Slide 1

DISTRIBUTED SYSTEMS (COMP9243)

Lecture 3: System Architecture

① System Architectures
- ➜ Client-server (and multi-tier)
- ➜ Peer to peer
- ➜ Hybrid architectures

② Processes & Server Architecture

Slide 2

ARCHITECTURE

Slide 3

BUILDING A DISTRIBUTED SYSTEM

Two questions:

① Where to place the hardware?

② Where to place the software?

Slide 4

System Architecture:
- ➜ identifying hardware and software elements
- ➜ placement of machines
- ➜ placement of software on machines
- ➜ communication patterns

Where to place?:
- ➜ processing capacity, load balancing
- ➜ communication capacity
- ➜ locality

Mapping of services to servers:
- ➜ Partitioning
- ➜ Replication
- ➜ Caching

Slide 5

Architecture Issues

Choosing the right architecture involves:

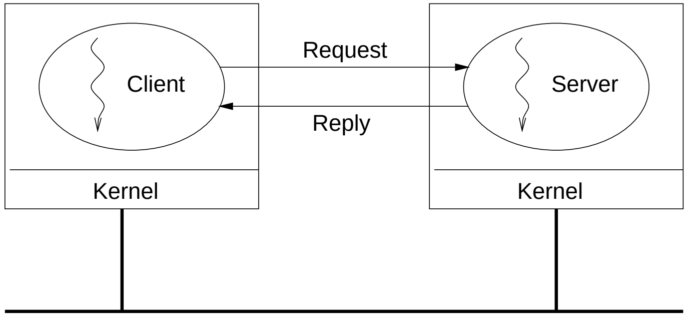
- Splitting of functionality
- Structuring the application
- Reducing complexity

Slide 6

Architectural Patterns

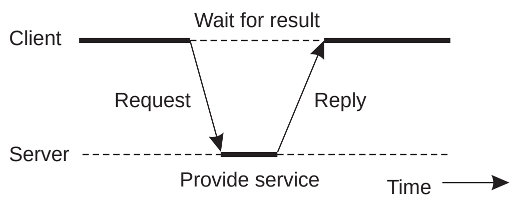
Slide 7

Client-Server



Slide 8

Client-Server from another perspective:



How scalable is this?

Slide 9

Example client-server code in C:

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

  ... // set server address in cin

  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 10

```
server(void) {
  struct sockaddr_in cin, sin;
  int sd, sd_client;
  ... // set server address in sin
  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 11

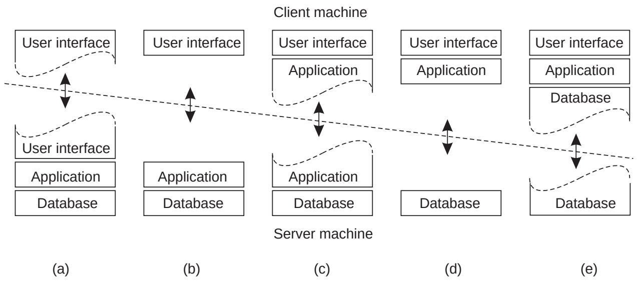
Example client-server code in Erlang:

```
% 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 12

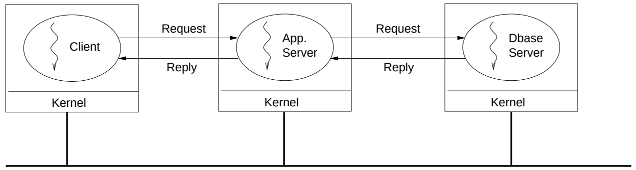
Splitting Functionality:



Which is the best approach?

