Principles of Distributed Systems

inft-3507

Dr. J.Burns

ADA University

Autumn 2025
Section 2: Architectures

Architectural styles

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.
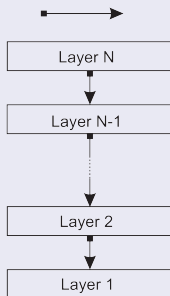
Connector

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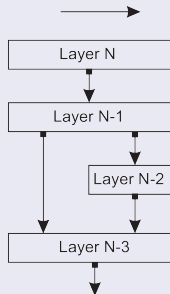
Layered architectures

Layered architecture

Different layered organizations

Request/Response
downcall

One-way call

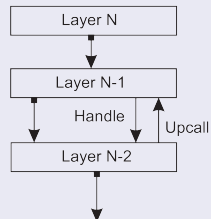|		
Layer N	Layer N	Layer N
Layer N-1	Layer N-1	Layer N-1
	Layer N-2	Handle
Layer 2		Layer N-2
Layer 1	Layer N-3	

(a)   (b)   (c) callback

Note in (c) the **handle** downward call does not return data. The upward callback is a separate invocation. No call/return pairing across layers, ulike the others.

Application Layering
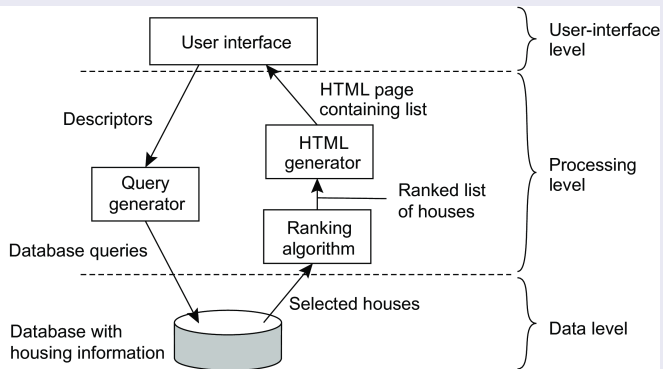
Traditional three-layered view

- **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
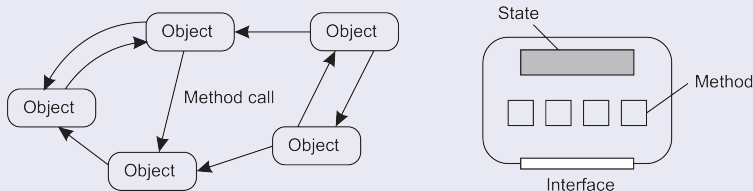
Example: a simple search engine

RPC and REST Architectures

RPC - 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. This is known as a Remote Procedure Call (RPC)

Encapsulation

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

Why RPC faded away

It didn't **modernize** well

- **Tight coupling** between clients and servers made traditional RPC systems difficult to evolve, increasing maintenance and versioning complexity.
- **Limited abstraction** of distributed system boundaries led developers to treat remote calls as local, resulting in poor handling of latency, failures, and partial outages.
- **Security concerns**, including excessive exposure of internal methods and weaker alignment with standardized authentication and authorization mechanisms, reduced suitability for public APIs.
- **Debugging and observability** challenges arose from opaque payloads and tooling that was less compatible with standard web diagnostics and monitoring practices.
- **RESTful architectures** aligned more effectively with web infrastructure, enabling better scalability, interoperability, and long-term API stability, leading to broader adoption.

RESTful architectures

Essence

- **Architectural Style**: REST is an architectural style for networked applications, focusing on stateless, client-server communication over HTTP, designed for scalability and simplicity.
- **Resource-Based**: Resources in REST are uniquely identified by URIs and manipulated using standard HTTP methods such as GET, POST, PUT, DELETE, and PATCH.
- **Statelessness**: Each client request to the server is independent, containing all necessary information, eliminating server-side session state to enhance scalability.
- **Layered System**: REST employs a layered architecture, allowing intermediaries like proxies or load balancers to improve performance and security transparently.
- **Cacheability**: REST responses can be cached when explicitly defined, improving performance by reducing server load and latency for frequently accessed resources.

REST Services

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

- REST is a scalable, stateless architectural style for networked applications using HTTP.

- It focuses on resources identified by URIs, manipulated via standard methods (GET, POST, PUT, DELETE).

- Features a layered, cacheable design to enhance performance and flexibility.

