Distributed Systems (COMP9243)

Lecture 2: System Architecture & Communication

Slide 1

① System Architectures
② Processes & Server Architecture
③ Communication in a Distributed System
④ Communication Abstractions

Principles

Slide 2

Several key principles underlying the functioning of all distributed systems

- ➜ System Architecture
- ➜ Communication
- ➜ Replication and Consistency
- ➜ Synchronisation
- ➜ Naming
- ➜ Fault Tolerance
- ➜ Security

Discussion of these principles will form the core content of the course

Building a Distributed System

Slide 3

Two questions:

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

Architecture

Slide 4

System Architecture:

- placement of machines
- placement of software on machines

Where to place?:

- processing capacity, load balancing
- communication capacity
- locality

Mapping of services to servers:

- Partitioning
- Replication
- Caching

Slide 5

Software Architecture:

Logical organisation and roles of software components

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

There is no *single* best architecture

- depends on application requirements
- and the environement!

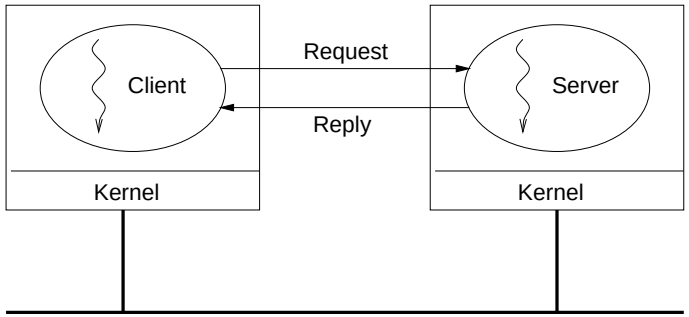
Slide 6

Slide 7

ARCHITECTURAL PATTERNS

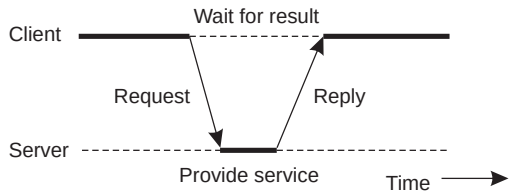
Slide 8

CLIENT-SERVER



Slide 9

Client-Server from another perspective:



Slide 10

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.

% 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).
```


Slide 11

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);
}
```

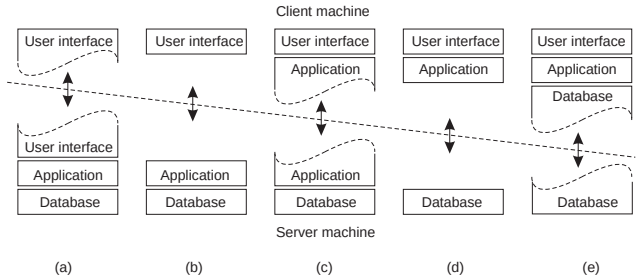
Slide 12

```
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,&addrlen));
    recv(sd_client,buffer,sizeof(buffer),0);
    DoService(buffer);
    send(sd_client,buffer,strlen(buffer),0);
    close (sd_client);
  }
  close (sd);
}
```

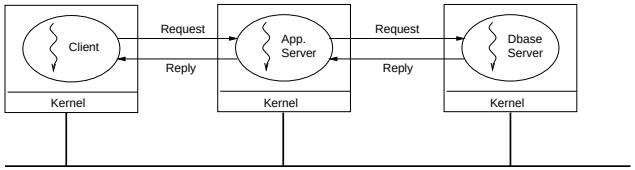
Slide 13

Splitting Functionality:



Slide 14

VERTICAL DISTRIBUTION (MULTI-TIER)



Three 'layers' of functionality:

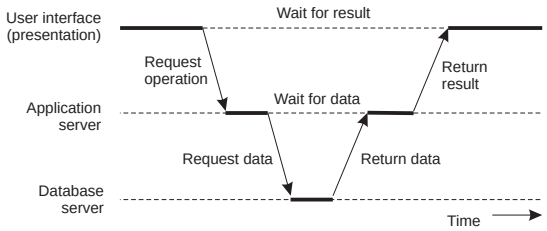
- User interface
- Processing/Application logic
- Data

➜ Logically different components on different machines

How scalable is this?

