
DISTRIBUTED SYSTEMS (COMP9243)

Lecture 2: System Architecture & Communication

Slide 1

- ① System Architectures
 - ② Processes
 - ③ Communication in a Distributed System
 - ④ Communication Abstractions
-
-

ARCHITECTURE

System Architecture:

- placement of machines
- placement of software on machines

Software Architecture:

Logical organisation of software components

- Layered
- Object-oriented
- Data-centered
- Event-based

There is no *single* best architecture

- depends on application requirements
 - and the environment!
-
-

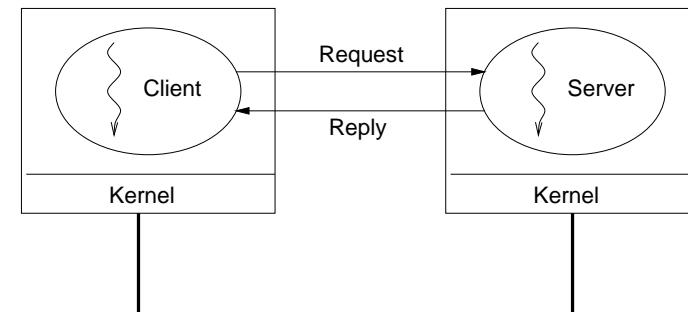
Slide 3

BUILDING A DISTRIBUTED SYSTEM

Two questions:

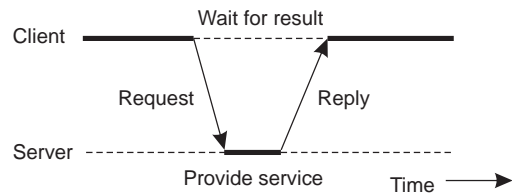
- ① Where to place the hardware?
 - ② Where to place the software?
-
-

CLIENT-SERVER



Slide 4

Client-Server from another perspective:



Slide 5

Example client-server code in C:

```
client(void) {
    struct sockaddr_in cin;
    char buffer[bufsize];
    int sd;

    sd = socket(AF_INET,SOCK_STREAM,0);
    connect(sd,(void *)&cin,sizeof(cin));
    send(sd,buffer,strlen(buffer),0);
    recv(sd,buffer,bufsize,0);
    close (sd);
}
```

Slide 7

Example client-server code in Erlang:

```
% Client code using the increment server
client (Server) ->
    Server ! {self (), 10},
    receive
        {From, Reply} -> io:format ("Result: ~w~n", [Reply])
    end.
```

Slide 6

```
% Server loop for increment server
loop () ->
    receive
        {From, Msg} -> From ! {self (), Msg + 1},
            loop ();
        stop -> true
    end.
% Initiate the server
start_server() -> spawn (fun () -> loop () end).
```

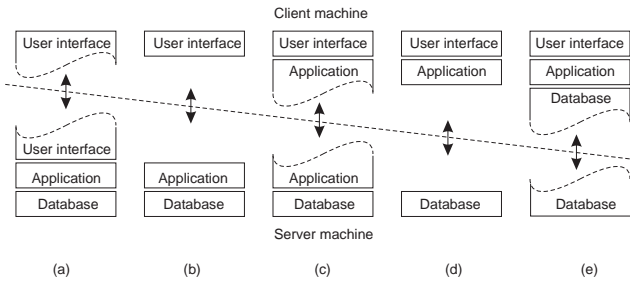
```
server(void) {
    struct sockaddr_in cin, sin;
    int sd, sd_client;

    sd = socket(AF_INET,SOCK_STREAM,0);
    bind(sd,(struct sockaddr *)&sin,sizeof(sin));
    listen(sd, queuesize);
    while (true) {
        sd_client = accept(sd,(struct sockaddr *)&cin,&addrllen);
        recv(sd_client,buffer,sizeof(buffer),0);
        DoService(buffer);
        send(sd_client,buffer,strlen(buffer),0);
        close (sd_client);
    }
    close (sd);
}
```

Slide 8

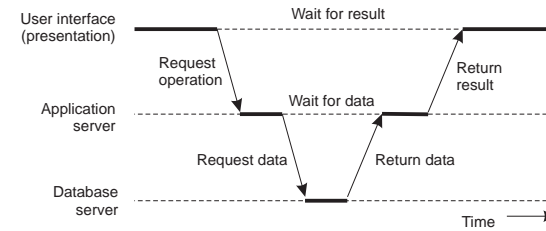
Splitting Functionality:

