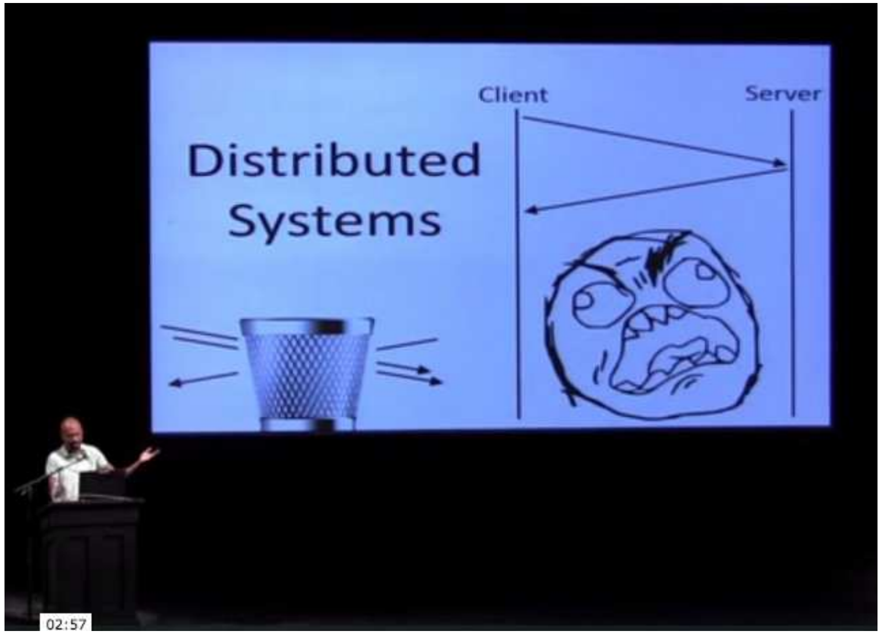
Lecture 2: System Architecture & Communication



① System Architectures

② Processes & Server Architecture

③ Communication in a Distributed System

④ Communication Abstractions

Architecture

BUILDING A DISTRIBUTED SYSTEM

Two questions:

① Where to place the hardware?

② Where to place the software?

System Architecture:

- ➜ placement of machines
- ➜ placement of software on machines

Where to place?:

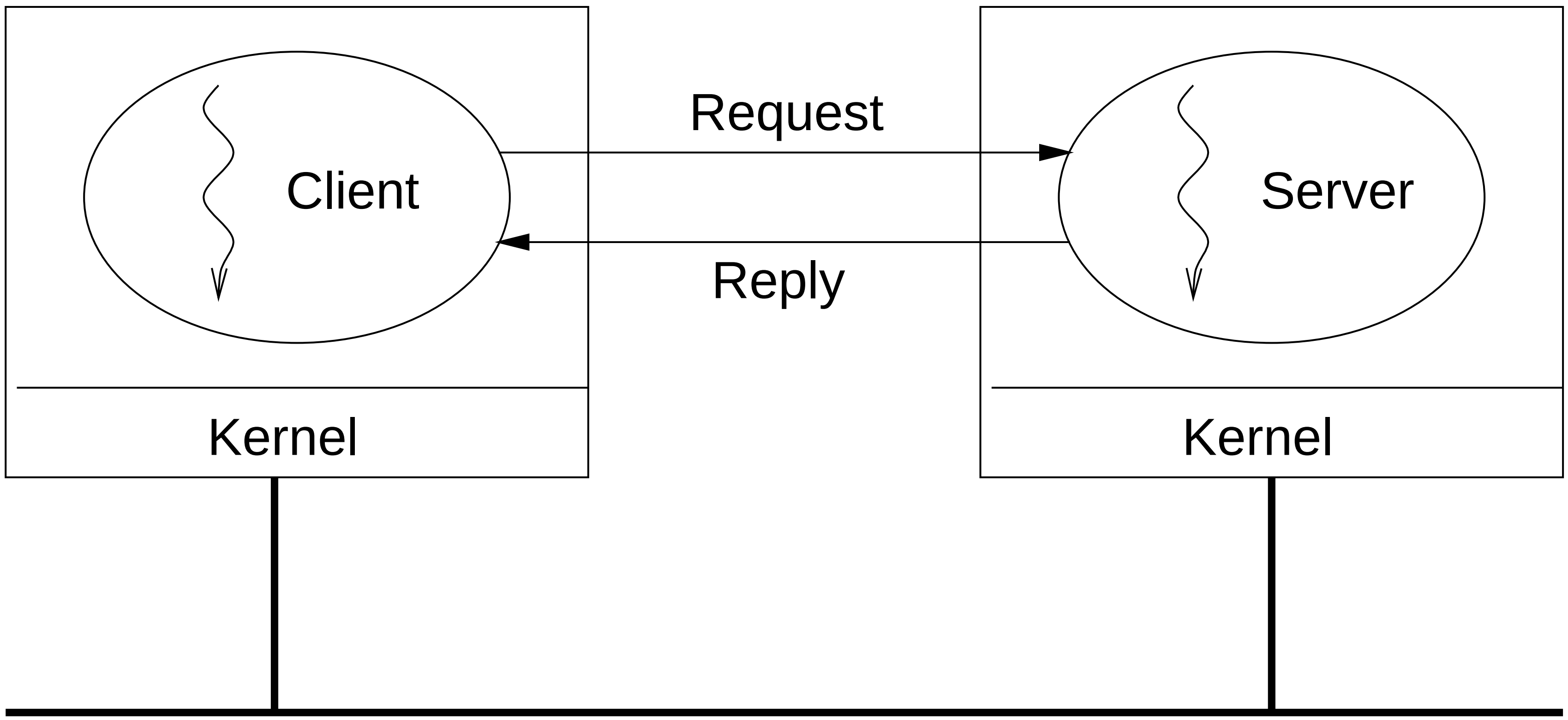
- ➜ processing capacity, load balancing
- ➜ communication capacity
- ➜ locality

Mapping of services to servers:

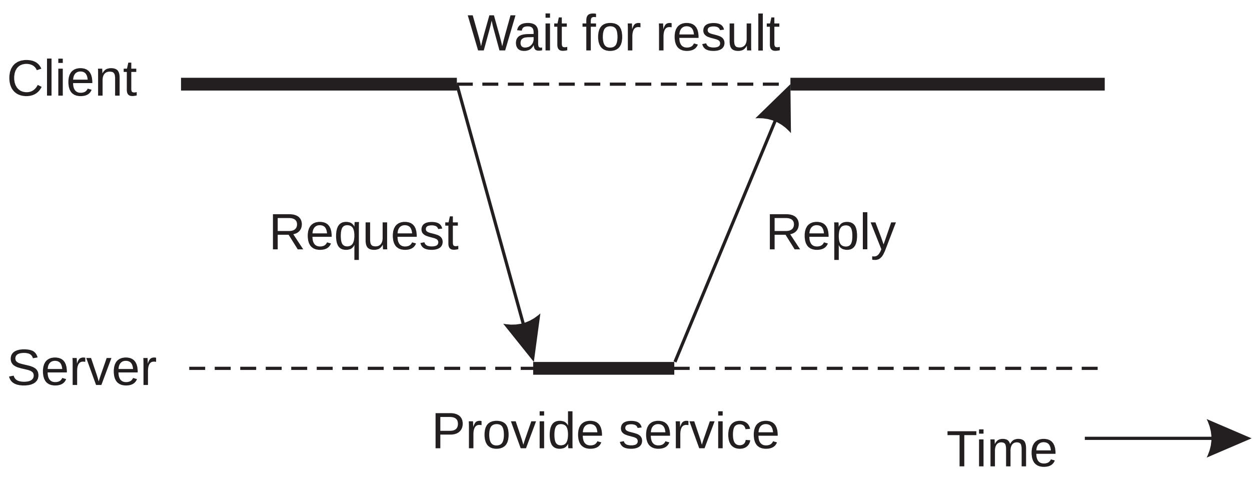
- ➜ Partitioning
- ➜ Replication
- ➜ Caching

Architectural Patterns

CLIENT-SERVER



Client-Server from another perspective:



How scalable is this?

Example client-server code in C:

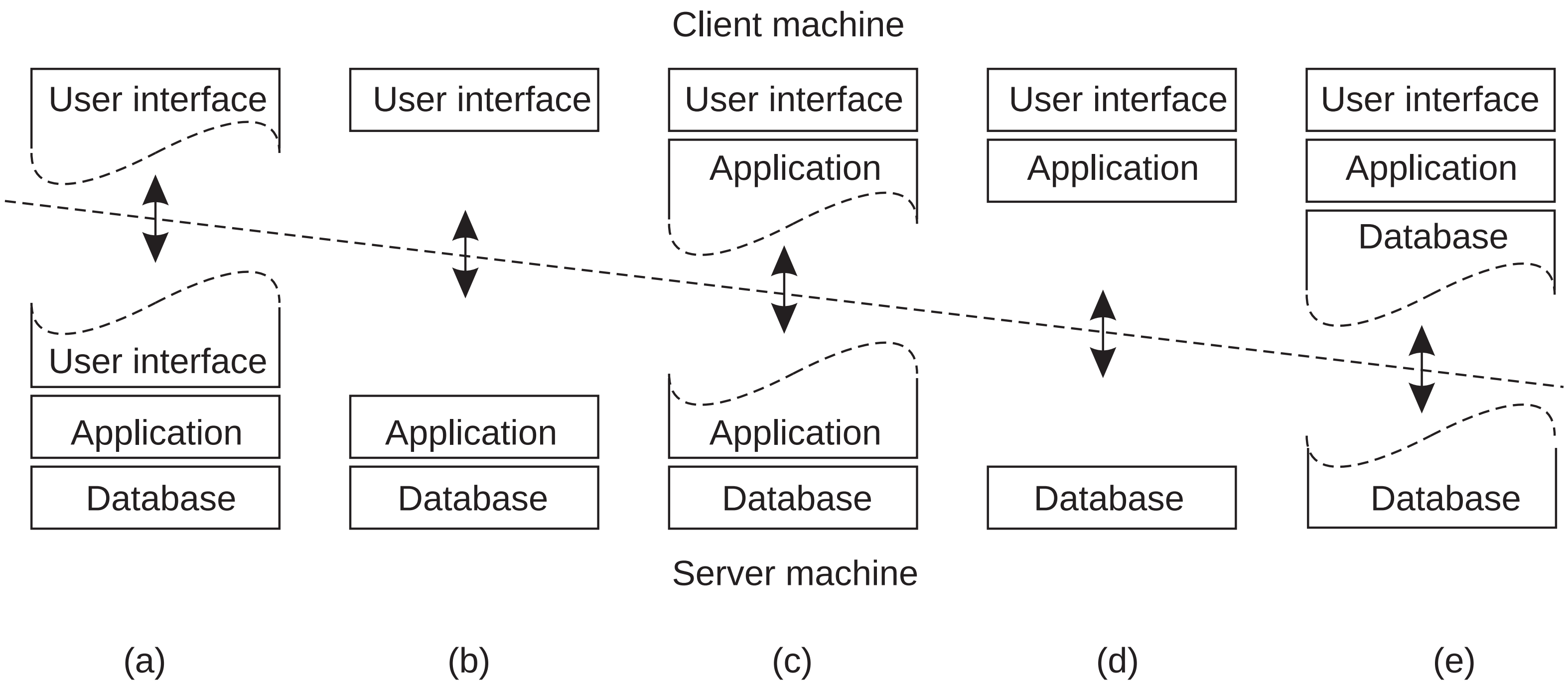
```
client(void) {
  struct sockaddr_in cin;
  char buffer[bufsize];
  int sd;

  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);
}
```

```
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);
}
```

