DHTs

Anne-Marie Kermarrec

You know you have a distributed system when the crash of a computer you have never heard of stops you from getting any work done. Leslie lamport

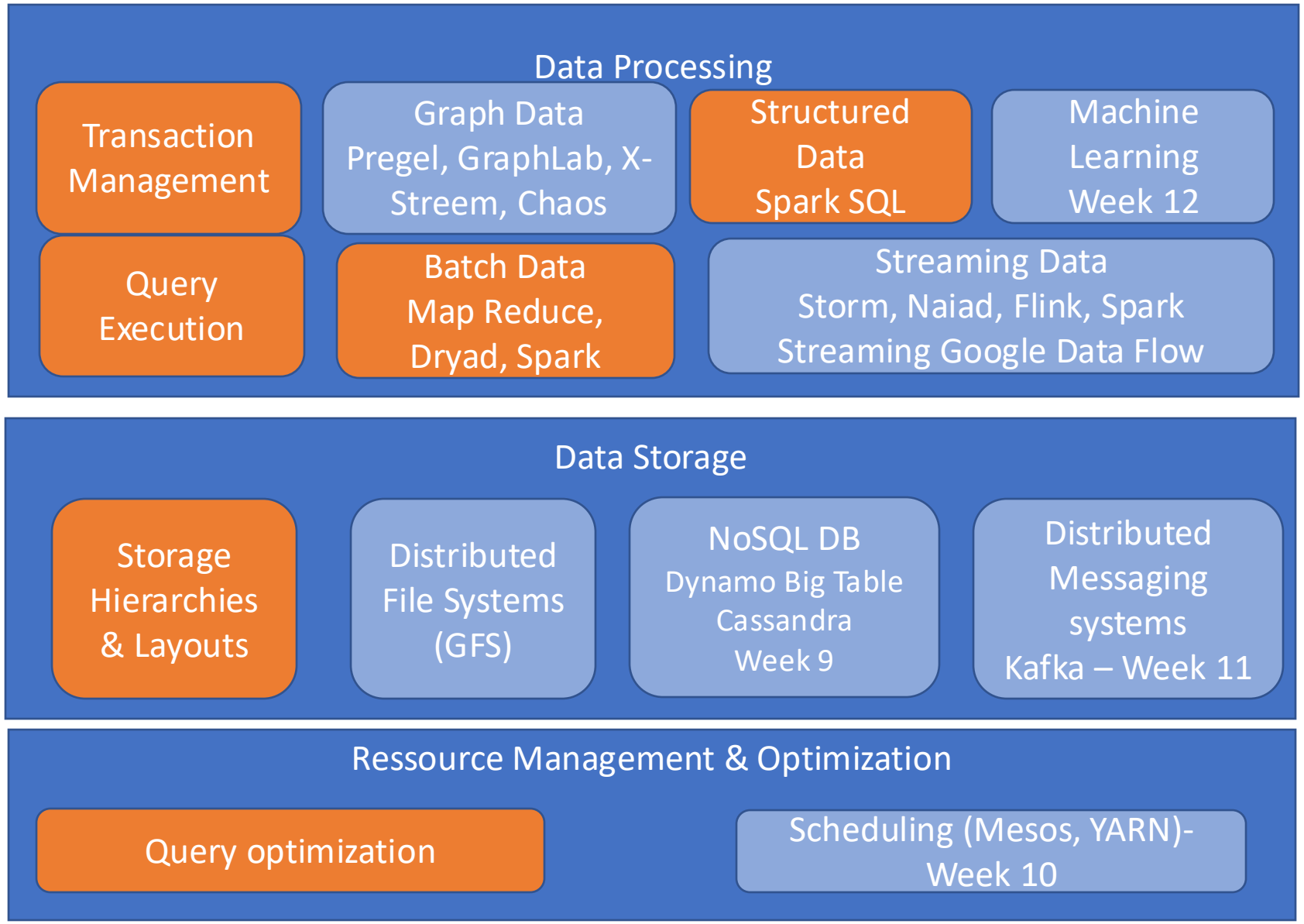
Where are we?

Gossip Protocols
Week 7

Consistency protocols
CAP Theorem
Week 9

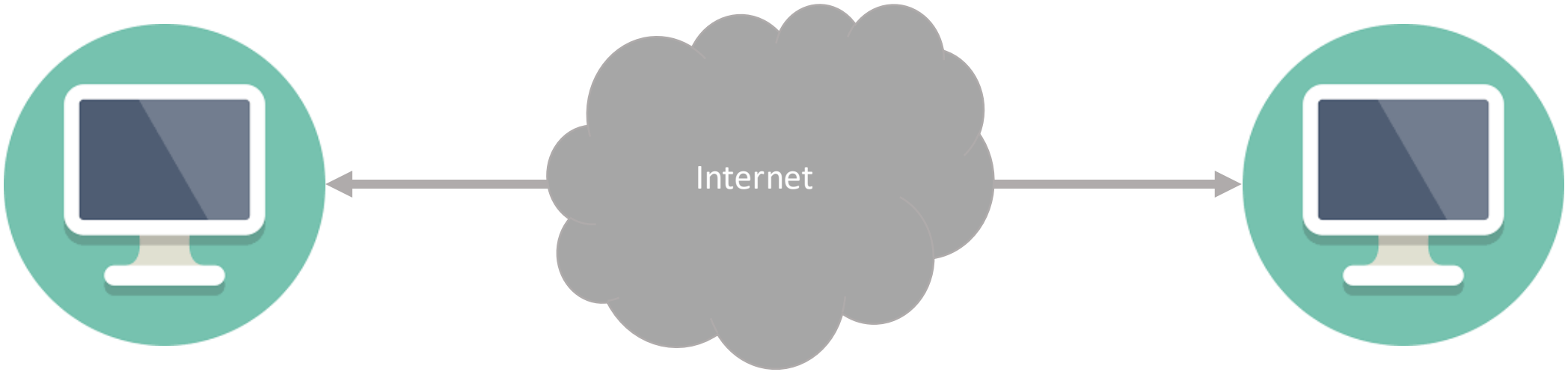
Distributed/decentralized systems
Week 8-12

Data science software stack

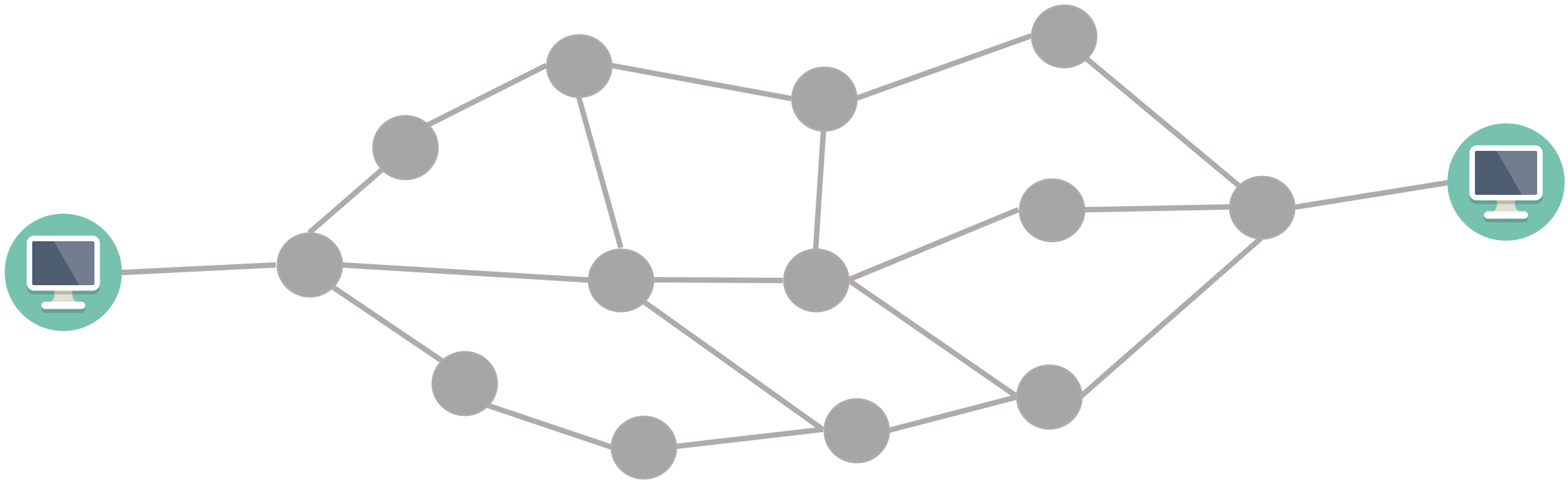


They had a dream

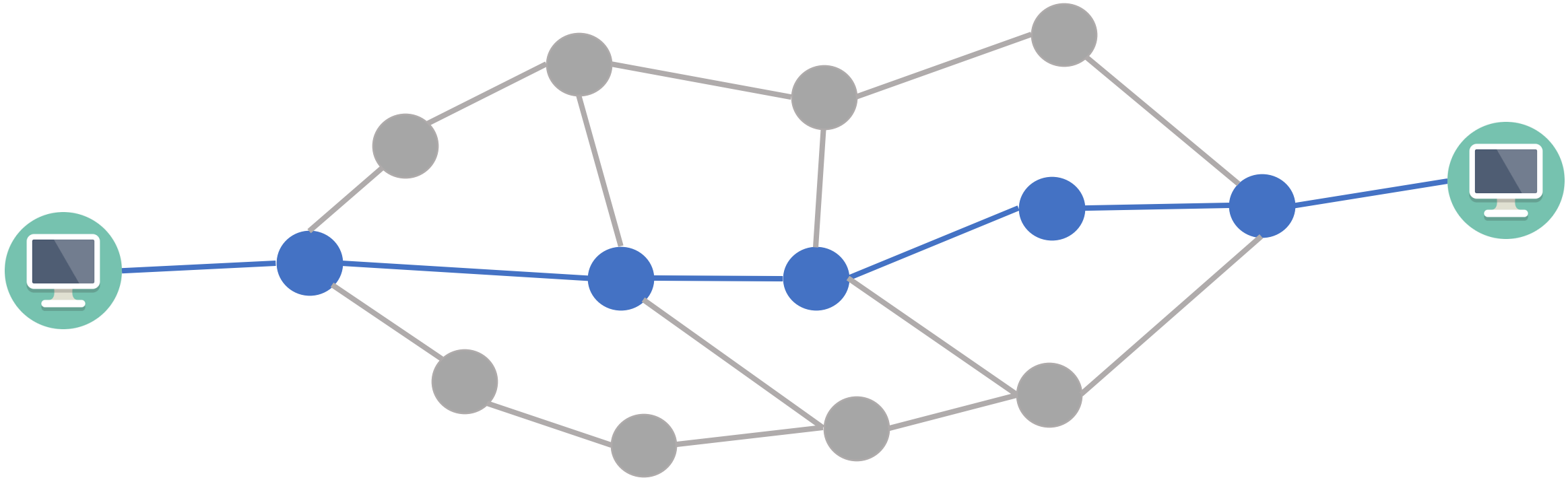
Share ressources among multiple machines