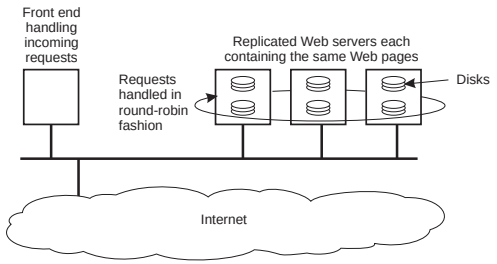
Slide 15

Vertical Distribution from another perspective:



Slide 16

HORIZONTAL DISTRIBUTION



➜ Logically equivalent components replicated on different machines

How scalable is this?

Note: Scaling Up vs Scaling Out?

Horizontal and Vertical *Distribution* not the same as Horizontal and Vertical *Scaling*.

Slide 17

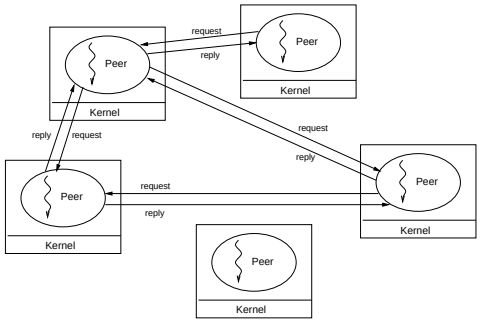
Vertical Scaling: Scaling UP Increasing the resources of a single machine

Horizontal Scaling: Scaling OUT Adding more machines.

Horizontal and Vertical Distribution are both examples of this.

Peer to Peer

Slide 18



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

Why is this special?

How do peers keep track of all other peers?

Slide 19

➜ static structure: you already know

➜ dynamic structure: *Overlay Network*

① structured

② unstructured

Overlay Network:

➜ Application-specific network

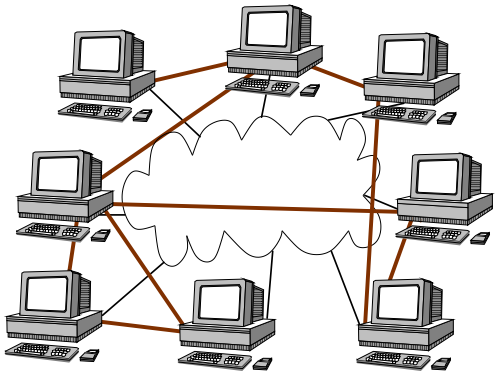
➜ Addressing

➜ Routing

➜ Specialised features (e.g., encryption, multicast, etc.)

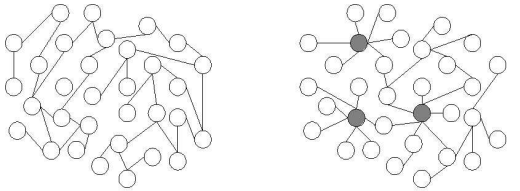
Example:

Slide 20



Slide 21

Unstructured Overlay



(a) Random network (b) Scale-free network

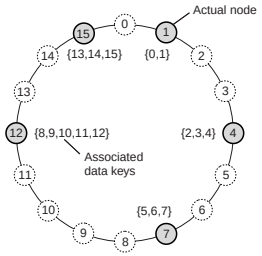
- ➜ Data stored at random nodes
- ➜ Partial view: node's list of neighbours
- ➜ Exchange partial views with neighbours to update

What's a problem with this?

Slide 22

Structured Overlay

Distributed Hash Table:



- ➜ Nodes have identifier and range, Data has identifier
- ➜ Node is responsible for data that falls in its range
- ➜ Search is routed to appropriate node
- ➜ Examples: Chord, Pastry, Kademlia

What's a problem with this?