Example client-server code in Erlang:

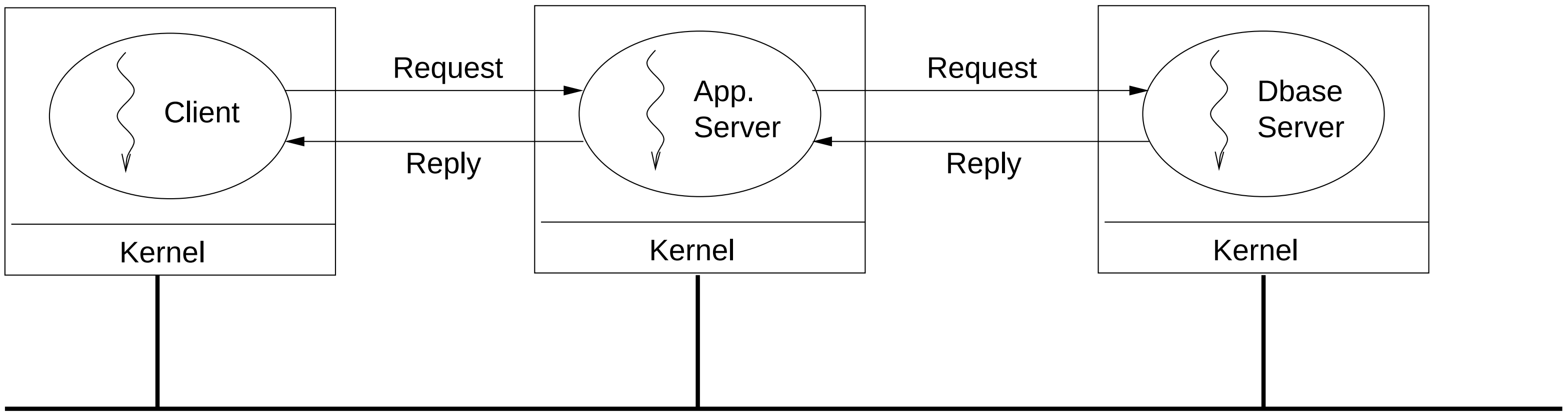
```
% Client code using the increment server
client (Server) ->
  Server ! {self (), 10},
  receive
    {From, Reply} -> io:format ("Result: ~w~n", [Reply])
  end.

% 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).
```

Splitting Functionality:



Which is the best approach?

VERTICAL DISTRIBUTION (MULTI-TIER)



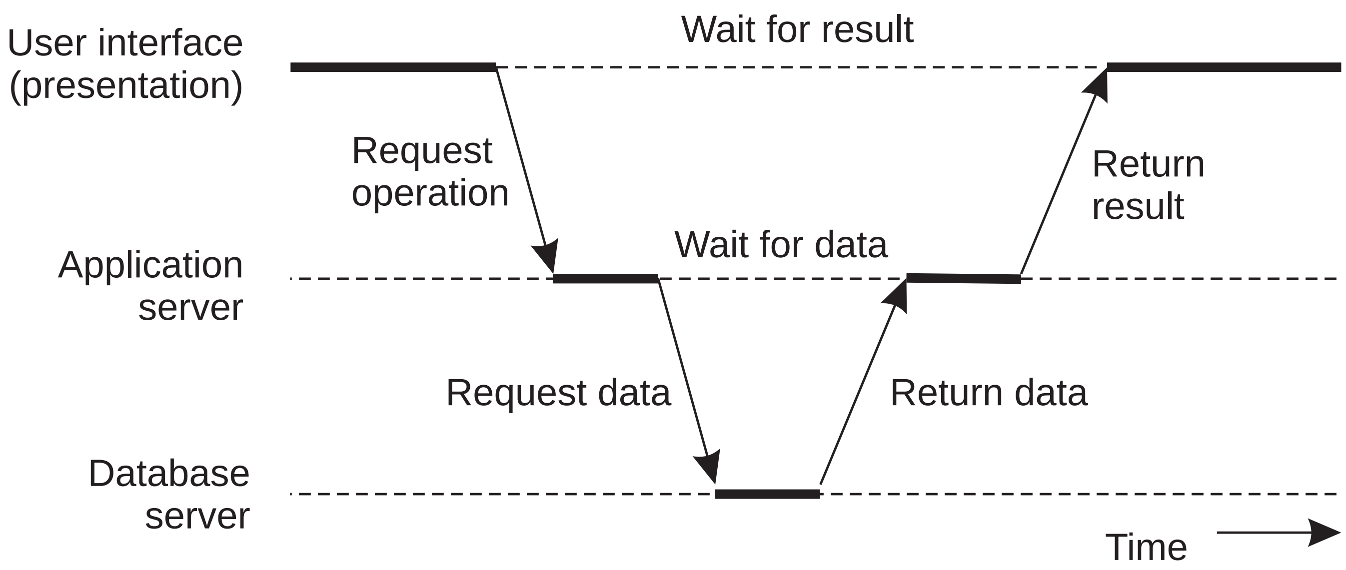
Three 'layers' of functionality:

- User interface
- Processing/Application logic
- Data

➜ Logically different components on different machines

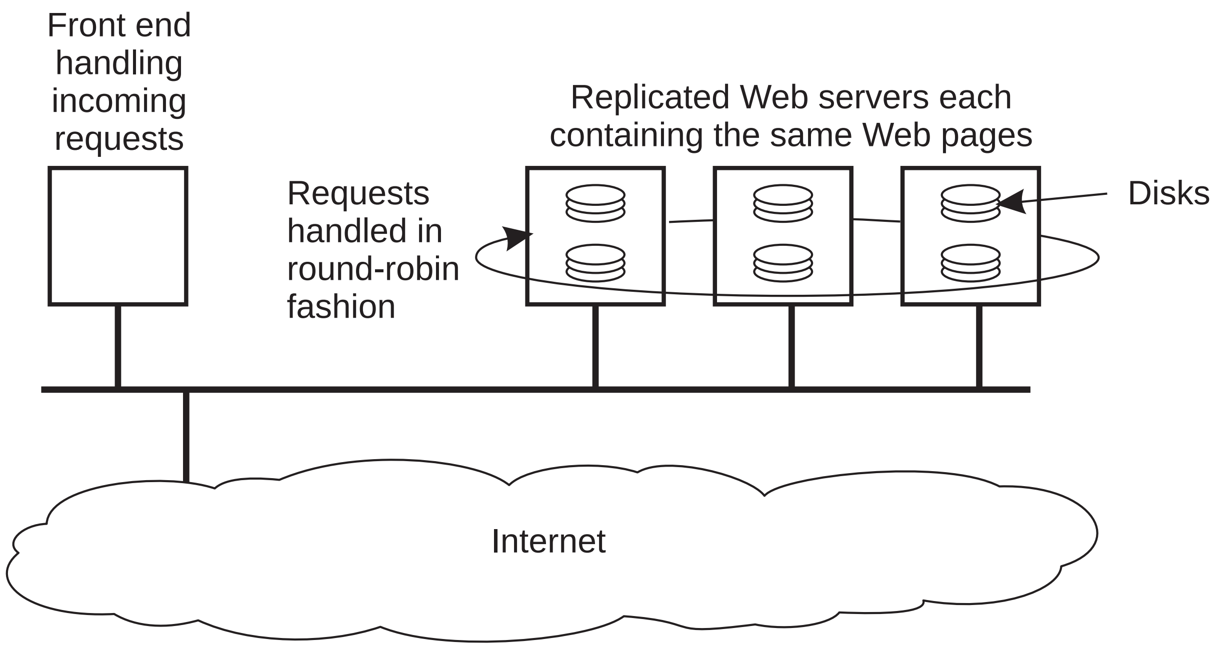
Leads to Service-Oriented architectures (e.g. microservices).

Vertical Distribution from another perspective:



How scalable is this?

Horizontal Distribution



➜ Logically equivalent components replicated on different machines

How scalable is this?

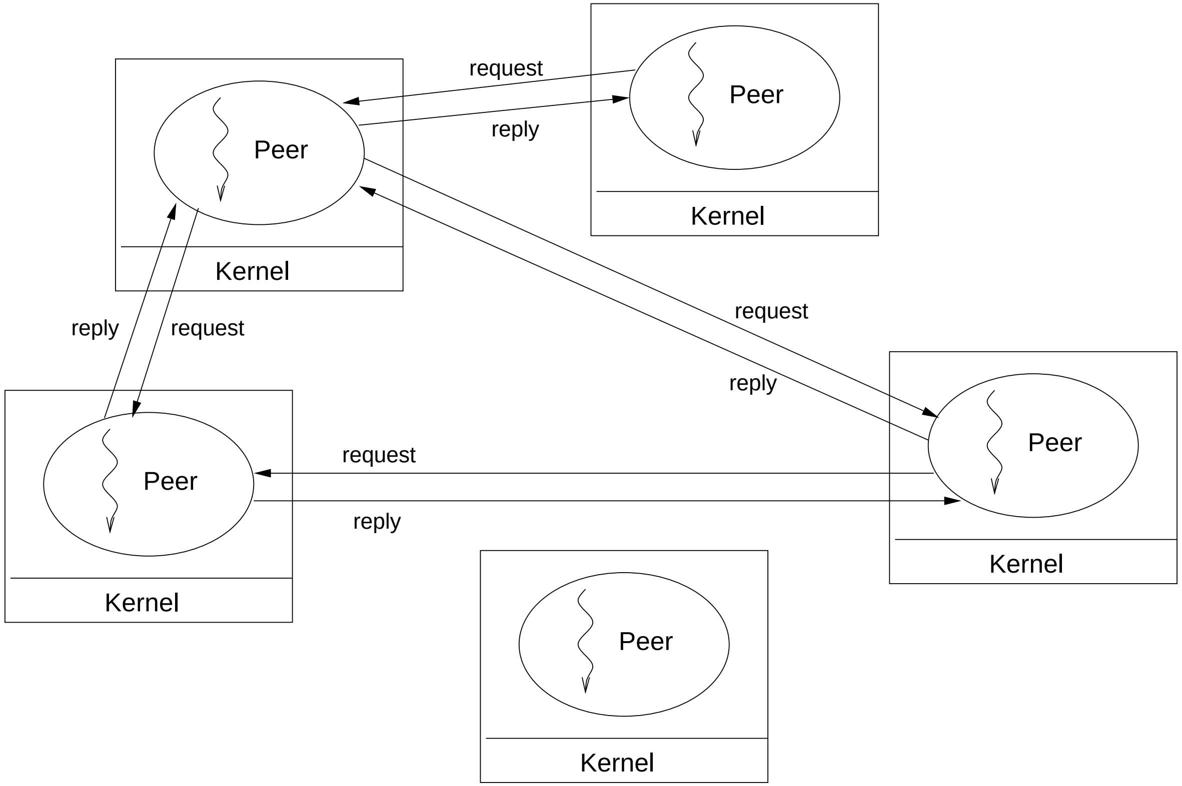
Note: Scaling Up vs Scaling Out?

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

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



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

Why is this special?