Slide 9



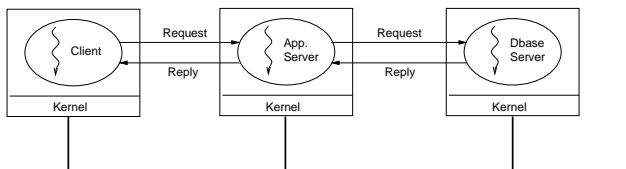
Vertical Distribution from another perspective:

Slide 11



VERTICAL DISTRIBUTION (MULTI-TIER)

Slide 10

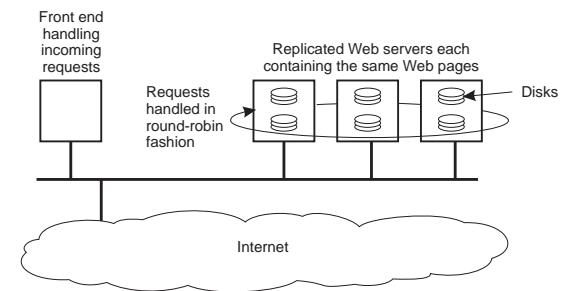


Three 'layers' of functionality:

- User interface
 - Processing/Application logic
 - Data
- Logically different components on different machines

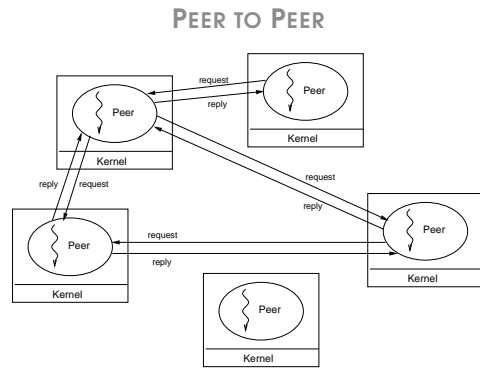
HORIZONTAL DISTRIBUTION

Slide 12



→ Logically equivalent components replicated on different machines

Slide 13

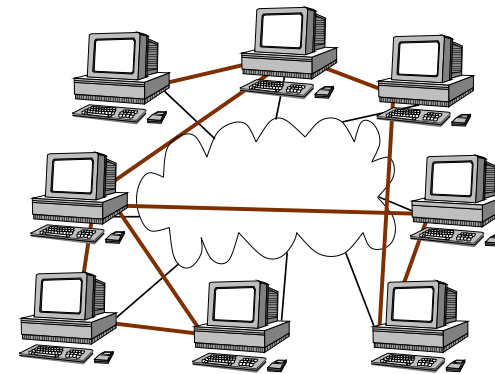


→ All processes have client and server roles: *servent*

Why is this special?

Example:

Slide 15



PEER TO PEER AND OVERLAY NETWORKS

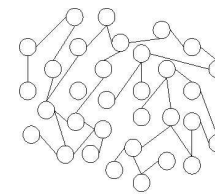
How do peers keep track of all other peers?

Slide 14

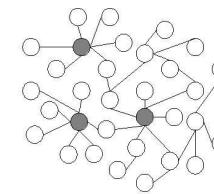
- static structure: you already know
- dynamic structure: *Overlay Network*
 - ① structured
 - ② unstructured

UNSTRUCTURED OVERLAY

Slide 16



(a) Random network

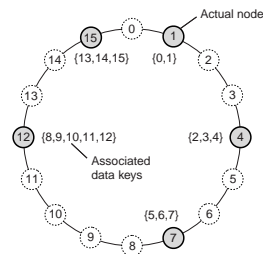


(b) Scale-free network

- Data stored at random nodes
- Partial view: node's list of neighbours
- Exchange partial views with neighbours to update

STRUCTURED OVERLAY

Distributed Hash Table:



Slide 17

- Nodes have identifier and range
- Data has identifier
- Node is responsible for data that falls in its range
- Search is routed to appropriate node

What's a problem with this?

HYBRID ARCHITECTURES

Combination of architectures.

