

Distributed System Architectures

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2023/24

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Software vs. system architecture of DS

- **Software architecture:** **logical organization** and **interaction** of software components that constitute the DS
- **System architecture:** **final instantiation** (including deployment) of a software architecture
 - Software components need to be placed on system resources
 - E.g., a container containing a microservice needs to be instantiated on a machine
- Let's first focus on software architectures for DS

Architectural styles for DS

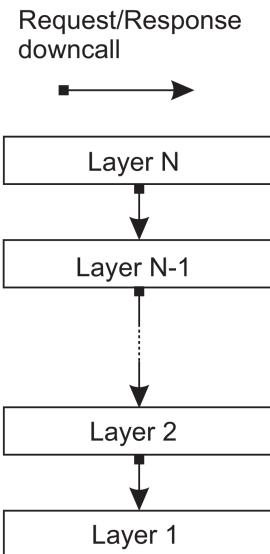
- Architectural style: set of design decisions concerning sw architecture
 - Mainly defined in terms of **components** and **connectors**
- **Component**
 - Modular unit with well-defined interfaces
 - Replaceable within its environment
- **Connector**
 - Mechanism for interaction among components, mediating communication, coordination or cooperation
 - Example: mechanisms for (remote) procedure call, messaging
- Plus:
 - How components are connected to each other
 - Data exchanged between components
 - How components and connectors are jointly configured into a system

Main architectural styles for DS

- Layered style
- Service-oriented style
 - Object-oriented
 - Microservices
 - RESTful
- Publish-subscribe style

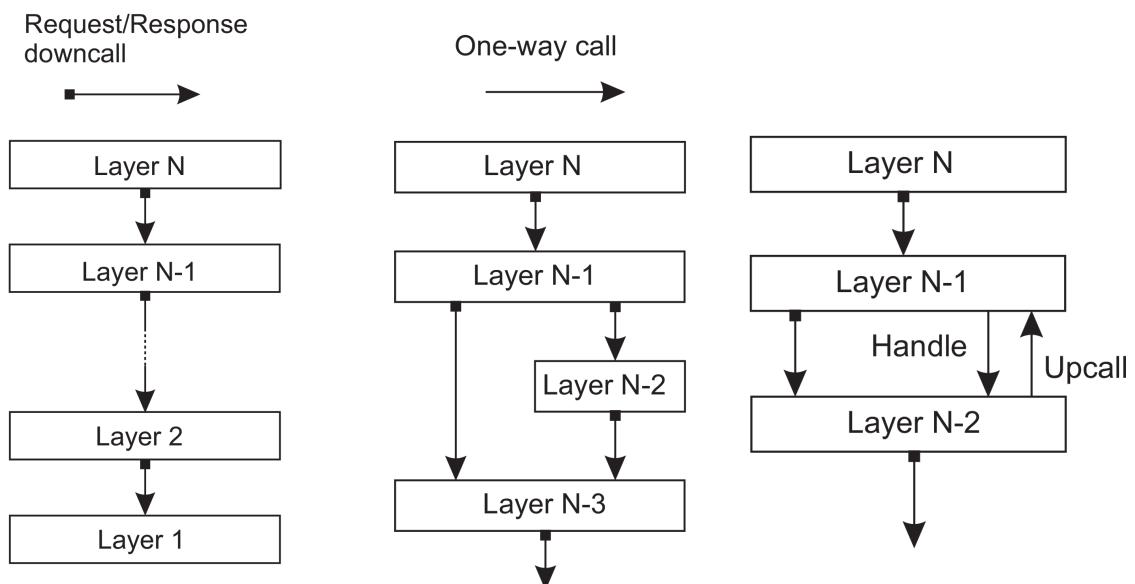
Layered style

- Components are organized in *layers*
 - Component at layer i invokes component at layer j (*with $j < i$*)
 - Components communicate by message exchange
 - Request/response downcall
- ✓ Separation of concerns among components
 - E.g., web app based on MVC design



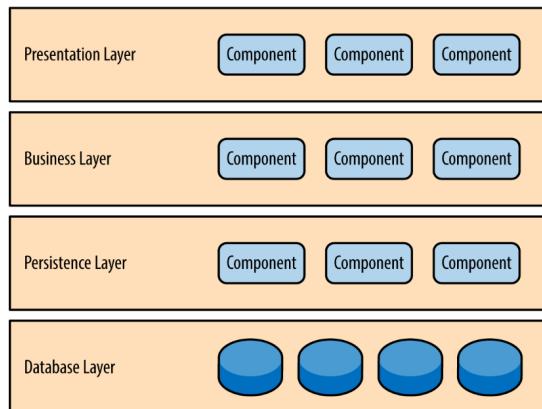
Layered style

- Different layered organizations



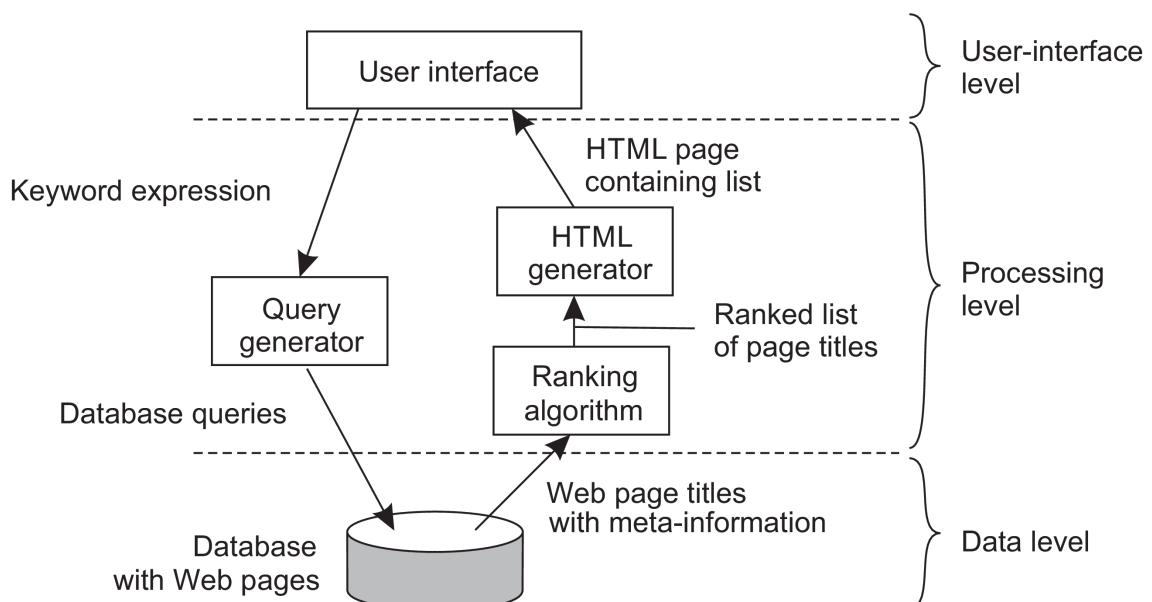
Application layering

- Traditional layered architecture: presentation, business, persistence, database
 - In some cases, business and persistence layers are combined into a single business layer
 - Found in many distributed information systems, using traditional DB technology and accompanying applications



Application layering: example

- A simple Web search engine

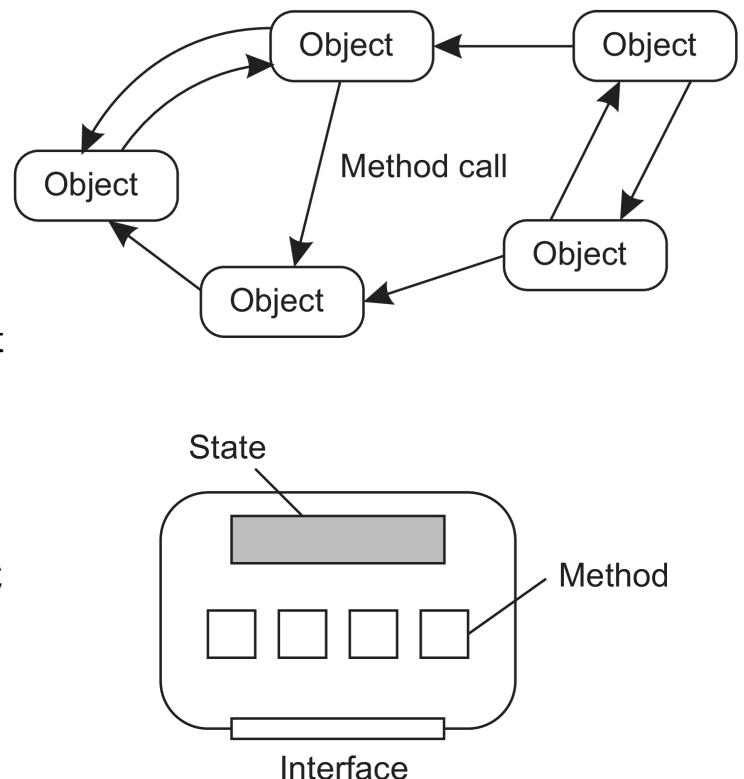


Service-oriented style

- A collection of separate, independent entities
- Each entity encapsulates a service
- Entity = service, object, or microservice
- Includes
 - Object-based architectural style
 - Microservices architectural style
 - RESTful architectural style

Object-based style

- Component = object:
encapsulates data and
offers methods on data
 - Encapsulation and information hiding reduce management complexity
 - Reusability among different apps
 - Wrapping of legacy components
- Communication between components through RPC

