Chapter 2

Architectures

Topics

- Software Architectural Styles
- Middleware
- Physical Distribution /Architecture/

Architectural styles

- Distributed systems are often complex pieces of software, of which the components are by definition dispersed across multiple machines.
- The organization of distributed systems is mostly about the software components that constitute the system.

Architectural styles: Software Architecture

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.

Most important architectural styles for distributed systems are:

- Layered architectures
- Service-oriented architectures
- Publish-subscribe architectures

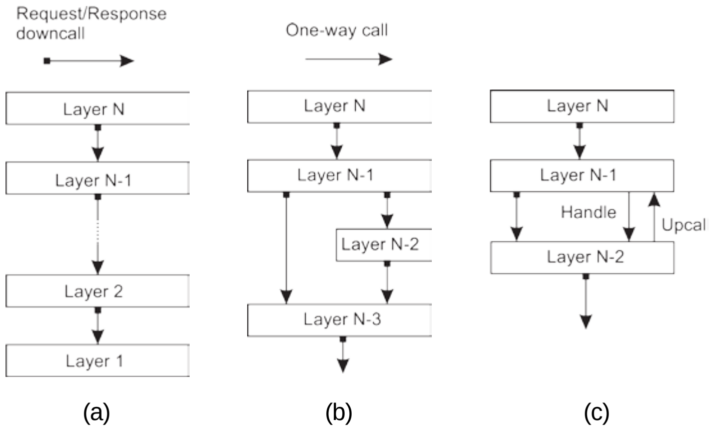
Note: in most real-world distributed systems, many styles are combined.

Layered Architecture:

- The basic idea for the layered style is: components are organized in a layered fashion where a component at layer Lj can make a downcall to a component at a lower-level layer Li (with $i < j$) and generally expects a response.
- Only in exceptional cases will an upcall be made to a higher-level component

Layered architecture

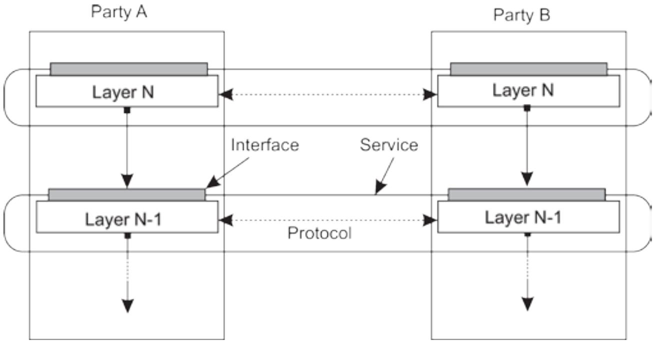
Different layered organizations



Layered Architecture:

Example: communication protocols

Protocol, service, interface



Example: 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

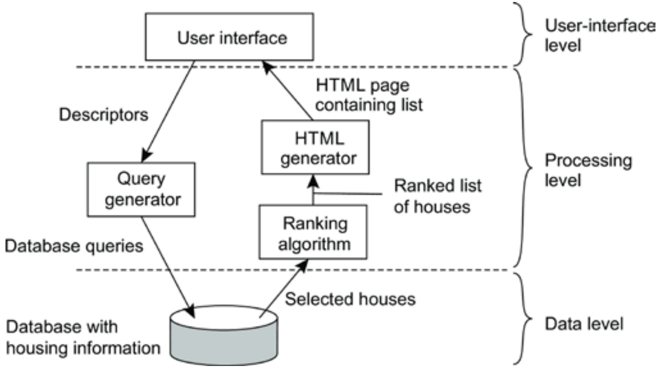
- Application-interface layer contains units for interfacing to users or external applications
- Processing layer contains the functions of an application, i.e., without specific data
- Data layer contains the data that a client wants to manipulate through the application components

Observation

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

Application Layering

Example: a simple search engine



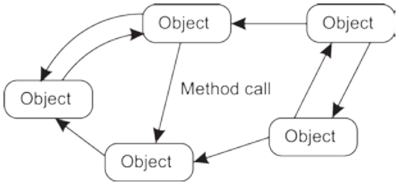
Service Oriented Style

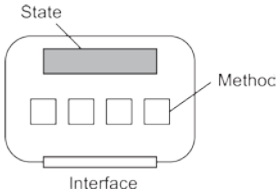
- Although the layered architectural style is popular, one of its major drawbacks is the often strong dependency between different layers.
- Such direct dependencies to specific components have led to an architectural style reflecting a more loose organization.