Examples:

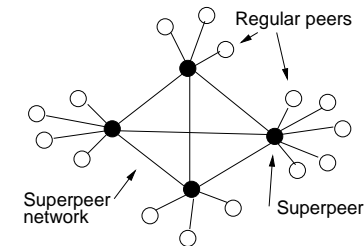
Slide 18

- Superpeer networks
- Edge-server systems
- Collaborative distributed systems

Superpeer Networks:

- Regular peers are clients of superpeers
- Superpeers are servers for regular peers
- Superpeers are peers among themselves
- Superpeers may maintain large index, or act as brokers

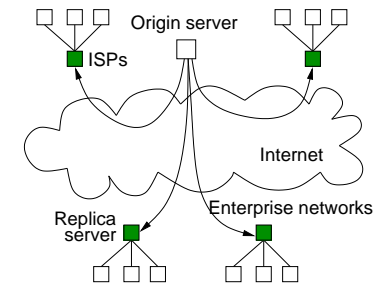
Slide 19



Edge-Server Networks:

- Servers placed at the edge of the network
- Servers replicate content
- Mostly used for content and application distribution
- *Content Distribution Networks*

Slide 20



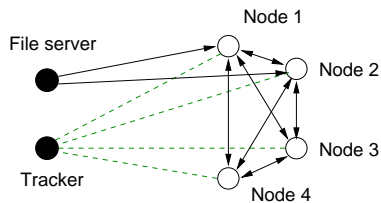
What are the challenges?

Collaborative Distributed Systems:

Example: BitTorrent

- Node downloads chunks of file from many other nodes
- Node provides downloaded chunks to other nodes
- Tracker keeps track of active nodes that have chunks of file
- Enforce collaboration by penalising selfish nodes

Slide 21



What problems does Bit Torrent face?

Slide 22

A CLOUD

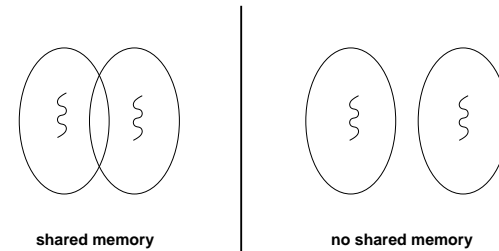
PROCESSES AND THREADS

Process Single thread of control per address space

Thread Multiple threads of control per address space

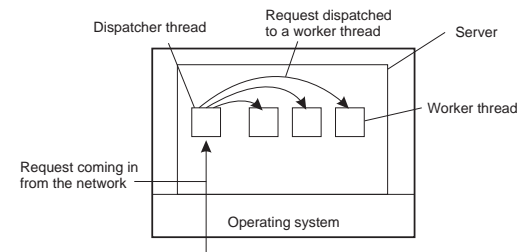
Memory access:

Slide 23



Will generally refer to threads of control as Processes, unless specifically mean Thread

SERVER DESIGN



Slide 24

Model	Characteristics
Single-threaded process	No parallelism, blocking system calls
Threads	Parallelism, blocking system calls
Finite-state machine	Parallelism, non-blocking system calls

STATEFUL VS STATELESS SERVERS

Stateful:

- Keeps persistent information about clients
- ✓ Improved performance
- ✗ Expensive crash recovery
- ✗ Must track clients

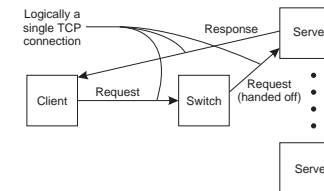
Slide 25

Stateless:

- Does not keep state of clients
- *soft state* design: limited client state
- ✓ Can change own state without informing clients
- ✓ No cleanup after crash
- ✓ Easy to replicate
- ✗ Increased communication

REQUEST SWITCHING

Transport layer switch:



Slide 27

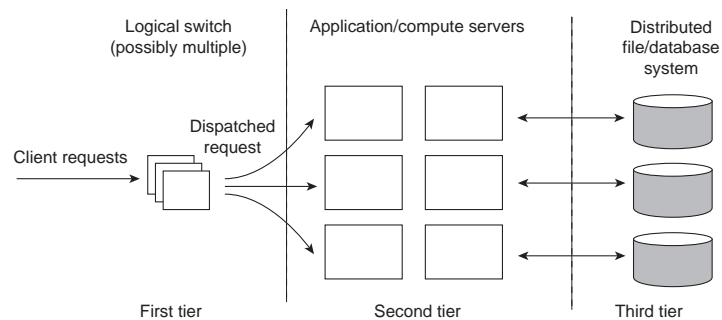
DNS-based:

- Round-robin DNS

Application layer switch:

- Analyse requests
- Forward to appropriate server

CLUSTERED SERVERS



Slide 26

CODE MOBILITY

Why move code?