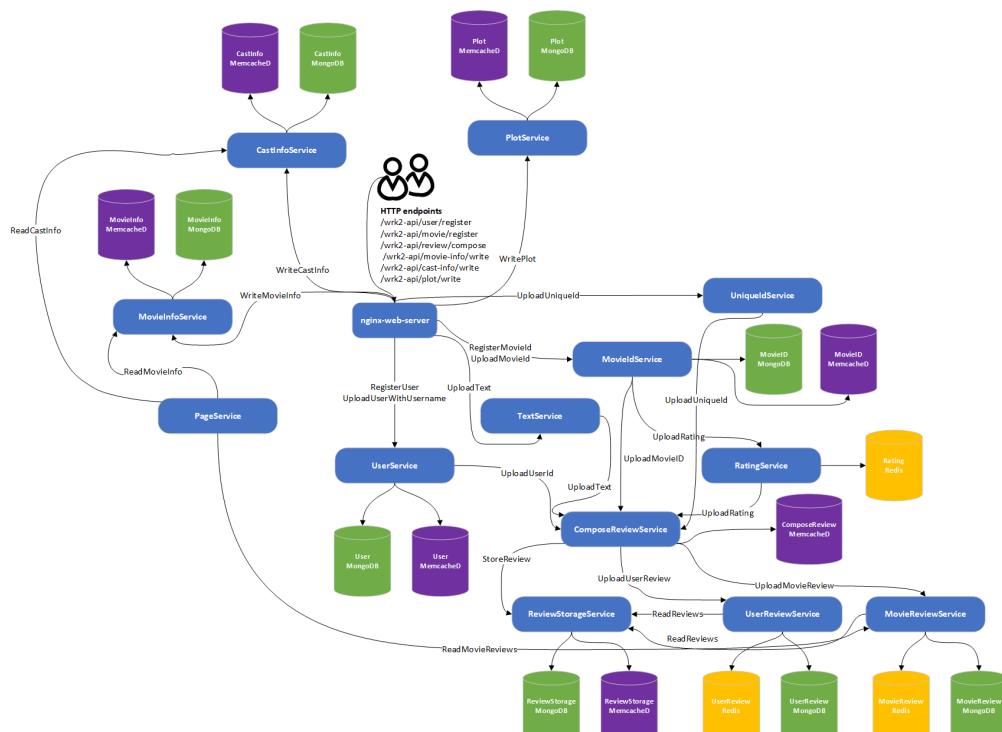


Microservices style

- A “new” emerging architectural style for distributed applications that structures an application as a **collection of loosely coupled services**
- Address how to build, manage, and evolve architectures out of **small, self-contained and independently-scalable services** that communicate over well-defined APIs
 - *Modularization*: decompose app into a set of **independently deployable services**, that are **loosely coupled** and **cooperating** and can be **rapidly deployed and scaled**
- See upcoming lessons

Microservices style: example

- A social-media microservice architecture



RESTful style

- DS as a collection of resources, individually managed by components
- **Representational State Transfer (REST)**: proposed by Roy Fielding, co-author of HTTP/1.1
 - Resources may be added, removed, retrieved, and modified by (remote) applications ([HTTP methods](#))
 - Resources are identified through a single naming scheme (Uniform Resource Identifier, [URI](#))
URI = scheme://authority[path[?query][#fragment]]
authority = [userinfo@]host[:port]
- Components expose a uniform interface
- Messages sent to/from component are self-described
- Interactions are [stateless](#)
 - State must be transferred from clients to servers

REST operations

- Basic operations
 - Use HTTP methods: GET, PUT, POST and DELETE

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: S3 REST API

- S3: cloud storage service by AWS, organized as a **key-based** object store
- **Objects (files)** are stored into **buckets** (directories)
 - Flat structure: no directory hierarchy
 - Logical hierarchy simulated by using object names with directory structure: photos/puppy.jpg
 - Object *objectname* stored in bucket *bucketname* is uniquely referred to by its URI:
`https://bucketname.s3.Region.amazonaws.com/objectname`
e.g., `https://example-bucket.s3.us-west-2.amazonaws.com/photos/puppy.jpg`
- Operations carried out through HTTP requests:
 - Create bucket/object: PUT, along with its URI
 - List objects in bucket: GET on bucket
 - Read object: GET on full URI

Example: S3 REST API

- Retrieve object from bucket (GetObject)
`GET /photos/puppy.jpg HTTP/1.1`
`Host: example-bucket.s3.us-west-2.amazonaws.com`
`Date: date`
`Authorization: authorization string`

- Add object to bucket (PutObject)
`PUT /photos/puppy.jpg HTTP/1.1`
`Host: example-bucket.s3.us-west-2.amazonaws.com`
`Date: date`
`Authorization: authorization string`

See [Amazon S3 REST API](#)

Notes:

- HTTP Authorization header to authenticate S3 request
- You need permission for operations (e.g., WRITE permission on bucket): create IAM roles and manage permissions

Decoupling

- Strong dependencies between components introduce limitations
- Solution: let components **indirectly communicate** through some intermediary
 - Clean separation between computation and coordination

“All problems in computer science can be solved by another level of indirection”

(David Wheeler, Titan project)

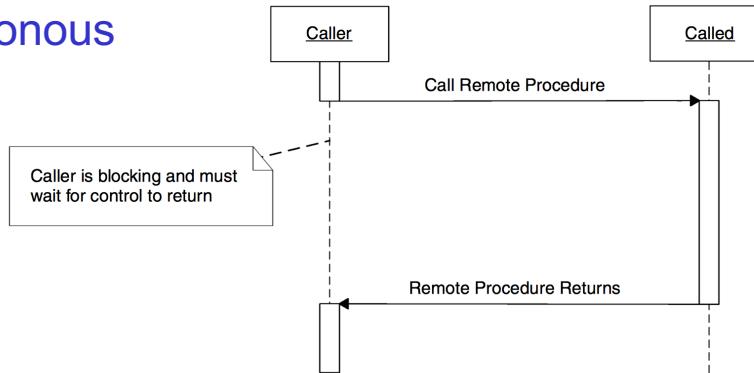
- **Decoupling**: enabling factor to
 - Achieve greater flexibility
 - Define architectural styles that allow to better exploit distribution, scalability, and elasticity

Decoupling properties

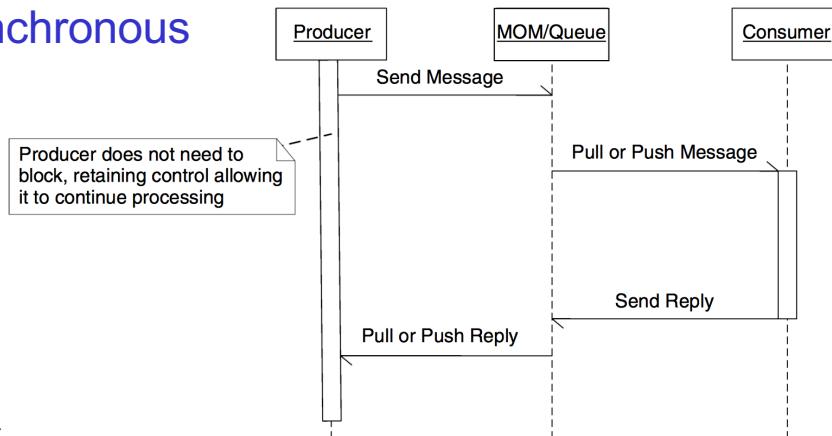
- **Space** (or **referential**) decoupling
 - Anonymous components: do not need to know each other in order to communicate and cooperate
- **Time** (or **temporal**) decoupling
 - Interacting components do not need to be present at the same time when communication occurs
- **Synchronization** decoupling
 - Interacting components do not need to wait each other and are not reciprocally blocked

Synchronous vs. asynchronous interaction

Synchronous



Asynchronous

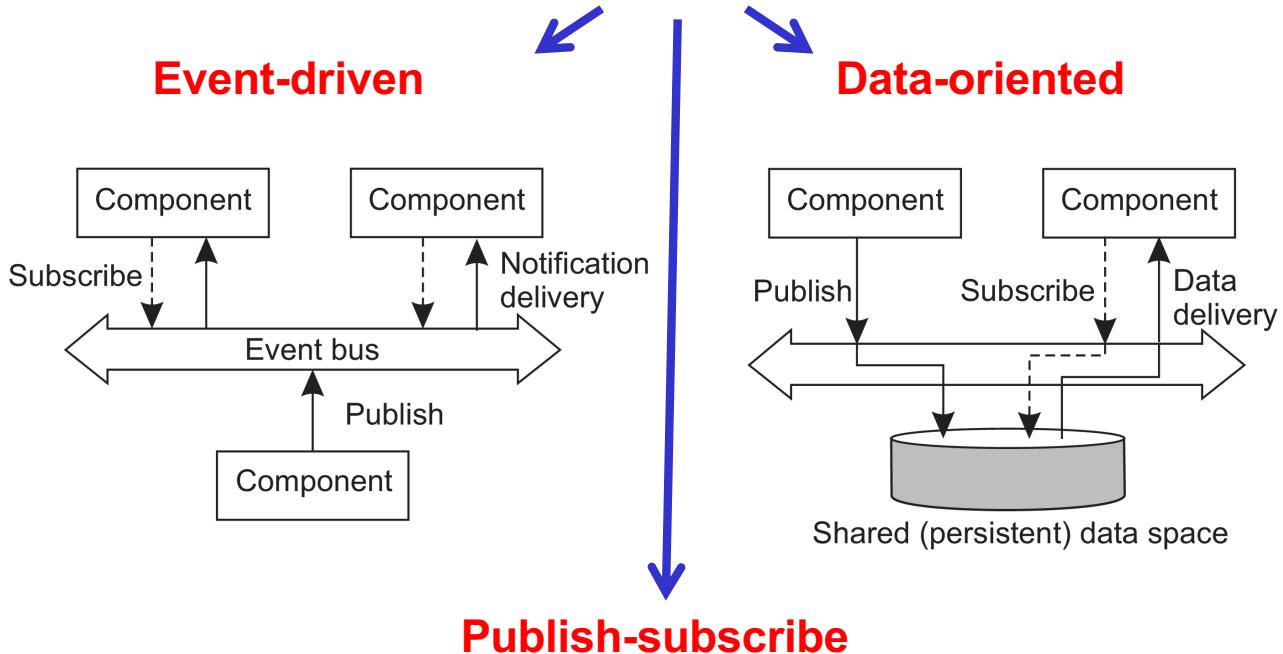


Decoupling: pros and cons

- ✓ Thanks to decoupling, DS can be flexible while dealing with changes and provide more dependable and elastic services
 - Space decoupling: components can be replaced, updated, replicated or migrated
 - Time decoupling: allows to manage volatility (senders and receivers can come and go)
 - Synchronization decoupling: no blocking
- ✗ Indirection can add performance overhead

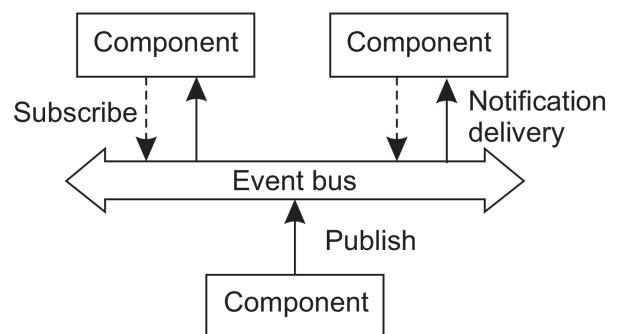
Architectural style evolution

- Introducing decoupling, alternative architectural styles where components communicate indirectly



Event-driven style

- Components communicate through an event bus
 - Event:** significant change in state (e.g., change in temperature, door opening)
- Components
 - Publish events
 - Subscribe to events they are interested in being notified
 - Receive notifications about events
- Communication
 - Anonymous
 - Based on message exchange
 - Asynchronous
 - Multicast



- Example: Java Swing

Which decoupling?