Slide 23

Hybrid Architectures

Combination of architectures.

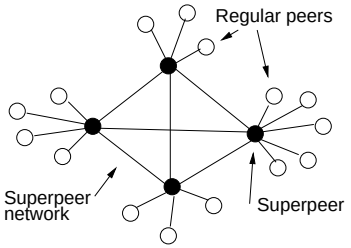
Examples:

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

Slide 24

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
- ➜ Example: Skype

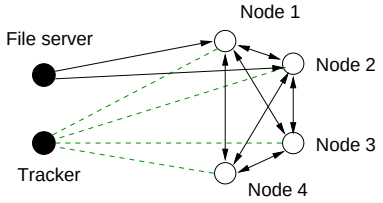


Collaborative Distributed Systems:

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 25

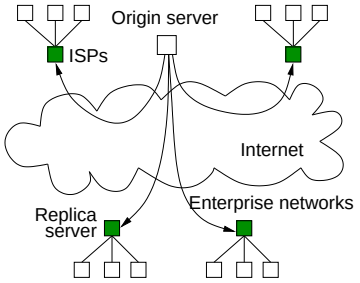


What problems does Bit Torrent face?

Edge-Server Networks:

- ➜ Servers placed at the edge of the network
- ➜ Servers replicate content
- ➜ Mostly used for content and application distribution
- ➜ *Content Distribution Networks*: Akamai, CloudFront, CoralCDN

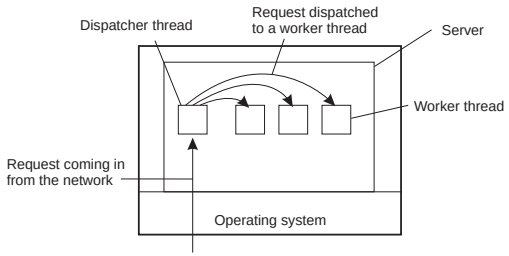
Slide 26



What are the challenges?

SERVER DESIGN

Slide 27



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

Slide 28

Stateful:

- ➜ Keeps persistent information about clients
- ☑ Improved performance
- ☒ Expensive crash recovery
- ☒ Must track clients

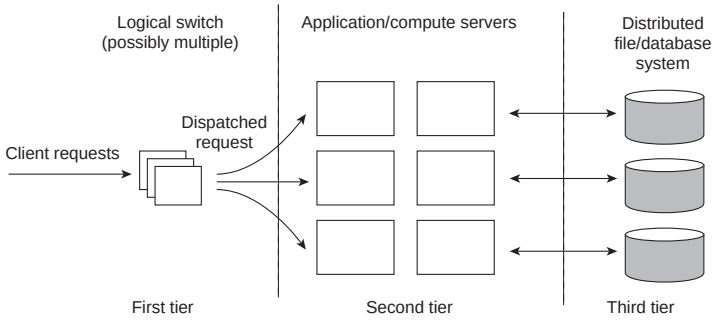
Stateless:

- ➜ 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

Note: Session state vs. Permanent state

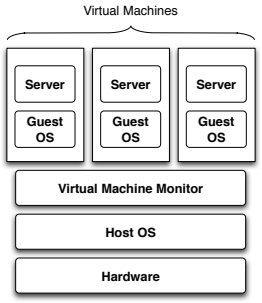
Slide 29

Clustered Servers



Slide 30

Virtualisation

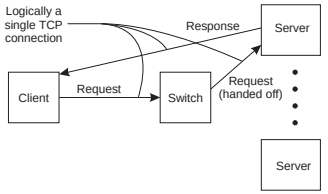


What are the benefits?

Slide 31

Request Switching

Transport layer switch:



DNS-based:

- ➜ Round-robin DNS

Application layer switch:

- ➜ Analyse requests
- ➜ Forward to appropriate server

Slide 32

Code Mobility

Why move code?

