Principles of Distributed Systems

inft-3507

Dr. J.Burns

ADA University

Autumn 2025

Section 2: Architectures

Ref: https://www.distributed-systems.net/index.php/books/ds4/

Architectural styles

Basic idea

A style is formulated in terms of

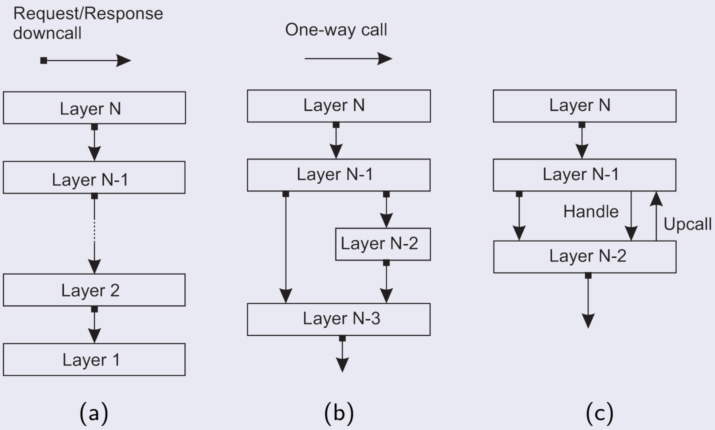
- (replaceable) components with well-defined interfaces
- the way that components are connected to each other
- the data exchanged between components
- how these components and connectors are jointly configured into a system.

Connector

A mechanism that mediates communication, coordination, or cooperation among components. Example: facilities for (remote) procedure call, messaging, or streaming.

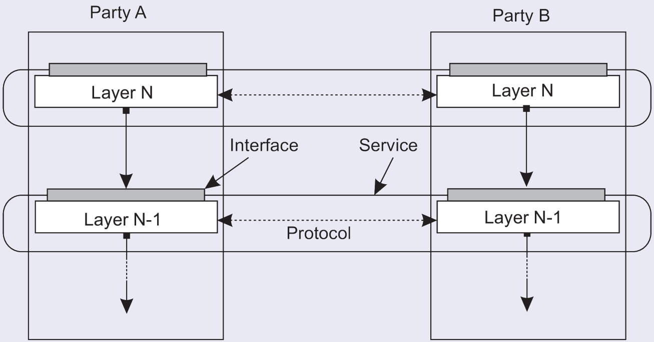
Layered architecture

Different layered organizations



Example: communication protocols

Protocol, service, interface



Two-party communication

Server

```
from socket  import *

s = socket(AF_INET, SOCK_STREAM)
(conn, addr) = s.accept()  # returns new socket and addr. client
while True:                # forever
  data = conn.recv(1024)   # receive data from client
  if not data: break       # stop if client stopped
  msg = data.decode()+"*"  # process the incoming data into a response
  conn.send(msg.encode())  # return the response
conn.close()               # close the connection
```

Client

```
from socket  import *

s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT)) # connect to server (block until accepted)
msg = "Hello World"     # compose a message
s.send(msg.encode())    # send the message
data = s.recv(1024)     # receive the response
print(data.decode())    # print the result
s.close()               # close the connection
```

Application Layering

Traditional three-layered view

- Application-interface layer contains units for interfacing to users or external applications
- Processing layer contains the functions of an application, i.e., without specific data
- Data layer contains the data that a client wants to manipulate through the application components

Application Layering

Traditional three-layered view

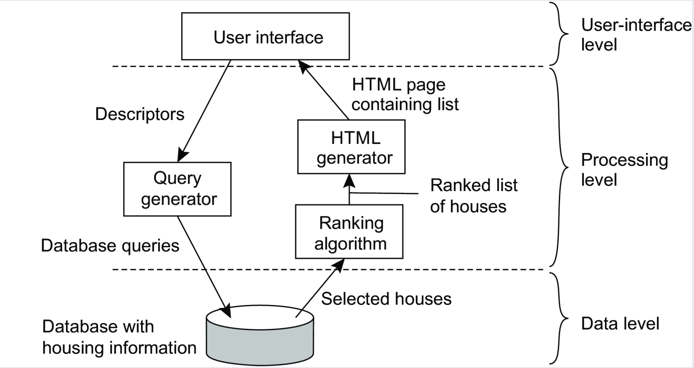
- Application-interface layer contains units for interfacing to users or external applications
- Processing layer contains the functions of an application, i.e., without specific data
- Data layer contains the data that a client wants to manipulate through the application components

Observation

This layering is found in many distributed information systems, using traditional database technology and accompanying applications.