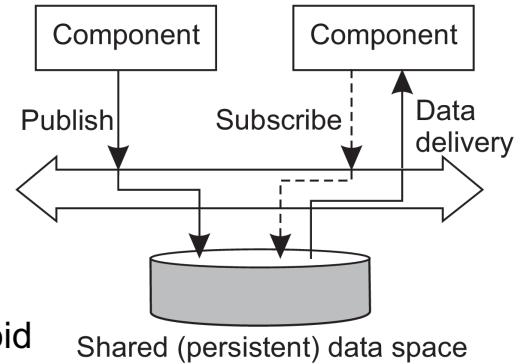
Data-oriented style

- Communication among components happens through shared data space: passive, sometimes (pro)active

- Data added to or removed from shared space

- Shared space API

- write, take, read and variants (takeIfExists, readIfExists)
 - If active space: notify or push (avoid polling)
 - Concurrency control



Which decoupling?

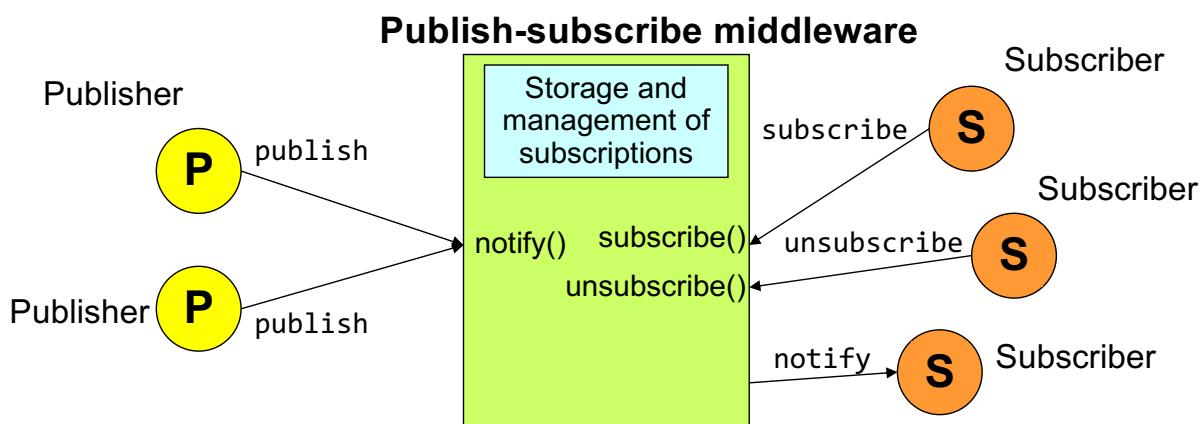
- Examples of shared data spaces:

- [GigaSpaces](#), [TIBCO ActiveSpaces](#)

How to implement shared space?

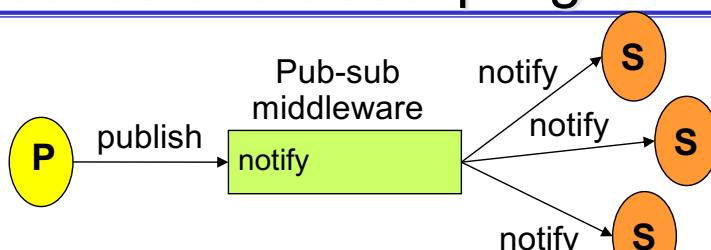
Publish-subscribe style

- Publishers** (aka **producers**) generate events (**publish**) and are not interested in their delivery to **subscribers** (aka **consumers**)
- Consumers** register as interested to events (**subscribe**) and are notified (**notify**) of their occurrence
- Full decoupling among components

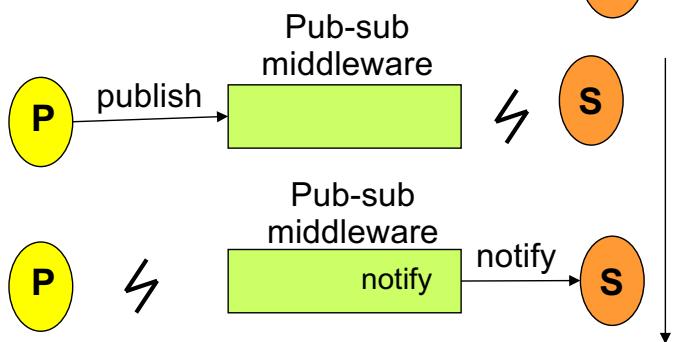


Publish-subscribe and decoupling

Space decoupling



Time decoupling



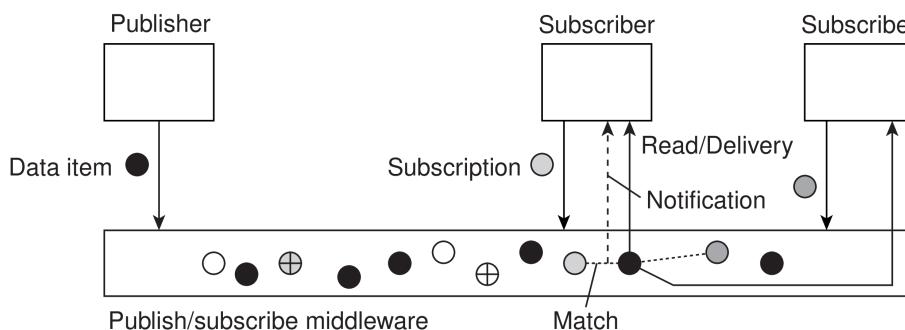
Synchronization decoupling



Eugster et al., [The many faces of publish/subscribe](#), ACM Comput. Surv., 2003

Publish-subscribe: subscription

- Issue: how to match events
 - Assume events are described by **(attribute,value)** pairs



- Topic-based subscription**
 - Specify a “**attribute = value**” series
 - X Expressiveness is limited
- Content-based subscription**
 - Specify a “**attribute ∈ range**” series, i.e., subscribers specify filters
 - X May easily have serious scalability problems, why?

Choosing an architectural style

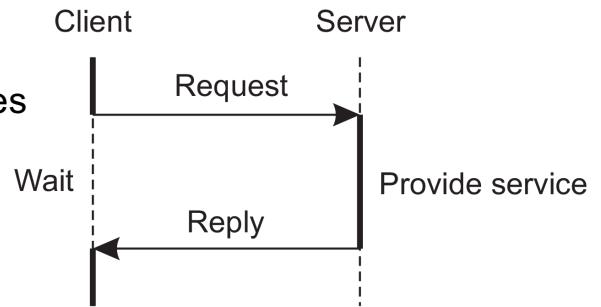
- **No single solution:** can tackle same problem with different architectural styles
- Choice depends often on **extra-functional requirements:**
 - Costs (resource usage, development effort needed)
 - Scalability and elasticity (effects of scaling and amount of available resources)
 - Performance (e.g., response time, latency)
 - Reliability and fault tolerance
 - Maintainability (extending system with new components)
 - Usability (ease of configuration and usage)
 - Reusability

System architecture of DS

- **Runtime instantiation** of DS software architecture
 - Which components?
 - How do they interact with each other?
 - Where to deploy them?
- Types of system architectures
 - Centralized architectures
 - Decentralized architectures
 - Hybrid architectures

Centralized system architectures

- Basic **client-server** model
 - Two groups: servers offer services and clients use services
 - Clients and servers can be on different machines
 - E.g., Web clients and servers
- **Request/reply** model
- Communication
 - based on message exchange
 - often *synchronous* and *blocking*
- Strong coupling: e.g., coexistence of interacting entities

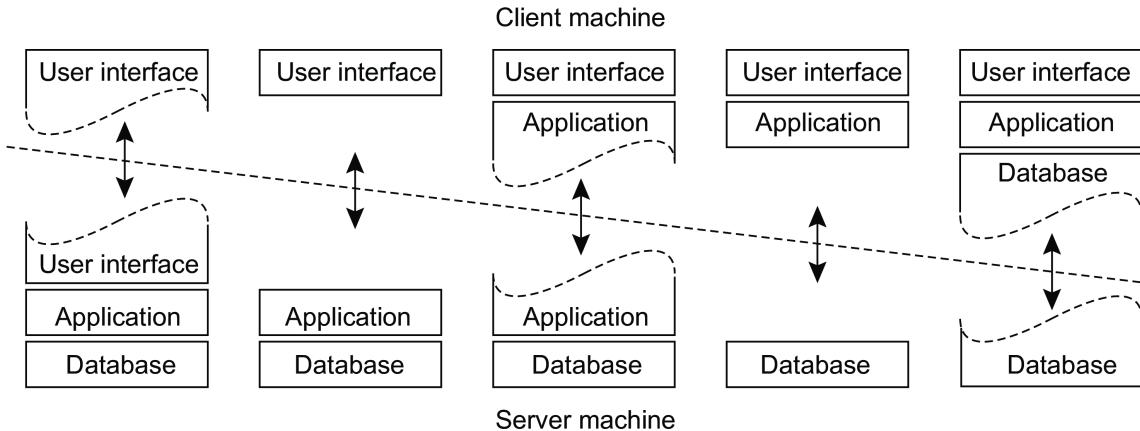


Multi-tiered client-server architectures

- How to map **logical levels (*layers*)** into **physical levels (*tiers*)**?
 - Two-tiered architectures
 - Three-tiered architectures
- Different configurations, depending on distribution of:
 1. presentation layer
 2. logic (aka application, business, processing) layer
 3. data layer

Multi-tiered client-server architectures

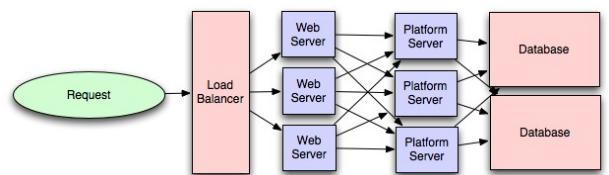
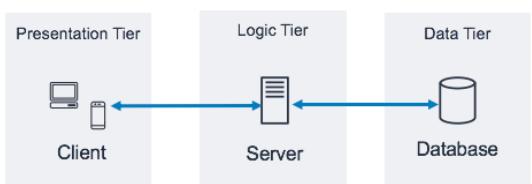
- Example: two-tiered configurations



- More than three tiers?

