mechanism for communication: procedure calls (but with doors, it is not transparent)

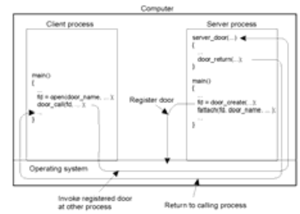
Server calls **door_create**: registers a door, an id is returned

fattach: associates a symbolic name with the id

Client invokes a door using **door_call**, the id and any parameters

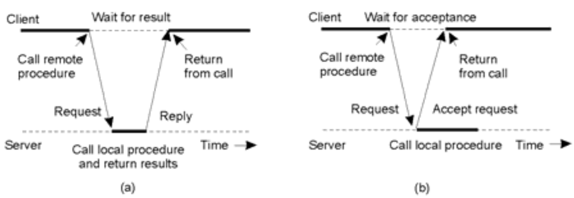
The OS does an upcall to the server

To return the result **door_return**



Asynchronous RPC

Try to get rid of the strict request-reply behavior, and let the client continue without waiting for ananswer from the server.

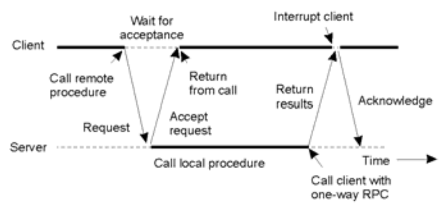


Traditional RPC

Asynchronous RPC

Asynchronous RPC: the server immediately sends a reply back to the client the moment the RPC request is received, after which it calls the requested procedure

Differed Synchronous RPC



Deferred Synchronous RPC: two asynchronous RPCs combined

The client uses asynchronous RPC to call the server

The server uses asynchronous RPC to send the reply

One way RPC: the client does not wait at all (reliability?)

Performing an RPC

At-most-one semantics: no call is ever carried out more than once, even in the case of system crashes

Idempotent remote procedure: a call may be repeated multiple times

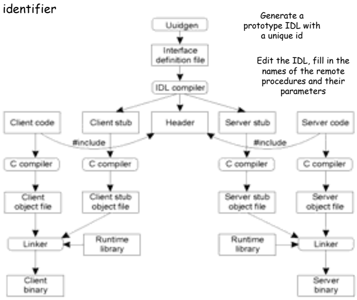
DCE RPC

Let the developer concentrate on only the client- and server-specific code; let the RPC system (generators and libraries) do the rest.

Writing a Client and a Server

IDL permits procedure declarations (similar to function prototypes in C).Type definitions, constant declarations, etc to provide information to correctly marshal/unmarshal paramters/results. Just the syntax (no semantics)

A globally unique identifier



The steps in writing a client and a server in DCE RPC.

Binding a Client to a Server

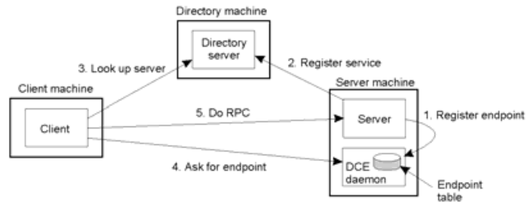
1. Locate the server machine

2. Locate the server on the machine: need to know an endpoint (port) on the server machine to which it can send messages

A table of (server, endpoints) is maintained on each server machine by a process called the DCE daemon

The server asks the OS for an endpoint and registers this endpoint with the DCE

The client asks the DCE daemon at the server's machine to lookup the endpoint



Remote Object Invocation

Distributed Objects
Remote Object Invocation
Parameter Passing

Distributed Objects

Expand the idea of RPCs to invocations on remote objects

- Data and operation encapsulated into an object
- Operations are implemented as **methods**, and are accessible through **interfaces**
- An object offers only its interface to clients
- **Object server** is responsible for a collection of objects
- **Client stub (proxy)** implements interface
- **Server skeleton** handles (un)marshaling and object invocation

Distributed Objects

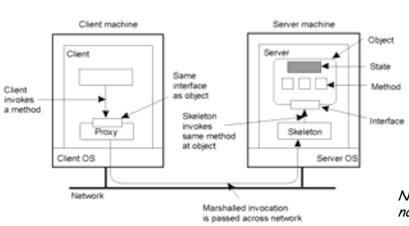
A client binds to a distributed object: an implementation of the object's interface, called a proxy, is loaded into the client's address space

Proxy (analog to a client stub)

Marshals method invocations into messages

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

Distributed Objects

Compile-time objects:

Objects defined as instances of a class
Compiling the class definition results in code that allows to instantiate Java objects
Language-level objects, from which proxy and skeletons are automatically generated.
Depends on the particular language

Runtime objects: Can be implemented in any language, but require use of an object adapter that makes the implementation *appear* as an object.
Adapter: objects defined based on their interfaces
Register an implementation at the adapter

Distributed Objects

Transient objects: live only by virtue of a server: if the server exits, so will the object.