Object Oriented 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

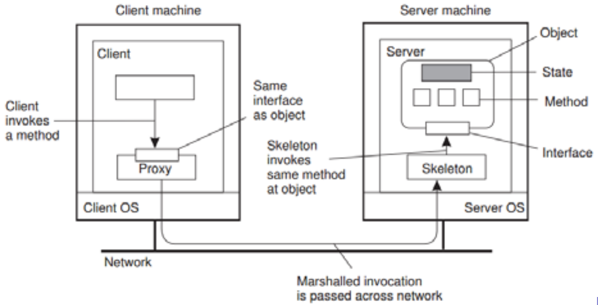
Objects are said to encapsulate data and offer methods on that data without revealing the internal implementation.

Object Oriented Style

Objects and interfaces for it might be found distributed over different machines.

The separation between interfaces and the objects implementing these interfaces allows placing an interface at one machine, while the object itself resides on another machine.

It is commonly referred to as a **distributed object**



Microservice architectural style

- By clearly separating various services such that they can operate independently, the road toward **service-oriented architectures** is paved.
- Generally abbreviated as **SOA**s.
- Despite no common agreement, in the end a microservice truly represents a separate, independent service.

Publish-subscribe architectures

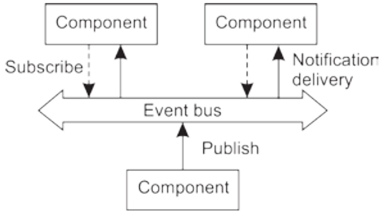
- An architecture in which there is a strong separation between *processing* and *coordination is required*.
- Different taxonomy of coordination models have been devised.
- A distinction between models can be made along two different dimensions, *temporal and referential*.

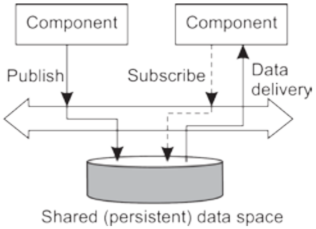
Coordination

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

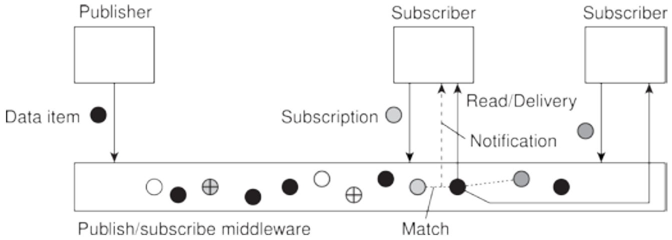




Publish and subscribe

When matching succeeds, there are two possible scenarios.

- the middleware may forward the published notification, along with data,
- The subscribers have a means to read data.



Middleware

The OS of distributed systems

Middleware: the OS of distributed systems

Distributed systems are often organized to have a separate layer of software logically placed on top of the respective operating systems of the computers that are part of the system.

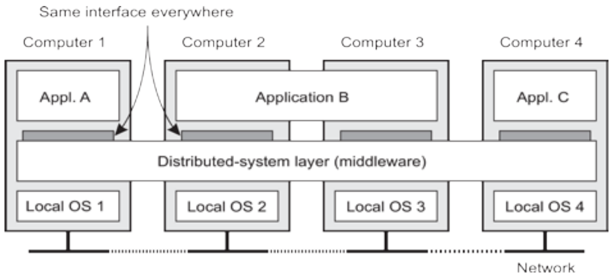
This is known us *middleware.*

Middleware: the OS of distributed systems

In a sense, middleware is the same to a distributed system as what an operating system is to a computer

- Transparent Resource Management (OS and Hardware)
- Facilities for inter application communication.
- Security services
- Accounting services
- Masking of and recovery from failures

Middleware: the OS of distributed systems



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

What makes a Middleware different from OS?