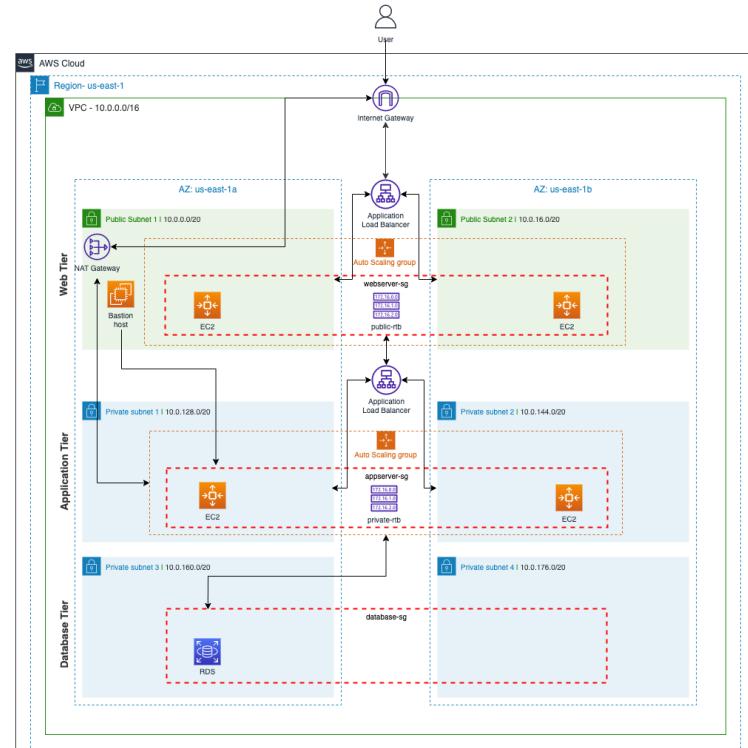
From multi-tiered architectures to...

- **Vertical distribution**
 - Divide distributed applications into 3 logical layers and run each layer on a different tier
- **Horizontal distribution**
 - Distribute each layer on multiple servers
 - Balance load among multiple servers through a **load balancer**
 - E.g.,: distributed Web cluster



Example: Web application in AWS

- Web application with horizontal distribution in AWS
 - **Elasticity**: tiers can scale out/in
 - **High availability**: replication in different availability zones
 - **Security**: tiers communicate with private IP



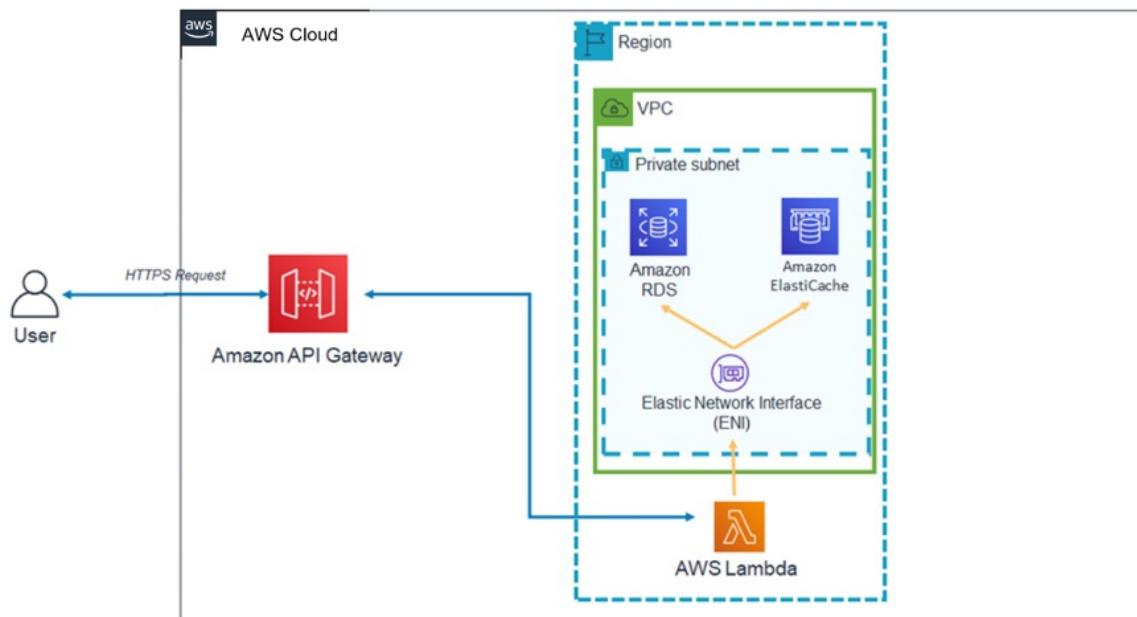
medium.com/@aaloktrivedi/building-a-3-tier-web-application-architecture-with-aws-eb5981613e30

Valeria Cardellini - SDCC 2023/24

32

Example: 3-tier serverless architecture

- Horizontal distribution is not visible: AWS Lambda has elastic scalability already built in



docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/three-tier-architecture-overview.html

Valeria Cardellini - SDCC 2023/24

33

Decentralized system architectures

- **Peer-to-peer (P2P) systems**
- P2P: class of systems and applications that use **distributed resources** to perform functions (even critical) in a **decentralized** way
 - “P2P is a class of applications that takes advantage of resources available at the edges of the Internet” (Clay Shirny, 2000)
- **Shared resources**: files, storage space, computing power, bandwidth
 - Give and receive resources from community of peers

P2P systems: features

- Peers are roughly **symmetric** in roles, privileges and responsibilities
 - **Autonomous** nodes located at network **edge**
 - **Hybrid** P2P system: differently from **pure** P2P system, some nodes are **super-peer** (have more functionalities than others)
- No centralized control
 - A peer behaves as client and server and shares resources and services (symmetric functionality: *servent* = server + client)
- Highly distributed
 - Up to hundreds of thousands of nodes
 - Highly dynamic and autonomous nodes
 - Node can enter or exit the P2P network at any time (**join/leave** operations)
 - Redundancy of information

P2P: applications

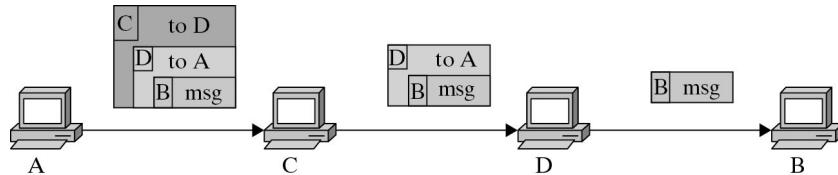
- Content distribution and storage
 - Content: file, video streaming, ...
 - Networks, protocols and clients for *file sharing* (P2P “killer application”): Gnutella, eMule, Kademlia, BitTorrent, uTorrent, ...
 - Video streaming: PPLive, ...
 - File storage: Freenet, ...
- Computing resource sharing
 - SETI@home (search for extraterrestrial intelligence), Folding@home (protein folding)
- Voice/video telephony
 - e.g., Chat/IRC, Instant Messaging, XMPP, Skype, ...
- Blockchain

P2P: challenges

- Heterogeneity in peer resources
 - Hardware, software and network heterogeneity
- Scalability
 - System scaling related to performance and bandwidth
- Location
 - Data location, data locality, network proximity, and interoperability
- Fault tolerance
 - Failure management
- Performance
 - Routing efficiency, load balancing, self-organization

P2P: challenges

- Free-riding avoidance
 - *Free-riders*: peers that are selfish and unwilling to contribute anything
- Anonymity and privacy
 - *Onion routing* for anonymous communications



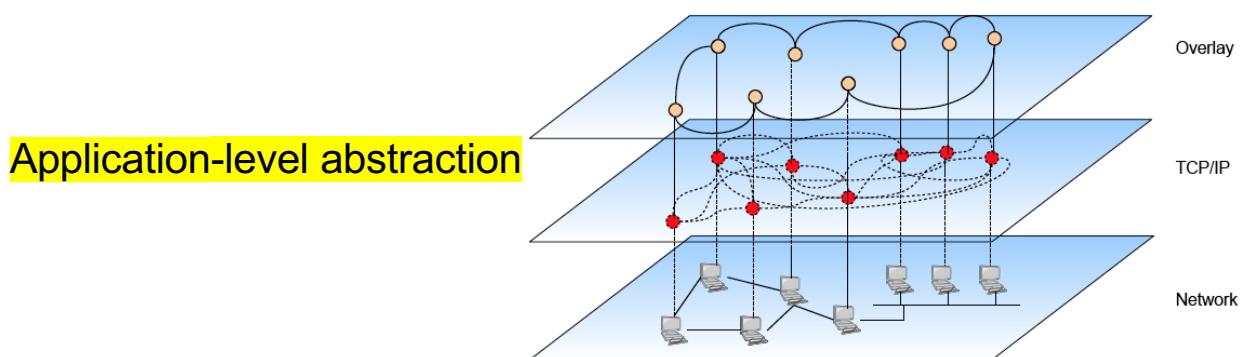
- Trust and reputation management
 - Lack of trust among peers who are strangers to each other
- Network threats and defense against attacks
- *Churn* resilience
 - Peers come, leave and even fail at random
 - Resources are dynamically added or removed

Main tasks of a P2P node

- Let's consider *file sharing*
- A P2P node performs the following operations:
 1. *Bootstrap*: how a new peer who intends to join a P2P system discover contact information for other peers in the network
 - Solutions: static configuration, pre-existing caches, well-known nodes
 2. *Resource lookup*: how to locate resources
 3. *Resource retrieval*: how to get localized resource
- We focus on resource lookup

P2P overlay

- P2P networks are commonly called overlays
- **Overlay network:** logical network connecting peers laid over the IP network
 - Based on underlying physical network
 - Logical links between peers, not corresponding to physical connections
 - Provides a resource location service by means of **application-level routing**



Overlay routing

- Basic idea:
 - The P2P system finds the path to reach a resource
- Compared to traditional routing
 - Resource: no network node address, but files, available CPUs, free disk space, ...
- We focus only on **routing**, not on interaction between peers to retrieve a localized resource
 - **Retrieval** typically occurs with a direct interaction between peers, e.g., using HTTP