- Optimise computation (load balancing)
- Optimise communication

Weak vs Strong Mobility:

Weak transfer only code

Strong transfer code and execution segment

Sender vs Receiver Initiated migration:

Sender Send program to compute server

Receiver Download applets

What are the challenges of code mobility?

COMMUNICATION

Cooperating processes need to communicate.

- For synchronisation and control
- To share data

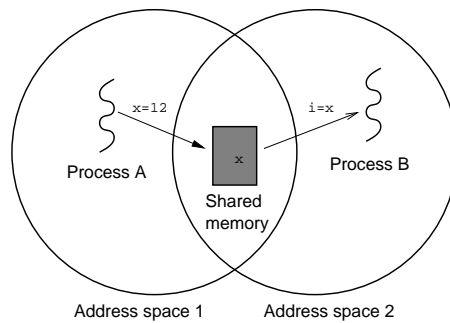
Two approaches to communication:

→ Shared memory

Slide 29

- Direct memory access (Threads)
- Mapped memory (Processes)

Shared Memory:



Slide 30

COMMUNICATION

Cooperating processes need to communicate.

- For synchronisation
- To share data

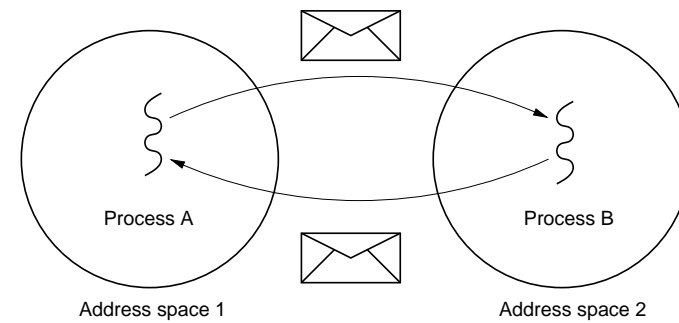
Two approaches to communication:

→ Shared memory

Slide 31

- Direct memory access (Threads)
- Mapped memory (Processes)
- Message passing
 - OS's IPC mechanisms

Message Passing:



Slide 32

COMMUNICATION IN A DISTRIBUTED SYSTEM

Previous slides assumed a uniprocessor or a multiprocessor.

In a distributed system (multicomputer) things change:

Shared Memory:

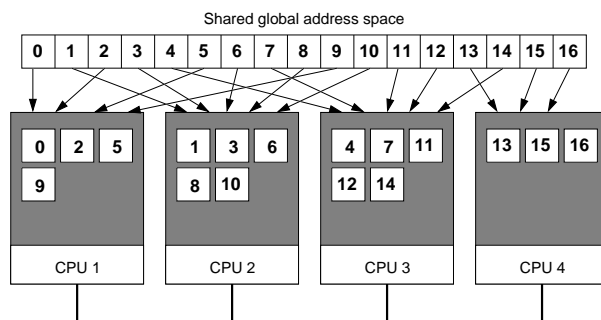
- There is no way to physically share memory
- Distributed Shared Memory

Message Passing:

- Over the network
- Introduces latencies
- Introduces higher chances of failure
- Heterogeneity introduces possible incompatibilities

Slide 33

DISTRIBUTED SHARED MEMORY (DSM)



Emulate shared memory over the network

Slide 34

MESSAGE PASSING

Basics:

- `send()`
- `receive()`

Variations:

- Connection oriented vs Connectionless
- Synchronous vs Asynchronous
- Buffered vs Unbuffered
- Reliable vs Unreliable
- Message ordering guarantees

Data Representation:

- Marshalling
- Endianness

Slide 35

COMMUNICATION MODES

Slide 36

- ① Data oriented vs control oriented communication
- ② Synchronous vs asynchronous communication
- ③ Transient vs persistent communication

Data-Oriented vs Control-Oriented Communication:

Data-oriented communication

- Facilitates data exchange between threads
- Shared address space, shared memory & message passing

Control-oriented communication

- Associates a transfer of control with communication
- Active messages, remote procedure call (RPC) & remote method invocation (RMI)

Slide 37

Observation:

- Hardware and OSes often provide data-oriented communication
- Higher-level infrastructure often provides control-oriented communication → **middleware**
- But some OSes provide RPC, MPI provides data-oriented communication

Synchronous vs Asynchronous Communication:

Synchronous

- Sender blocks until message received
 - Often sender blocked until message is processed and a reply received
- Sender and receiver must be active at the same time
- Receiver waits for requests, processes them (ASAP), and returns reply
- Client-Server generally uses synchronous communication

Slide 38

Asynchronous

- Sender continues execution after sending message (does not block waiting for reply)
- Message may be queued if receiver not active
- Message may be processed later at receiver's convenience

When is Synchronous suitable? Asynchronous?