Peer to Peer and Overlay Networks

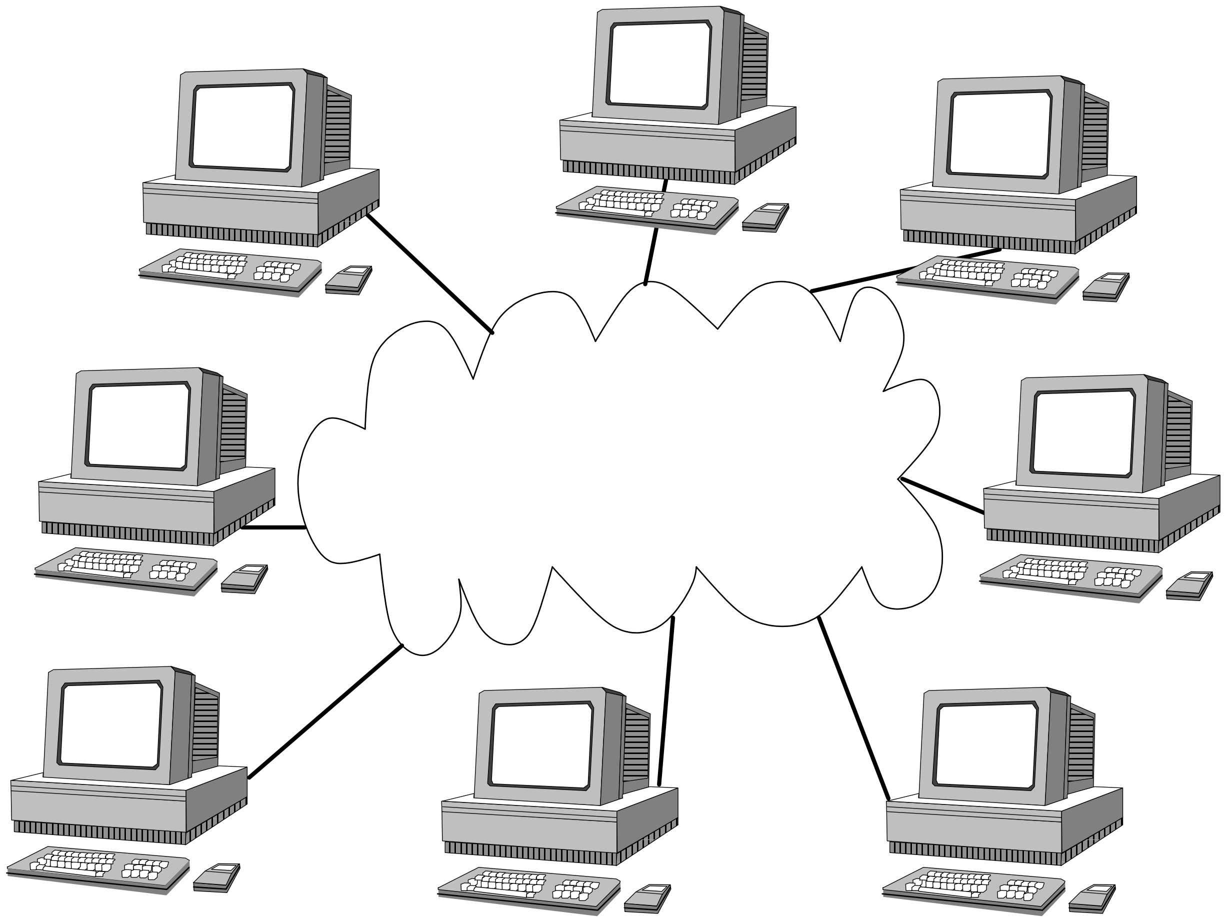
How do peers keep track of all other peers?

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

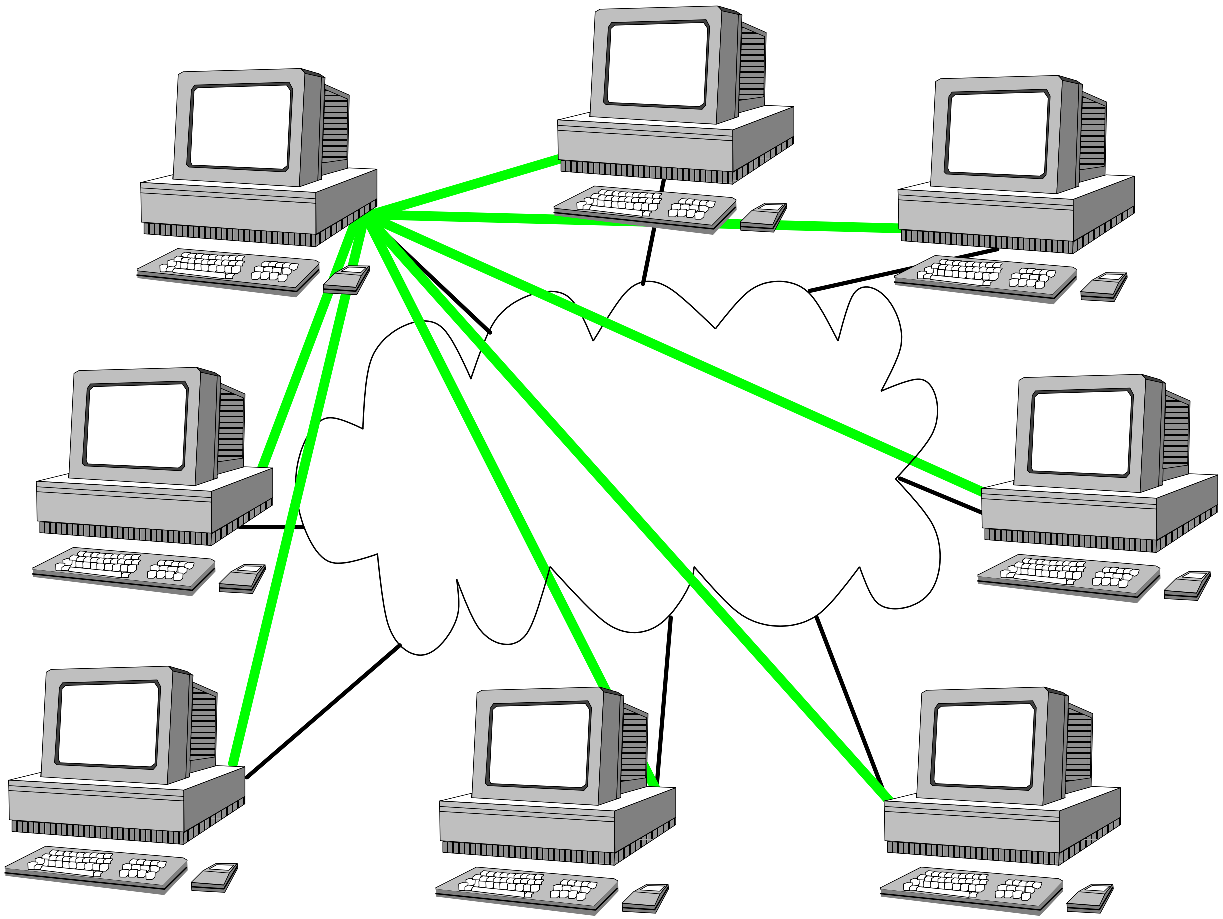
Overlay Network:

- ➜ Application-specific network
- ➜ Addressing
- ➜ Routing
- ➜ Specialised features (e.g., encryption, multicast, etc.)

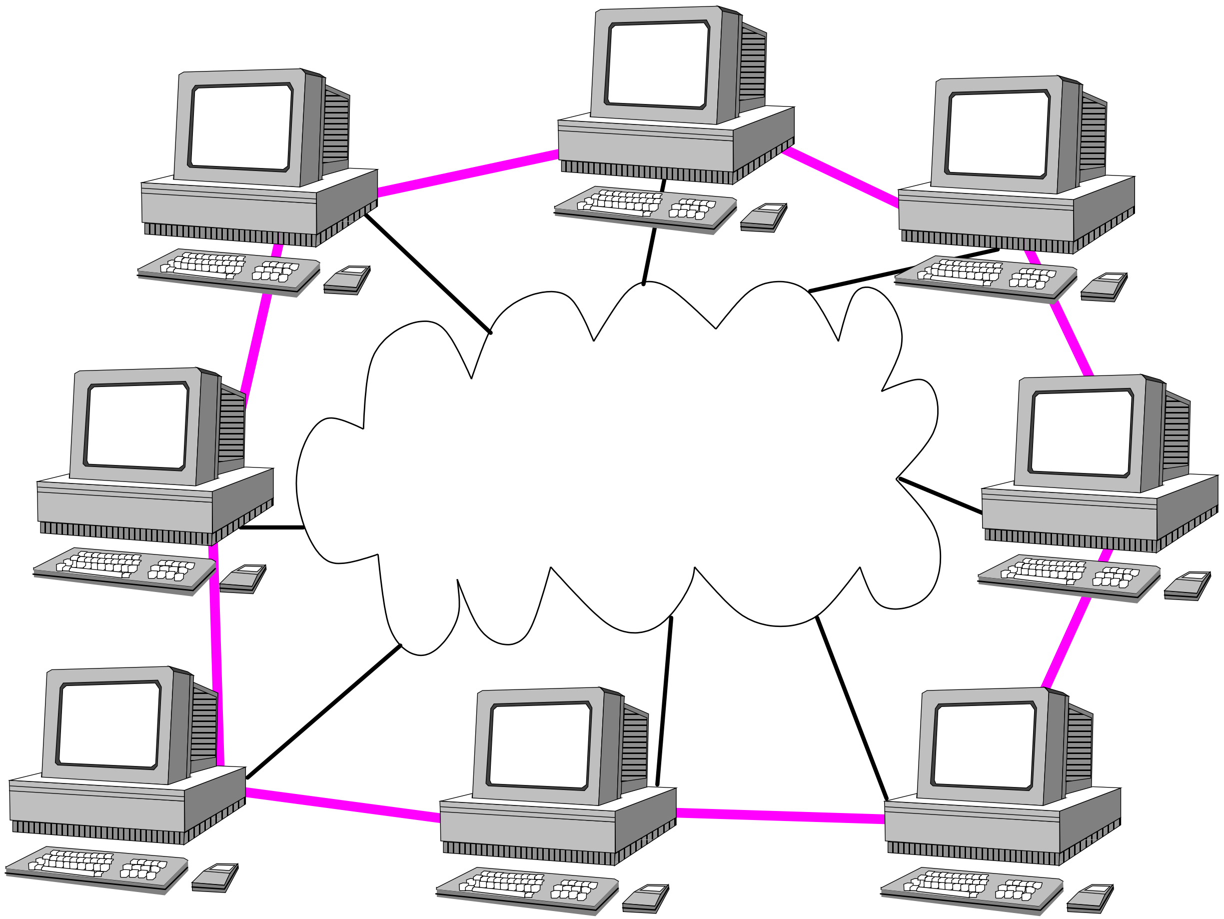
Example:



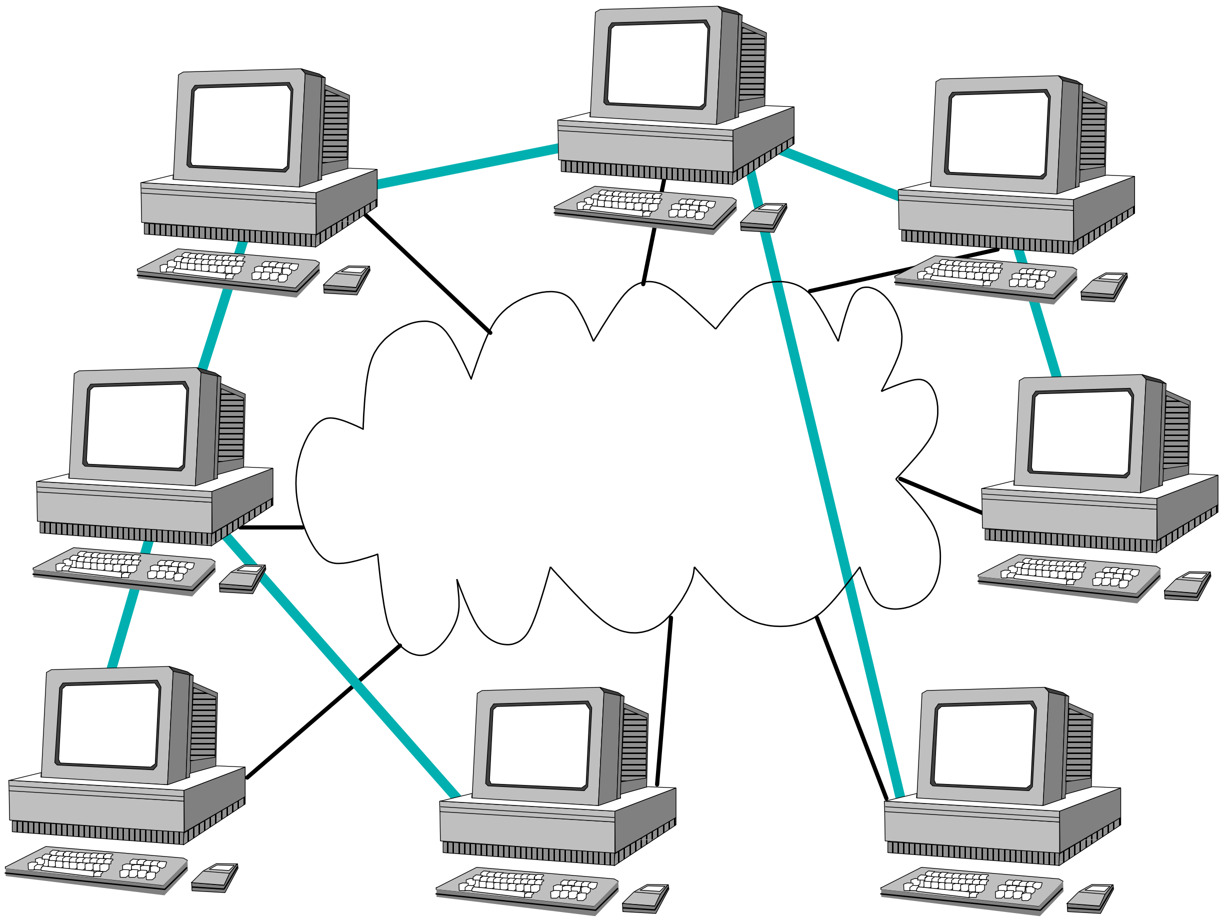
Example:



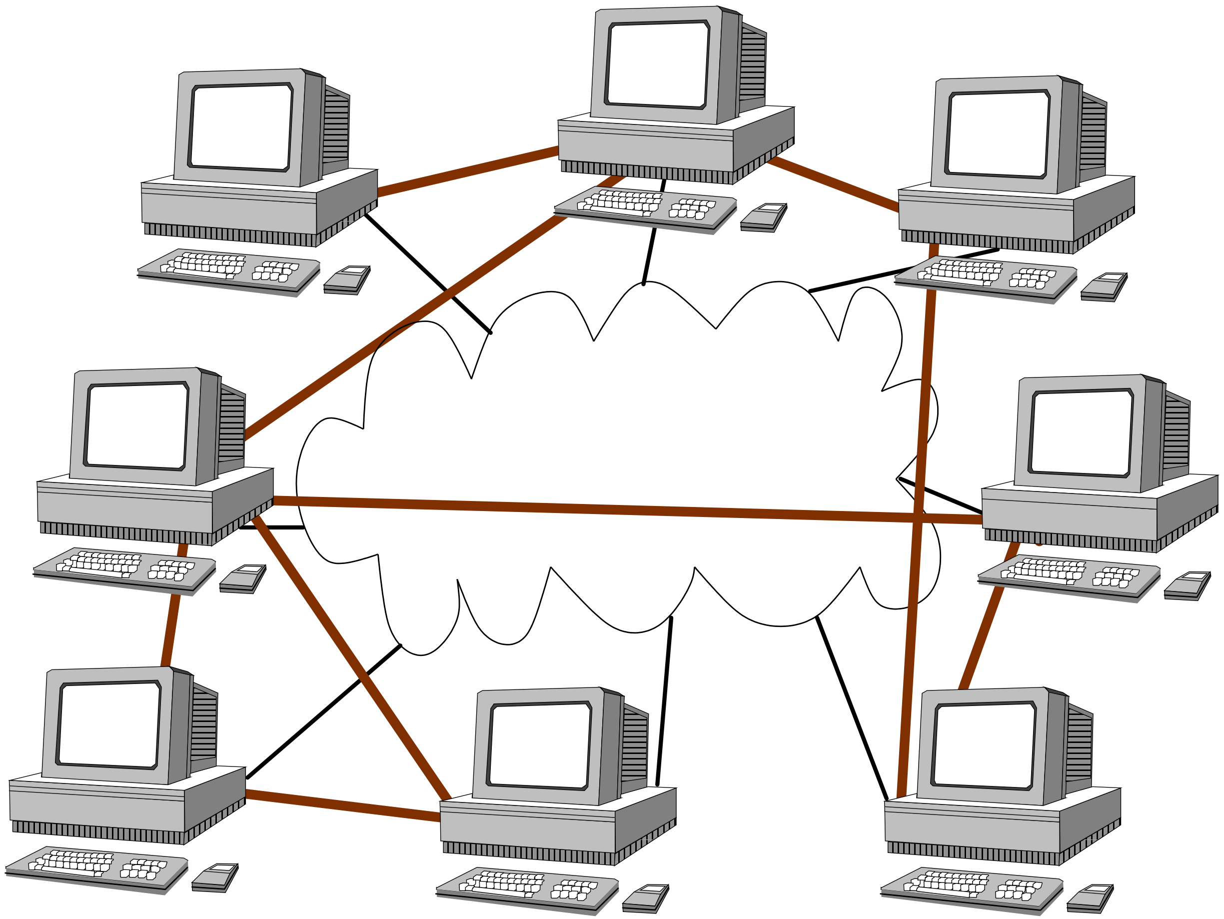
Example:



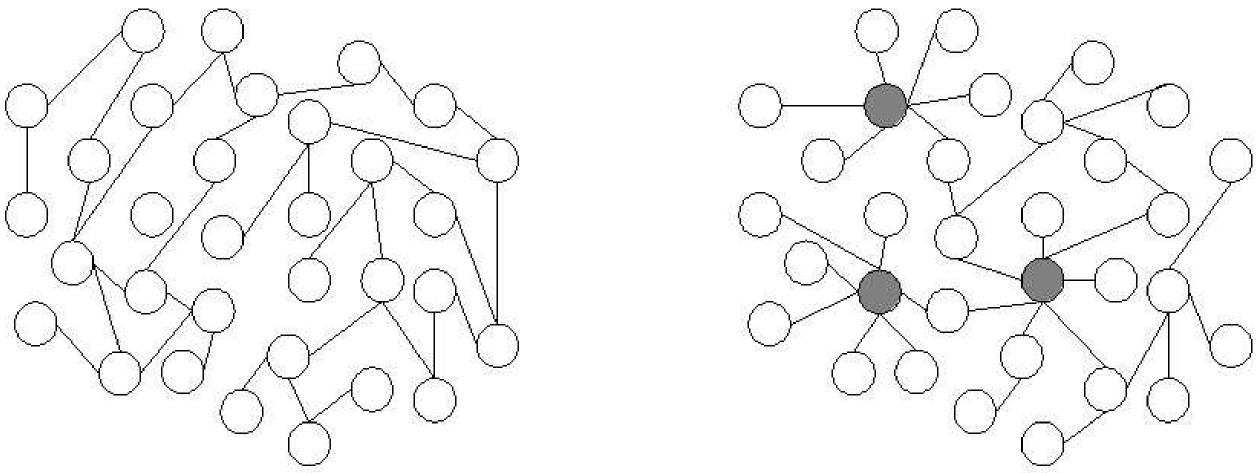
Example:



Example:



Unstructured Overlay



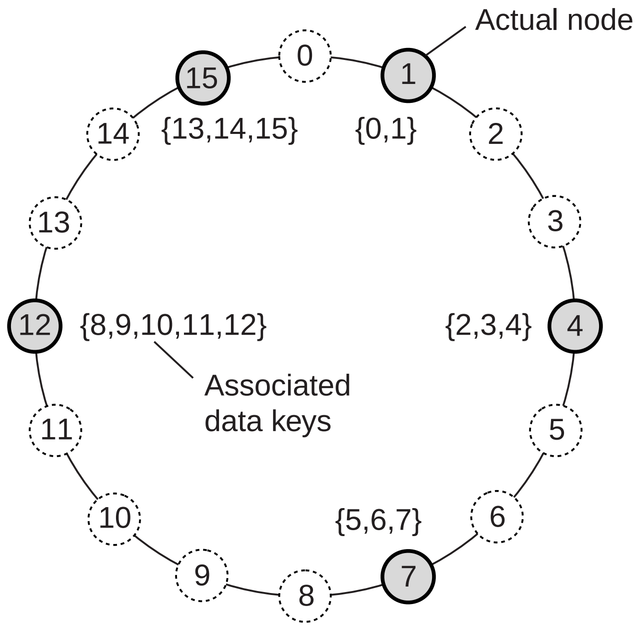
(a) Random network (b) Scale-free network

- ➜ 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?

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?

Hybrid Architectures

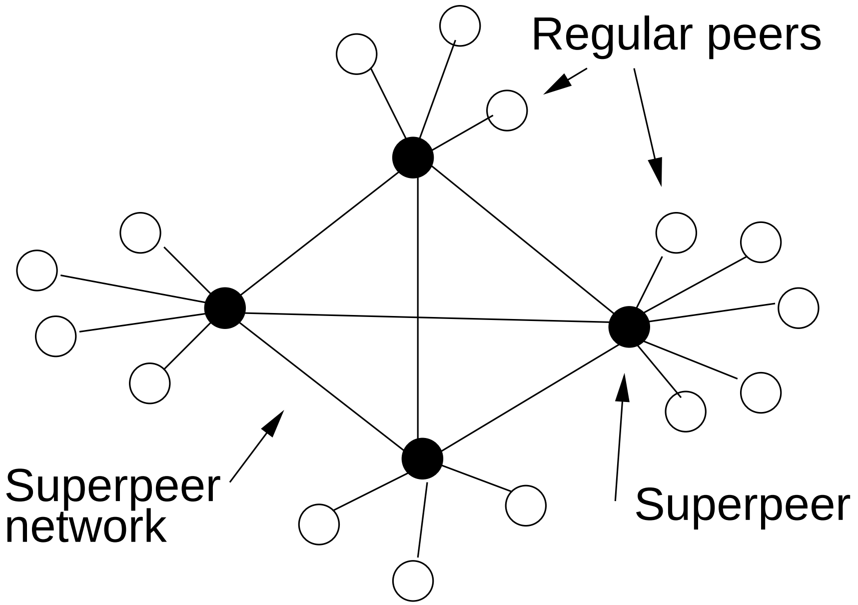
Combination of architectures.