Internet: A collaborative decentralized system

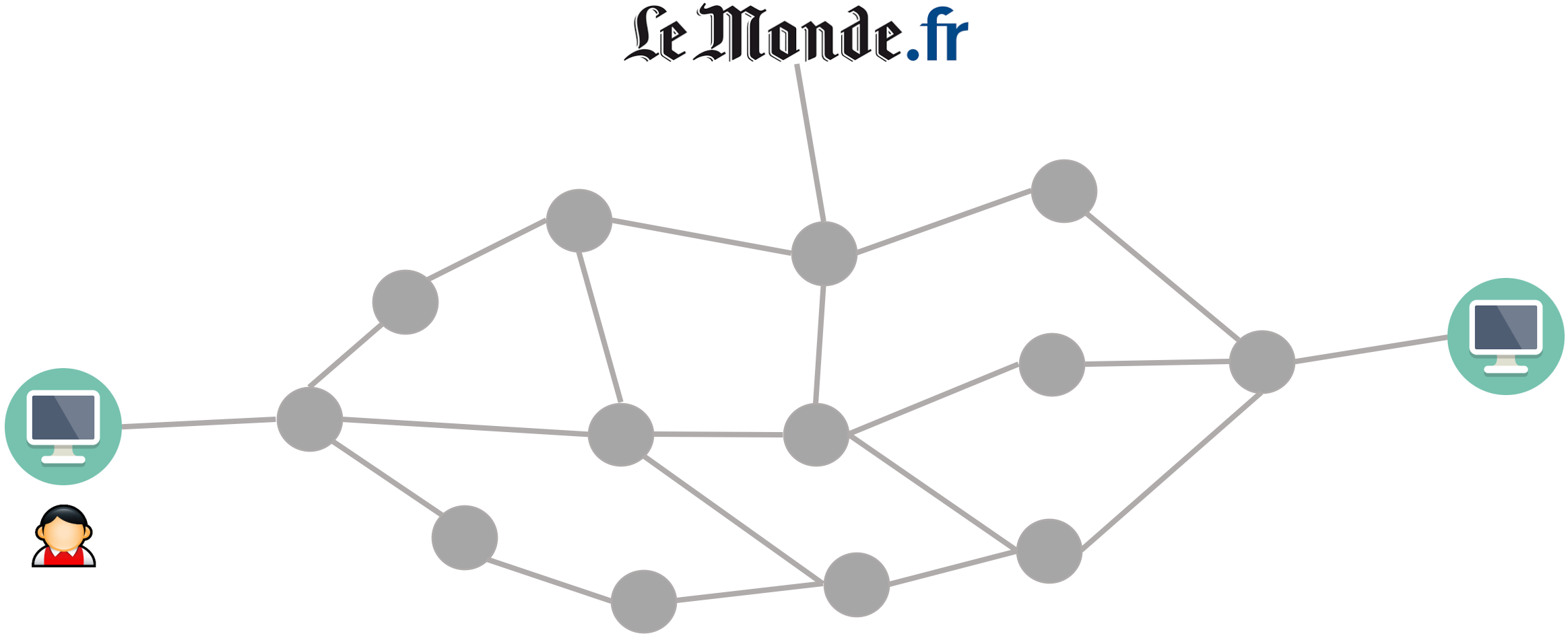


Internet: A collaborative decentralized system



He had a similar dream

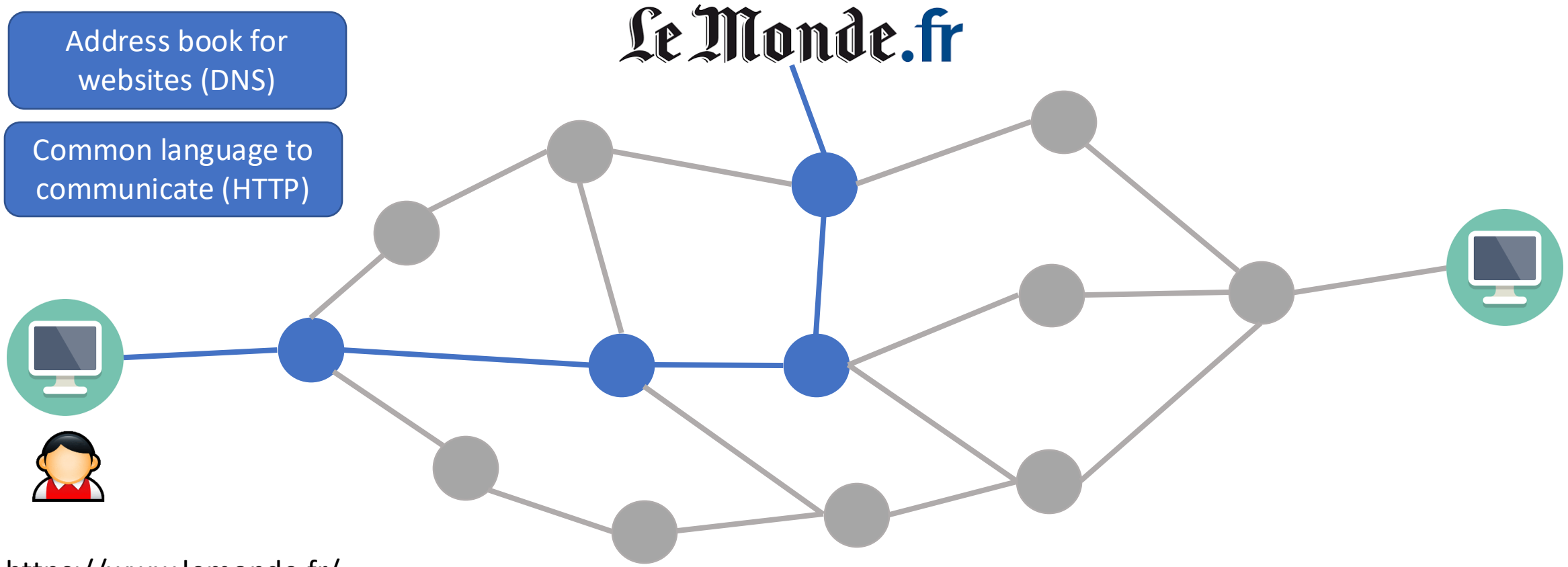
The Web: A decentralized system



The Web: A decentralized system

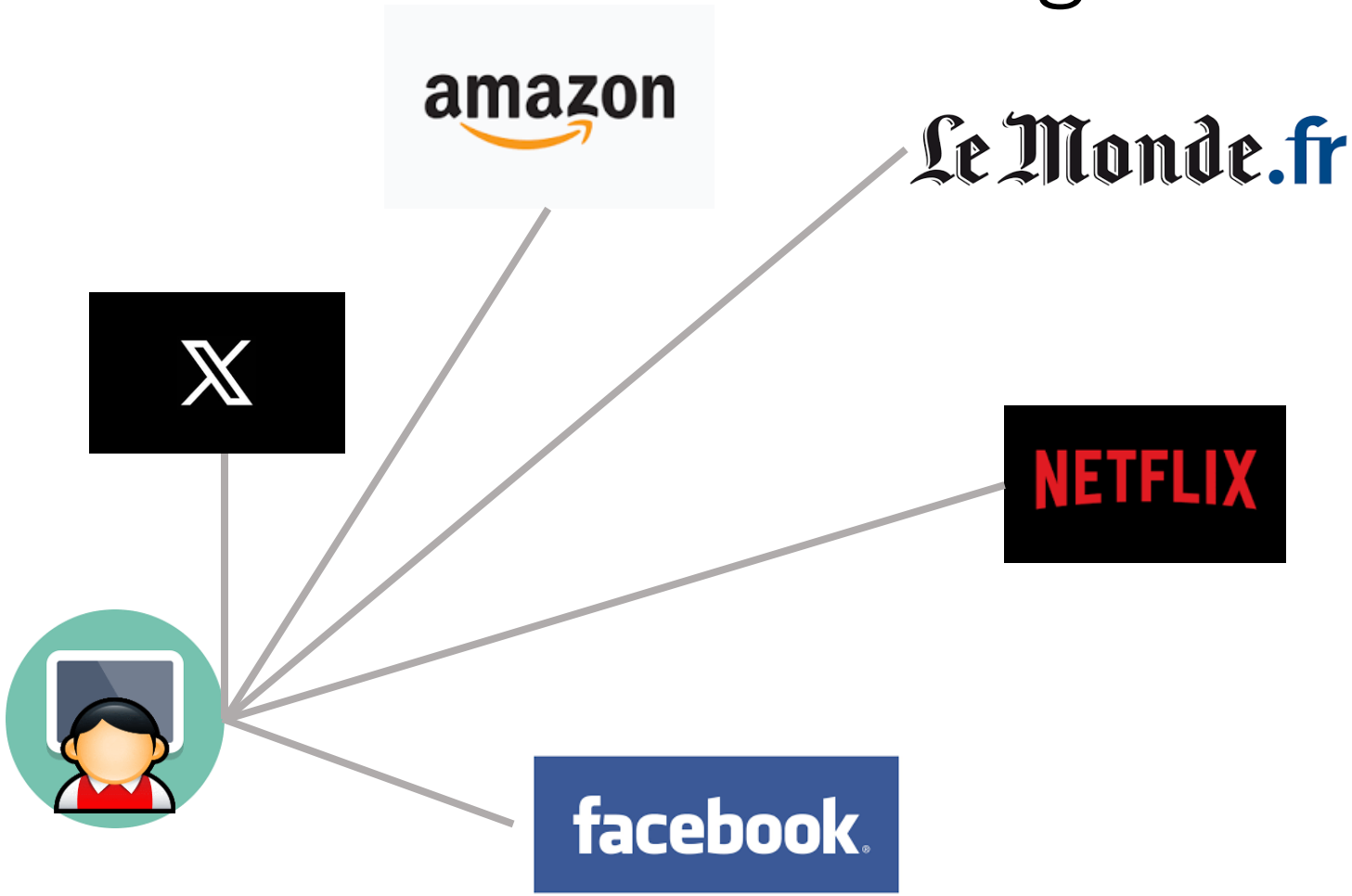
Address book for websites (DNS)

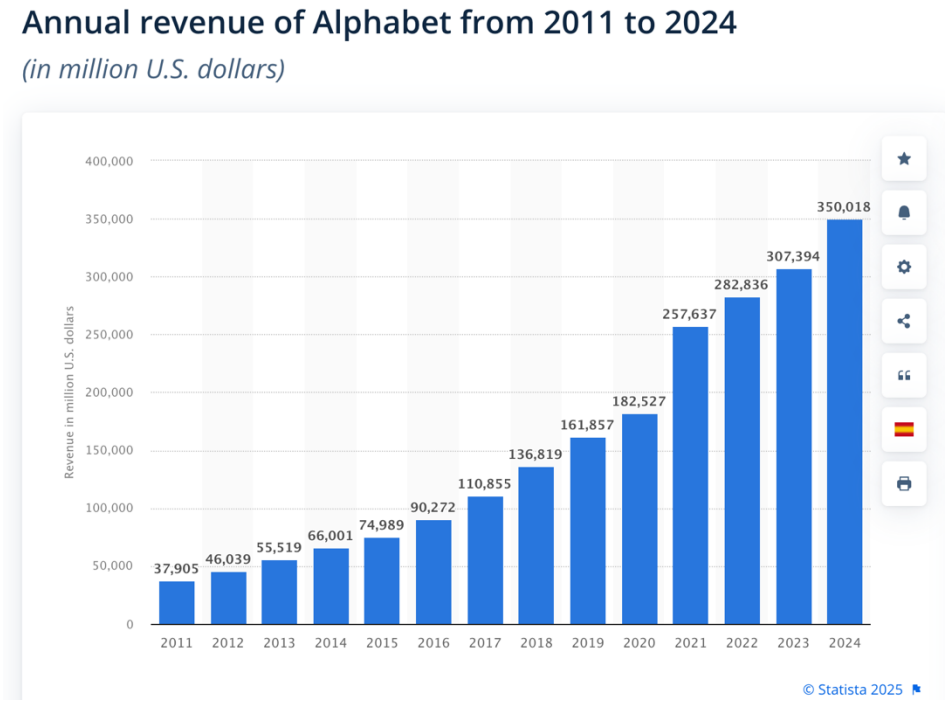
Common language to communicate (HTTP)



https://www.lemonde.fr/

The Web turned extremely centralized, now in the hand of a few giants





An increasingly popular alternative

- Citizen-friendly alternative
- Decentralized infrastructure
- Privacy-aware

(Fully) Distributed architectures

Aka P2P/decentralized

Distributed systems

- Use several machines
- Yet: appears to the users as a single computer: your FB wall, your Netflix Interface, etc
- Name it: The Web, The Internet, A wireless network, Bitcoin, A cloud Amazon EC2/S3 or Microsoft Azure, A datacenter