- ➜ Optimise computation (load balancing)
- ➜ Optmise communication

Weak vs Strong Mobility:

Weak transfer only code

Strong transfer code and execution segment

Sender vs Receiver Intitated migration:

Sender Send program to compute server

Receiver Download applets

Examples: Java, JavaScript, Virtual Machines, Mobile Agents

What are the challenges of code mobility?

Slide 33

COMMUNICATION

Slide 34

Why Communication:

Cooperating processes need to communicate.

- ➜ For synchronisation and control
- ➜ To share data

Slide 35

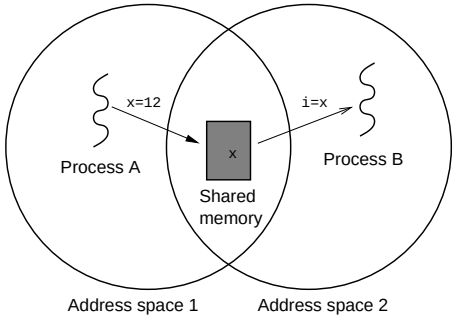
In a Non-Distributed System:

Two approaches to communication:

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

Slide 36

Shared Memory:



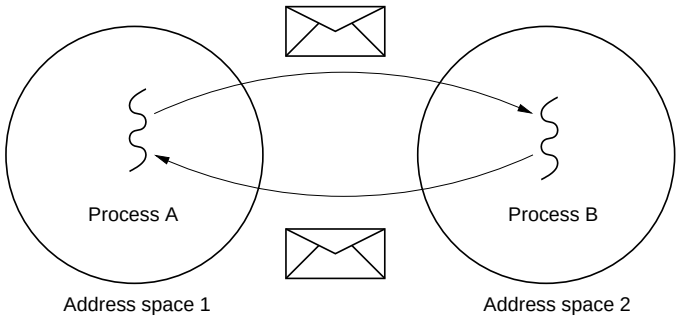
In a Non-Distributed System:

Two approaches to communication:

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

Slide 37

Message Passing:



Slide 38

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

Slide 39

Message Passing:

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

MESSAGE PASSING

Basics:

- ➜ `send()`
- ➜ `receive()`

Variations:

- ➜ Connection oriented vs Connectionless
- ➜ Point-to-point vs Group
- ➜ Synchronous vs Asynchronous
- ➜ Buffered vs Unbuffered
- ➜ Reliable vs Unreliable
- ➜ Message ordering guarantees

Data Representation:

- ➜ Marshalling
- ➜ Endianness

Slide 40

Coupling

Slide 41

Dependency between sender and receiver

Temporal do sender and receiver have to be active at the same time?

Spatial do sender and receiver have to know about each other? explicitly address each other?

Semantic do sender and receiver have to share knowledge of content syntax and semantics?

Platform do sender and receiver have to use the same platform?

Tight vs Loose coupling: yes vs no

Communication Modes

Slide 42

① Data oriented vs control oriented communication
② Synchronous vs asynchronous communication
③ Transient vs persistent communication
④ Provider-initiated vs consumer-initiated communication
⑤ Direct -addressing vs indirect-addressing communication

Slide 43

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 44

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

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?

Slide 45

Transient vs Persistent Communication:

Transient

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

Persistent

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

Coupling: Time

Slide 46

Provider-Initiated vs Consumer-Initiated Communication:

Provider-Initiated

- ➜ Message sent when data is available
- ➜ Example: notifications

Consumer-Initiated

- ➜ Request sent for data
- ➜ Example: HTTP request

Slide 47

Direct-Addressing vs Indirect-Addressing Communication:

Direct-Addressing

- ➜ Message sent directly to receiver
- ➜ Example: HTTP request

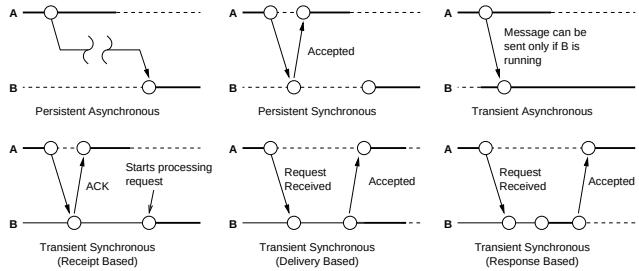
Indirect-Addressing

- ➜ Message not sent to a particular receiver
- ➜ Example: broadcast, publish/subscribe

Coupling: Space

Slide 48

Combinations:



Examples?