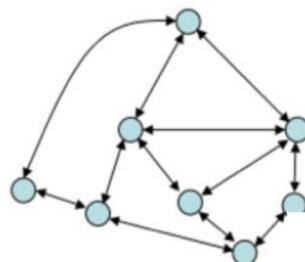
Tasks of overlay network

- Besides routing of requests to resources, an overlay network also allows to:
 - Insert and delete resources
 - Add and remove nodes
 - Identify resources and nodes
- How to identify resources?
 - **Globally Unique IDentifier (GUID)**: usually obtained by applying a secure hash function to some (or all) of the resource's state
- We also need to identify peers through a unique identifier
 - Again, usually computed through a hash function

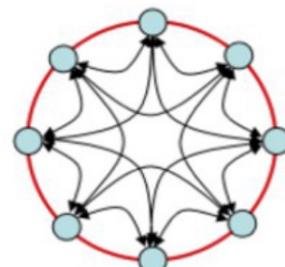
P2P overlay classification

- How to manage resources and nodes?
 - Depends on the overlay network type

Unstructured overlay networks



Structured overlay networks

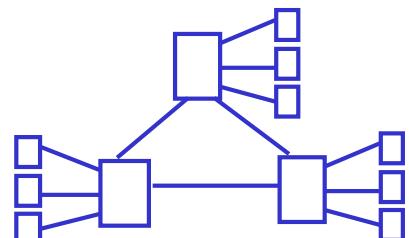
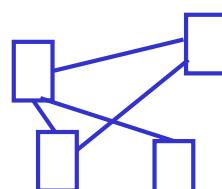
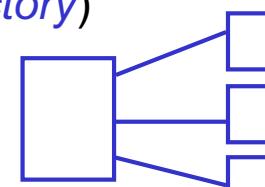


Unstructured overlay network

- Overlay network built on **random graphs**
 - No structure of overlay network *by design*
 - Peers are arbitrarily connected: each peer joins the network following some simple and local rule
 - A joining node contacts a set of neighbors, somehow selected
 - No control over resource placement on nodes
- Goal: manage nodes with highly dynamic behavior
- Examples: Gnutella, Bitcoin
- Pros and cons:
 - ✓ Easy maintenance because insertion and deletion of nodes and resources are easily managed
 - ✓ Highly resilient
 - ✗ High lookup cost: resource location is complicated by the lack of structure

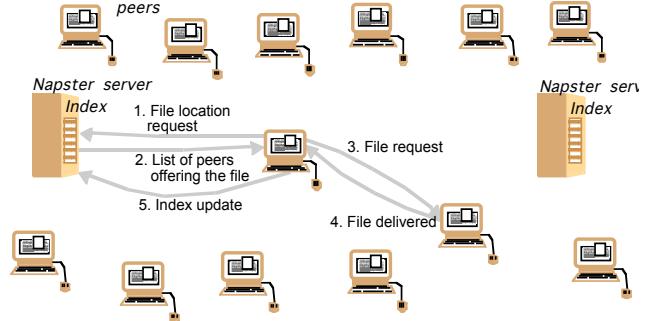
Unstructured overlay network: routing

- Let's classify unstructured overlays according to **distribution** of peer-resource index (*directory*)
- **Centralized** unstructured overlay:
central directory (e.g., Napster)
- **Decentralized** unstructured overlay:
distributed directory (e.g., Gnutella)
- **Hybrid** unstructured overlay: semi-centralized directory
 - Routing limited to *super-peers*



Centralized unstructured overlay

- **Directory server** responsible for resource-peer index:
 $\text{lookup}(\text{resource name}) \rightarrow \{\text{list of peers}\}$
- ✓ Simple: search is centralized on a single directory server
- ✓ Directory server is a single point of control: provides definitive answer to query
- ✗ Expensive management of centralized directory
- ✗ Single directory server: performance bottleneck (limited scalability) and SPOF (technical and legal reasons)



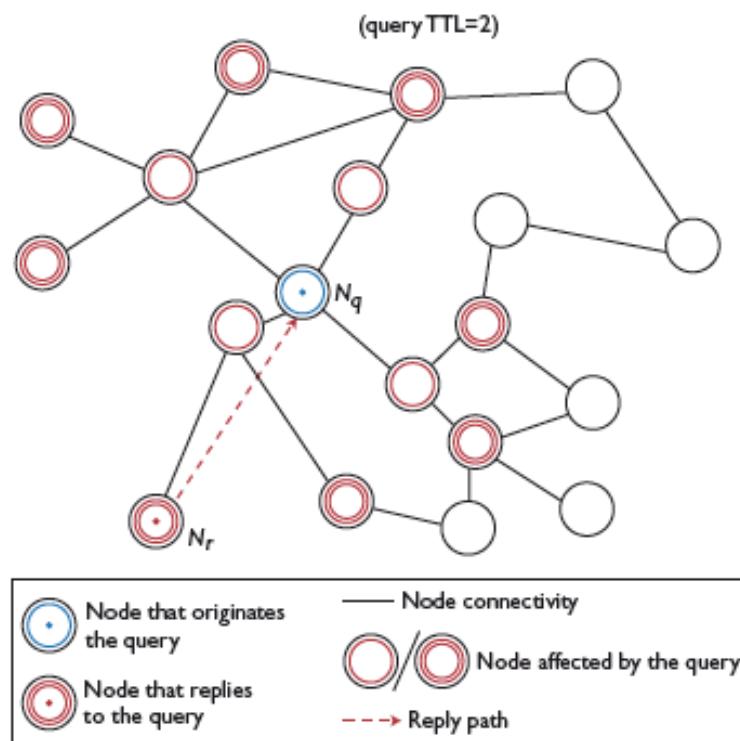
Decentralized unstructured overlay

- Fully decentralized approach to lookup resources
- How to lookup resources?
 - Query flooding
 - Random walk
 - Gossiping (upcoming lesson)

Query flooding

- Originator sends lookup query to its **neighboring peers**
- Each peer either responds if it owns the resource or **forwards** the query to its neighbors (excluding the neighbor from whom it received the query)
- Optimization #1: avoid indefinite query forwarding
 - Use **TTL** to limit the search range
 - At each forwarding, decrease TTL by 1; when TTL=0, the query is no longer forwarded
- Optimization #2: avoid cyclic paths
 - Assign a **unique query ID** so to not process the query again
- Lookup cost: $O(N)$, N = number of nodes in P2P network

Query flooding: example



Query flooding

- Options for sending back the response to the query originator:
 - Direct routing: from the peer that owns the resource to the query originator
 - Backward routing
- Backward routing
 - Response is forwarded back along the same path followed by the query until it reaches its originator
 - Query ID can be used to locate the backward path
 - Which pros wrt direct routing?

Query flooding: cons

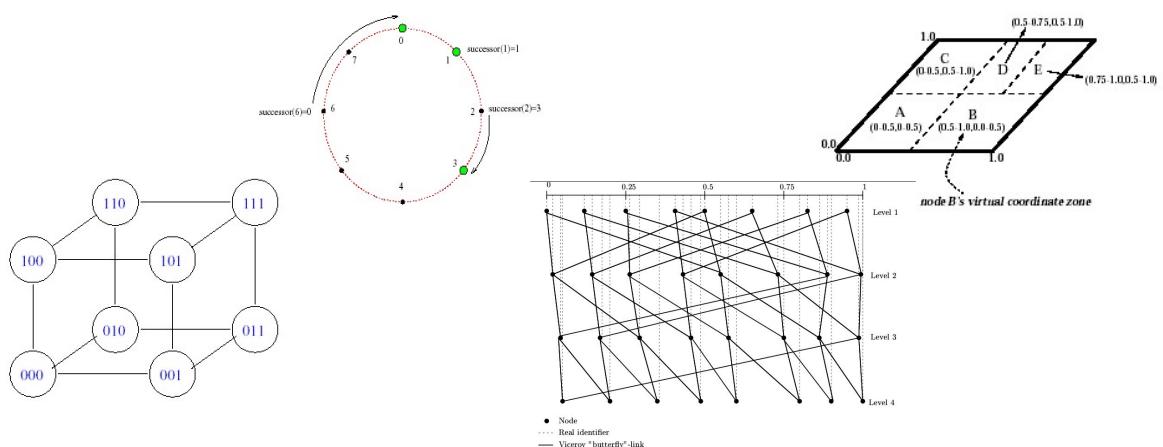
- Communication overhead
 - Large number of messages
 - Unsuccessful messages consume network bandwidth
- High lookup cost
 - How to choose the right TTL value?
- Denial-of-service attacks are possible
 - Black-hole nodes in case of congestion
- False negatives
 - No guarantee that (all) nodes that own the resource will be queried
- Lack of relationship between overlay and physical network topology
 - How far apart are “neighbor” peers?

Random walk

- In standard random walk, the originator forwards the lookup query to one randomly chosen neighbor
 - This neighbor randomly chooses one of its neighbors and forwards the request to that neighbor
 - This procedure continues until the resource is found
- With respect to flooding
 - ✓ Message traffic is cut down
 - ✗ Lookup time increases
- To decrease lookup time, the querying peer can start k random walks simultaneously
 - With k random walks the originator forwards k copies of the query to k randomly selected neighbors
 - Then, each request takes its own random walk

Structured overlay networks

- Lookup query is forwarded using a well-defined set of information about other peers in the network
- The overlay network is structured
 - Constraints on how resources and peers are positioned on the network
 - Overlay network topology: ring, tree, hypercube, grid, ...