Transient vs Persistent Communication:

Transient

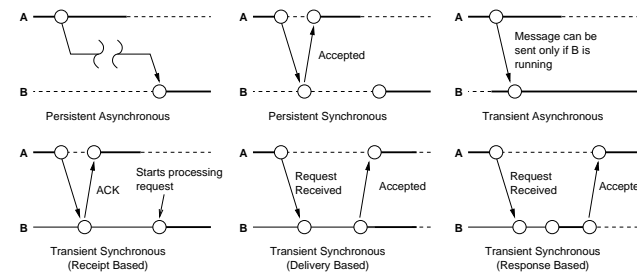
- Message discarded if cannot be delivered to receiver immediately
- Example: HTTP request

Slide 39

Persistent

- Message stored (somewhere) until receiver can accept it
- Example: email

Combinations:



Slide 40

Examples?

COMMUNICATION ABSTRACTIONS

Abstractions above simple message passing make communication easier for the programmer.

Provided by higher level APIs

- ① Remote Procedure Call (RPC)
- ② Remote Method Invocation (RMI)
- ③ Message-Oriented Communication
- ④ Group Communication
- ⑤ Streams

Slide 41

REMOTE PROCEDURE CALL (RPC)

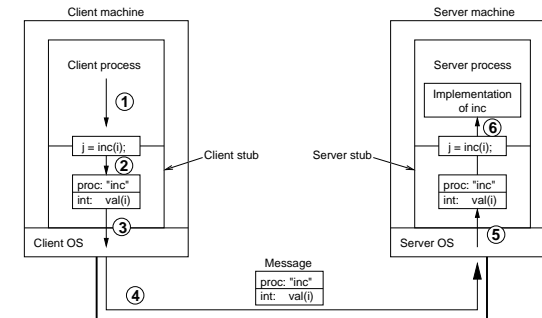
Idea: Replace I/O oriented message passing model by execution of a procedure call on a remote node (BN84):

- Synchronous - based on blocking messages
- Message-passing details hidden from application
- Procedure call parameters used to transmit data
- Client calls local "stub" which does messaging and marshalling

Confusion of local and remote operations can be dangerous. More on that later...

Slide 42

RPC Implementation:



Slide 43

Example client stub in Erlang:

```
% Client code using RPC stub
client (Server) ->
    register(server, Server),
    Result = inc (10),
    io:format ("Result: ~w~n", [Result]).
```

Slide 44

```
% RPC stub for the increment server
inc (Value) ->
    server ! {self (), inc, Value},
    receive
        {From, inc, Reply} -> Reply
    end.
```

Example server stub in Erlang:

```
% increment implementation
inc (Value) -> Value + 1.
```

Slide 45

```
% RPC Server dispatch loop
server () ->
  receive
    {From, inc, Value} ->
      From ! {self(), inc, inc(Value)}
  end,
  server().
```

Parameter marshalling:

- stub must pack ("marshal") parameters into message structure
- message data must be pointer free (by-reference data must be passed by-value)
- may have to perform other conversions:
 - byte order (big endian vs little endian)
 - floating point format
 - dealing with pointers
 - convert everything to standard ("network") format, or
 - message indicates format, receiver converts if necessary
- stubs may be generated automatically from interface specs

Examples of RPC frameworks:

- SUN RPC (aka ONC RPC): Internet RFC1050 (V1), RFC1831 (V2)
 - Based on XDR data representation (RFC1014)(RFC1832)
 - Basis of standard distributed services, such as NFS and NIS
- Distributed Computing Environment (DCE) RPC
- XML (data representation) and HTTP (transport)
 - Text-based data stream is easier to debug
 - HTTP simplifies integration with web servers and works through firewalls
 - For example, XML-RPC (lightweight) and SOAP (more powerful, but often unnecessarily complex)