Middleware Organization

Two important types of *design patterns* that are often applied to the organization of middleware:

- Wrappers and
- Interceptors

Each targets different problems, yet addresses the same goal for middleware

Middleware Organization

Wrappers

Problem

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

Solution

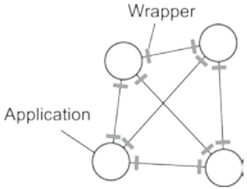
A wrapper or adapter offers an interface acceptable to a client application. Its functions are transformed into those available at the component.

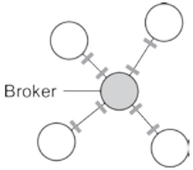
Devising a wrapper for every application doesn’t scale.

Facilitating a reduction of the number of wrappers is typically done through middleware.

Organizing wrappers

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





Complexity with *N* applications

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

Interceptors

Interceptor is nothing but a software construct that will break the usual flow of control and allow other code to be executed.

Are a primary means for adapting middleware to the specific needs of an application.

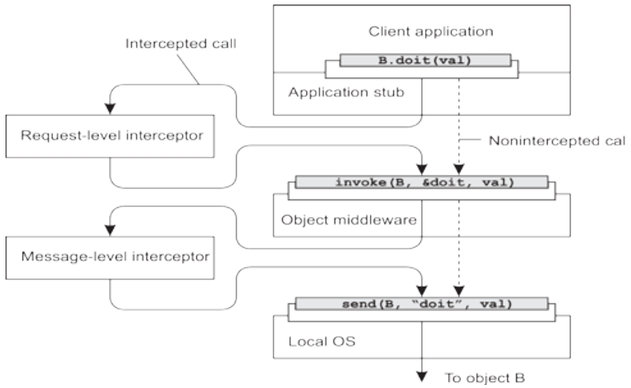
Interceptors

Such a remote-object invocation is carried out in three steps:

If Object A wants to call a method on object B:

1. Object A is offered a same interface as offered by object B. A calls the method available in that interface.
2. The call by A is transformed into a generic object invocation offered by the middleware at the machine where A resides.
3. The generic object invocation is transformed into a message that is sent through the transport-level network interface as offered by A’s local operating system.

Intercept the usual flow of control



Object B might be replicated. In that case, each replica should actually be invoked.

Physical Distribution of Systems

- Layered Architectures
- Symmetrically distributed system architectures
- Hybrid Systems

Layered System Architectures

Basic Client–Server Model

In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups.

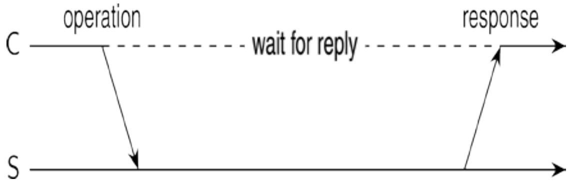
- Servers
- Clients

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

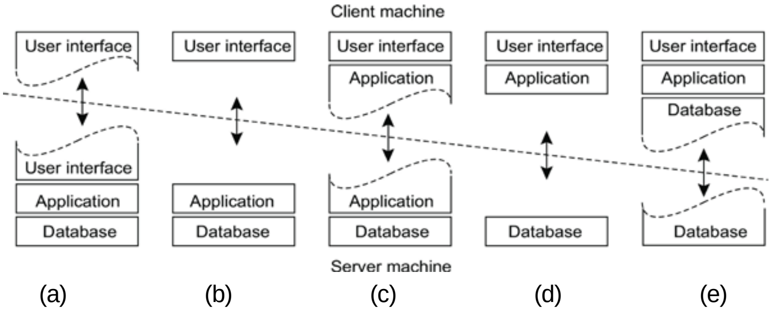


Multi-tiered centralized system architectures

Some traditional organizations

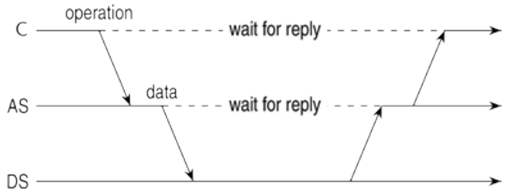
- Single-tiered: dumb terminal/mainframe configuration
- Two-tiered: client/single server configuration
- Three-tiered: each layer on separate machine