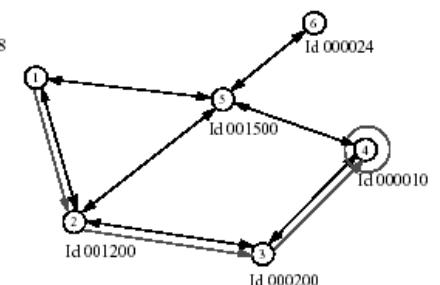


Structured overlay networks

- Goals: improve scalability by lowering lookup cost and reduce communication overhead with respect to unstructured overlays
 - Efficient key-based resource lookup
 - Overlay structure keeps lookup cost limited
 - Complexity guarantees also for peer join and leave
- Cons: peer join and leave become more expensive operations
 - Topology structure must be maintained

Routing in structured overlays

- Basic ideas
 - Each peer is responsible for some resources and knows some peers according to the overlay structure
 - Each resource is assigned a GUID
 - Each peer is assigned a GUID
 - GUIDs are computed using a hash function
 - Same large identifier space used for peers and resources GUIDs
 - Lookup query is routed to the peer whose GUID is the "closest" to the resource GUID
 - Closest: according to some **distance metric**
- Routing is based on **Distributed Hash Table (DHT)**: a distributed key-value data store

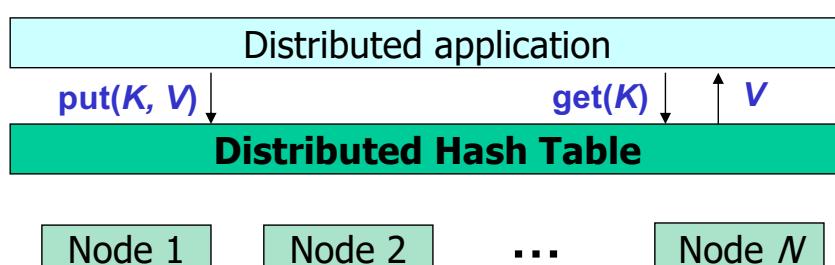


Distributed Hash Table

- Distributed abstraction of conventional hash table (HT) that maps keys to values
- Recall conventional HT
 - Table of $(key, value)$ tuples of size M
 - Key lookup: hash function maps keys to range $0 \dots M-1$
 - Lookup is very efficient: $O(1)$
 - Need to handle collisions because multiple keys may hash to same value
- DHT
 - Lookup similar to conventional HT: map the resource key to find the bucket (or slot) containing that resource
 - But **DHT buckets are spread** across multiple nodes (peers): how to map the resource key to find the peer responsible of the bucket?

Distributed Hash Table: API

- **Key-value pairs** (key K , value V) stored in DHT
 - K is the key that identifies the resource (contained in V) and corresponds to the resource GUID
- API for accessing DHT (common to many DHT-based systems)
 - **$V = get(K)$** : retrieve V associated with K from the node that stores it
 - **$put(K, V)$** : store the resource V in the node responsible for the resource identified by K
 - **$remove(K)$** : delete the reference to K and the associated V



Why might DHT design be hard?

- Decentralized: no central authority
- Scalable: low network traffic overhead
- Efficient: find items quickly (latency)
- Dynamic: nodes fail, new nodes join

Designing a DHT

- Resources and nodes are mapped onto the same identifier space using a hash function
 - GUID composed of m bits (usually $m = 128$ or 160)
 - E.g., SHA-1 cryptographic hash function
 - Hash function applied on metadata and/or data of resources (name, creation date, content, ...) and nodes
- Resources are partitioned among nodes: each node manages a portion of the resources stored in DHT
 - Each node is assigned a **contiguous portion** of keys and stores information about resources mapped to its own portion of keys
- Routing in DHT: given K , map it into the GUID of the node “closest” to K
- Resource replication can be exploited to improve availability

Issues related to DHTs

- Avoid hotspots by evenly distributing key responsibility among peers
- Avoid remapping all keys if DHT size changes (i.e., when peers join or leave)
 - **Consistent hashing** to address these issues
- Only directly support **exact-match** search
 - Since each resource is identified only by its key, to lookup for a resource we need to know its key
 - Easy to make exact-match search queries, e.g. based on resource name
 - Difficult and expensive to support more complex queries
 - E.g., wildcard query, range query
 - We will consider only exact-match

P2P systems based on DHTs

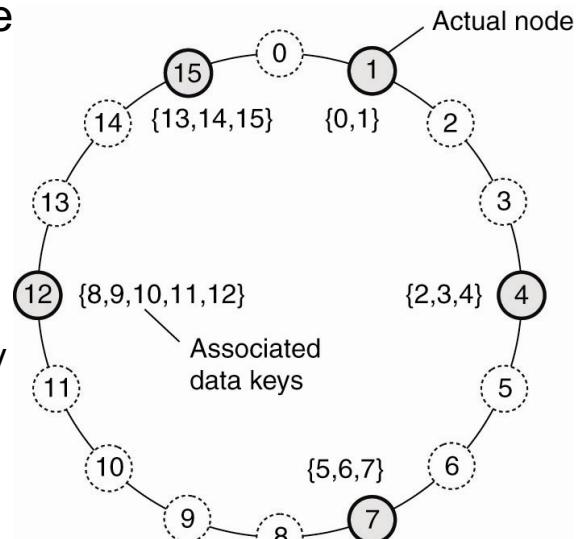
- Characterized by **high scalability** with respect to system size (i.e., N)
- Several proposals for DHT-based P2P systems
 - How do they differ?
 1. Definition of identifier space (and therefore **network topology**)
 2. Selection of peers to communicate with (i.e., **distance metric**)
 - More than 20 protocols and implementations for structured P2P networks, including:
 - **Chord** (MIT)
 - Pastry (Rice Univ., Microsoft)
 - Tapestry (Berkeley Univ.)
 - CAN (Berkeley Univ.)
 - Kademlia (NY Univ.)

Why Chord?

- Elegant resource lookup algorithm for DHT
- Efficient: $O(\log N)$ message per lookup
- Scalable: $O(\log N)$ state per node
- Robust: survives massive failures
- Simple to analyze

Chord <https://github.com/sit/dht/wiki>

- Nodes (peers) and resources are mapped onto a **ring** using **consistent hashing**
- Each node is responsible for the keys placed between itself and the preceding node in counter-clockwise direction
 - Resource with key K is managed by the node whose identifier is the smallest id $\geq K$
 - This node is called $\text{succ}(K)$, **successor** of key K
 - E.g., $\text{succ}(1)=1$, $\text{succ}(10)=12$
- Distance metric: based on **linear difference** between identifiers



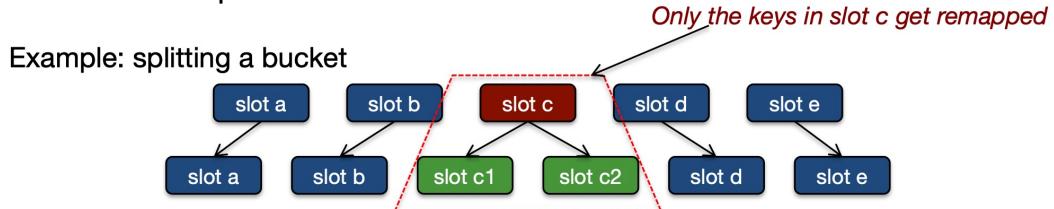
Stoica et al., [Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications](#), IEEE/ACM TON, 2003

Consistent hashing

- A special hashing technique
 - Both items (resources) and buckets (nodes) are uniformly mapped on the **same identifier space** (ring) using a standard hash function (e.g., SHA-1, MD5)
 - Each node manages an interval of consecutive hash keys, not a set of sparse keys
- Original devised by Karger et al. at MIT for distributed caching
[Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web](#), STOC 1997
- Gave birth to [Akamai](#)
- Some details and Java implementation
www.tom-e-white.com/2007/11/consistent-hashing.html
- Repurposed for new technologies and largely used in real systems, e.g., [Amazon Dynamo](#) and [memcached](#)

Chord: consistent hashing

- Consistent hashing is integral to Chord robustness and performance
 1. In case of DHT **resizing** (adding or removing a bucket): most keys will hash to the same bucket as before
 - Practical impact: peers can join and leave the network with minimal disruption



2. All buckets get roughly the same number of keys: **load balancing** among nodes

Chord: towards routing

- The simplest approach: lookup can be performed by traversing the ring, going one node at a time
- Can we do better than $O(N)$ lookup?
- Simple approach for great performance
 - Have all nodes know about each other
 - When a peer gets a query, it searches its table of nodes for the node that owns that key
 - Gives us $O(1)$ performance
 - Join/leave node operations must inform everyone
 - Maybe not a good solution if we have lots of peers (large tables)
- Chord uses a compromise to avoid large tables at each node: **finger table**
 - A partial list of nodes, progressively more distant

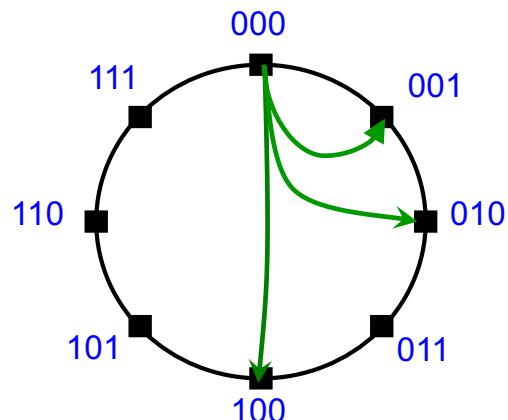
Chord: finger table

- **Finger table (FT)**: routing table of each node
 - FT has m rows, with $m = \#$ GUID bits
 - If FT_p is the FT of node p , then $FT_p[i] = succ(p + 2^{i-1}) \bmod 2^m$, $1 \leq i \leq m$
 - $succ(p+1), succ(p+2), succ(p+4), succ(p+8), succ(p+16), \dots$
- Example with $m=3$
 - Finger table of node 0?

$$FT_0[1]=0+2^0=1$$

$$FT_0[2]=0+2^1=2$$

$$FT_0[3]=0+2^2=4$$



Chord: FT's characteristics

- Each node stores information about only a small number of other nodes
 - Only m rows
- Each node knows more about nodes closely following it than about nodes farther away
- However, a node's FT generally does not contain enough information to directly determine the successor of any key
 - We need a routing algorithm to map each key K into $\text{succ}(K)$