Slide 47

Sun RPC - interface definition:

```
program DATE_PROG {
  version DATE_VERS {
    long BIN_DATE(void) = 1; /* proc num = 1 */
    string STR_DATE(long) = 2; /* proc num = 2 */
  } = 1; /* version = 1 */
} = 0x31234567; /* prog num */
```

Slide 48

Sun RPC - client code:

```
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */
...
main(int argc, char **argv) {
    CLIENT *cl; /* RPC handle */
    ...
    cl = clnt_create(argv[1], DATE_PROG, DATE_VERS, "udp");

    lresult = bin_date_1(NULL, cl);
    printf("time on host %s = %ld\n", server, *lresult);

    sresult = str_date_1(lresult, cl);
    printf("time on host %s = %s", server, *sresult);

    clnt_destroy(cl); /* done with the handle */
}
```

Slide 49

Sun RPC - server code:

```
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */

long * bin_date_1() {
    static long timeval; /* must be static */
    long time(); /* Unix function */
    timeval = time((long *) 0);
    return(&timeval);
}

char ** str_date_1(long *bintime) {
    static char *ptr; /* must be static */
    char *ctime(); /* Unix function */
    ptr = ctime(bintime); /* convert to local time */
    return(&ptr); /* return the address of pointer */
}
```

Slide 50

DYNAMIC BINDING

How to locate a service?

→ Well-known naming service, "binder":

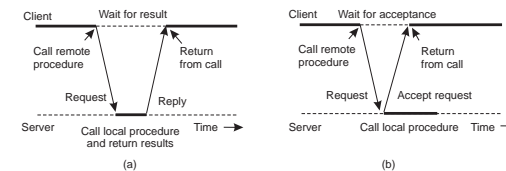
- register(name, version, handle, UID)
- deregister(name, version, UID)
- lookup(name, version) → (handle, UID)

→ handle is some physical address (IP address, process ID, ...)

→ UID is used to distinguish between servers offering the same service

Slide 51

ASYNCHRONOUS RPC



Slide 52

→ When no reply is required

→ When reply isn't needed immediately (2 asynchronous RPCs - deferred synchronous RPC)

REMOTE METHOD INVOCATION (RMI)

The transition from Remote Procedure Call (RPC) to Remote Method Invocation (RMI) is a transition from the server metaphor to the object metaphor.

Why is this important?

- Slide 53**
- RPC: explicit handling of host identification to determine the destination
 - RMI: addressed to a particular object
 - Objects are first-class citizens
 - Can pass object references as parameters
 - More natural resource management and error handling
 - **But still**, only a small evolutionary step
-

TRANSPARENCY CAN BE DANGEROUS

Why is the transparency provided by RPC and RMI dangerous?

- Slide 54**
- Remote operations can fail in different ways
 - Remote operations can have arbitrary latency
 - Remote operations have a different memory access model
 - Remote operations can involve concurrency in subtle ways

What happens if this is ignored?

- Unreliable services and applications
- Limited scalability
- Bad performance

See "A note on distributed computing" (WWWK94)

MESSAGE-ORIENTED MIDDLEWARE (MOM)

Middleware services to provide message passing.

Traditional `send()/receive()` provides:

- Asynchronous and Synchronous communication
- Transient communication

Slide 55

What more does it provide than `send()/receive()`?

- Persistent communication (Message queues)
 - Hides implementation details
 - Marshalling
-

Example: Message Passing Interface (MPI):

- Designed for parallel applications
- Makes use of available underlying network
- Tailored to transient communication
- No persistent communication
- Primitives for all forms of transient communication
- Group communication

Slide 56

MPI is BIG. Standard reference has over 100 functions and is over 350 pages long!

MPI primitives:

Slide 57

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_issend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Basic queue interface:

Slide 59

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

MESSAGE QUEUING SYSTEMS

Provides:

- Persistent communication
- Message Queues
- Transfer of messages between queues

Slide 58

Model:

- Application-specific queues
- Messages addressed to specific queues
- Only guarantee delivery to queue. Not when.
- Message transfer can be in the order of minutes

Very similar to email but more general purpose (i.e., enables communication between applications and not just people)

Message Queue Architecture — Concepts:

Queue

- Source queue (local)
- Destination queue (remote)