Slide 13

VERTICAL DISTRIBUTION (MULTI-TIER)



Three 'layers' of functionality:

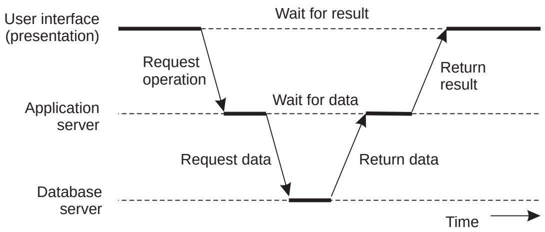
- User interface
- Processing/Application logic
- Data

➜ Logically different components on different machines

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

Slide 14

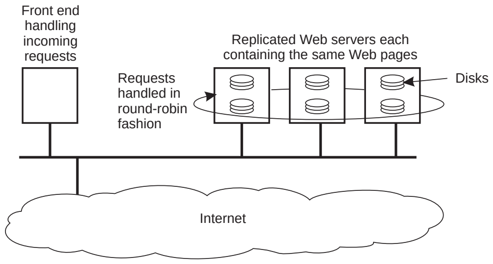
Vertical Distribution from another perspective:



How scalable is this?

Slide 15

HORIZONTAL DISTRIBUTION



➜ Logically equivalent components replicated on different machines

How scalable is this?

Slide 16

Note: Scaling Up vs Scaling Out?

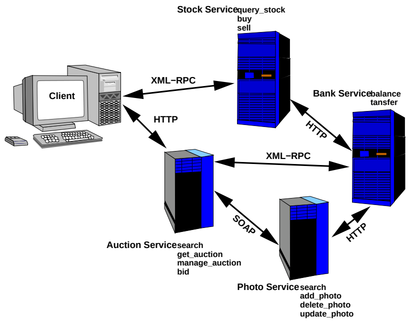
Horizontal and Vertical *Distribution* not the same as Horizontal and Vertical *Scaling*.

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.

Slide 17

Service Oriented Architecture (SOA)



Slide 18

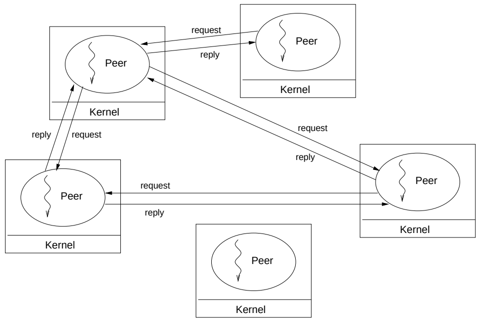
Microservices

'Extreme' vertical distribution

- ➜ split application logic into many (reusable) services
- ➜ services limited in scope: single-purpose, do one thing really well
- ➜ orchestrate execution of services

Slide 19

Peer to Peer



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

Why is this special?

Slide 20

Peer to Peer and Overlay Networks

How do peers keep track of all other peers?

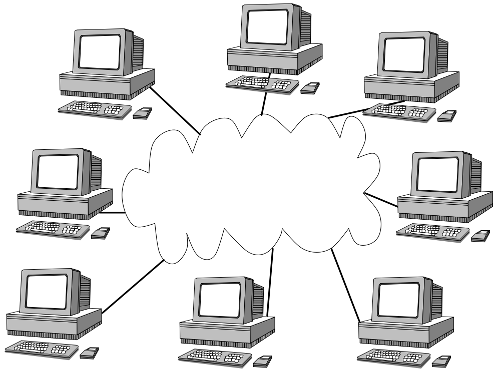
- ➜ static structure: you already know
- ➜ dynamic structure: *Overlay Network*
  - ① structured
  - ② unstructured

Overlay Network:

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

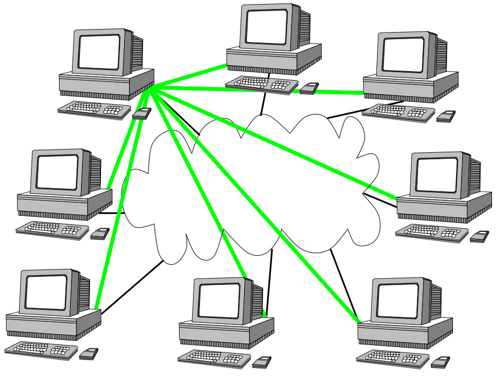
Slide 21

Example:



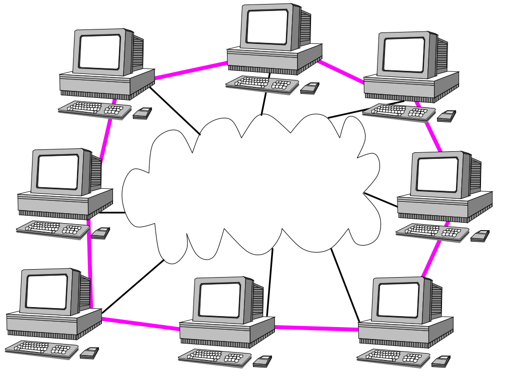
Slide 22

Example:



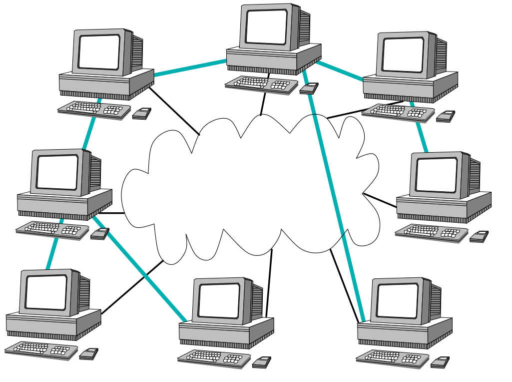
Slide 23

Example:



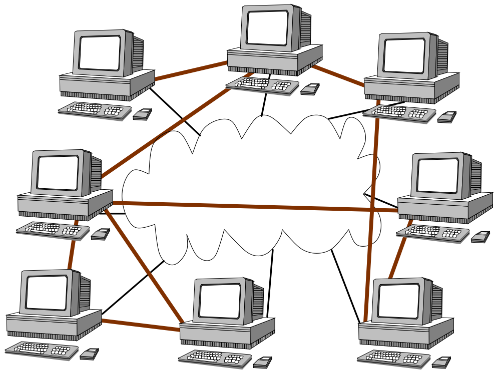
Slide 24

Example:



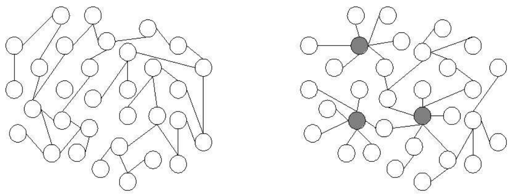
Slide 25

Example:



Slide 26

Unstructured Overlay



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

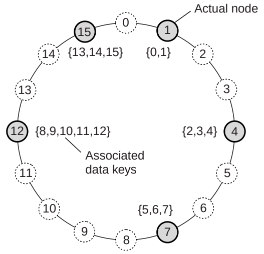
- ➜ Data stored at random nodes
- ➜ Partial view: node's list of neighbours
- ➜ Exchange partial views with neighbours to update

What's a problem with this?

Slide 27

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 28

Hybrid Architectures

Combination of architectures.

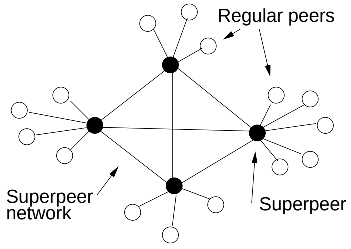
Examples:

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

Slide 29

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



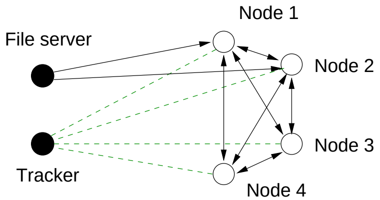
What are potential issues?

Slide 30

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

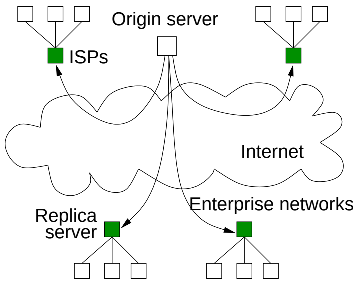


What problems does Bit Torrent face?