Characteristics

- Aggregate resources
- Scalability
- Speed
- Reliability

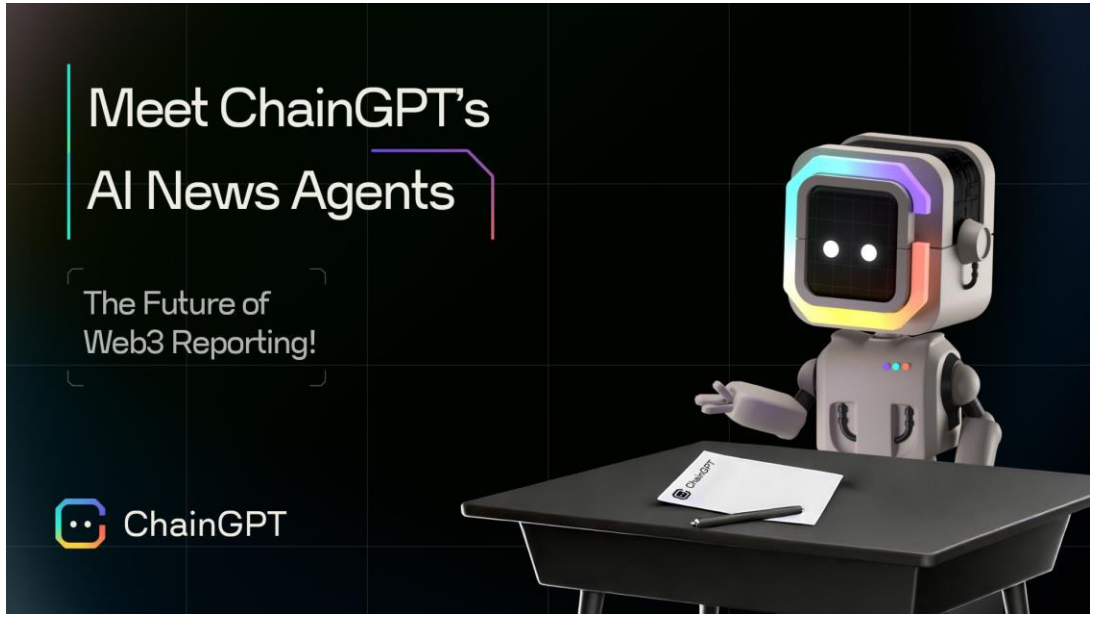
- At the price of
  - Complexity
  - Cost of maintenance

Why are they more complex?

- No global clock; no single global notion of the correct time (asynchrony)
- Unpredictable failures of components: lack of response may be due to either failure of a network component, network path being down, or a computer crash
- Highly variable bandwidth: from 16Kbps (slow modems or Google Balloon) to Gbps (Internet2) to Tbps (in between DCs of same big company)
- Possibly large and variable latency: few ms to several seconds
- Large numbers of hosts: 2 to several millions

P2P applications

- Large contributor of Internet traffic (~50% of the Internet traffic)
- Applications
  - Bitcoin & Blockchain
  - File sharing applications (Gnutella, Kazaa, Edonkey, Bit Torrent…)
  - Archival systems
  - Application level multicast
  - Streaming protocols
  - Telco applications (Skype)
  - Recommenders
  - Decentralized AI



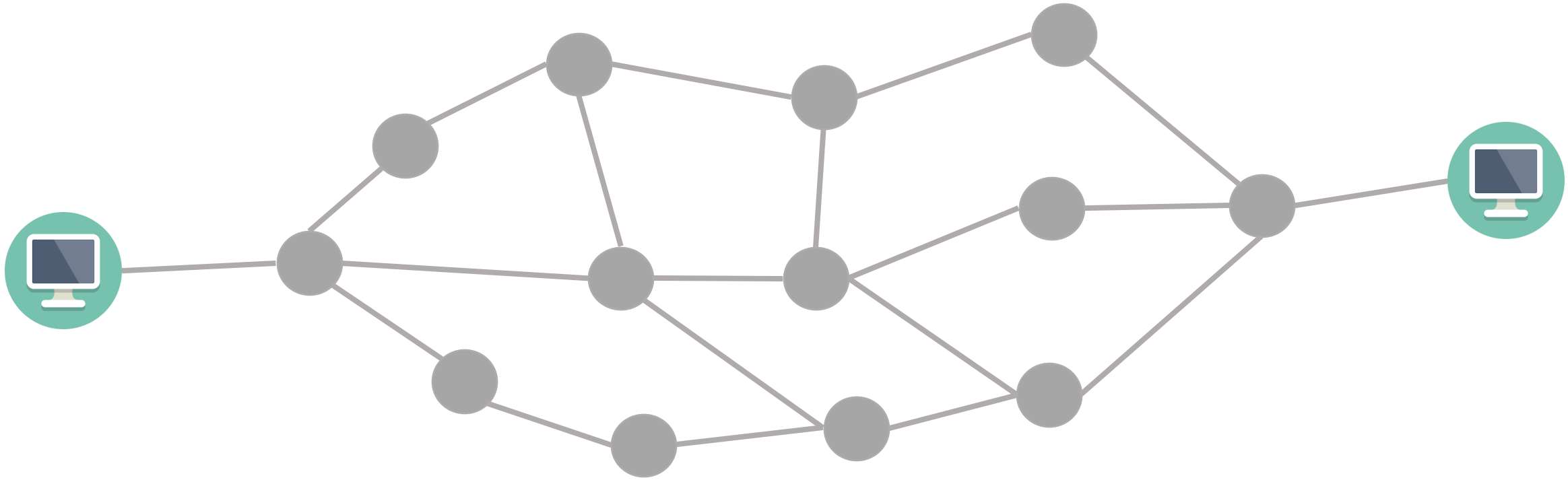
Why do I tell you about P2P systems ?

- First distributed systems that seriously focused on scalability
- P2P techniques are widely used in cloud computing systems
  - Key-value stores (e.g., Cassandra, Riak, Voldemort) use p2p hashing

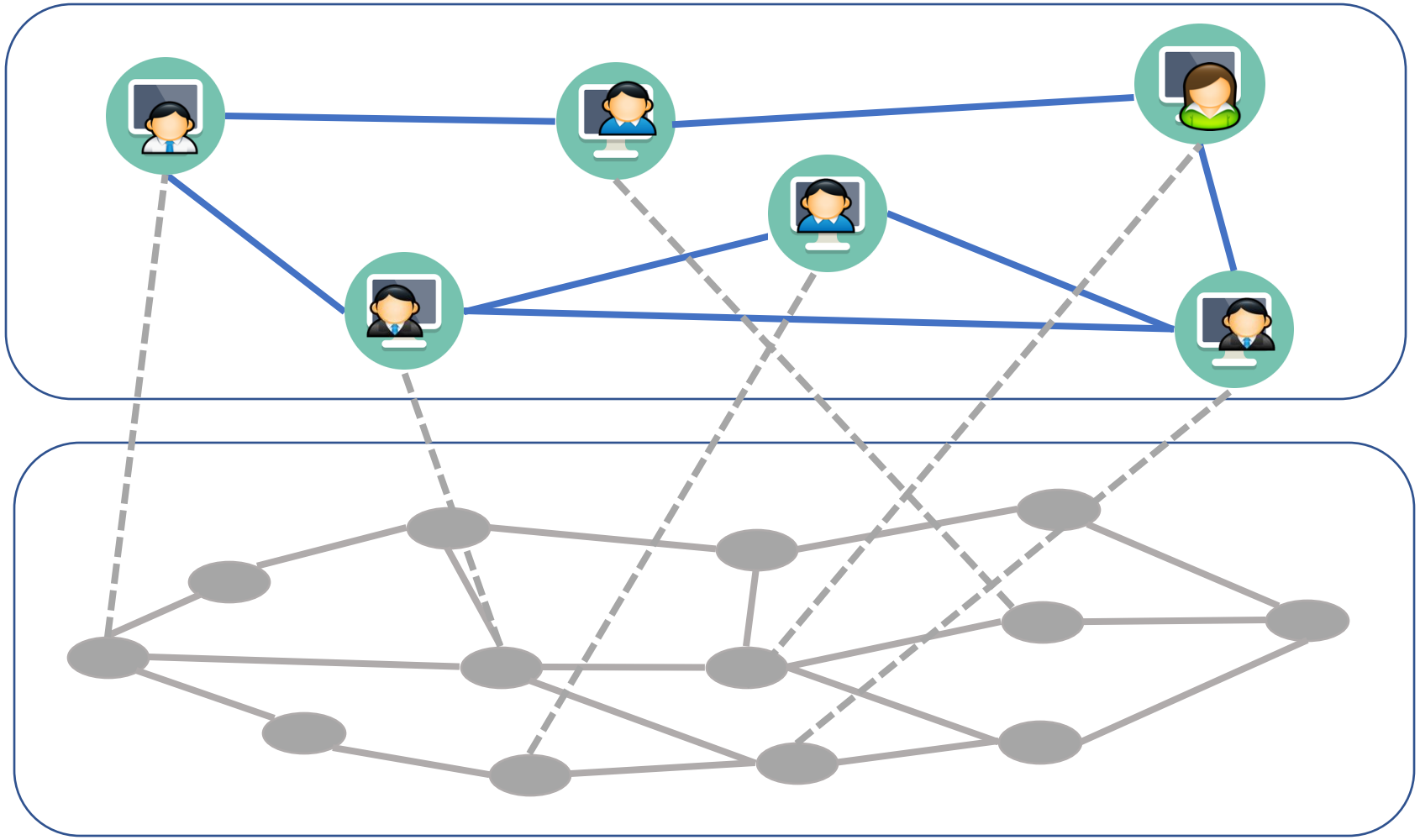
What makes P2P interesting?

- End-nodes are promoted to active components
- Nodes participate, interact, contribute to the services they use.
- Harness huge pools of resources accumulated in millions of end-nodes.
- Avoid a central/master entity
- Irregularities and dynamicity are treated as the norm