Application Layering

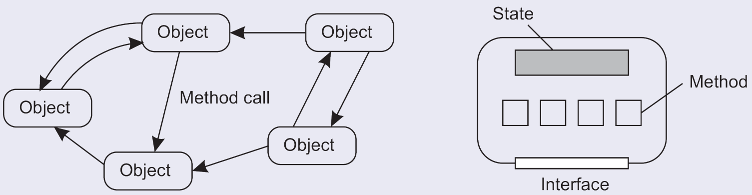
Example: a simple search engine



Object-based style

Essence

Components are objects, connected to each other through procedure calls. Objects may be placed on different machines; calls can thus execute across a network.



Encapsulation

Objects are said to encapsulate data and offer methods on that data without revealing the internal implementation.

RESTful architectures

Essence

View a distributed system as a collection of resources, individually managed by components. Resources may be added, removed, retrieved, and modified by (remote) applications.

1. Resources are identified through a single naming scheme
2. All services offer the same interface
3. Messages sent to or from a service are fully self-described
4. After executing an operation at a service, that component forgets everything about the caller

Basic operations

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state

Example: Amazon's Simple Storage Service

Essence

Objects (i.e., files) are placed into buckets (i.e., directories). Buckets cannot be placed into buckets. Operations on `ObjectName` in bucket `BucketName` require the following identifier:

`http://BucketName.s3.amazonaws.com/ObjectName`

Typical operations

All operations are carried out by sending HTTP requests:

- Create a bucket/object: `PUT`, along with the URI
- Listing objects: `GET` on a bucket name
- Reading an object: `GET` on a full URI

On interfaces

Issue

Many people like RESTful approaches because the interface to a service is so simple. The catch is that much needs to be done in the parameter space.

Amazon S3 SOAP interface

Bucket operations	Object operations
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetObjectAccessControlPolicy

On interfaces

Simplifications

Assume an interface `bucket` offering an operation `create`, requiring an input string such as `mybucket`, for creating a bucket "mybucket."

On interfaces

Simplifications

Assume an interface `bucket` offering an operation `create`, requiring an input string such as `mybucket`, for creating a bucket "mybucket."

SOAP

```
import bucket
bucket.create("mybucket")
```

On interfaces

Simplifications

Assume an interface bucket offering an operation create, requiring an input string such as mybucket, for creating a bucket "mybucket."

SOAP

```
import bucket
bucket.create("mybucket")
```

RESTful

```
PUT "https://mybucket.s3.amazonsws.com/"
```

On interfaces

Simplifications

Assume an interface `bucket` offering an operation `create`, requiring an input string such as `mybucket`, for creating a bucket “mybucket.”

SOAP

```
import bucket
bucket.create("mybucket")
```

RESTful

```
PUT "https://mybucket.s3.amazonsws.com/"
```

Conclusions

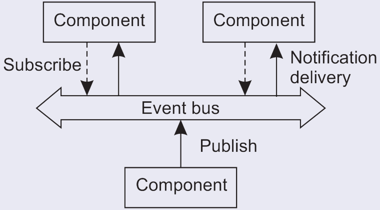
Are there any to draw?