Examples:

- Superpeer networks
- Collaborative distributed systems
- Edge-server 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
- ➜ Example: Skype

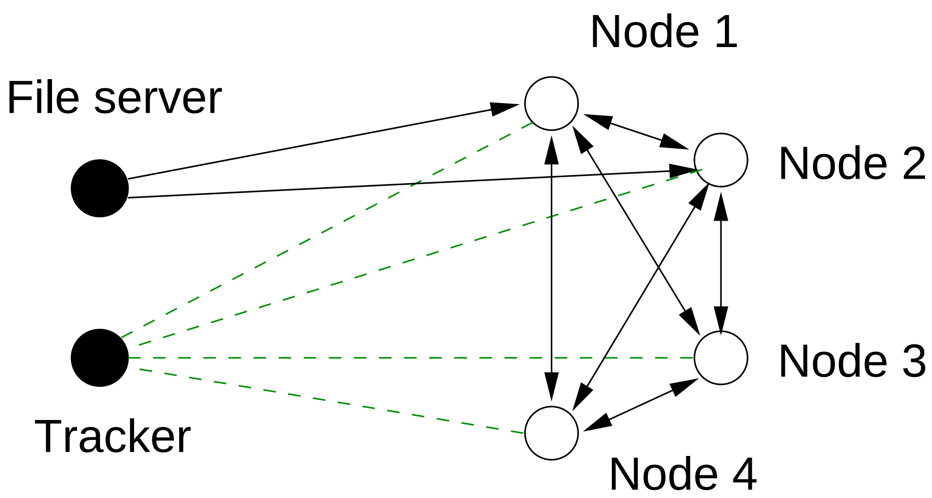


What are potential issues?

Collaborative Distributed Systems:

Example: BitTorrent

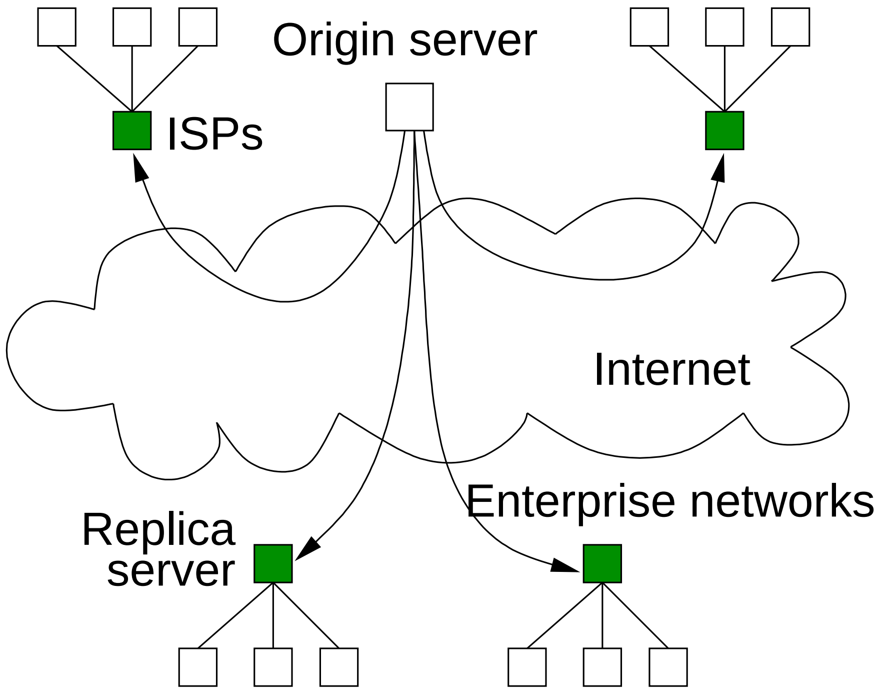
- ➜ Node downloads chunks of file from many other nodes
- ➜ Node provides downloaded chunks to other nodes
- ➜ *Tracker* keeps track of active nodes that have chunks of file
- ➜ Enforce collaboration by penalising selfish nodes



What problems does Bit Torrent face?

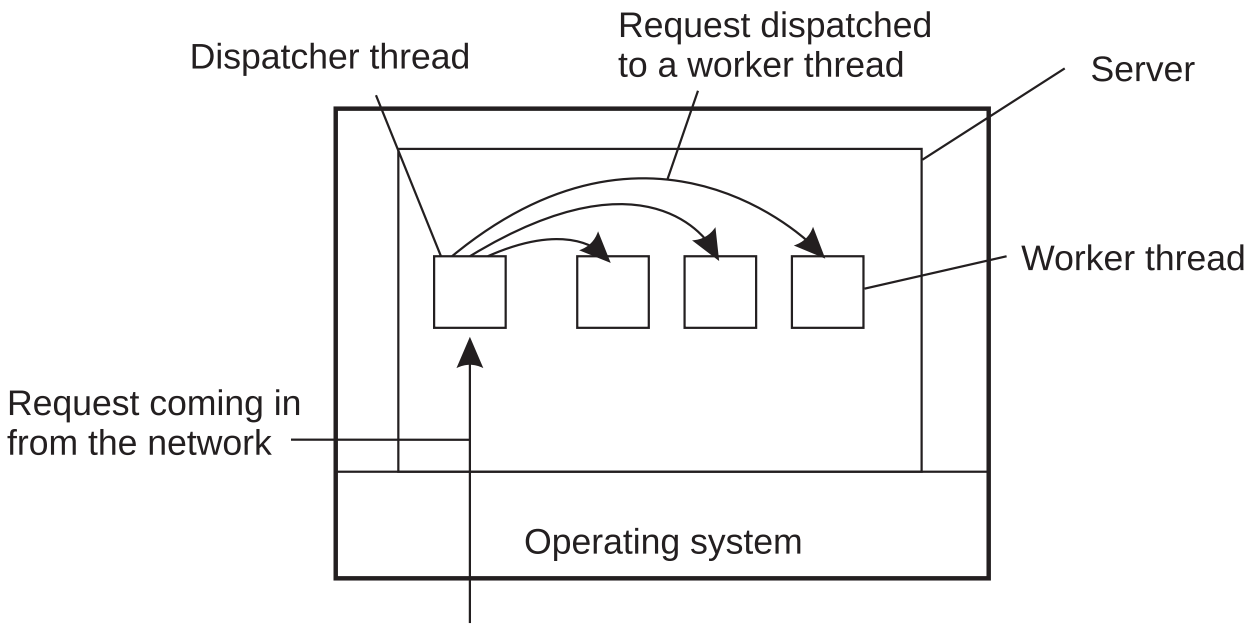
Edge-Server Networks:

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



What are the challenges?

SERVER DESIGN



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:

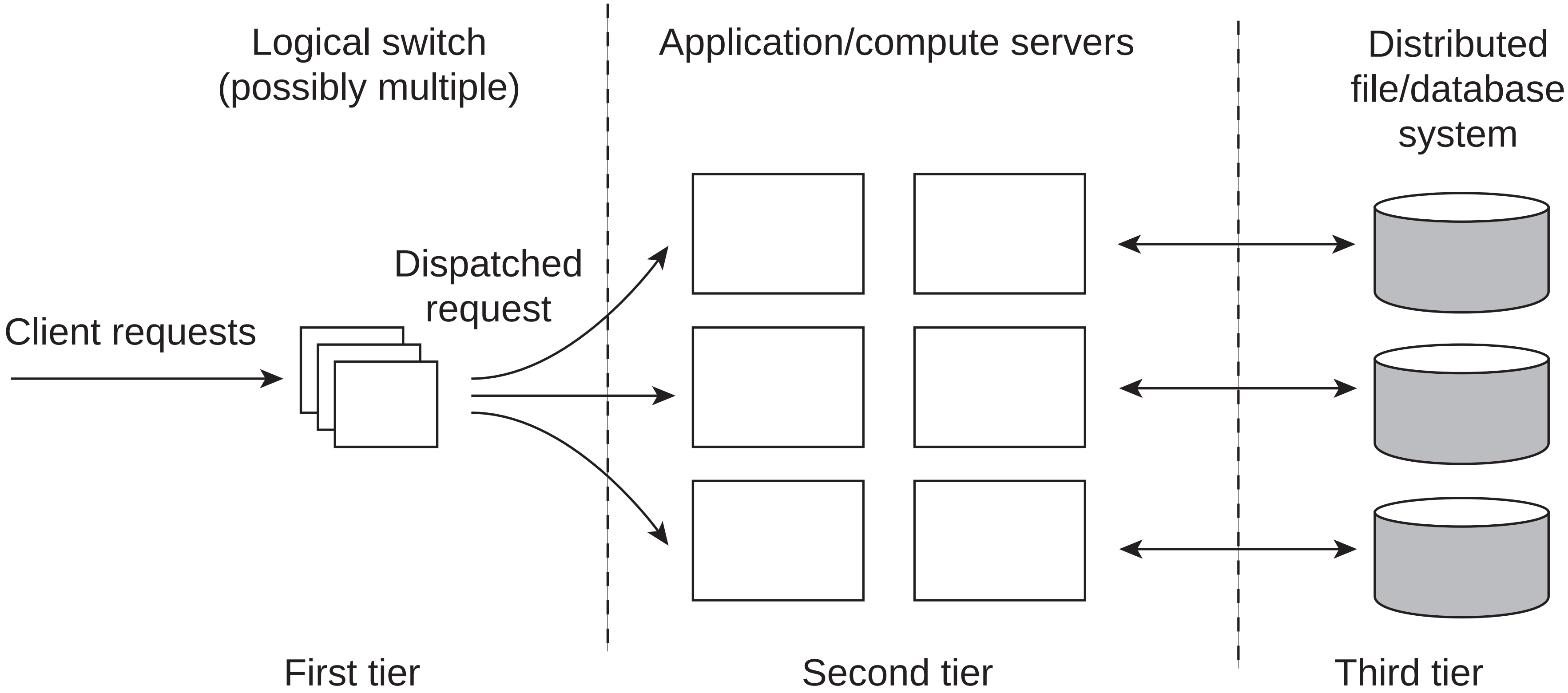
- ➜ Keeps persistent information about clients
- ☑ Improved performance
- ☒ Expensive crash recovery
- ☒ Must track clients

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

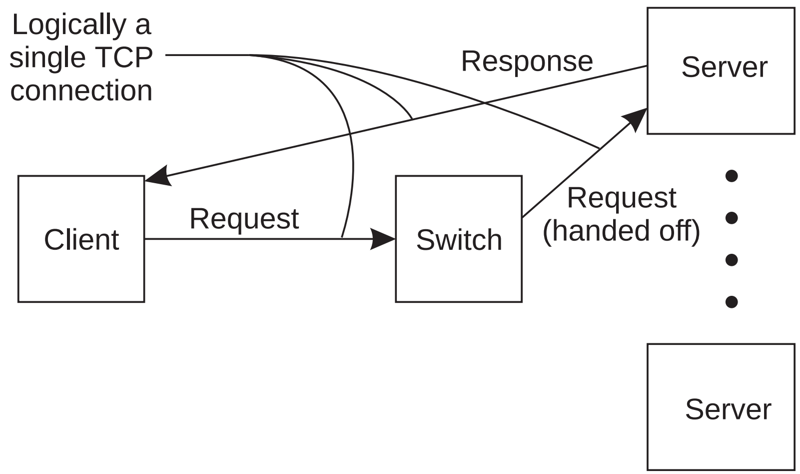
Note: Session state vs. Permanent state

CLUSTERED SERVERS



Request Switching

Transport layer switch:



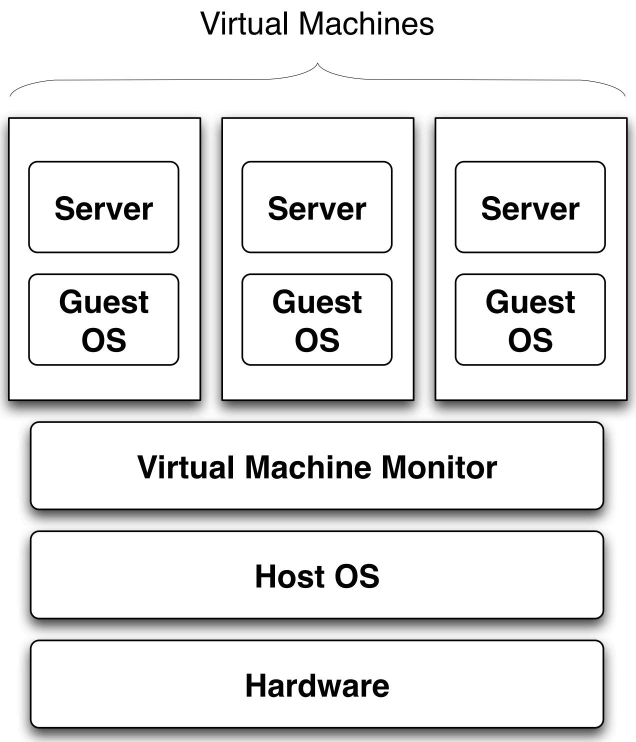
DNS-based:

- ➜ Round-robin DNS

Application layer switch:

- ➜ Analyse requests
- ➜ Forward to appropriate server

VIRTUALISATION



What are the benefits?

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 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?

COMMUNICATION

Why Communication?

Cooperating processes need to communicate.

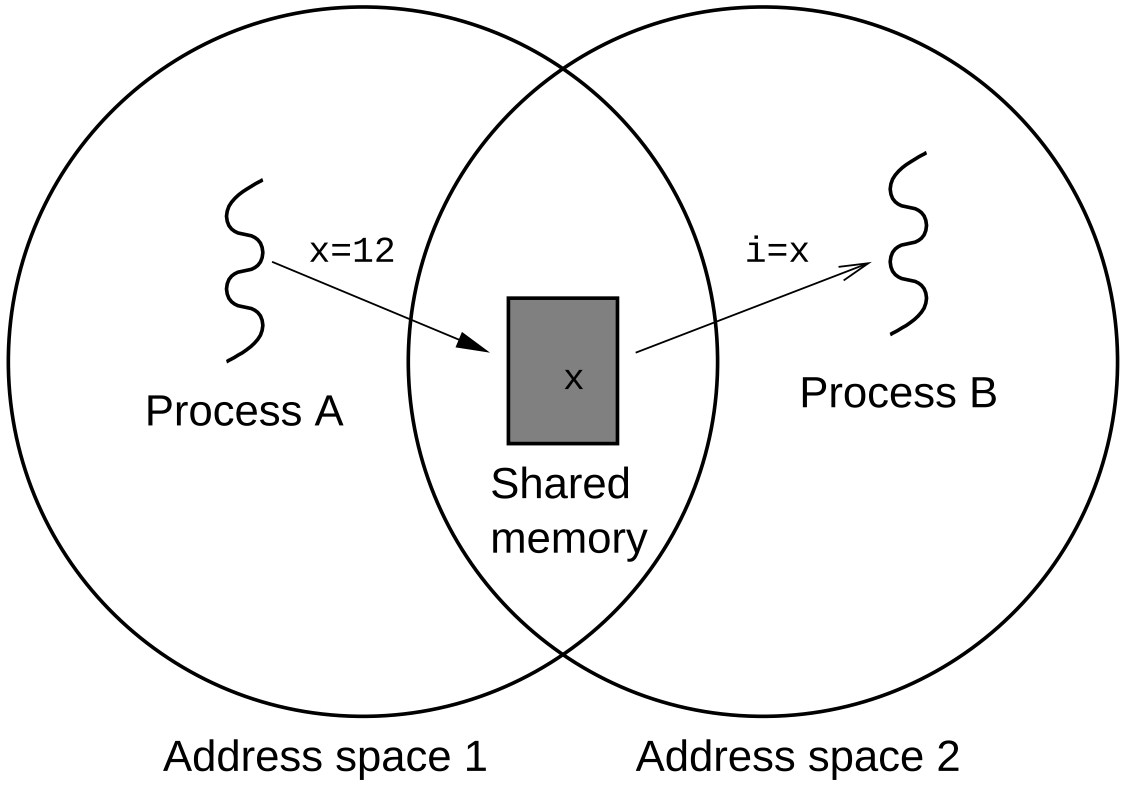
- ➜ For synchronisation and control
- ➜ To share data

In a Non-Distributed System:

Two approaches to communication:

- ➜ Shared memory

Shared Memory:

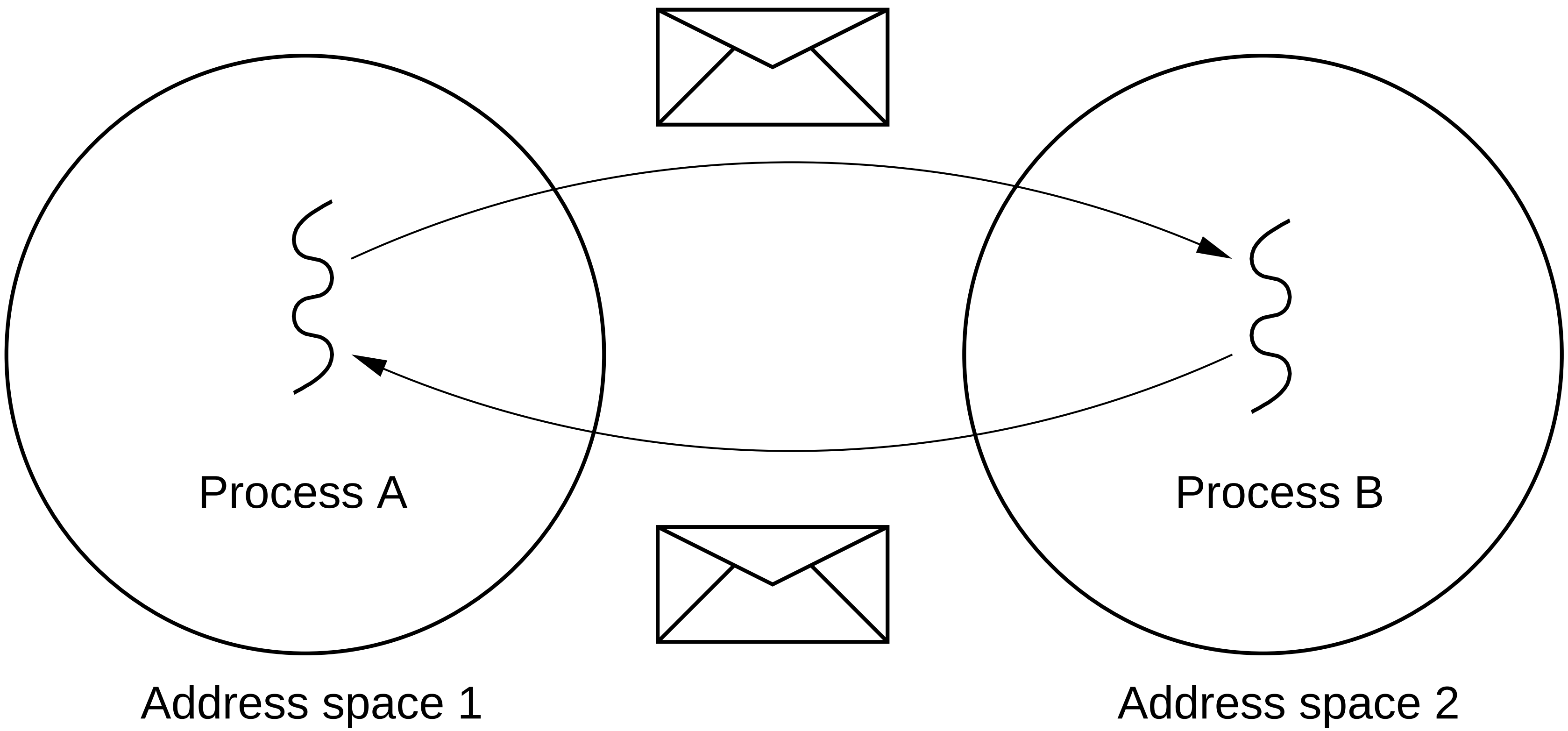


In a Non-Distributed System:

Two approaches to communication:

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

Message Passing:



In a Non-Distributed System:

Two approaches to communication:

➜ Shared memory

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

➜ Message passing

- OS's IPC mechanisms

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

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

Coupling

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

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)

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?

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?

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

Direct-Addressing vs Indirect-Addressing Communication:

Direct-Addressing

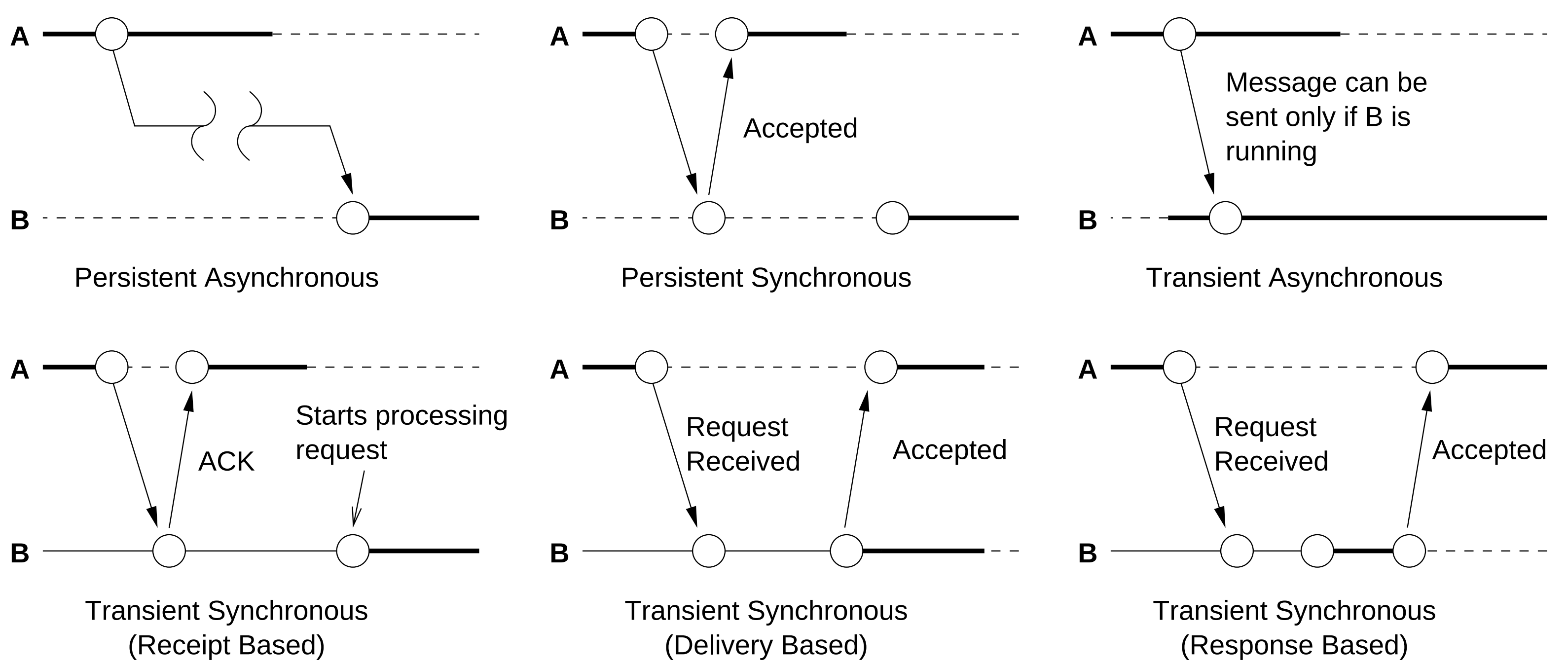
- ➜ Message sent directly to receiver
- ➜ Example: HTTP request

Indirect-Addressing

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

Coupling?