Queue Manager

Slide 60

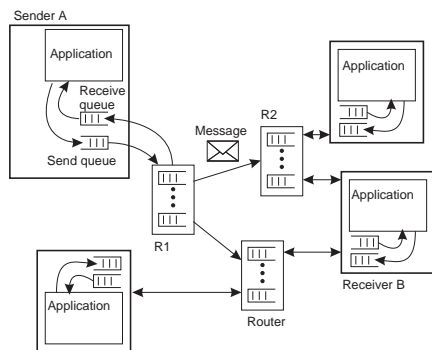
- Direct (manages queues on its machine)
- Relay/Router (routes to given destination queue's machine)
- Multicast support (routes to multiple destination queues)

Message Broker

- Translates between application specific message formats
- Application gateway

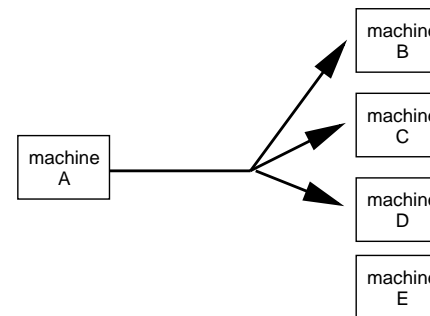
Message Queue Architecture Example:

Slide 61



GROUP COMMUNICATION

Slide 63



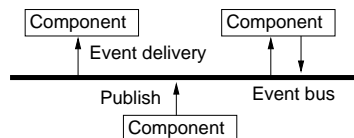
→ Sender performs a single send()

What are the difficulties with group communication?

EVENT-BASED COMMUNICATION

Slide 62

- Communication through propagation of events
- Generally associated with *publish/subscribe* systems
- Sender processes publish events
- Receiver processes subscribe to events and receive only the ones they are interested in.



Two kinds of group communication:

- Broadcast (message sent to everyone)
- Multicast (message sent to specific group)

Used for:

- Replication of services
- Replication of data
- Service discovery
- Event notification

Slide 64

Issues:

- Reliability
- Ordering

Example:

- IP multicast

GOSSIP-BASED COMMUNICATION

Technique that relies on *epidemic behaviour*, e.g. spreading diseases among people.

Variant: *rumour spreading*, or *gossiping*.

Slide 65

- When node P receives data item x , it tries to push it to arbitrary node Q .
- If x is new to Q , then P keeps on spreading x to other nodes.
- If node Q already has x , P stops spreading x with certain probability.

Analogy from real life: Spreading rumours among people.

STREAMS

Slide 66

- Support for Continuous Media
 - Between applications
 - Between devices
-

Continuous Media:

- Data represented as single stream rather than discrete chunks
- Temporal relationship between data (timing has an effect on correctness)
- Minimum and maximum end-to-end time delays

Slide 67

Contrast to Discrete Media:

- No temporal relationship between data (timing does not have an effect on correctness)
-
-

Data Streams:

- Sequence of data units
- Can apply to discrete and continuous media (e.g., TCP connection is a stream)
- Simple stream: single sequence of data
- Complex stream: several related streams (substreams) (e.g., audio and video)

Slide 68

Transmission modes:

Asynchronous no timing constraints (e.g., file transfer)

Synchronous maximum end-to-end delay for each data unit (e.g., sending sampled data. Must not be too old when it reaches destination)

Isochronous maximum and minimum end-to-end delay (e.g., video)

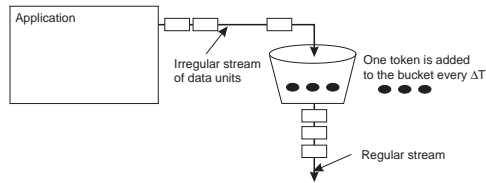
Will concentrate on **Isochronous streams**

Quality of Service:

Time dependent requirements

- Specified as flow specification
- Bandwidth requirements, transmission rates, delays, etc.
- Token bucket model

Slide 69



Stream setup requires:

- Reservation of communication resources
-

Stream Synchronisation:

Maintaining temporal relationships between substreams

Examples: synchronised audio and video, stereo audio

Client based

- Client receives multiple streams
- Uses synchronisation profile to synchronise
- Example: based on timestamps
- ✗ Different streams may have different delays

Slide 70

Server based

- Multiplex data streams into one stream
 - ✗ What's the problem with this?
-

READING LIST

Slide 71 **Implementing Remote Procedure Calls** A classic paper about the design and implementation of one of the first RPC systems.