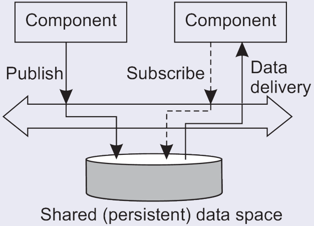
Coordination

Temporal and referential coupling

	Temporally coupled	Temporally coupled
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

Event-based and Shared data space





Example: Linda tuple space

Three simple operations

- `in(t)`: remove a tuple matching template `t`
- `rd(t)`: obtain copy of a tuple matching template `t`
- `out(t)`: add tuple `t` to the tuple space

More details

- Calling `out(t)` twice in a row, leads to storing two copies of tuple `t` $\Rightarrow$ a tuple space is modeled as a multiset.
- Both `in` and `rd` are blocking operations: the caller will be blocked until a matching tuple is found, or has become available.

Example: Linda tuple space

Bob:

```
import linda
linda.connect()

blog = linda.TupleSpace()
linda.universe._out(("MicroBlog",blog))

blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

blog._out(("bob","distsys","I am studying chap 2"))
blog._out(("bob","distsys","The linda example's pretty simple"))
blog._out(("bob","gtcn","Cool book!"))
```

Alice:

```
import linda
linda.connect()

blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

blog._out(("alice","gtcn","This graph theory stuff is not easy"))
blog._out(("alice","distsys","I like systems more than graphs"))
```

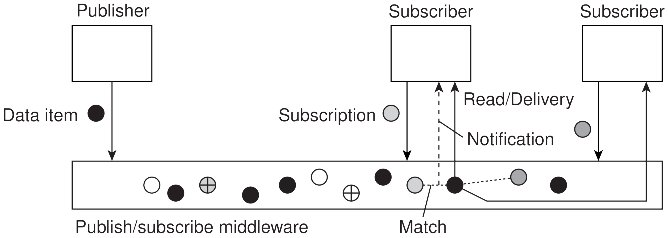
Chuck:

```
import linda
linda.connect()

blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

t1 = blog._rd(("bob","distsys",str))
t2 = blog._rd(("alice","gtcn",str))
t3 = blog._rd(("bob","gtcn",str))

print t1
print t2
```

Publish and subscribe

Issue: how to match events?

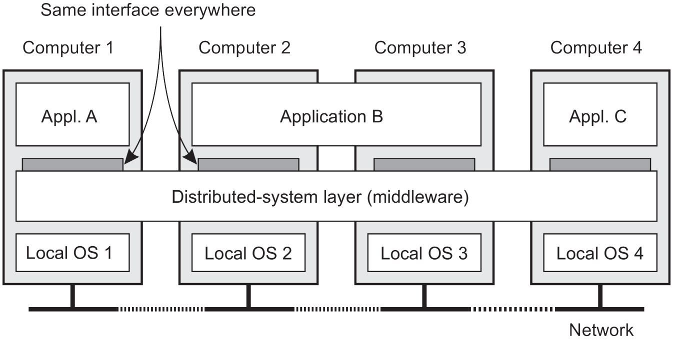
- Assume events are described by (attribute,value) pairs
- topic-based subscription: specify a “attribute = value” series
- content-based subscription: specify a “attribute ∈ range” series



Observation

Content-based subscriptions may easily have serious scalability problems (why?)

Middleware: the OS of distributed systems



What does it contain?

Commonly used components and functions that need not be implemented by applications separately.

Using legacy to build middleware

Problem

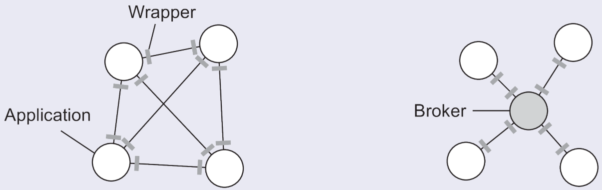
The interfaces offered by a legacy component are most likely not suitable for all applications.

Solution

A wrapper or adapter offers an interface acceptable to a client application. Its functions are transformed into those available at the component.