Chord: routing algorithm

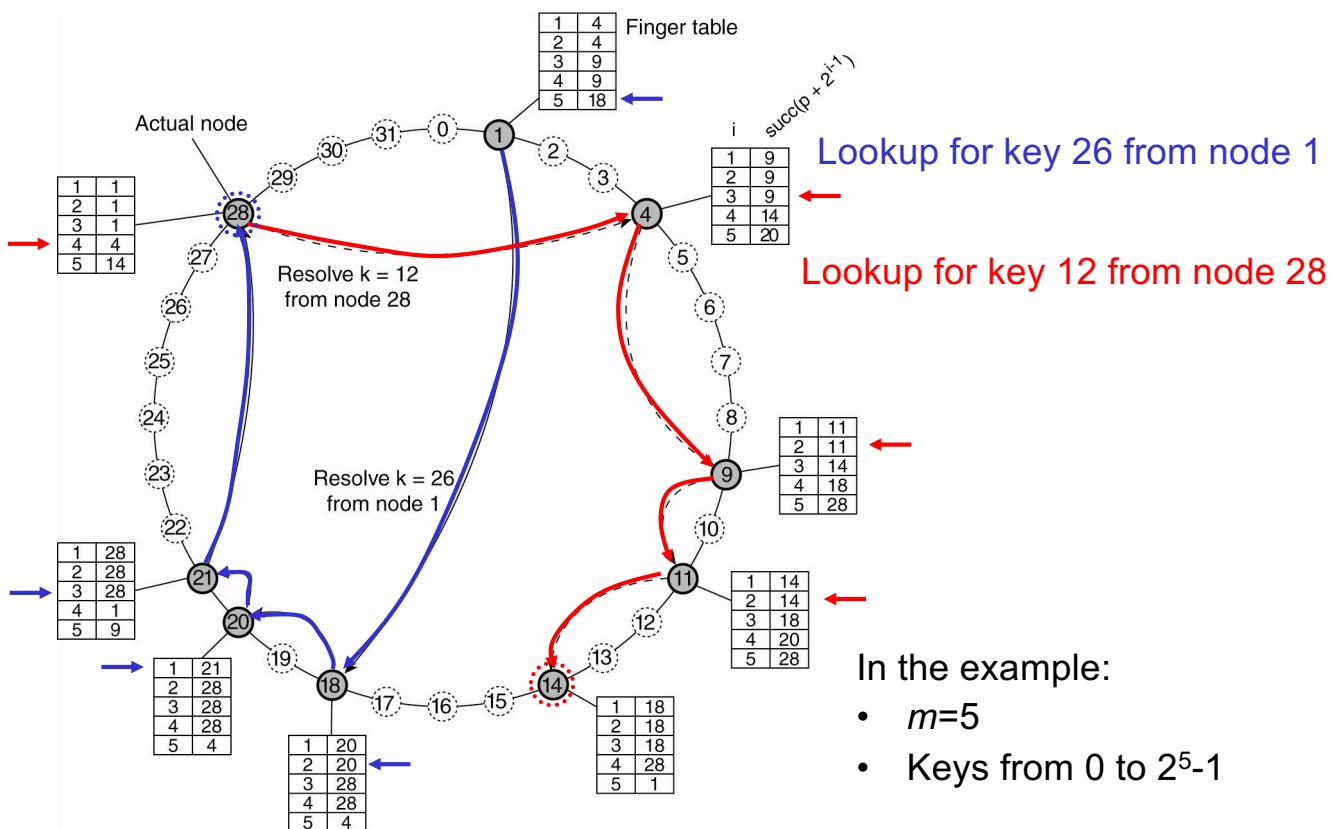
- How to map key K into $\text{succ}(K)$ starting from node p
 - If K belongs to the ring portion managed by p , lookup ends
 - If $p < K \leq FT_p[1]$, p forwards the request to its successor
 - Else p forwards the request to node q with index j in FT_p by considering the clockwise ordering

$$FT_p[j] \leq K < FT_p[j+1]$$

q is the farthest node from p whose key is less than or equal to K

- Features
 - It quickly reaches the vicinity of the searched point, and then proceeds with gradually smaller jumps
 - Lookup cost: $O(\log N)$, being N the number of nodes
 - ... not as cool as $O(1)$ but way better than $O(N)$

Chord: routing algorithm example

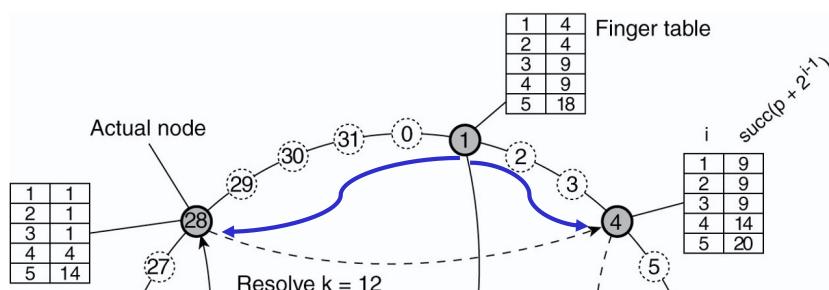


Valeria Cardellini - SDCC 2023/24

70

Chord: node join and leave

- In addition to successor pointer, each node also keeps the pointer to its predecessor (i.e., linked list) so to simplify ring maintenance operations
 - Predecessor of node p is the first node met in counter-clockwise direction starting at $p-1$
 - When a node joins or leaves, **successor and predecessor pointers should be updated**

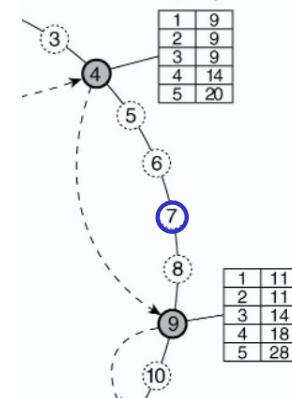


Valeria Cardellini - SDCC 2023/24

71

Chord: node join and leave

- When node p joins the overlay network, it has to find its place in the Chord ring:
 - Asks to a node to find its successor $\text{succ}(p+1)$ on the ring
 - Joins the ring linking to its successor and informs its successor of its presence
 - Initialize its FT looking for $\text{succ}(p + 2^{i-1})$, $2 \leq i \leq m$
 - Informs its predecessor to update the FT
 - Transfers from its successor to itself the keys for which it becomes responsible
- Example: node 7 joins
 - Node 7 successor is node 9
 - Node 9 predecessor changes to node 7
 - Node 4 successor changes to node 7
 - Keys 5, 6 and 7 are transferred to node 7

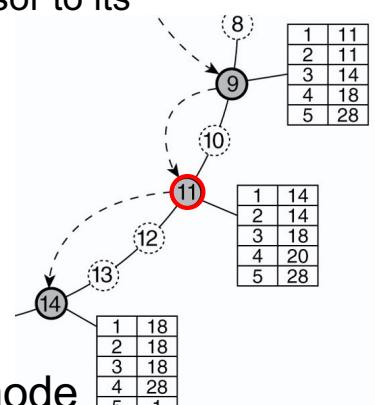


Valeria Cardellini - SDCC 2023/24

72

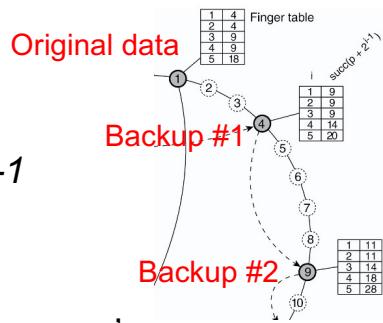
Chord: node join and leave

- When node p voluntary leaves the overlay network:
 - Transfers the keys it is responsible for to its successor
 - Updates the predecessor pointer held by its successor to the node that precedes p
 - Updates the successor pointer of its predecessor to its successor
- Example: node 11 leaves
 - Keys 10 and 11 are transferred to node 14
 - Node 14 predecessor changes to node 9
 - Node 9 successor changes to node 14
- Join/leave operations require $O(\log^2 N)$
- To keep the finger tables updated, each node periodically executes a ring stabilization procedure
 - Nodes can also leave the network abruptly because of failure



Chord: fault tolerance

- Nodes might crash
 - (K, V) data should be replicated
 - Create R replicas, storing each one at $R-1$ successor nodes in the ring
- Need to know multiple successors
 - A node needs to know how to find its successor's successor (or more)
 - Easy only if it knows all nodes!
 - When a node is back up, it needs to:
 - Check with successors for updates of data it owns
 - Check with predecessors for updates of data it stores as backups



Chord: summing up

- Pros
 - ✓ Simple and elegant
 - ✓ Load balancing
 - Keys are evenly distributed among nodes
 - ✓ Scalability
 - Efficient lookup operations: $O(\log(N))$
 - ✓ Robustness
 - Periodically update of nodes finger tables to reflect changes in the network
- Cons
 - ✗ Proximity in the underlying Internet is not considered
 - ✗ Expensive support for searches without exact matching
 - ✗ Original Chord ring-maintenance protocol is not correct
 - “Reasoning about Identifier Spaces: How to Make Chord Correct” <https://arxiv.org/pdf/1610.01140.pdf>

DHTs: summing up

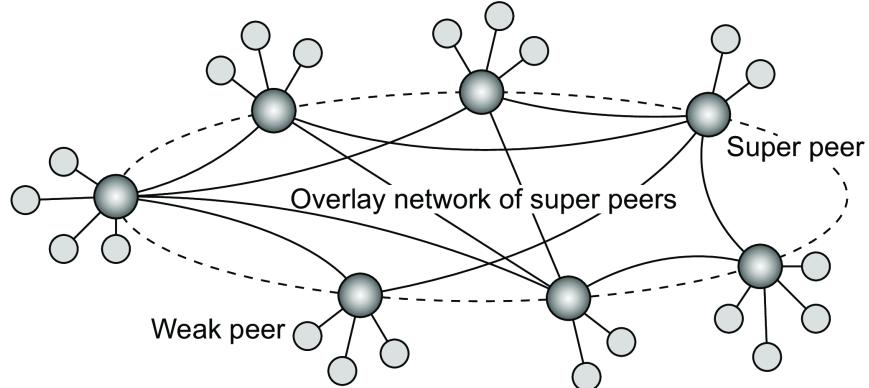
- DHTs in retrospective
 - Seem promising for finding data in large P2P systems
 - Decentralization is good for load balancing and fault tolerance
 - But: security problems are difficult
 - But: churn is a problem, particularly if $\log(N)$ is big
 - DHTs have not had the hoped-for impact
- However, DHTs got right for
 - Consistent hashing: elegant way to spread load across machines (e.g., used in Amazon Dynamo, Cassandra)
 - Incremental scalability: add nodes, capacity increases
 - Replication for high availability, efficient recovery after node failures
 - Self-management: minimal configuration
 - No single server to shut down/monitor

Hybrid architectures

- So far we have considered centralized and decentralized architectures
- In hybrid architectures, elements from centralized and decentralized organizations are combined
- Goal: take the benefits of both
- 3 examples of hybrid architectures (with different degree of decentralization)
 - Super-peer network
 - BitTorrent
 - Blockchain