Communication Abstractions

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

Provided by higher level APIs

Slide 49

① Message-Oriented Communication
② Request-Reply, Remote Procedure Call (RPC) & Remote Method Invocation (RMI)
③ Group Communication
④ Event-based Communication
⑤ Shared Space

Message-Oriented Communication

Communication models based on message passing

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

Slide 50

➜ Asynchronous and Synchronous communication
➜ Transient communication

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

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

Example: Message Passing Interface (MPI)

Slide 51

➜ 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

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

MPI primitives:

Slide 52

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_isend	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

Example: Message Queuing Systems

Slide 53

Provides:

- ➜ Persistent communication
- ➜ Message Queues: store/forward
- ➜ Transfer of messages between queues

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)

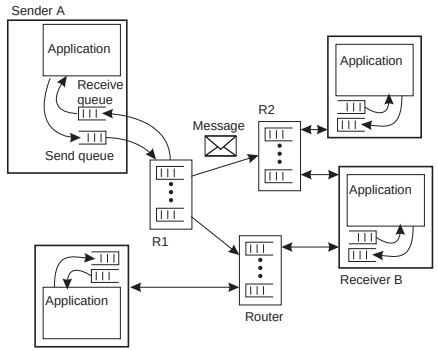
Basic queue interface:

Slide 54

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 Queue Architecture Example:

Slide 55



Examples:

IBM MQSeries

Slide 56

- ➜ Message channels connect queue managers
- ➜ Message Channel Agent (MCA) manages a channel end (sends and receives transport level messages)
- ➜ Source and destination MCAs must be running to use channel
- ➜ Queue manager responsible for routing

Java Message Service

- ➜ Java API to messaging system (e.g., MQ Series)
- ➜ Implementation of own messaging system
- ➜ Provides Point-to-point (usual message queues) and Publish/Subscribe

Others: Amazon SQS, Advanced Message Queuing Protocol, MQTT, STOMP

Request-Reply Communication

Slide 57

Request:

- ➜ a service
- ➜ data

Reply:

- ➜ result of executing service
- ➜ data

Requirement:

- ➜ Message formatting
- ➜ Protocol

Remote Procedure Call (RPC)

Slide 58

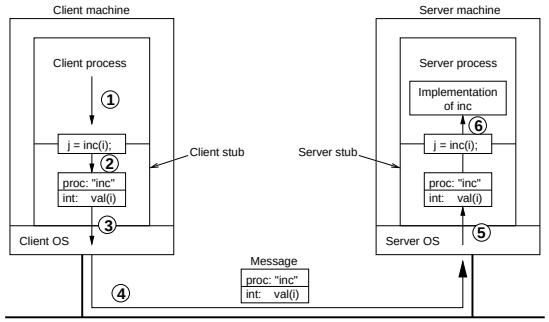
Idea: Replace I/O oriented message passing model by execution of a procedure call on a remote node (BN84):

- ➜ 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 59

RPC Implementation:



Slide 60

RPC Implementation:

① client calls client stub (normal procedure call)
② client stub packs parameters into message data structure
③ client stub performs `send()` syscall and blocks
④ kernel transfers message to remote kernel
⑤ remote kernel delivers to server stub, blocked in `receive()`
⑥ server stub unpacks message, calls server (normal proc call)
⑦ server returns to stub, which packs result into message
⑧ server stub performs `send()` syscall
⑨ kernel delivers to client stub, which unpacks and returns