Organizing wrappers

Two solutions: 1-on-1 or through a broker



Complexity with N applications

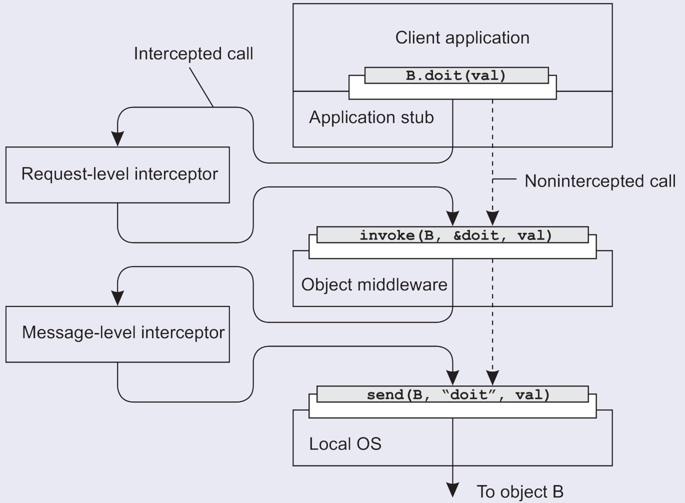
- 1-on-1: requires $N \times (N-1) = \mathcal{O}(N^2)$ wrappers
- broker: requires $2N = \mathcal{O}(N)$ wrappers

Developing adaptable middleware

Problem

Middleware contains solutions that are good for most applications $\Rightarrow$ you may want to adapt its behavior for specific applications.

Intercept the usual flow of control

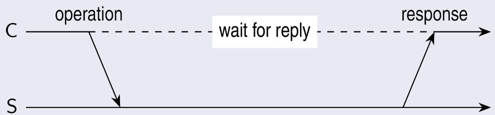


Centralized system architectures

Basic Client–Server Model

Characteristics:

- There are processes offering services (servers)
- There are processes that use services (clients)
- Clients and servers can be on different machines
- Clients follow request/reply model regarding using services

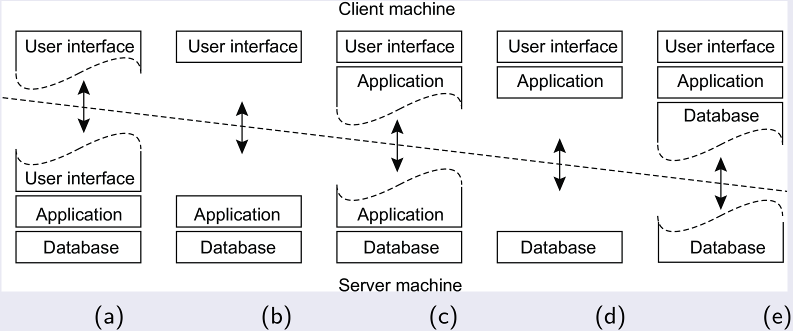


Multi-tiered centralized system architectures

Some traditional organizations

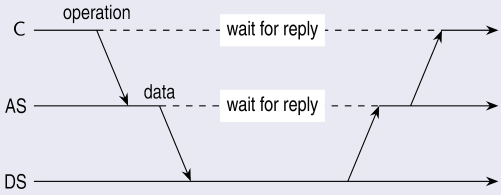
- Single-tiered: dumb terminal/mainframe configuration
- Two-tiered: client/single server configuration
- Three-tiered: each layer on separate machine

Traditional two-tiered configurations



Being client and server at the same time

Three-tiered architecture

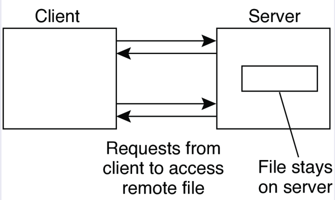


Example: The Network File System

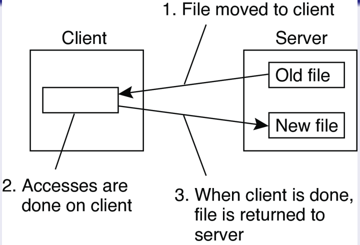
Foundations

Each NFS server provides a standardized view of its local file system: each server supports the same model, regardless the implementation of the file system.