Persistent objects: live independently from a server: if a server exits, the object's state and code remain (passively) on disk.

Binding a Client to an Object

Provide system-wide object references, freely passed between processes on different machines
Reference denotes the server machine plus an endpoint for the object server, an id of which object

When a process holds an object reference, it must first bind to the object

Bind: the local proxy (stub) is instantiated and initialized for specific object – implementing an interface for the object methods

Two ways of binding:
Implicit binding: Invoke methods directly on the referenced object

Explicit binding: Client must first explicitly bind to object before invoking it (generally returns a pointer to a proxy that then becomes locally available

Binding a Client to an Object

```
Distr_object* obj_ref;          //Declare a systemwide object reference
obj_ref = …;                        // Initialize the reference to a distributed object
obj_ref-> do_something();           // Implicitly bind and invoke a method
                                (a)
Distr_object objPref;           //Declare a systemwide object reference
Local_object* obj_ptr;          //Declare a pointer to local objects
obj_ref = …;                        //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);            //Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();          //Invoke a method on the local proxy
                                (b)
```

(a) Example with implicit binding using only global references
(b) Example with explicit binding using global and local references

Static vs Dynamic RMI

Remote Method Invocation (RMI)

Static invocation: the interfaces of an object are known when the client application is being developed

If interfaces change, the client application must be recompiled

Dynamic invocation: the application selects at runtime which method it will invoke at a remote object

invoke(object, method, input_parameters, output_parameters)

method is a parameter, **input_parameters**, **output_parameters** data structures

Static: **fobject.append(int)**

Dynamic: **invoke(fobject, id(append), int)**

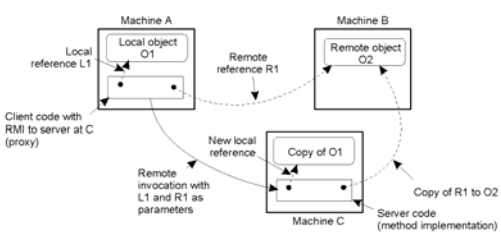
id(append) returns an id for the method append

Example uses: browsers, batch processing service to handle invocation requests

Object References as Parameters

When invoking a method with an object reference as a parameter, when it refers to *a remote object*, the reference is copied and passed as a value parameter (pass-by-reference)

When the reference refers to a *local object* (i.e., an object in the same address space as the client) the referred object is copied as a whole and passed along with the invocation (pass-by-value)



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

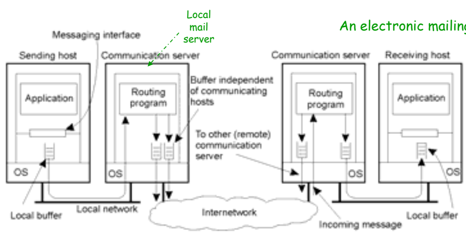
Synchronous communication: Sender blocks until the message arrives at the receiving host or is actually delivered and processed by the receiver

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

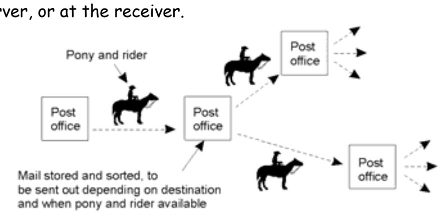
An electronic mailing system



Persistent vs Transient Communication

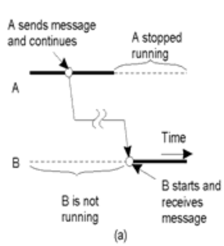
Persistent communication: A message is stored at a communication server as long as it takes to deliver it at the receiver.

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.