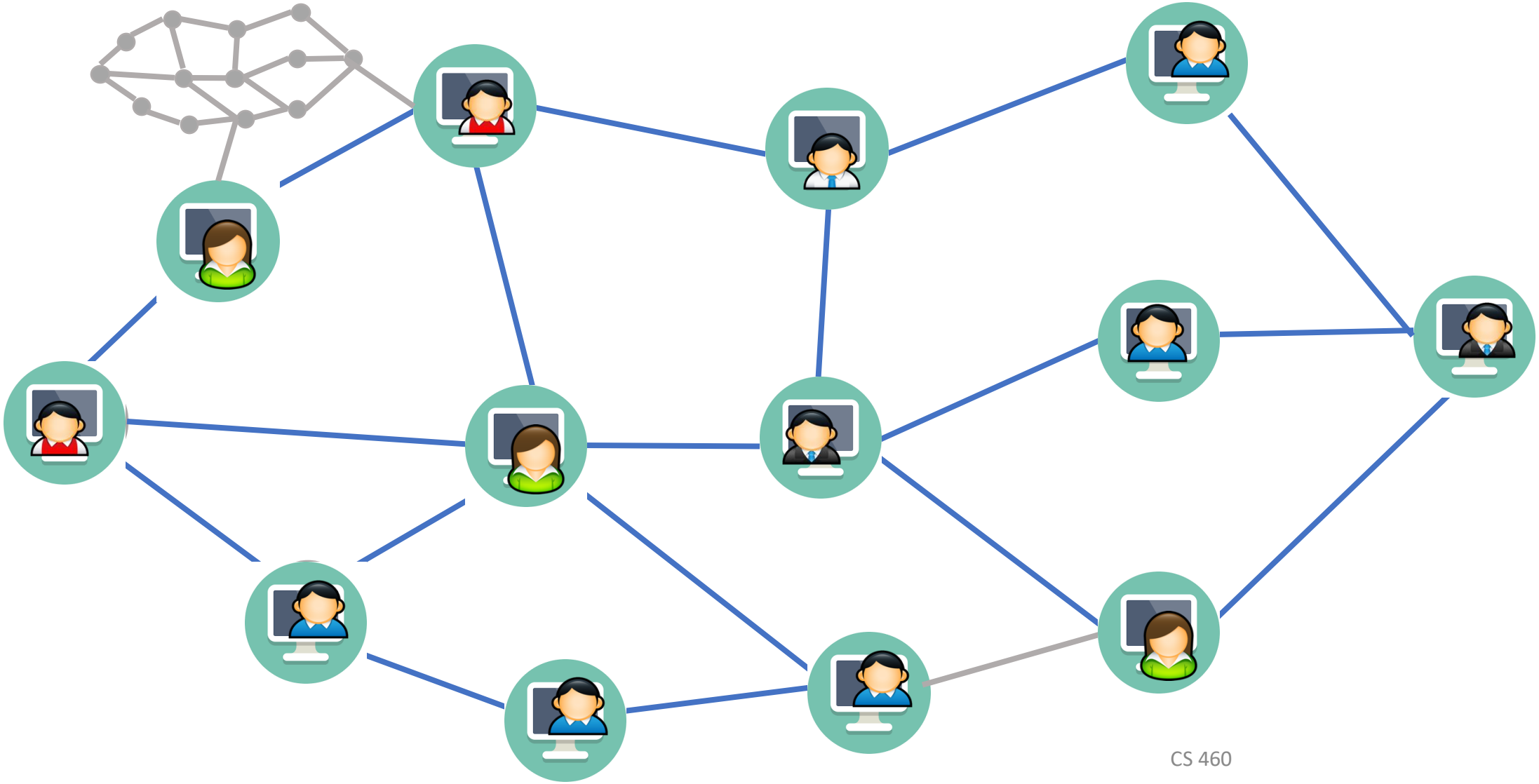
The Internet: A decentralized system



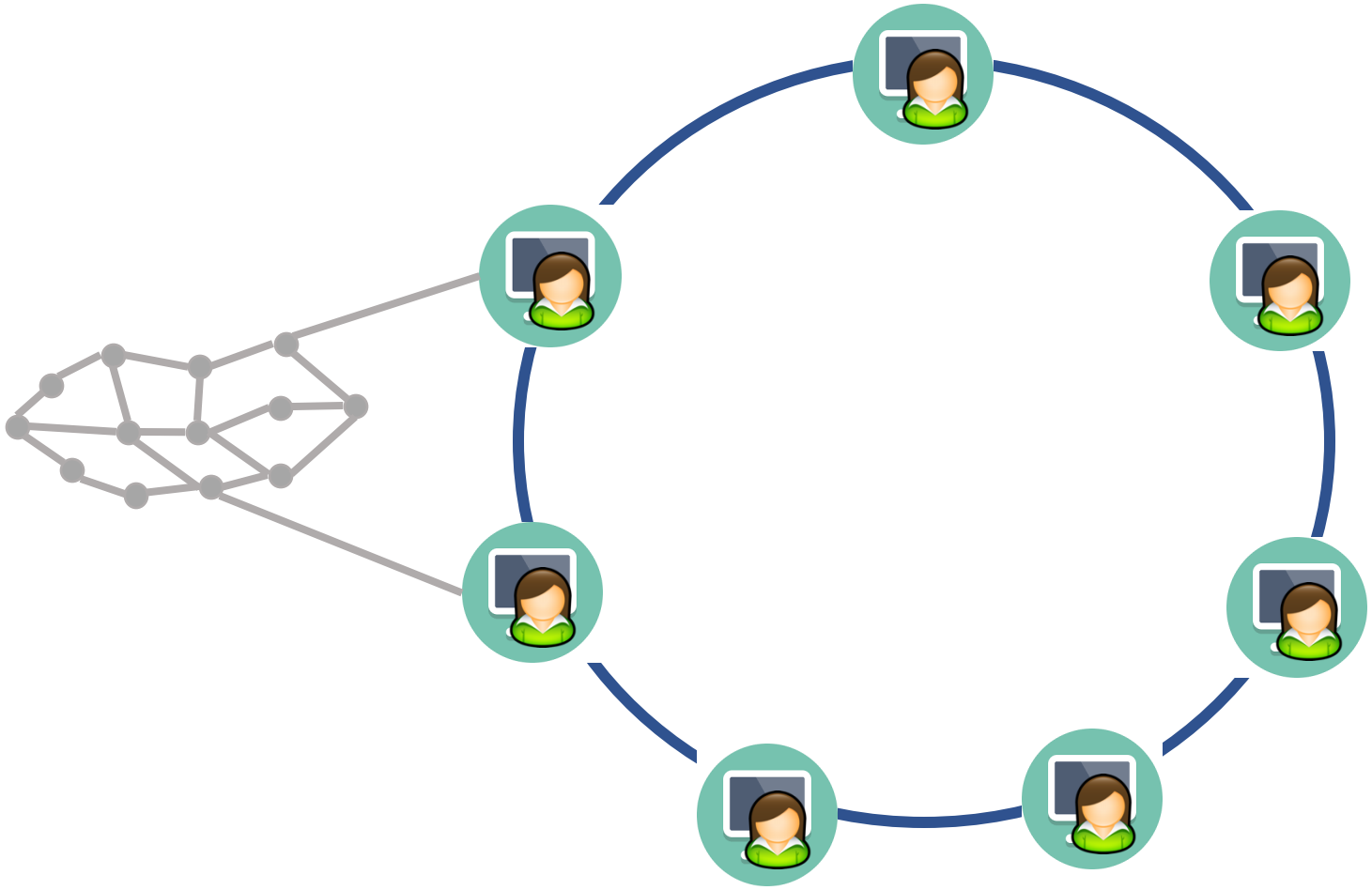
Overlay networks



Unstructured overlays



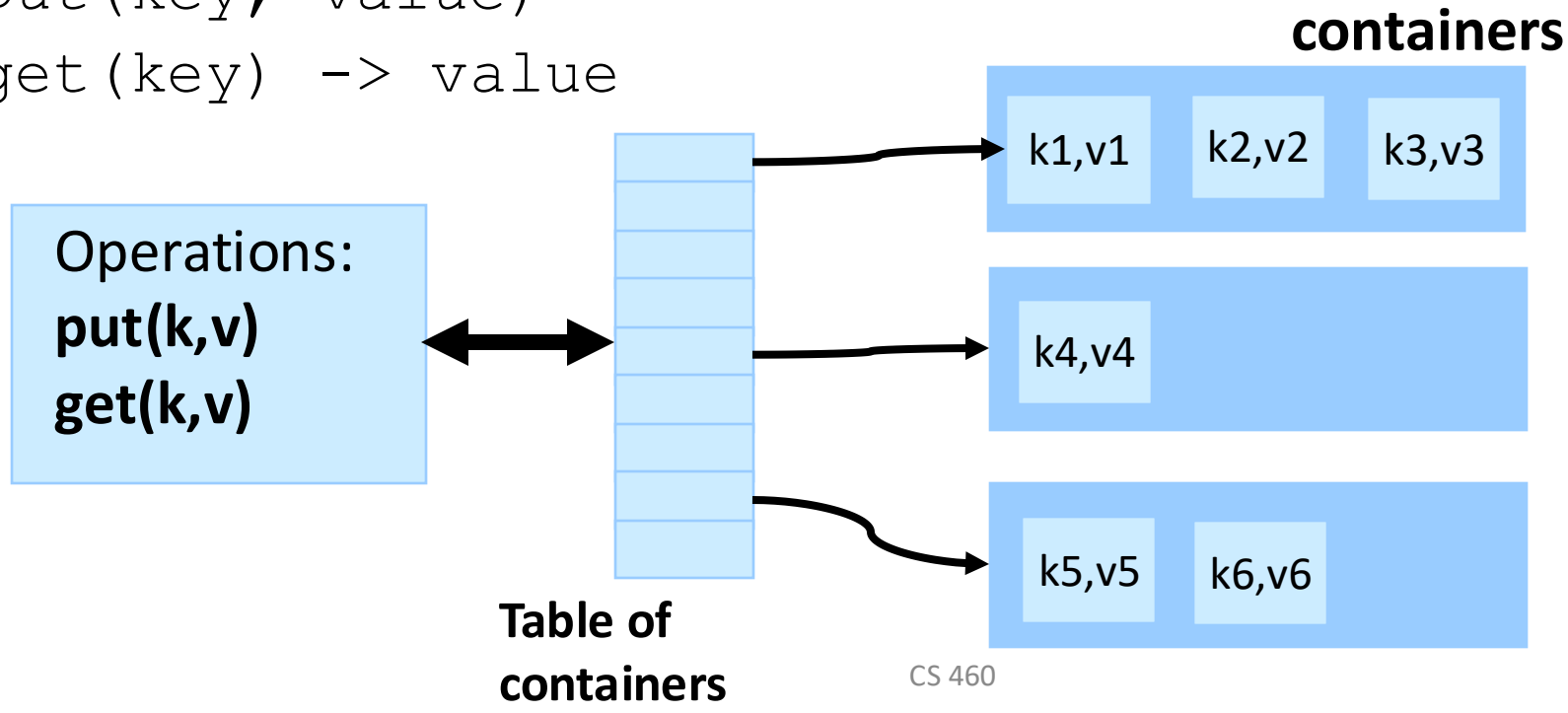
Structured overlays



Hash Table

- Structured overlay network
- A hash table: insert, lookup, delete object with keys

```
key = Hash(name)
put(key, value)
get(key) -> value
```

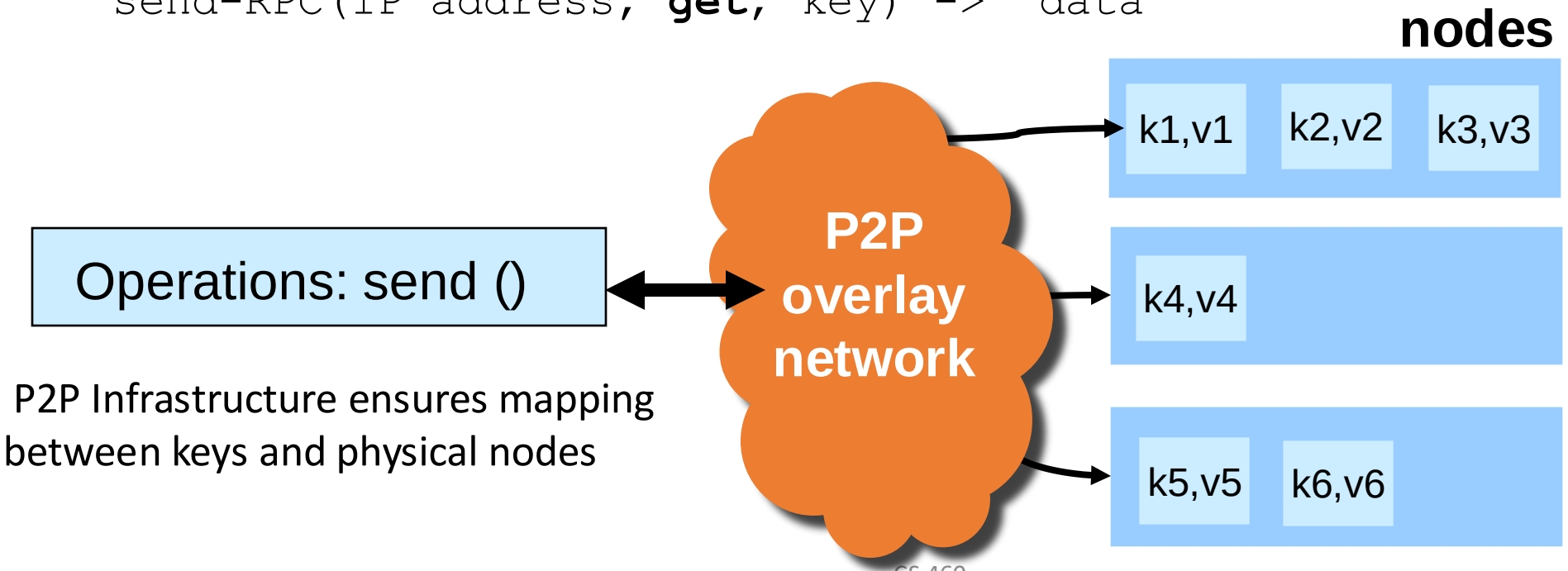


- Efficient access to a value given a key
- Mapping key-value ensured by the table of containers