Combinations:



Examples?

Communication Abstractions

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

Provided by higher level APIs

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:

- ➜ Asynchronous and Synchronous communication
- ➜ Transient communication

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

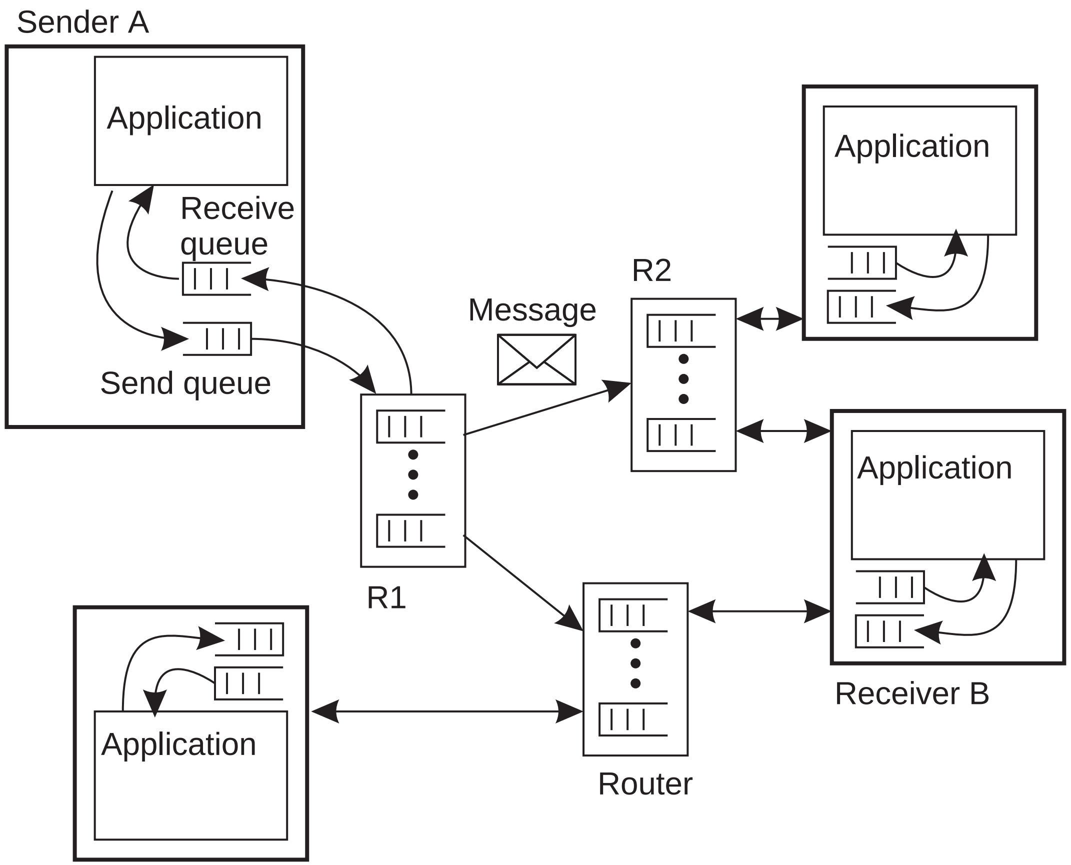
- ➜ Persistent communication (Message queues)
- ➜ Hides implementation details
- ➜ Marshalling

EXAMPLE: MESSAGE PASSING INTERFACE (MPI)

- ➜ 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!

EXAMPLE: MESSAGE QUEUING SYSTEMS



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

Examples:

- ➜ IBM MQSeries, Java Message Service, Amazon SQS, Advanced Message Queuing Protocol, MQTT, STOMP

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

Request-Reply Communication

Request:

- ➜ a service
- ➜ data

Reply:

- ➜ result of executing service
- ➜ data

Requirement:

- ➜ Message formatting
- ➜ Protocol

EXAMPLE: 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

Confusing local and remote operations can be dangerous, why?

Remember Erlang client/server example?:

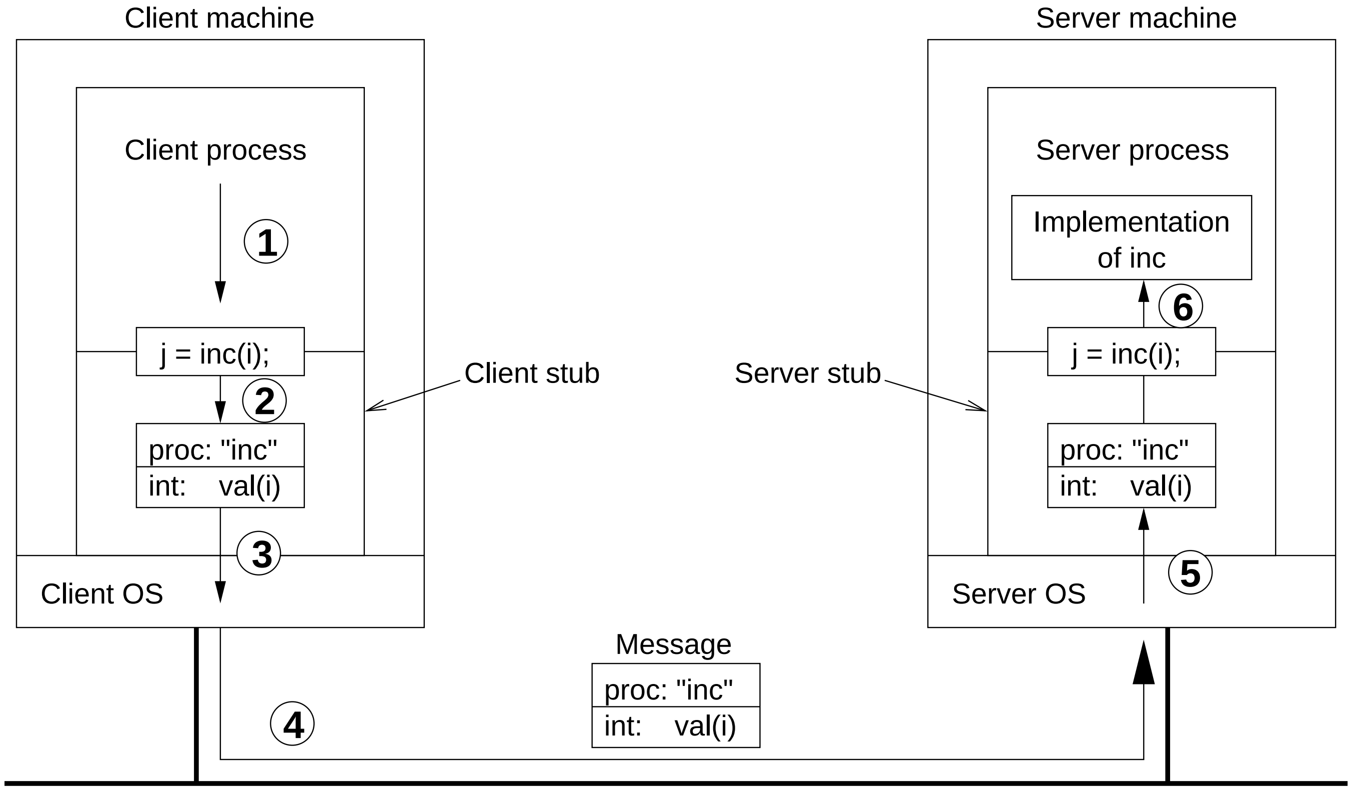
```
% Client code using the increment server
client (Server) ->
  Server ! {self (), 10},
  receive
    {From, Reply} -> io:format ("Result: ~w~n", [Reply])
  end.

% 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).
```

This is what it's like in RPC:

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

% Server code
inc (Value) -> Value + 1.
```

Where is the communication?

RPC Implementation:



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]).

% 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.

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

Sun RPC Example:

Run example code from website

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 */
```

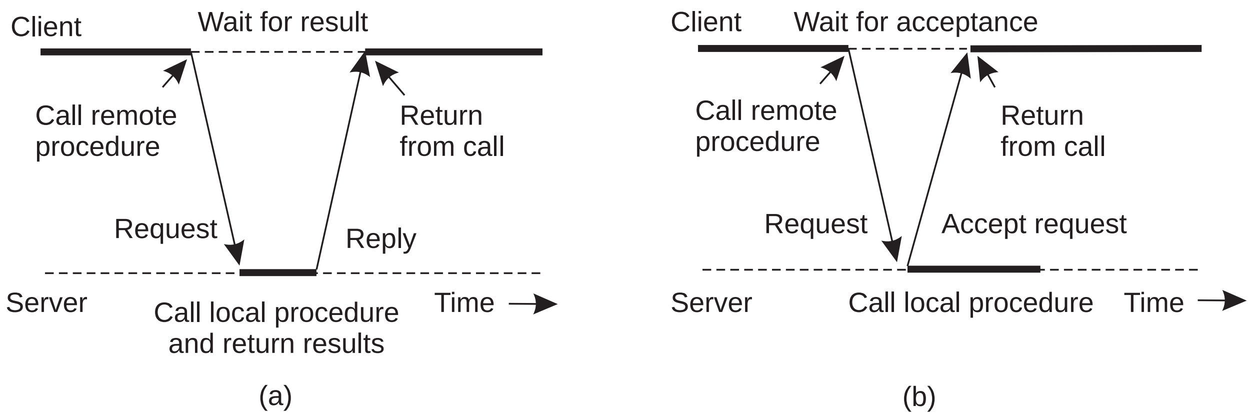
Sun RPC - client code:

```
#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 */
}
```