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

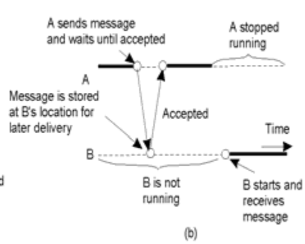
Messaging Combinations



Persistent asynchronous

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

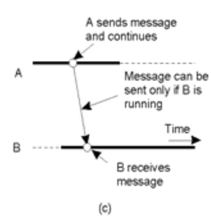
e.g., electronic mail systems



Persistent synchronous

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

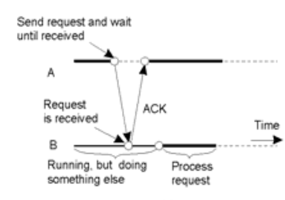
Messaging Combinations



Transient asynchronous

Transport-level datagram services (such as UDP)

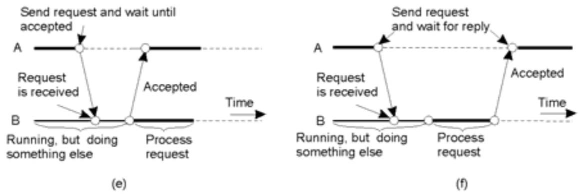
One-way RPC



Receipt-based transient synchronous

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

Messaging Combinations



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

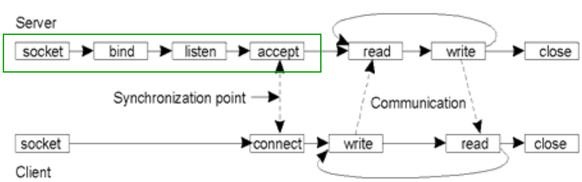
Socket: a communication endpoint to which an application can write data to be sent out over the network and from which incoming data may be read

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.

Berkeley Sockets



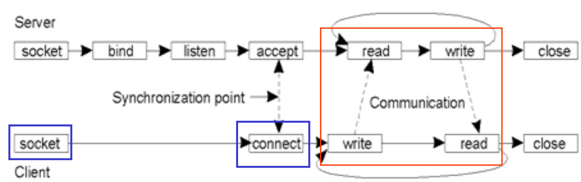
socket: creates a new communication endpoint for a specific transport protocol (the local OS reserves resources to accommodate sending and receiving messages for the specified protocol)

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

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

The Message-Passing Interface (MPI)

- Suitable for COWs and MPPs
- MPI designed for parallel applications and thus tailored to transient communication
- Assumes communication within a known group of processes, a (group_ID, process_ID) uniquely identifies a source or destination of a message

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

Outline

Message-Oriented Transient Communication

Transport-level sockets

Message-Passing Interface (MPI)

→ Message-Oriented Persistent Communication

Message Queuing Model

General Architecture

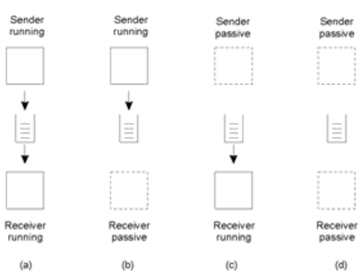
Example (IBM MQSeries: check the textbook)

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.

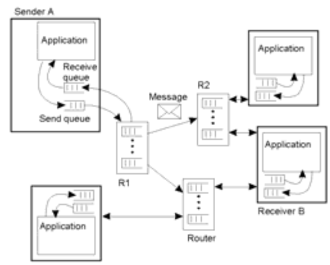
Primitive	Meaning
Put	Call by the sender Append a message to a specified queue Non-blocking
Get	Block until the specified queue is nonempty, and remove the first message Variations allow searching for a specific message in the queue
Poll	Check a specified queue for messages, and remove the first. Never 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 o 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 address
- A (possibly distributed) database of queue names to network locations
- Queues are managed by queue managers
- Relays: special queue managers that operate as routers and forward incoming messsges to other queue managers ⇒ overlay network

General Architecture of a Message-Queuing System

Why routers?

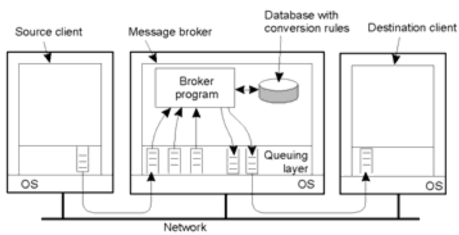


- Only the routers need to be updated when queues are added or removes
- Allow for secondary processing of messages (e.g., logging for fault tolerance)
- Used for multicasting purposes
- Act as message brokers

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



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 pf 1/T images per sec*

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

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

  E.g., sensor data

- 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)