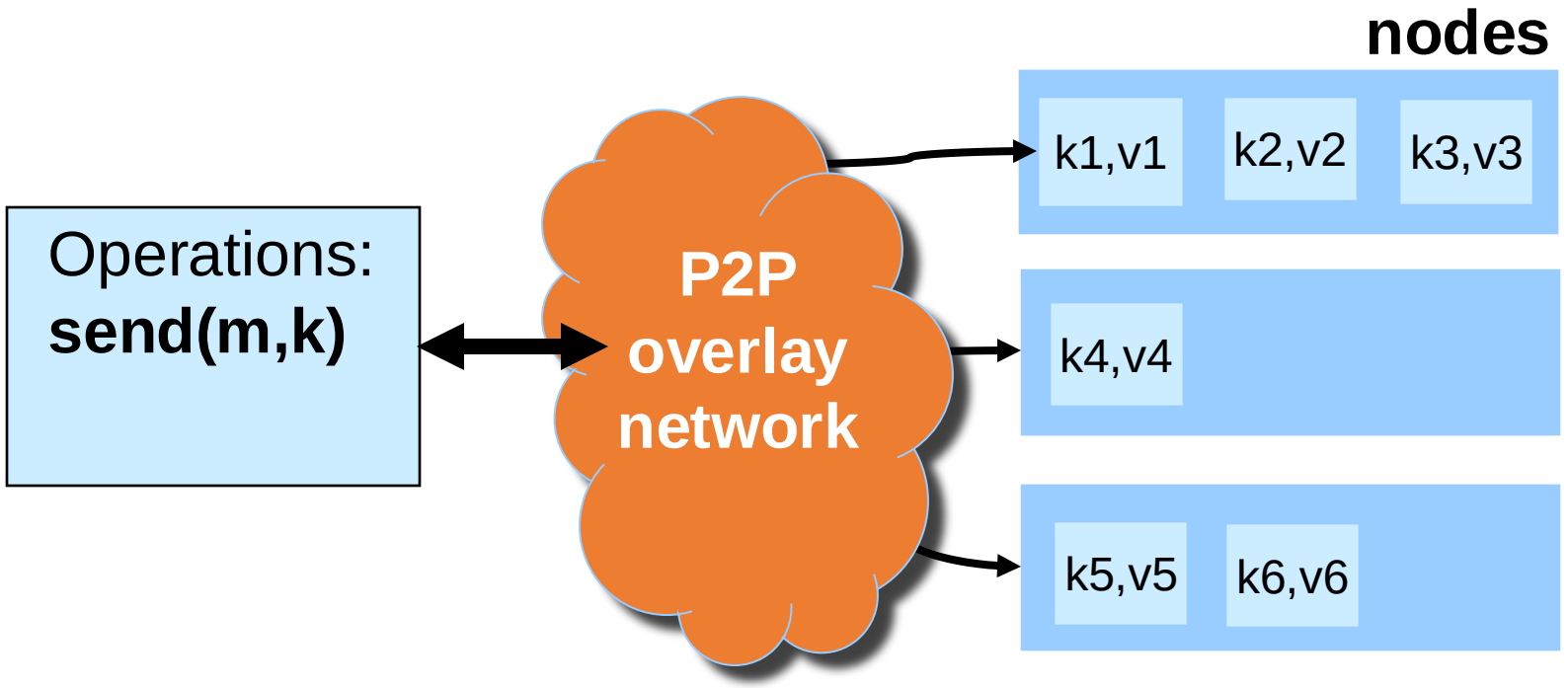
Distributed Hash Table

- A DHT does the same in a distributed setting across millions of hosts on the Internet

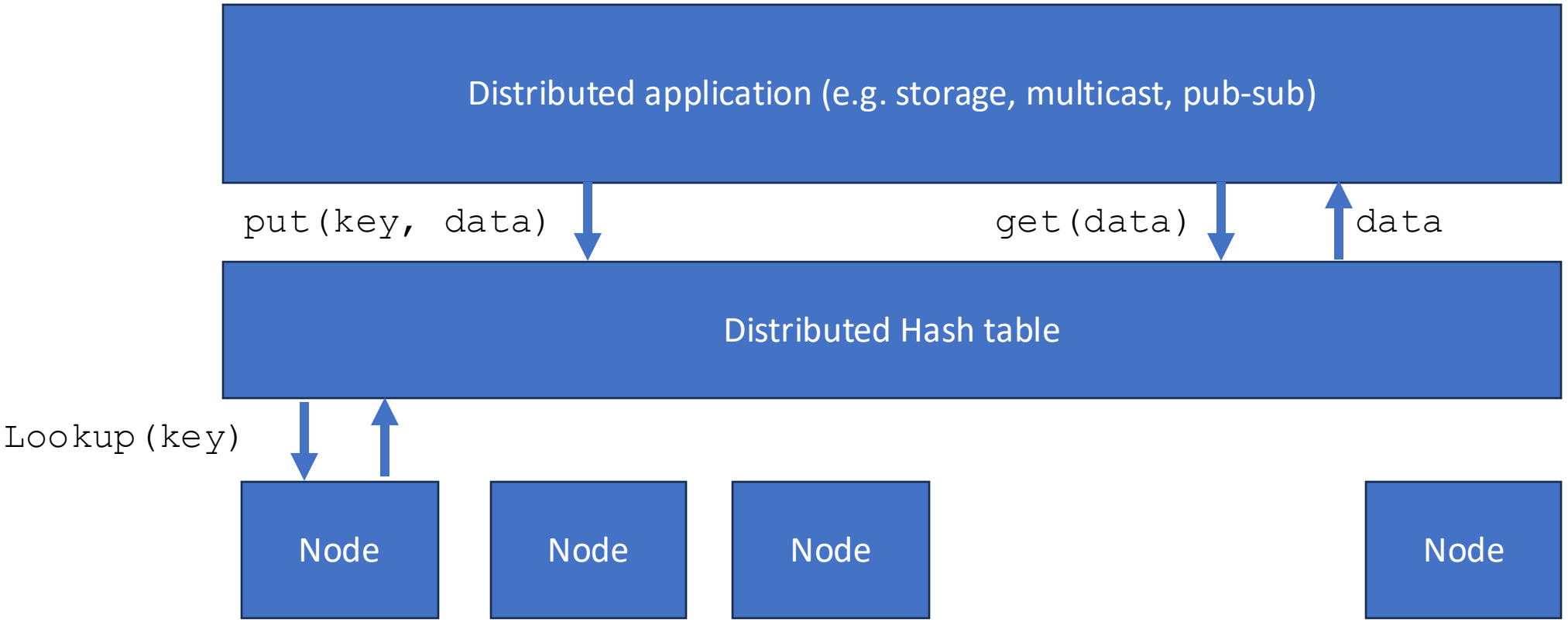
```
key = hash(data)
lookup(key) -> IP addr (DHT lookup service)
send-RPC(IP address, put, key, data)
send-RPC(IP address, get, key) ->  data
```



Distributed Hash Table



- Message sent to keys: implementation of a DHT
- P2P Infrastructure ensures mapping between keys and physical nodes
- Fully decentralized: peer to peer communication paradigm


Distributed application (e.g. storage, multicast, pub-sub)
put(key, data)
get(data)
data
Distributed Hash table
Lookup(key)
Node
Node
Node
Node

Pastry

Designed by A. Rowstron (MSR) and P. Druschel (Rice Univ.)

P2P routing infrastructure

- Overlay: network abstraction on top of IP
- Basic functionality: distributed hash table

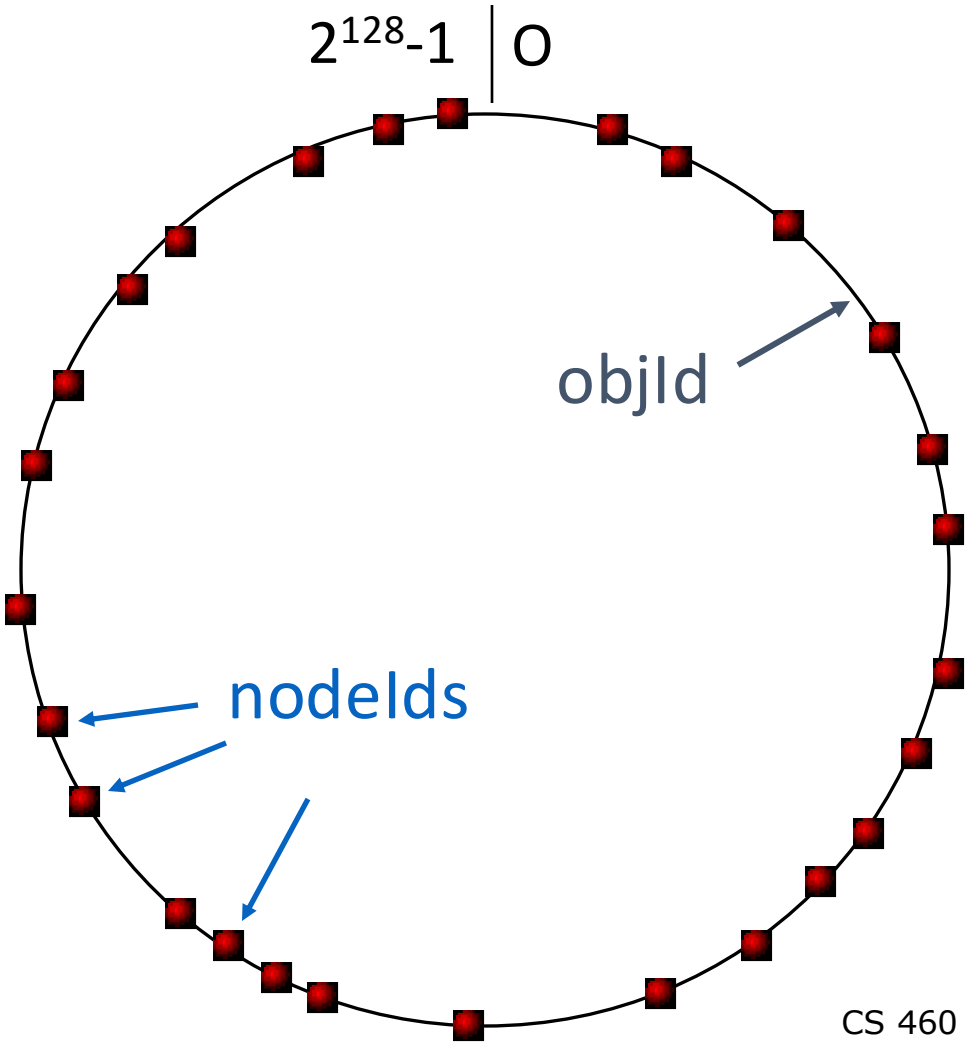
```
key = SHA-1(data)
```

- An identifier is associated to each node

```
nodeId = SHA-1(IP address)
```

- Large identifier space (keys and nodeId)
- A node is responsible for a range of keys
- Routing: search efficiently for keys

Object distribution



Consistent hashing
[*Karger et al. '97*]

128 bit circular id space

- *nodeIds* (uniform random)
- *objIds* (uniform random)

Invariant: node with numerically closest nodeId maintains object.