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

Publish-subscribe architectures
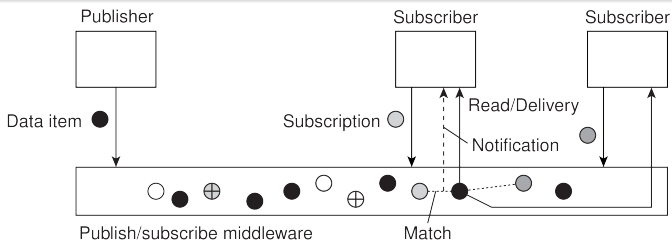
Publish and Subscribe

Pub-Sub Model: Publishers send messages to topics, subscribers receive them asynchronously.

- **Decoupling**: Publishers and subscribers operate independently, enhancing scalability.
- **Asynchronous Communication**: Messages are delivered without requiring immediate response.
- **Topics**: Messages are categorized into topics for targeted delivery.
- **Loose Coupling**: Reduces dependencies between system components.
- **Event-Driven**: Triggers actions based on message events.

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?**)

Symmetrically distributed system architectures

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**).
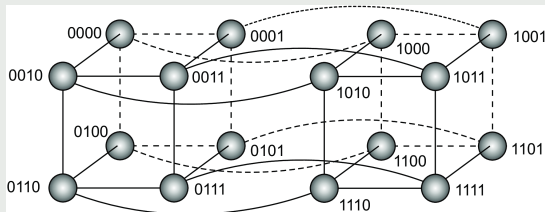
Structured P2P

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**

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

P2P system now responsible for storing (*key*,*value*) pairs.
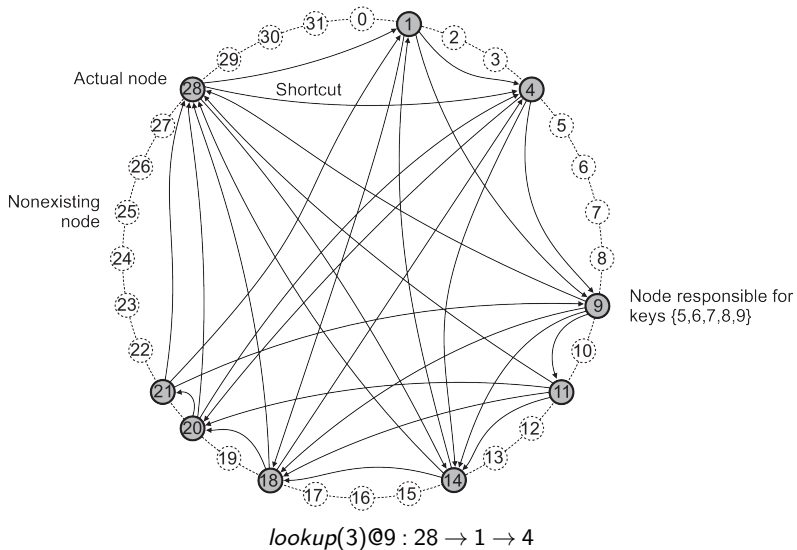
Simple example: hypercube

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.
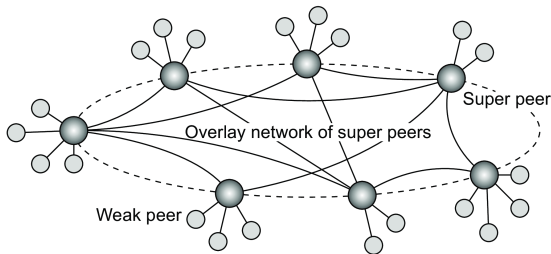
Comparison

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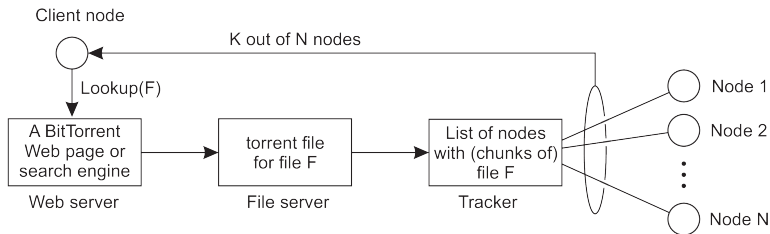
- When searching in unstructured P2P systems, having **index servers** improves performance
- Deciding where to store data can often be done more efficiently through **brokers**.
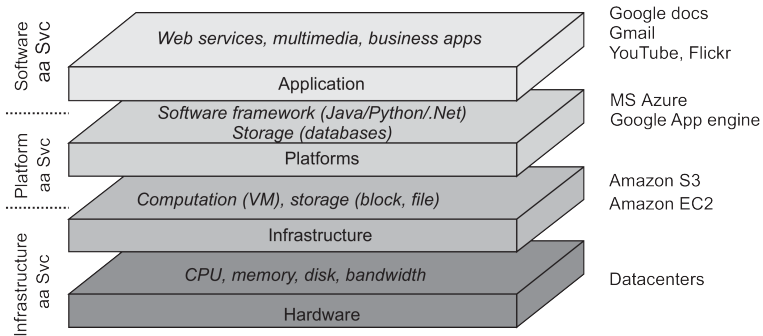
Collaboration: The BitTorrent case