The NFS remote access model



Remote access

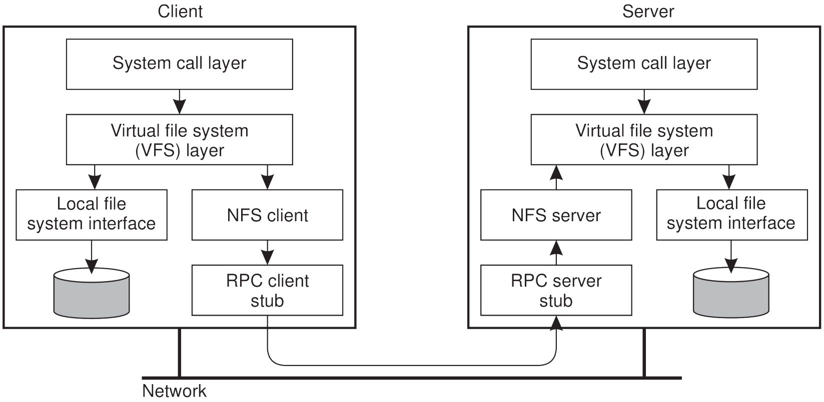


Upload/download

Note

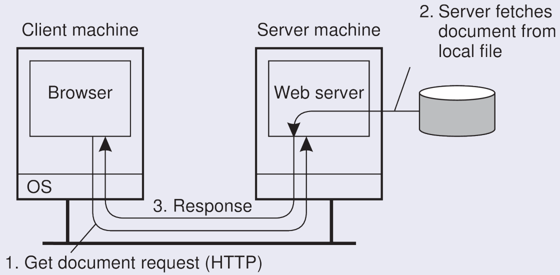
FTP is a typical upload/download model. The same can be said for systems like Dropbox.

NFS architecture



Example: Simple Web servers

Back in the old days...

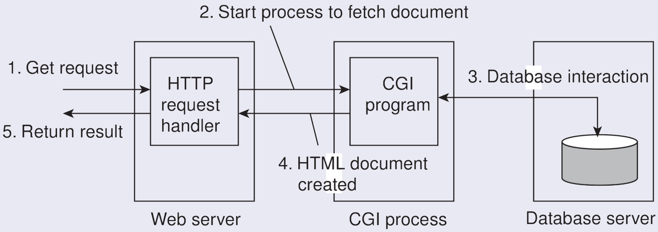


...life was simple:

- A website consisted as a collection of HTML files
- HTML files could be referred to each other by a hyperlink
- A Web server essentially needed only a hyperlink to fetch a file
- A browser took care of properly rendering the content of a file

Example (cnt'd): Less simple Web servers

Still back in the old days...



...life became a bit more complicated:

- A website was built around a database with content
- A Webpage could still be referred to by a hyperlink
- A Web server essentially needed only a hyperlink to fetch a file
- A separate program (Common Gateway Interface) composed a page
- A browser took care of properly rendering the content of a file

Alternative organizations

Vertical distribution

Comes from dividing distributed applications into three logical layers, and running the components from each layer on a different server (machine).

Horizontal distribution

A client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set.

Peer-to-peer architectures

Processes are all equal: the functions that need to be carried out are represented by every process $\Rightarrow$ each process will act as a client and a server at the same time (i.e., acting as a servant).

Structured P2P

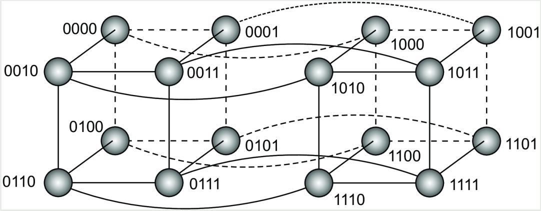
Essence

Make use of a semantic-free index: each data item is uniquely associated with a key, in turn used as an index. Common practice: use a hash function

$$key(data\ item) = hash(data\ item\text{'s}\ value).$$

P2P system now responsible for storing $(key, value)$ pairs.