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

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

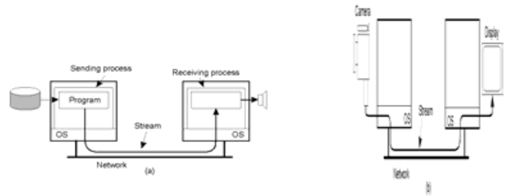
Stream Types

- 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



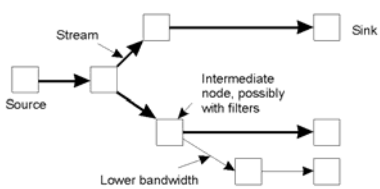
Data Streams

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 fo outgoing streams



Quality of Service

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

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

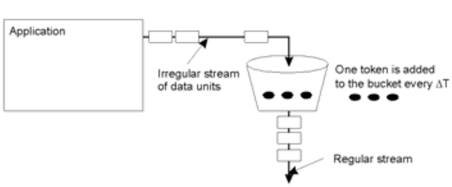
Flow Specification of QoS

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



Flow Specification of QoS

Characteristics of the Input	Service Required
•maximum data unit size (bytes) •Token bucket rate (bytes/sec)	•Loss sensitivity (bytes) •Loss interval (μsec) maximum acceptable loss rate
•Token bucket size (bytes)	•Burst loss sensitivity (data units) How many consecutive data units may be lost
•Maximum transmission rate (bytes/sec)	•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 guarentess

Implementing QoS

QoS specifications translate to resource reservations in the underlying communication system

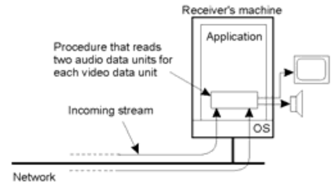
Resources: bandwidth, buffers, processing capacity

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

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

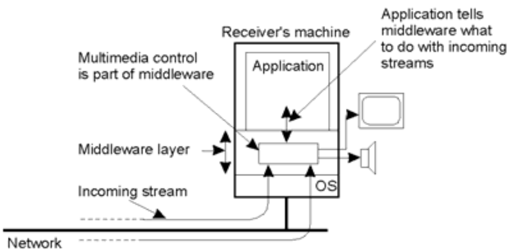
The principle of explicit synchronization on the level of data units.



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

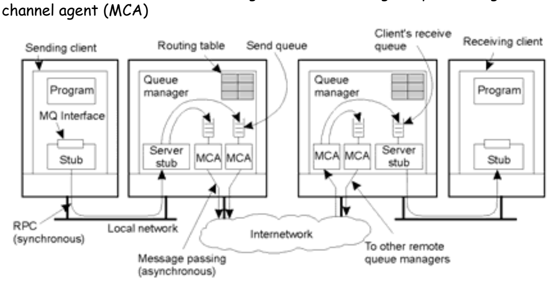
The principle of synchronization as supported by high-level interfaces.



Extra Slides

Example: IBM MQSeries

- All queues are managed by queue managers
- Queue managers are pair-wise connected through message channels
- Each of the two ends of a message channel is managed by a message channel agent (MCA)

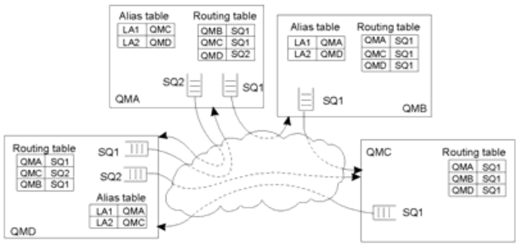


Example: IBM MQSeries

Attribute	Description
Transport type	Determines the transport protocol to be used
FIFO delivery	Indicates that messages are to be delivered in the order they are sent
Message length	Maximum length of a single message
Setup retry count	Specifies maximum number of retries to start up the remote MCA
Delivery retries	Maximum times MCA will try to put received message into queue

Some attributes associated with message channel agents.

Example: IBM MQSeries



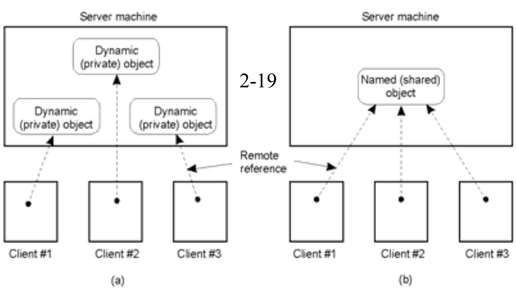
The general organization of an MQSeries queuing network using routing tables and aliases.

Example: IBM MQSeries

Primitive	Description
MQopen	Open a (possibly remote) queue
MQclose	Close a queue
MQput	Put a message into an opened queue
MQget	Get a message from a (local) queue

Primitives available in an IBM MQSeries MQI

The DCE Distributed-Object Model



a) Distributed dynamic objects in DCE.
b) Distributed named objects

Implementing QoS

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