Pastry

- Naming space :
  - Ring of 128 bit integers
  - *nodeIds* chosen at random
- Key/node mapping
  - key associated to the node with the numerically closest node id
- Routing table

- *Leaf set*
  - 8 or 16 closest numerical neighbors in the naming space

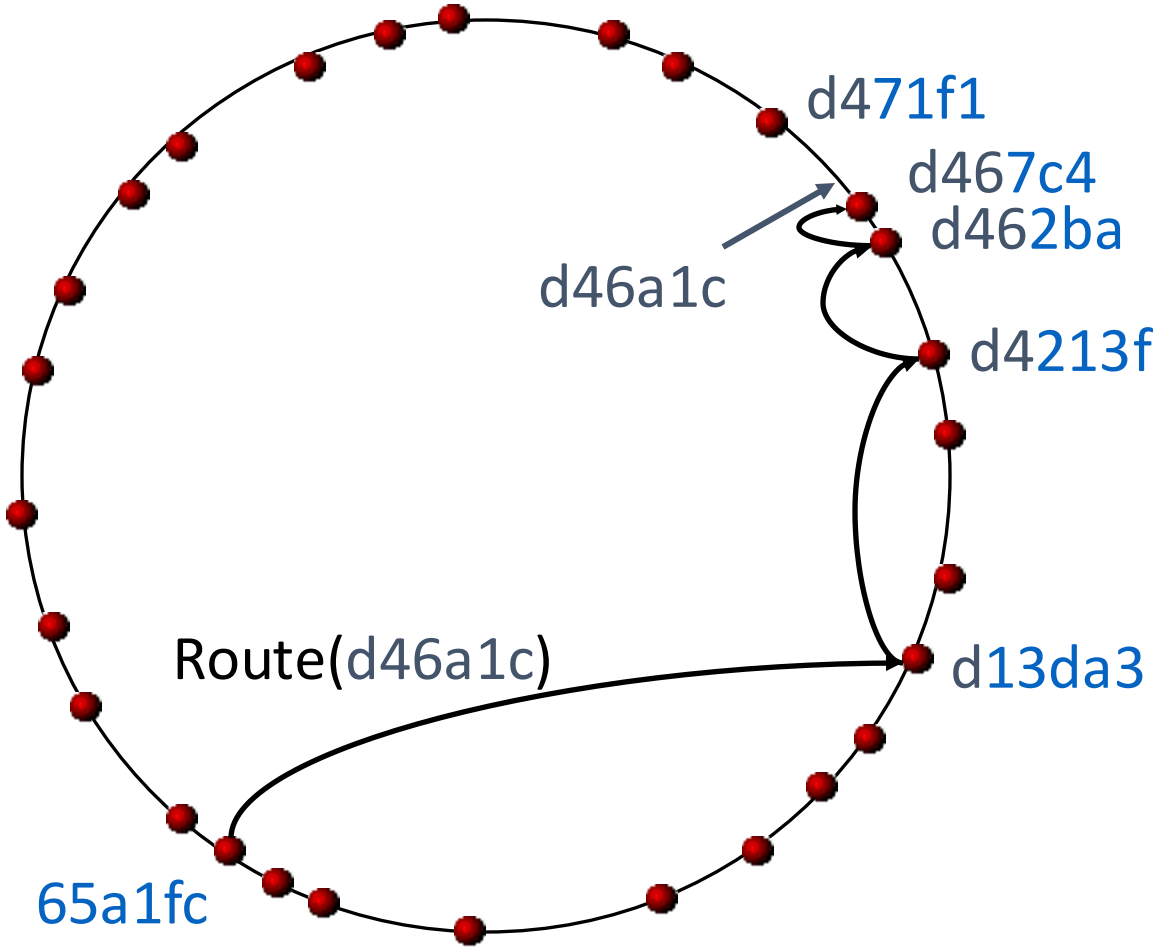
Pastry routing table

- Routing tables based on **prefix matching**
  - Identifiers are a set of digits in base 16
  - Matrix of 128/4 lines et 16 columns
  - `routeTable(i,j)`:
    - *nodeId* matching the current node identifier up to level *I*
    - with the next digit is *j*

Simple example

- Consider a peer with id 01110100101
- Maintains a neighor peer in each of the following prefixes
  - 1
  - 00
  - 010
  - 0110
  - ….
- At each routing step, forward to a neighbor with the largest matching prefix

Pastry: Routing



Properties

- $\log_{16}$ N hops
- Size of the state maintained (routing table): O(*log N*)

Search takes *O(log(N))* time

(intuition): *at each step, distance between query and peer-with-file reduces by a factor of at least 2*

Pastry: Routing table(#65a1fcx)

Line 0	0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
Line 1	60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
Line 2	650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
Line 3	65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx

$\log_{16} N$ lines

Routing algorithm, notations

R_l^i : entry of the routing table R, $0 \leq i \leq 2^b$,

line l, $0 \leq l \leq \lfloor 128/b \rfloor$

L_i : ith closest nodeId in the leafset

D_l : value of the l digits of key D

$SHL(A, B)$: length of the shared prefix between A and B

Routing algorithm (on node A)

(1) if ($L_{-\lfloor |L|/2 \rfloor} \leq D \leq L_{\lfloor |L|/2 \rfloor}$) {
(2) // D is within range of our *leaf set*
(3) forward to L_i, s.th. $|D - L_i|$ is minimal;
(4) } else {
(5) // use the routing table
(6) Let $l = shl(D, A)$;
(7) if ($R_l^{D_l} \neq null$) {
(8) forward to $R_l^{D_l}$;
(9) }
(10) else {
(11) // rare case
(12) forward to $T \in L \cup R \cup M$, s.th.
(13) $shl(T, D) \geq l$,
(14) $|T - D| < |A - D|$
(15) }
(16) }

R_l^i : entry of the routing table R, $0 \leq i \leq 2^b$,
line l, $0 \leq l \leq \lfloor 128/b \rfloor$
L_i : ith closest nodeId in the leafset
D_l : value of the l digits of key D
$SHL(A, B)$: length of the shared prefix between A and B

Node departure

- Explicit departure or failure
- Graceful replacement of a node
- The leafset of the closest node in the leafset contains the closest new node, not yet in the leafset
- Update from the leafset information
- Update the application

Failure detection

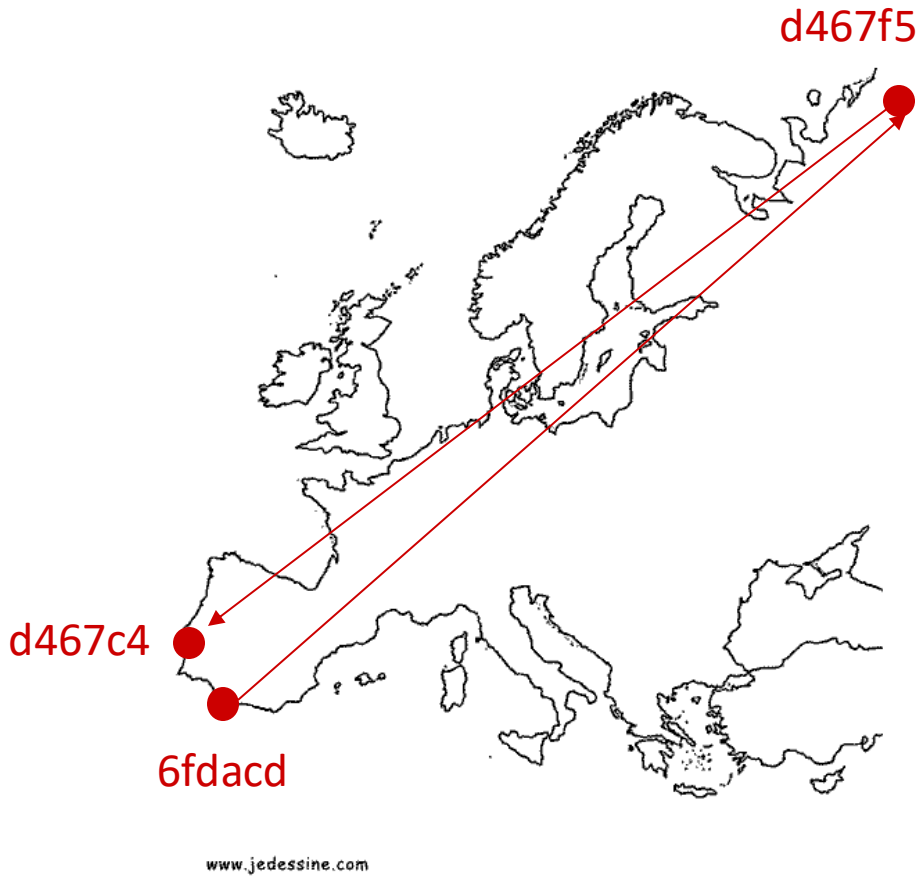
- Detected when immediate neighbors in the name space (leafset) can no longer communicate
- Detected when a contact fails during the routing
  - Routing uses an alternative route

State maintenance

- ***Leaf set***
  - **is aggressively monitored and fixed**
  - Eventual guarantee up to L/2 nodes with adjacent nodeIds fail simultaneously
- **Routing table**
  - **are lazily repaired**
    - When a hole is detected during the routing
  - Periodic gossip-based maintenance

Reducing latency

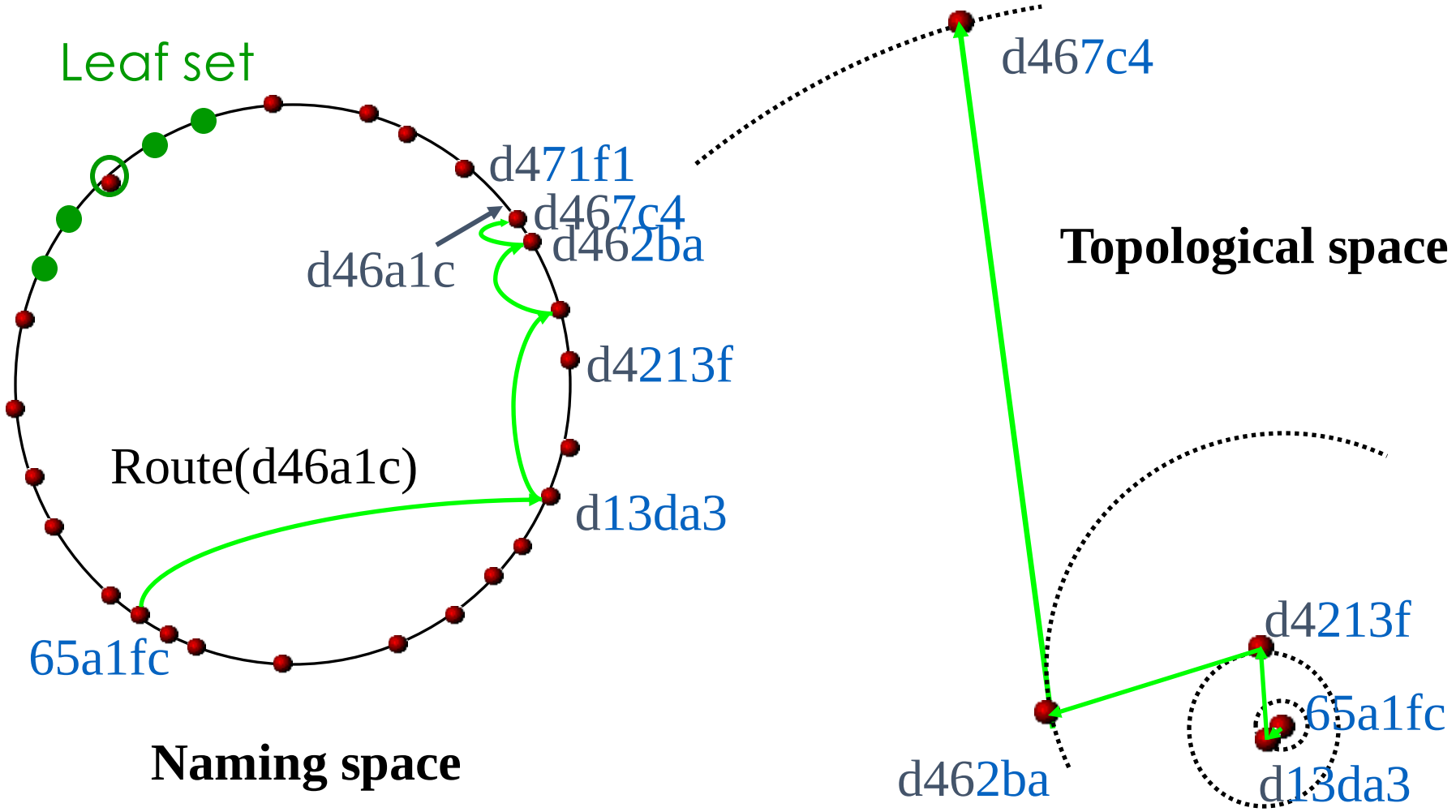
- **Random assignment of nodeId**: Nodes numerically close are geographically (topologically) distant
- **Objective**: fill the routing table with nodes so that routing hops are as short (latency wise) as possible
- **Topological Metric**: latency



Exploiting locality in Pastry

- Neighbor selected based of a network proximity metric:
  - Closest topological node
  - Satisfying the constraints of the routing table routeTable(i,j):
    - *nodeId* corresponding to the current *nodeId* courant up to level i
    - next digit = j
  - nodes are close at the top level of the routing table
  - random nodes at the bottom levels of the routing tables