Hybrid architectures: super-peer network

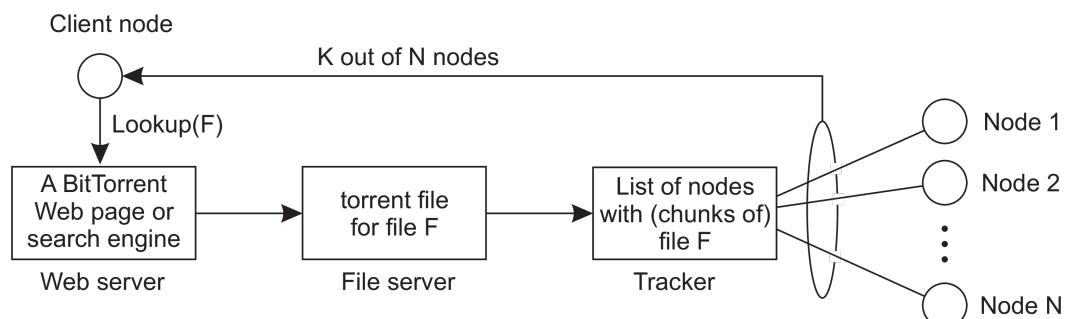
- It is sometimes sensible to break the symmetry in pure P2P networks: **super peers**
- Super peers (index servers) improve lookup performance



- Issues to address
 - Static or dynamic association of peer-super peer
 - How to select super peers

Hybrid architectures: BitTorrent

- Unstructured P2P system for file sharing
- Steps to search for file F



1. User clicks on download link
 - BT client gets **torrent file** containing **tracker** reference
 - Tracker: a server keeping an accurate account of active nodes that have (chunks of) F; bootstrapping node for the torrent
2. BT client contacts tracker
 - Tracker replies with a list of peers who have (chunks of) F

Hybrid architectures: BitTorrent

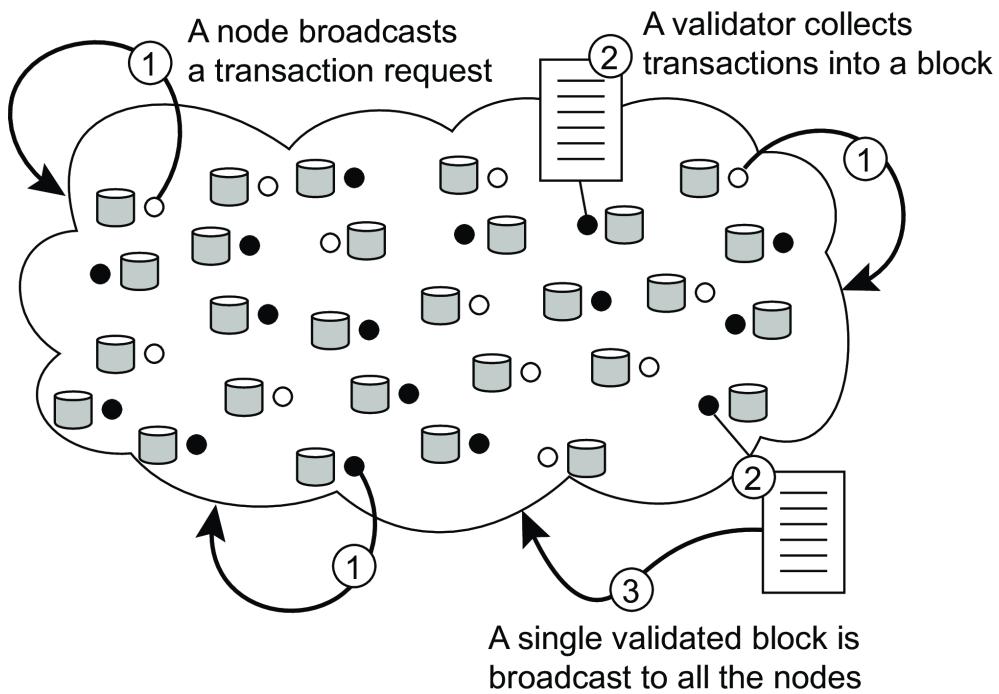
3. BT client downloads chunks of F from peers, joining a **swarm** of downloaders, who in parallel get file chunks but also distribute downloaded chunks amongst each other
 - BitTorrent incentivizes peers to exchange data
 - Chunk selection based on **rarest piece first**
 - Bandwidth allocation based on **tit-for-tat**
 - **Rarest piece first**
 - Chunks that are most uncommon in the network are preferably selected for download
 - Goal: make file exchange more robust against node churn
 - **Tit-for-tat**
 - Peers decide to whom they upload data based on downloaded data from a peer
 - Goal: prevent peers from only downloading without providing any resources to others

Blockchain

- A transaction (e.g., Alice transfers €10 to Bob) needs to be validated and then stored for auditing purposes
 - Validation: verify that transaction is legal (not malicious, no double spending, ...)
 - **How to validate transactions and where to store transactions?**
- Which kind of transactions?
 - Not only transfer of cryptocurrency (e.g., Bitcoin)
 - Also identification documents, resource usage and allocation, electronic voting, health records, etc.
- A blockchain provides a kind of collaborative **data store of transactions** replicated among **untrusted peers** and guarantees a consistent view of all transactions by peers
 - A type of **distributed ledger**

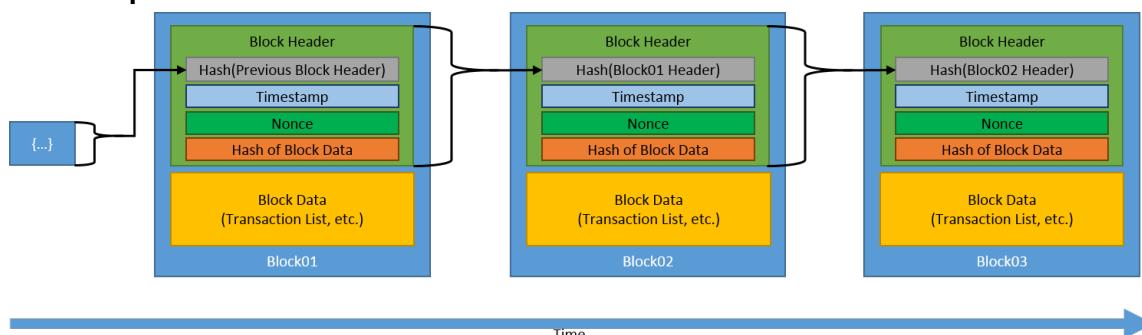
Blockchain: working principle

- Each peer stores a local replica of the ledger 



Blockchain: blocks

- Transactions are grouped into **blocks**
 - Block: header + body (set of transactions)
- Blocks are organized into an **unforgeable append-only chain**
 - A block is connected to the previous one by including a unique identifier (hash) based on previous block
 - Changing a block invalidates all subsequent blocks
- Each block in the blockchain is **immutable** \Rightarrow massive replication

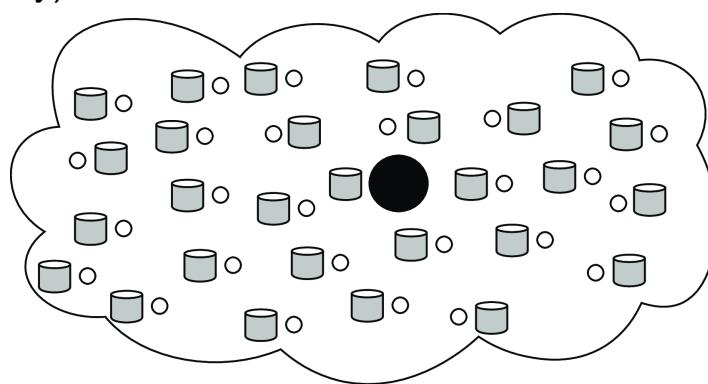


Blockchain: the key aspect

- Which validator is allowed to append a block of validated transactions to the chain?
- Deciding on which validator can move ahead requires (**distributed**) consensus

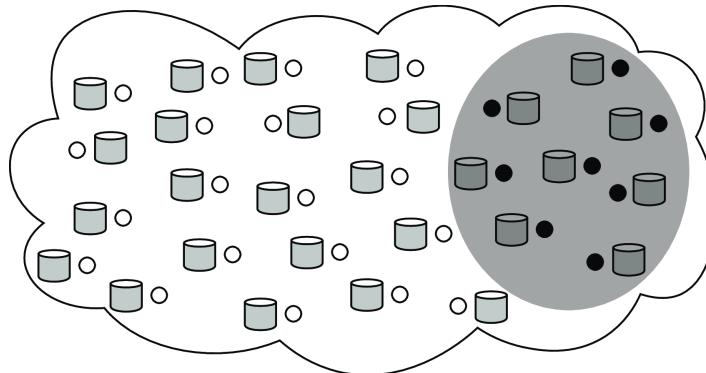
Appending a block: (distributed) consensus

- Centralized solution
 - A trusted **single** entity decides on which validator can go ahead and append a block
- X** Does not fit the design goals of blockchain (no central authority)



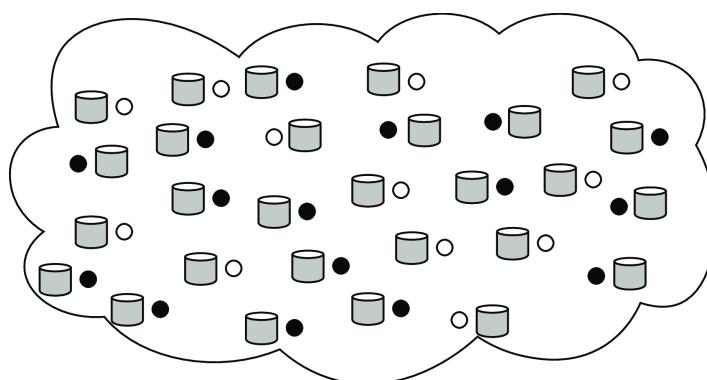
Appending a block: distributed consensus

- Distributed solution (**permissioned** blockchain)
 - 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 **2/3** behave according to their specifications
 - In practice, only a few tens of servers can be accommodated



Appending a block: distributed consensus

- Decentralized solution (**permission-less** blockchain)
 - All nodes collectively participate to validate transactions and engage in a **leader election**. Only the elected leader is allowed to append a block of validated transactions
 - E.g., Bitcoin, Ethereum
 - ✗ Large-scale, decentralized leader election that is fair, robust, secure, energy-efficient and so on, is far from trivial
 - We will study proof-of-work and proof-of-stake



References

- Chapter 2 and Section 6.2.3 of van Steen & Tanenbaum book
- [The many faces of publish/subscribe](#)
- [Looking up data in P2P systems](#)
- [Chord: a scalable peer-to-peer lookup protocol for Internet applications](#)
- [Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web](#)
- [Blockchain technology overview](#)