Traditional two-tiered configurations



Being client and server at the same time

Three-tiered architecture



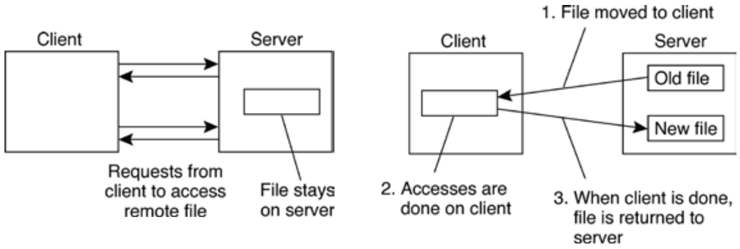
One example where we often see a three-tiered architecture is in the organization of Websites.

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

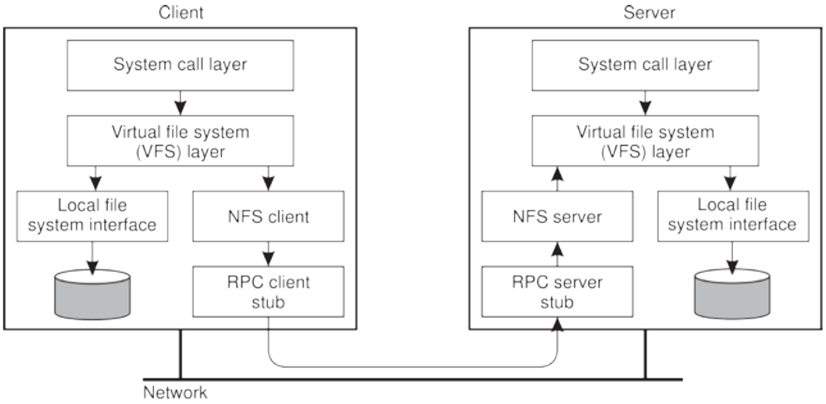

Client
Server
Requests from client to access remote file
File stays on server
1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to server

Remote access Upload/download

Note

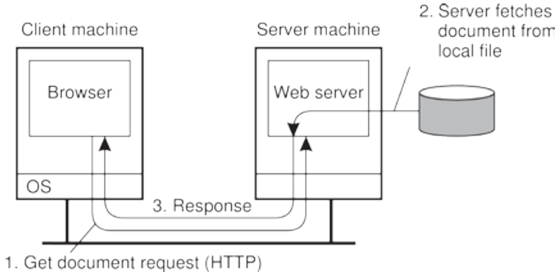
FTP is a typical upload/download model. The same can be said for systems like Dropbox.

NFS architecture


Client
Server
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

Example: Simple Web servers

Back in the old days...

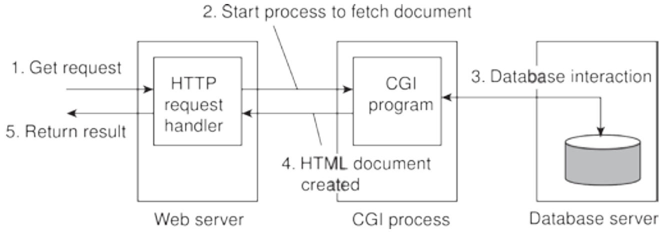

2. Server fetches document from local file
Client machine
Server machine
Browser
Web server
OS
3. Response
1. Get document request (HTTP)

...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

Symmetrically Distributed Architectures

Alternative organizations

Vertical distribution

Comes from dividing distributed applications into multiple 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 ⇒ each process will act as a client and a server at the same time (i.e., acting as a servant).

Alternative organizations

- From a high-level perspective, the processes that constitute a peer-to-peer system are all equal.
- The functions that need to be carried out are represented by every process that constitutes the distributed system.
- Much of the interaction is symmetric
- Each process will act as a servant

Structured P2P

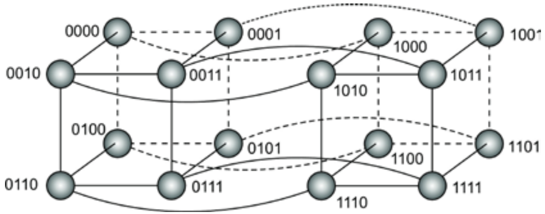
Essence

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.