Slide 31

Edge-Server Networks:

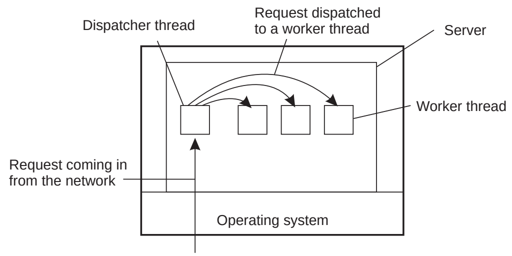
- ➜ Servers placed at the edge of the network
- ➜ Servers replicate content
- ➜ Mostly used for content and application distribution
- ➜ *Content Distribution Networks*: Akamai, CloudFront, CoralCDN



What are the challenges?

Slide 32

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

Slide 33

Stateful vs Stateless Servers

Stateful:

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

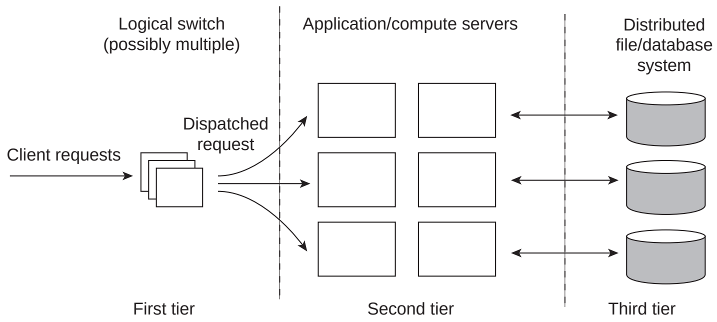
Stateless:

- ➜ 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

Slide 34

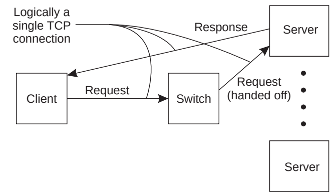
Clustered Servers



Slide 35

Request Switching

Transport layer switch:



DNS-based:

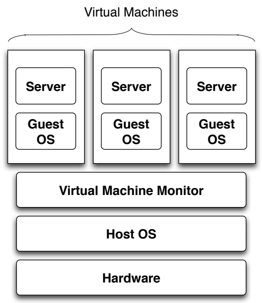
- ➜ Round-robin DNS

Application layer switch:

- ➜ Analyse requests
- ➜ Forward to appropriate server

Slide 36

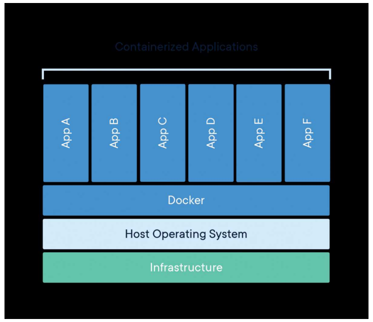
Virtualisation



What are the benefits?

Slide 37

Containerisation



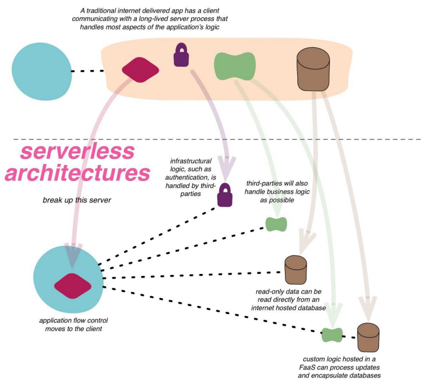
What are the benefits?

What are the drawbacks?

(from `https://www.docker.com/resources/what-container`)

Slide 38

Serverless



(from `https://martinfowler.com/bliki/Serverless.html`)

Slide 39

Serverless does use servers!

➜ You don't maintain them yourself

➜ You only provide functions to run

➜ Transparently run on servers

➜ Functions as a Service (FaaS)

- code components have a short lifecycle (per request)
- environment manages loading, starting, stopping code
- client-side management of control-flow, application logic

Slide 40

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

Examples: Java, JavaScript, Virtual Machines, Mobile Agents

What are the challenges of code mobility?

HOMEWORK

Client Server:

Slide 41

- ➜ Do Exercise *Client server exercise (Erlang)* Part A

Hacker's Edition: Client-Server vs Ring:

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