Proximity routing in Pastry

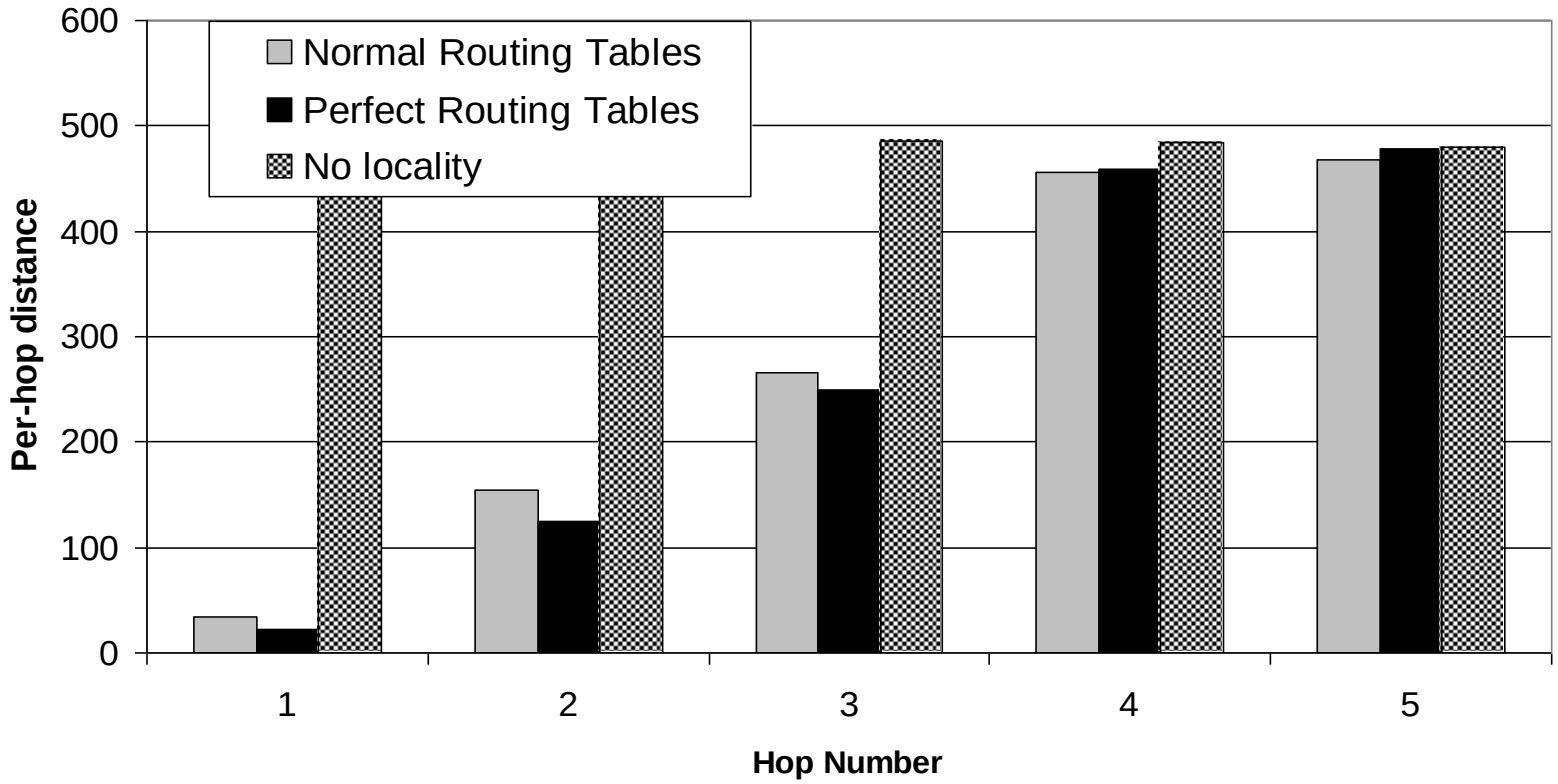


Joining the network

- Node X joins through a nearby node A
- Node X routes to node A
  - Path A,B,… -> Z
  - Z numerically closest to X
  - Initialisation of the line *i* of the routing table with the contents of line *i* of the routing table of the *ith* node encountered on the path
- Improving the quality of the routing table
  - X asks to each node of its routing table its own routing state and compare distances
  - Gossip-based update for each line (every 20mn)
    - Periodically, an entry is chosen at random in the routing table
    - Corresponding line of this entry sent over
    - Evaluation of potential candidates
    - Replacement of better candidates
    - New nodes gradually integrated

Performance

1.59 slower than IP on average



References

- *Antony I. T. Rowstron, Peter Druschel: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Middleware 2001: 329-350*
- *Ion Stoica, Robert Tappan Morris, David R. Karger, M. Frans Kaashoek, Hari Balakrishnan: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM 2001: 149-160*
- *Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, Scott Shenker : A scalable content-addressable network. SIGCOMM 2001: 161-172*
- *Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph , John Kubiatowicz: Tapestry: a resilient global-scale overlay for service deployment. IEEE J. Sel. Areas Commun. 22(1): 41-53 (2004)*

Consistency models

Anne-Marie Kermarrec

CS-460

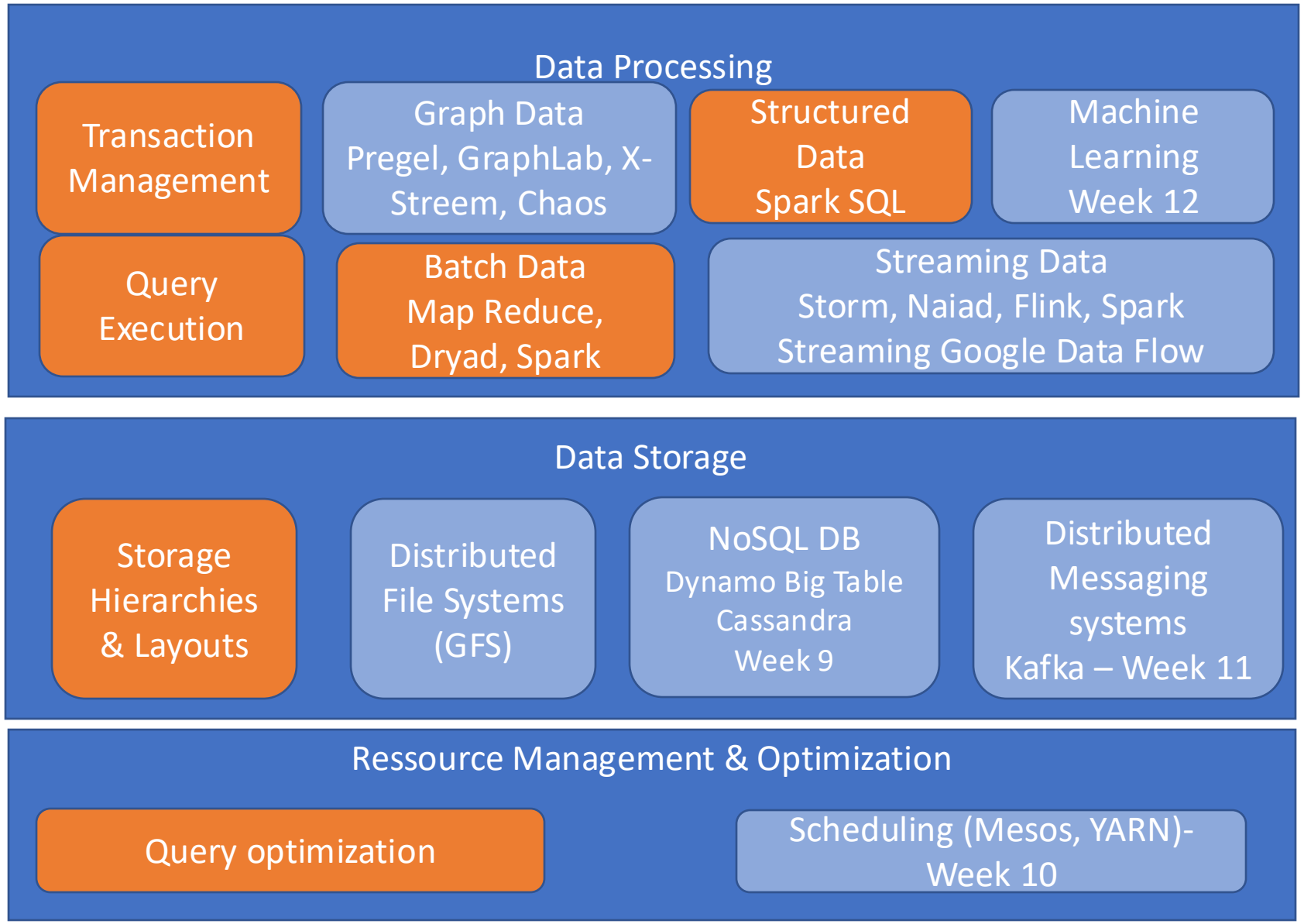
Where are we?

Gossip Protocols
Week 7

Consistency models
CAP Theorem
Week 8-9

Distributed/decentralized systems
Week 8-12

Data science software stack



Replication

- Replication is key to availability (low latency, failure resilience, load balancing)
- But creates inconsistencies due to concurrent accesses

What is a consistency model?

- Describes a contract between a client application and the data store
  - States how the memory behaves
  - States what the application can expect from the underlying storage systems and the associated rules

When is it needed?

Whenever objects are replicated

Replicas must be consistent in some way

- Modifications have to be carried out on all copies
- In the presence of concurrent updates/reads

Different consistency models

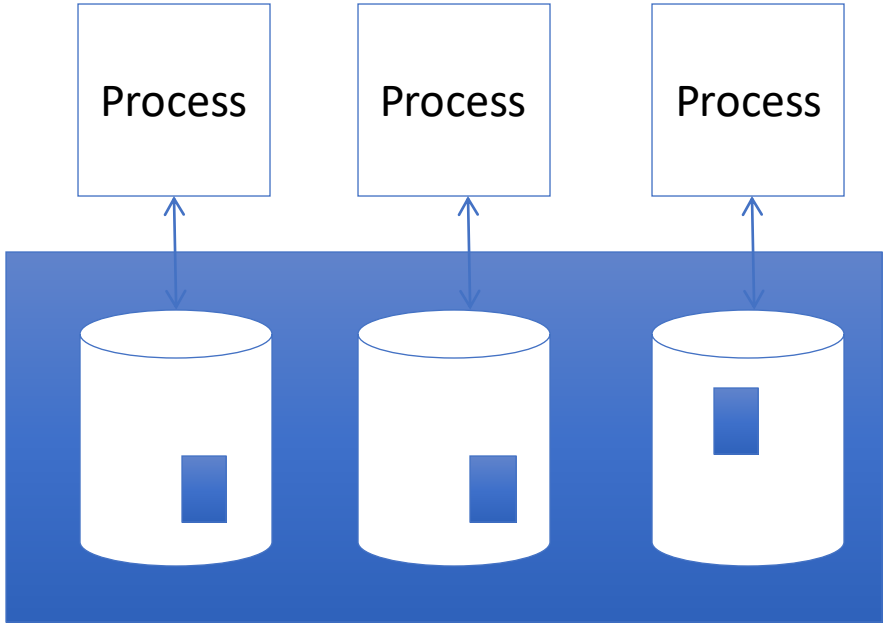
- A consistency model is a set of rules that process obeys while accessing data