Simple example: hypercube



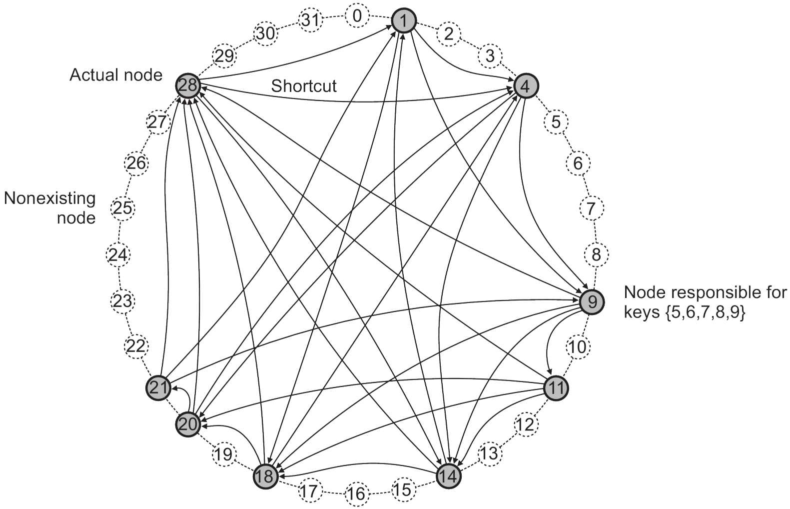
Looking up d with key $k \in \{0, 1, 2, \ldots, 2^4 - 1\}$ means routing request to node with identifier k.

Example: Chord

Principle

- Nodes are logically organized in a ring. Each node has an m-bit identifier.
- Each data item is hashed to an m-bit key.
- Data item with key k is stored at node with smallest identifier $id \geq k$, called the successor of key k.
- The ring is extended with various shortcut links to other nodes.

Example: Chord



$lookup(3)@9 : 28 \rightarrow 1 \rightarrow 4$

Unstructured P2P

Essence

Each node maintains an ad hoc list of neighbors. The resulting overlay resembles a random graph: an edge $\langle u, v \rangle$ exists only with a certain probability $\mathbb{P}[\langle u, v \rangle]$.

Searching

- Flooding: issuing node u passes request for d to all neighbors. Request is ignored when receiving node had seen it before. Otherwise, v searches locally for d (recursively). May be limited by a Time-To-Live: a maximum number of hops.
- Random walk: issuing node u passes request for d to randomly chosen neighbor, v. If v does not have d, it forwards request to one of *its* randomly chosen neighbors, and so on.

Flooding versus random walk

Model

Assume N nodes and that each data item is replicated across r randomly chosen nodes.

Random walk

$\mathbb{P}[k]$ probability that item is found after k attempts:

$$\mathbb{P}[k] = \frac{r}{N}(1 - \frac{r}{N})^{k-1}.$$

S ("search size") is expected number of nodes that need to be probed:

$$S = \sum_{k=1}^{N} k \cdot \mathbb{P}[k] = \sum_{k=1}^{N} k \cdot \frac{r}{N}(1 - \frac{r}{N})^{k-1} \approx N/r \text{ for } 1 \ll r \leq N.$$

Flooding versus random walk

Flooding

- Flood to d randomly chosen neighbors
- After k steps, some $R(k) = d \cdot (d-1)^{k-1}$ will have been reached (assuming k is small).
- With fraction r/N nodes having data, if $\frac{r}{N} \cdot R(k) \geq 1$, we will have found the data item.

Comparison

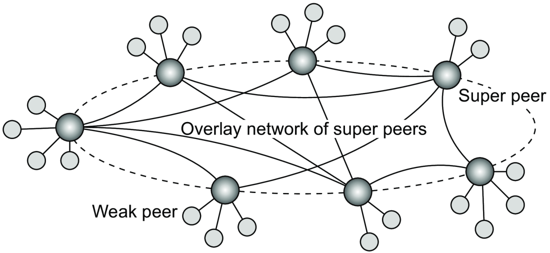
- If $r/N = 0.001$, then $S \approx 1000$
- With flooding and $d = 10, k = 4$, we contact 7290 nodes.
- Random walks are more communication efficient, but might take longer before they find the result.