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 61

```
% 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 62

```
% 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

Slide 63

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)
- ➜ Many More: Facebook Thrift, Google Protocol Buffers RPC, Microsoft .NET

Slide 64

Slide 65

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 66

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 67

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 68

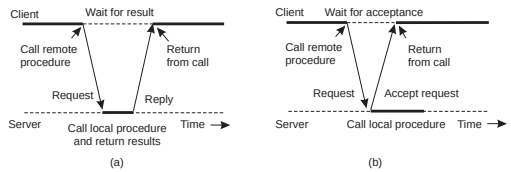
DYNAMIC BINDING

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

Asynchronous RPC

Slide 69



- 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 70

- 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 71

- 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

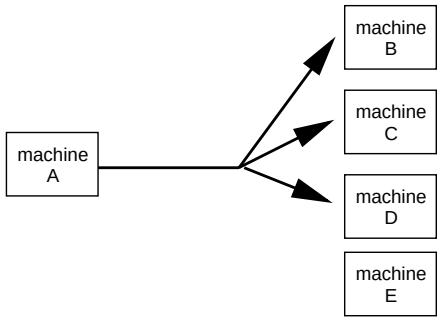
What happens if this is ignored?

- Unreliable services and applications
- Limited scalability
- Bad performance

See "A note on distributed computing" (Waldo et al. 94)

Group-Based Communication

Slide 72



- Sender performs a single `send()`

What are the difficulties with group communication?

Two kinds of group communication:

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

Used for:

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

Slide 73

Issues:

- ➜ Reliability
- ➜ Ordering

Example:

- ➜ IP multicast
- ➜ Flooding

Gossip-Based Communication

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

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

Slide 74

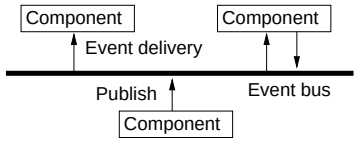
- When node P receives data item x, it tries to push it to arbitrary node Q.
- If x is new to Q, then P keeps on spreading x to other nodes.
- If node Q already has x, P stops spreading x with certain probability.

Analogy from real life: Spreading rumours among people.

Event-Based Communication

- ➜ Communication through propagation of events
- ➜ Generally associated with *publish/subscribe* systems
- ➜ Sender process publishes events
- ➜ Receiver process subscribes to events and receives only the ones it is interested in.
- ➜ Loose coupling
- ➜ Example: OMG Data Distribution Service (DDS), JMS, Tibco

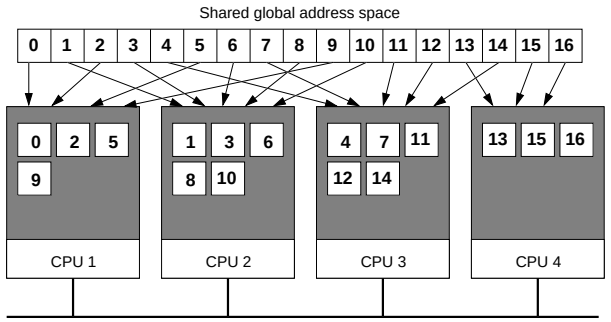
Slide 75



Shared Space

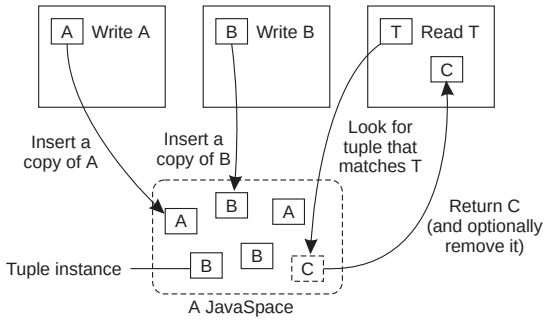
Distributed Shared Memory:

Slide 76



Slide 77

Tuple Space:



READING LIST

Slide 78

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