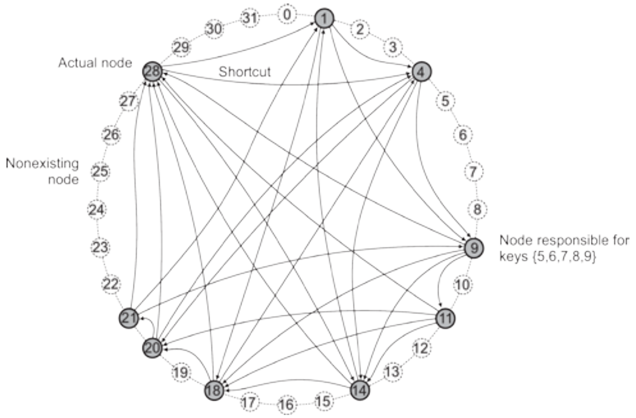
Simple example: hypercube



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

Example: Chord

Take it as a reading assignment!



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 $\mathrm{P}[\langle u, v \rangle]$.

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

P[k] probability that item is found after k attempts:

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

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

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

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

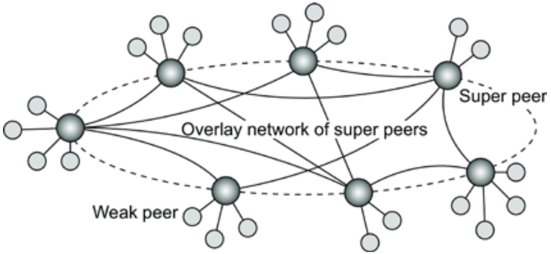
- If $r/N = 0.001$, then $S \approx 1000$
- With flooding and $d = 10, k = 4$, we contact 7290 nodes.
- Random walks are more communication efficient, but might take longer before they find the result.

Hierarchical: Super-peer networks

Essence

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

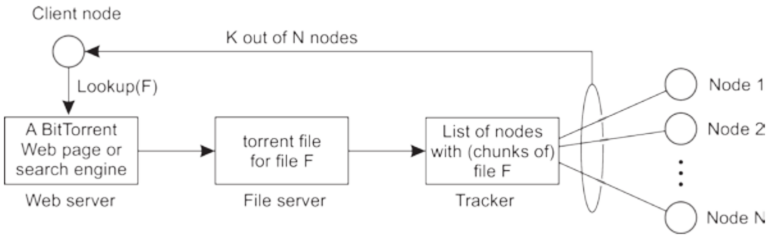
- When searching in unstructured P2P systems, having index servers improves performance
- Deciding where to store data can often be done more efficiently through brokers.



Collaboration: The BitTorrent case

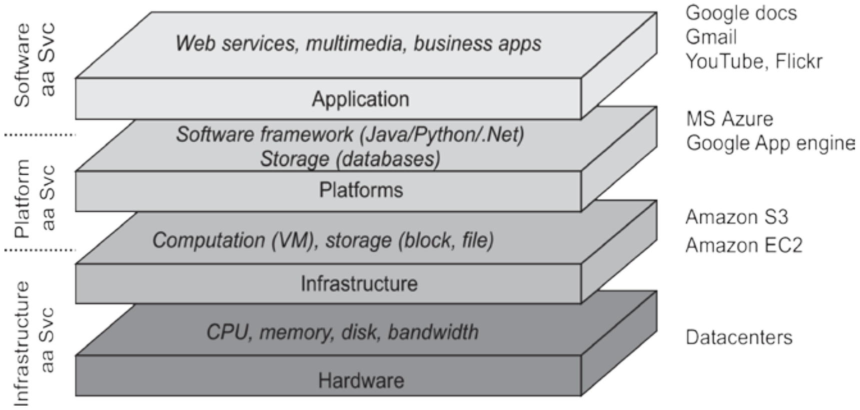
Principle: search for a file *F*

- Lookup file at a global directory ⇒ 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.