Super-peer networks

Essence

It is sometimes sensible to break the symmetry in pure peer-to-peer networks:

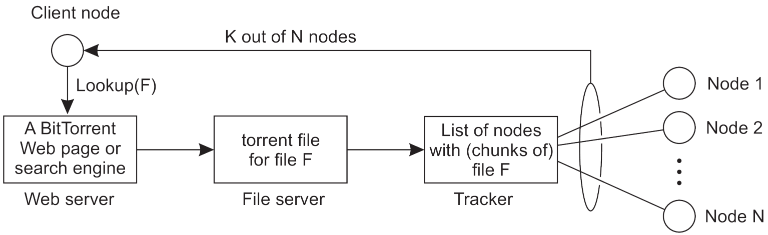
- When searching in unstructured P2P systems, having index servers improves performance
- Deciding where to store data can often be done more efficiently through brokers.



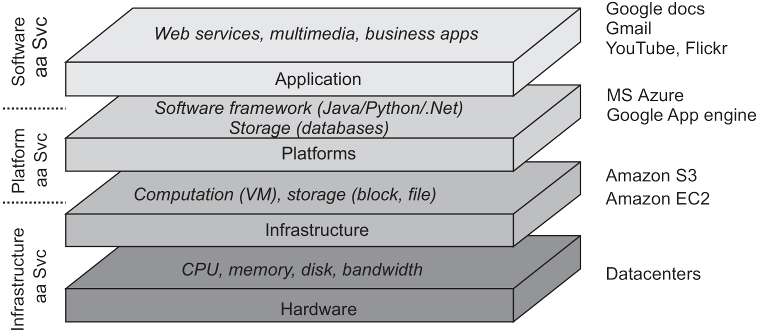
Collaboration: The BitTorrent case

Principle: search for a file F

- Lookup file at a global directory $\Rightarrow$ returns a torrent file
- Torrent file contains reference to tracker: a server keeping an accurate account of active nodes that have (chunks of) F.
- P can join swarm, get a chunk for free, and then trade a copy of that chunk for another one with a peer Q also in the swarm.



Cloud computing



Cloud computing

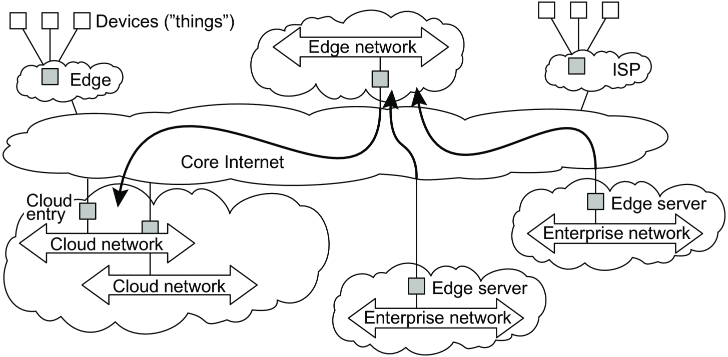
Make a distinction between four layers

- Hardware: Processors, routers, power and cooling systems. Customers normally never get to see these.
- Infrastructure: Deploys virtualization techniques. Evolves around allocating and managing virtual storage devices and virtual servers.
- Platform: Provides higher-level abstractions for storage and such. Example: Amazon S3 storage system offers an API for (locally created) files to be organized and stored in so-called buckets.
- Application: Actual applications, such as office suites (text processors, spreadsheet applications, presentation applications). Comparable to the suite of apps shipped with OSes.

Edge-server architecture

Essence

Systems deployed on the Internet where servers are placed at the edge of the network: the boundary between enterprise networks and the actual Internet.



Reasons for having an edge infrastructure

Commonly (and often misconceived) arguments

- Latency and bandwidth: Especially important for certain real-time applications, such as augmented/virtual reality applications. Many people underestimate the latency and bandwidth to the cloud.
- Reliability: The connection to the cloud is often assumed to be unreliable, which is often a false assumption. There may be critical situations in which extremely high connectivity guarantees are needed.
- Security and privacy: The implicit assumption is often that when assets are nearby, they can be made better protected. Practice shows that this assumption is generally false. However, securely handling data operations in the cloud may be trickier than within your own organization.