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.

Hybrid system architectures

Cloud computing

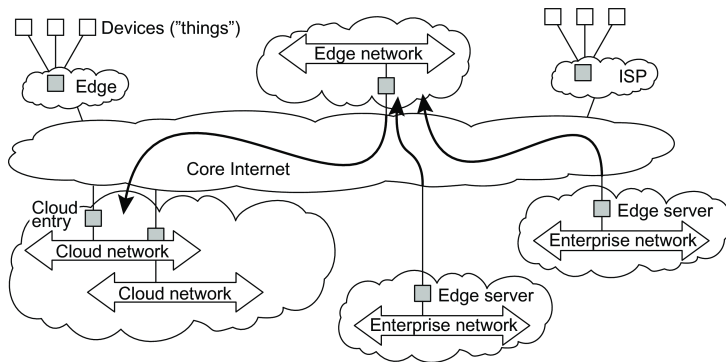|		
Software aa Svc	Web services, multimedia, business apps / Application	Google docs / Gmail / YouTube, Flickr
Platform aa Svc	Software framework (Java/Python/.Net) Storage (databases) / Platforms	MS Azure / Google App engine
Infrastructure aa Svc	Computation (VM), storage (block, file) / Infrastructure	Amazon S3 / Amazon EC2
	CPU, memory, disk, bandwidth / Hardware	Datacenters

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.
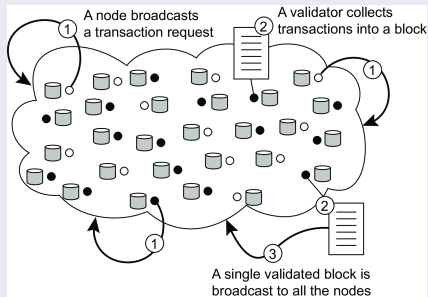
Observation

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

Blockchain architectures

Blockchains

Principle working of a blockchain system

A node broadcasts a transaction request

A validator collects transactions into a block

A single validated block is broadcast to all the nodes
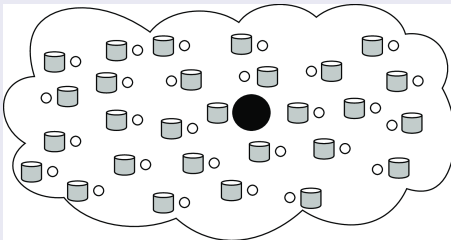
Observations

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

Appending a block: distributed consensus
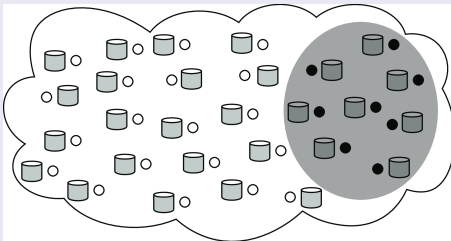
Centralized solution

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

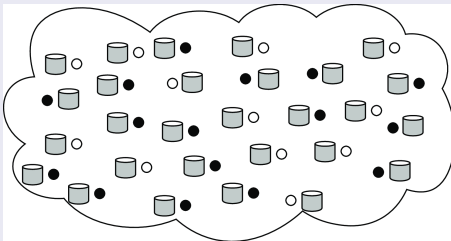### Distributed solution (permissioned)

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.

Summary

Summary and Conclusions

We have discussed architectureal principles in Distributed Systems, namely:

- Layered
- RPC and RESTful styles
- Publish and Subscribe
- Structured and Unstructured P2P
- Cloud Edge Computing
- Blockchains

As with all topics on this course, please make sure you understand the *why*, of each of these architectures, not just the *how/what*