Hybrid System Architecture

Cloud computing



Cloud computing

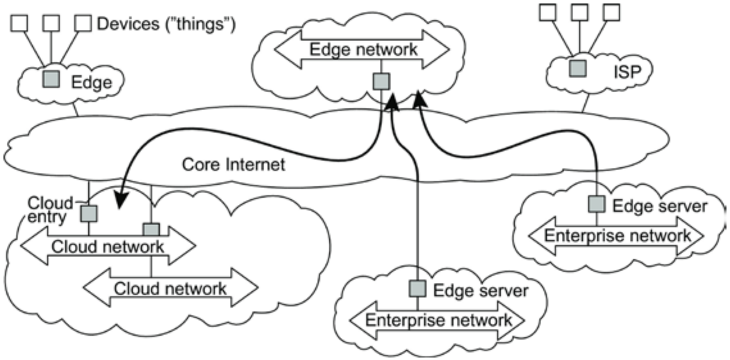
Make a distinction between four layers

- 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

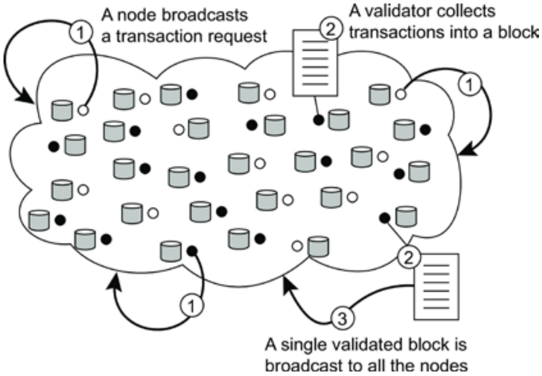
- Resource allocation: we need to guarantee the availability of the resources required to perform a service.
- Service placement: we need to decide when and where to place a service. This is notably relevant for mobile applications.
- Edge selection: we need to decide which edge infrastructure should 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.

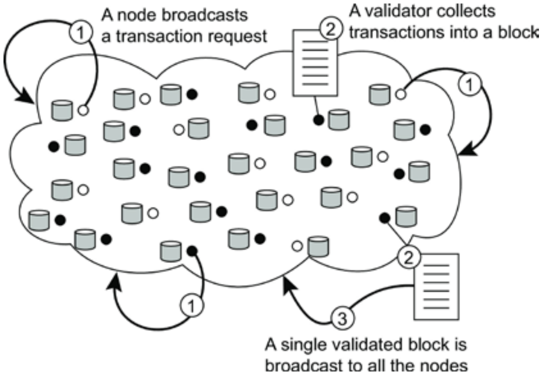
Blockchains

Principle working of a blockchain system


1 A node broadcasts a transaction request
2 A validator collects transactions into a block
1
1
2
3 A single validated block is broadcast to all the nodes

Blockchains

Principle working of a blockchain system

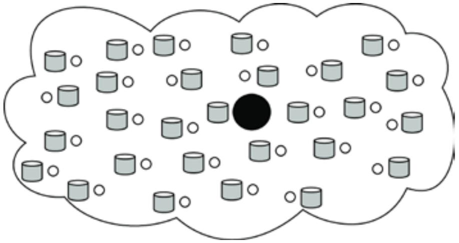

1
A node broadcasts a transaction request
2
A validator collects transactions into a block
3
A single validated block is broadcast to all the nodes

Observations

- Blocks are organized into an unforgeable append-only chain
- Each block in the blockchain is immutable ⇒ massive replication
- The real snag lies in who is allowed to append a block to a chain

Appending a block: distributed consensus

Centralized solution

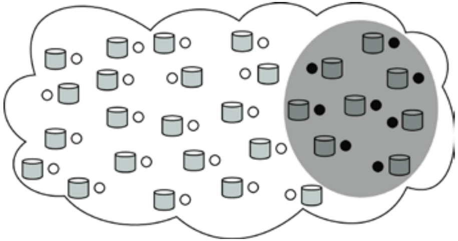


Observation

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)

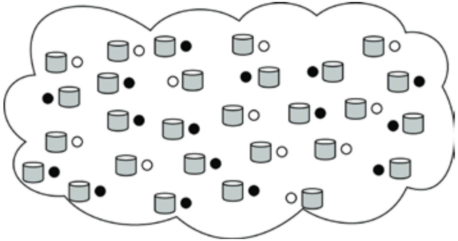


Observation

- 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.