Edge orchestration

Managing resources at the edge may be trickier than in the cloud

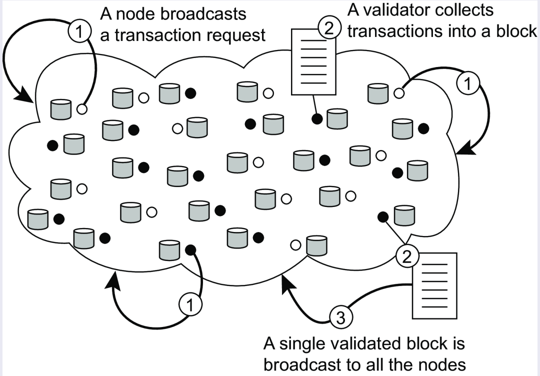
- Resource allocation: we need to guarantee the availability of the resources required to perform a service.
- Service placement: we need to decide when and where to place a service. This is notably relevant for mobile applications.
- Edge selection: we need to decide which edge infrastructure should be used when a service needs to be offered. The closest one may not be the best one.

Observation

There is still a lot of buzz about edge infrastructures and computing, yet whether all that buzz makes any sense remains to be seen.

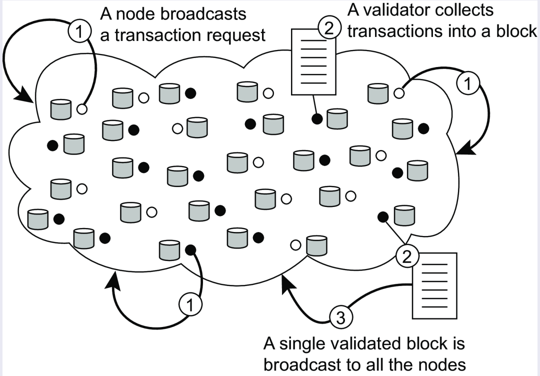
Blockchains

Principle working of a blockchain system



Blockchains

Principle working of a blockchain system

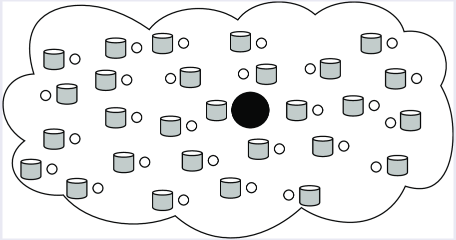


Observations

- Blocks are organized into an unforgeable append-only chain
- Each block in the blockchain is immutable ⇒ massive replication

Appending a block: distributed consensus

Centralized solution

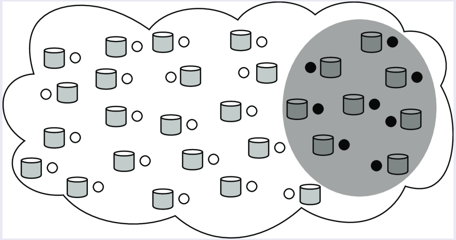


Observation

A single entity decides on which validator can go ahead and append a block. Does not fit the design goals of blockchains.

Appending a block: distributed consensus

Distributed solution (permissioned)

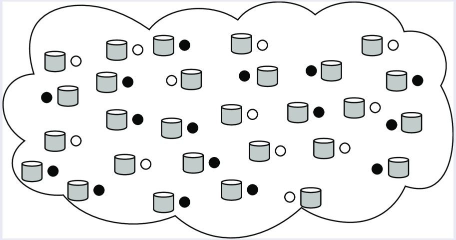


Observation

- A selected, relatively small group of servers jointly reach consensus on which validator can go ahead.
- None of these servers needs to be trusted, as long as roughly two-thirds behave according to their specifications.
- In practice, only a few tens of servers can be accommodated.

Appending a block: distributed consensus

Decentralized solution (permisionless)



Observation

- Participants collectively engage in a leader election. Only the elected leader is allowed to append a block of validated transactions.
- Large-scale, decentralized leader election that is fair, robust, secure, and so on, is far from trivial.