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 */
}
```

ONE-WAY (ASYNCHRONOUS) RPC



- ➜ When no reply is required
- ➜ When reply isn't needed immediately (2 asynchronous RPCs - deferred synchronous RPC)

Remote Method Invocation (RMI)

Like RPC, but transition from the server metaphor to the object metaphor.

Why is this important?

- ➜ 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?

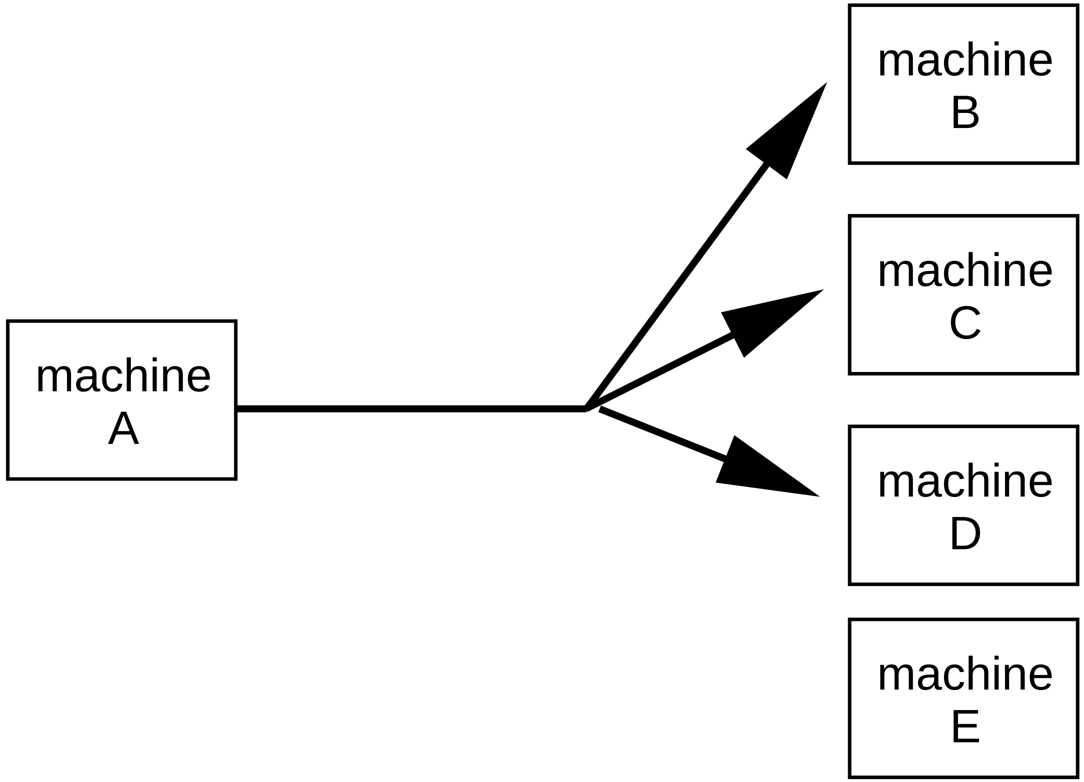
- ➜ Remote operations can fail in different ways
- ➜ Remote operations can have arbitrary latency
- ➜ Remote operations have a different memory access model
- ➜ Remote operations can involve concurrency in subtle ways

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



➜ Sender performs a single `send()`

What are the difficulties with group communication?

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

Issues:

- ➜ Reliability
- ➜ Ordering

Example:

- ➜ IP multicast
- ➜ Flooding

Example: Gossip-Based Communication

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

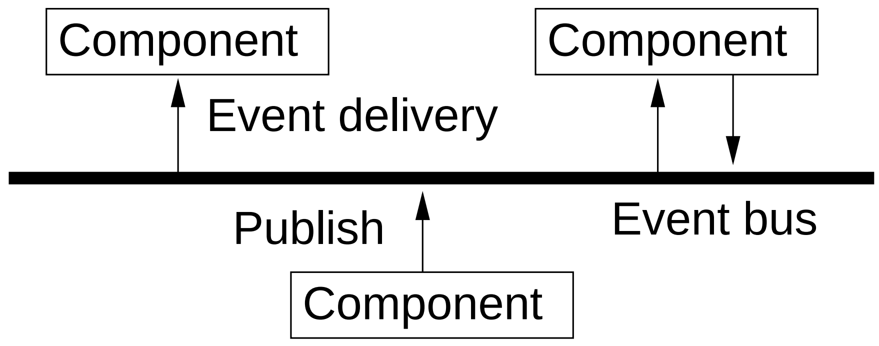
Variant: *rumour spreading*, or *gossiping*.

- 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.

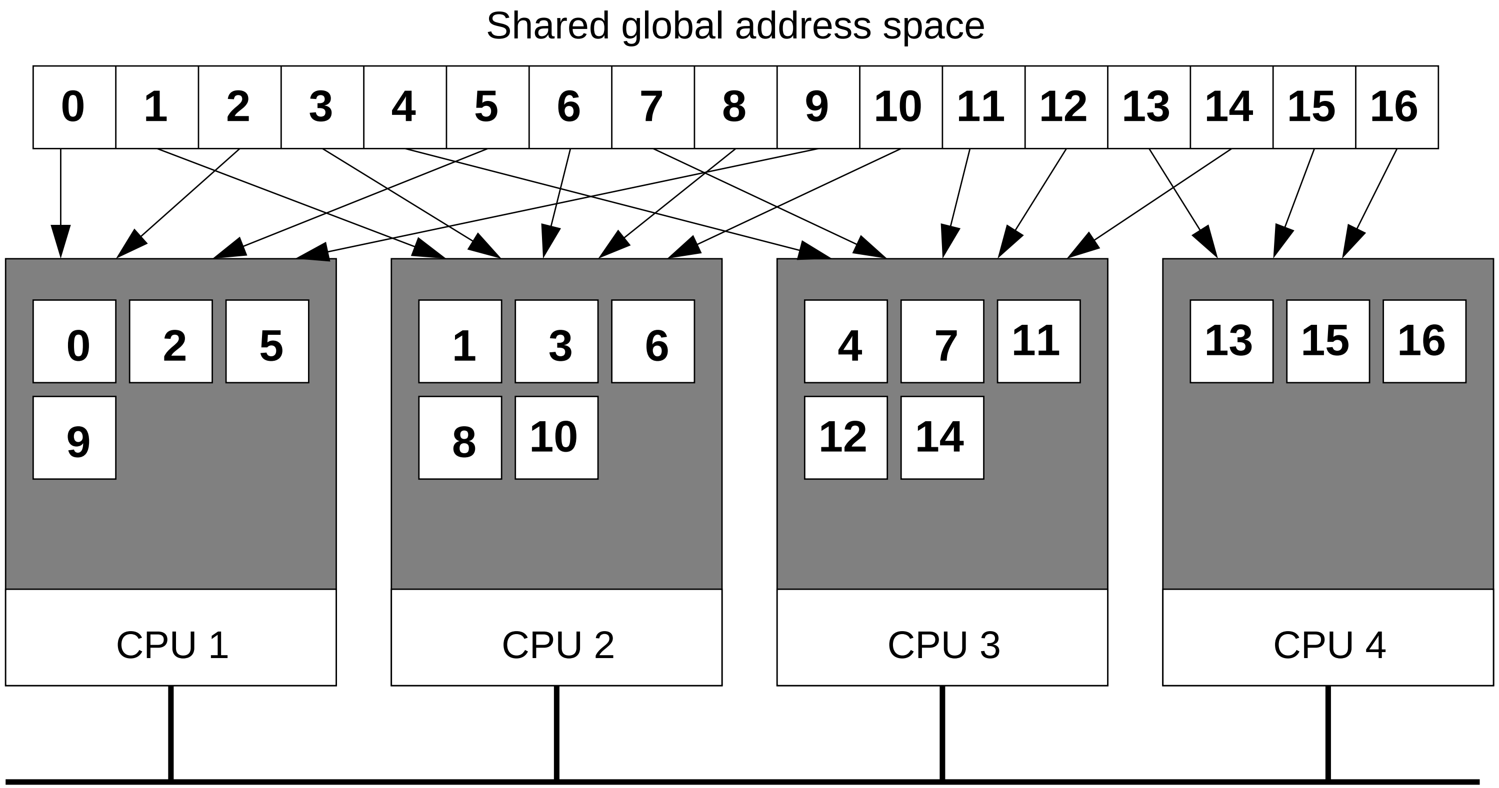
Event-Based Communication

- 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: space, time
- Example: OMG Data Distribution Service (DDS), JMS, Tibco



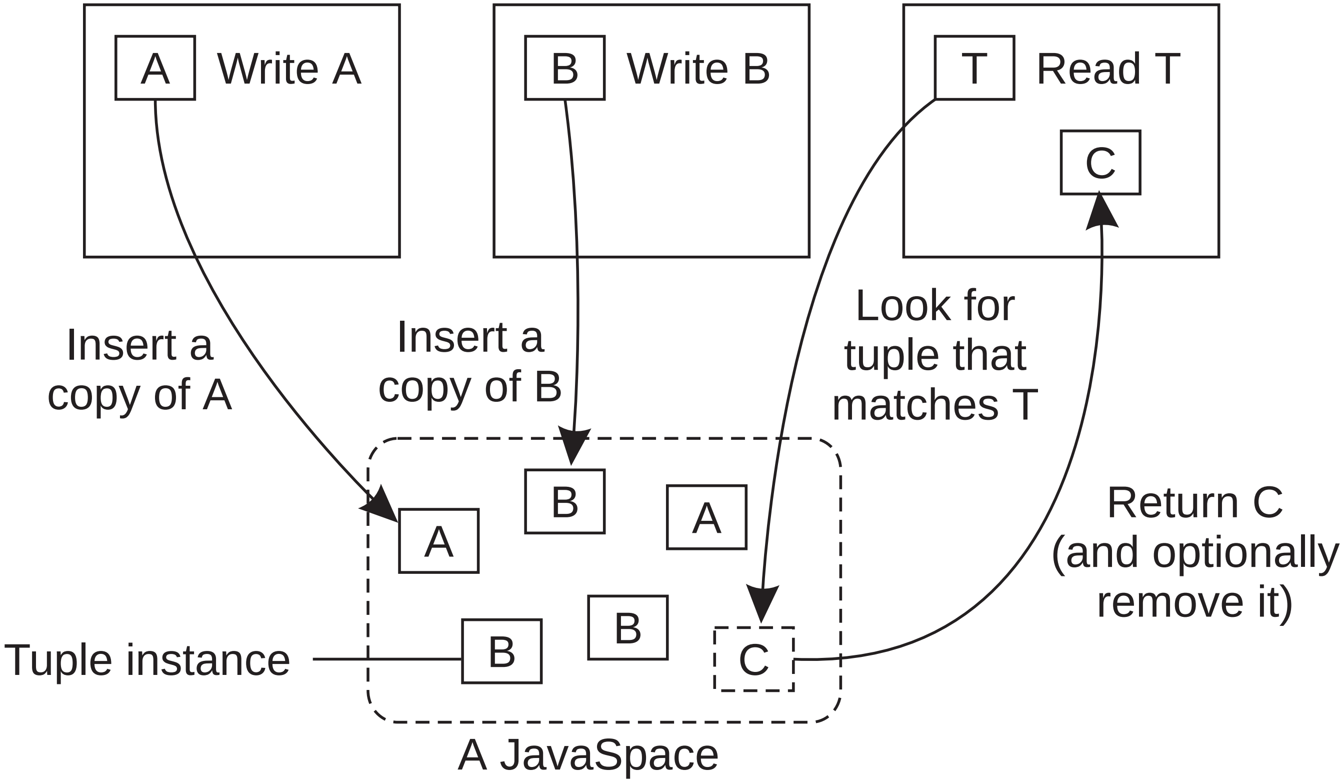
Shared Space Communication

Example: Distributed Shared Memory:



Coupling?

Example: Tuple Space:



Coupling?

Reading List

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

HOMEWORK

RPC:

➜ Do Exercise *Client server exercise (Erlang)* Part B

Synchronous vs Asynchronous:

➜ Explain how you can implement synchronous communication using only asynchronous communication primitives.

➜ How about the opposite?

Hacker's Edition: Client-Server vs Ring:

➜ Do Exercise *Client-Server vs. Ring (Erlang)*