A large spectrum of consistency models

Strong Consistency

-
-
-

Eventual Consistency



Distributed data storage

Faster reads and writes

More consistency

Eventual

Strong

Examples of onsistency guarantees

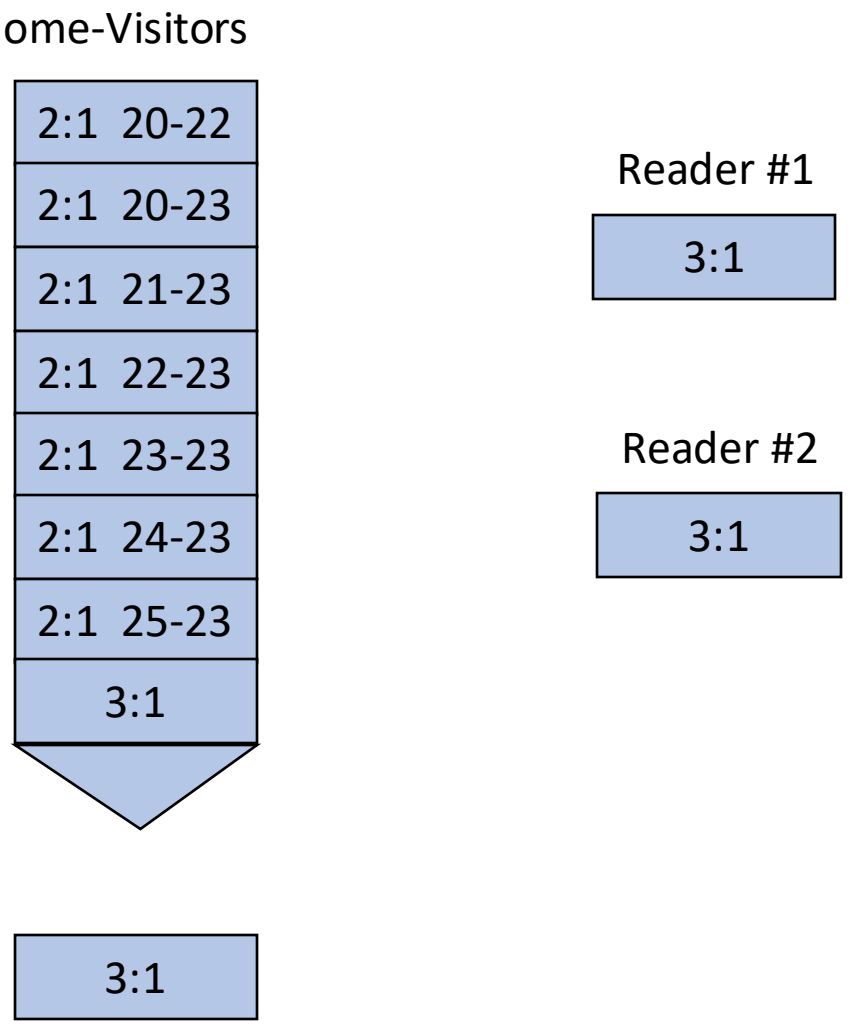
Strong consistency	See all previous writes
Eventual consistency	See subset of previous writes
Consistent prefix	See initial sequence of writes
Monotonic Freshness	See increasing sequence of writes
Read my writes	See all writes performed by reader
Bounded Staleness	See all “old”writes

Consistency requirements in a volley-ball game

- The first team to reach 25 points and by at least two points wins a set (for the first 4 sets)
- The first team to reach 15 points and by at least two points wins the 5th set
- The first team to win 3 sets wins the game
- Imagine the score is stored and replicated in the cloud

Inspired from Replicated Data Consistency Explained Through Baseball Doug Terry MSR Technical Report, October 2011

Strong consistency



Aka linearizability , one-copy serializability

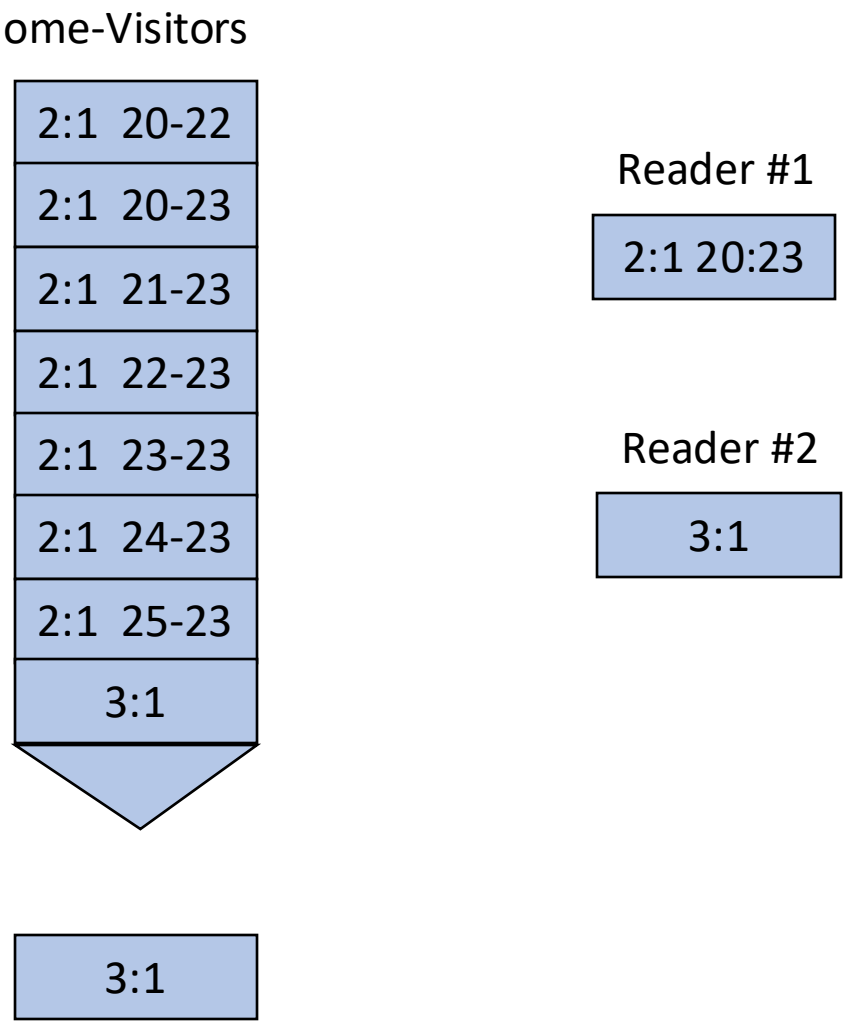
The responses to the operations invoked in an execution are the same as if all operations were executed in a sequential order and this order respects those specified by each process

Strong consistency is impossible to achieve in the presence of partition (CAP-next lecture)

Strong consistency is impossible to achieve in an asynchronous system without assumptions on message delivery latencies (FLP)

Guarantee: see all previous writes. All reads at time *t* should reflect all the writes that happened before *t.*

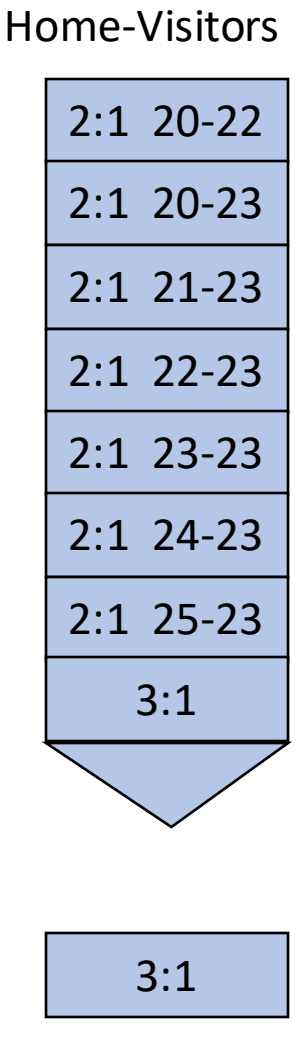
Eventual consistency



Eventually, in the absence of operations, replicas will be consistent

Guarantee: see some previous writes. Eventually (in the absence of new writes), all the reads will return the correct and most recent state.

Consistent Prefix



Reader #1

2:1 20:23

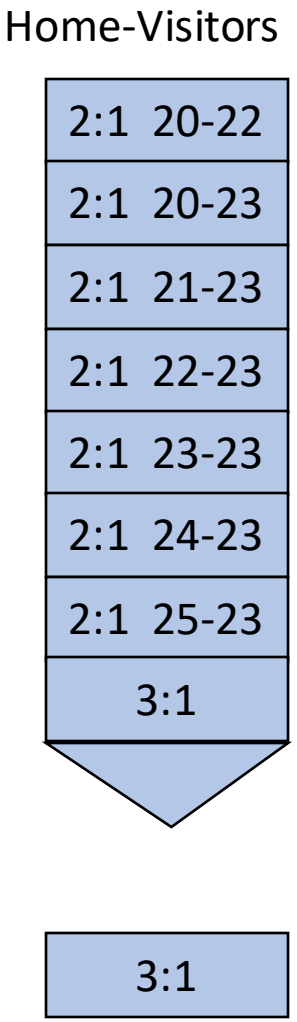
Reader #2

2:1 22:23

Snapshot isolation, ordered delivery

Guarantee: see initial sequence of writes that existed at some point in time. If a reader issues a read request at time *t,* it should read the result of any of the prefixes of the sequence of writes.

Monotonic Freshness



Reader #1 at time t1

2:1 20:23

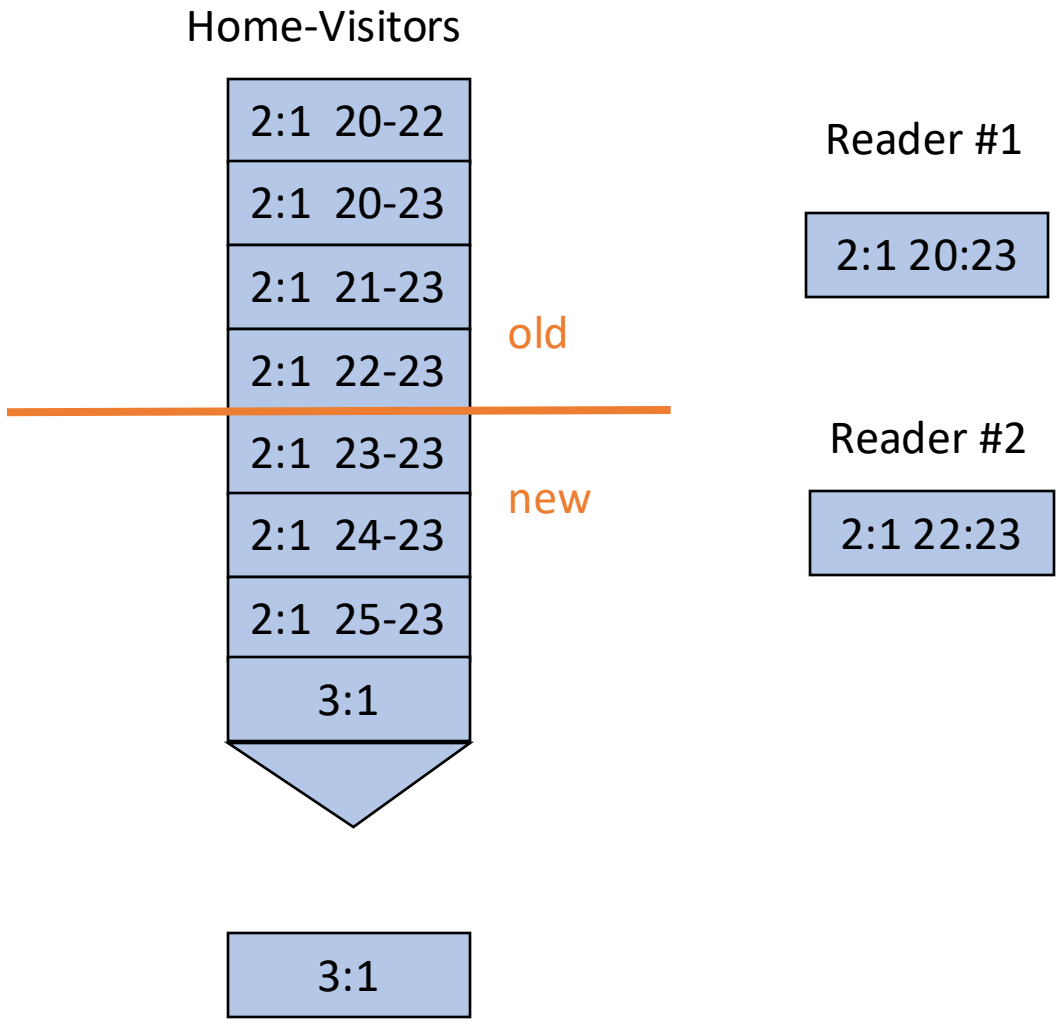
Reader #1 at time t2

3:1

If a process reads the value of a data item x, any successive operation on x by that process will always return the same or a more recent value

Guarantee: see increasing subset of previous writes (local guarantee from a given reader)

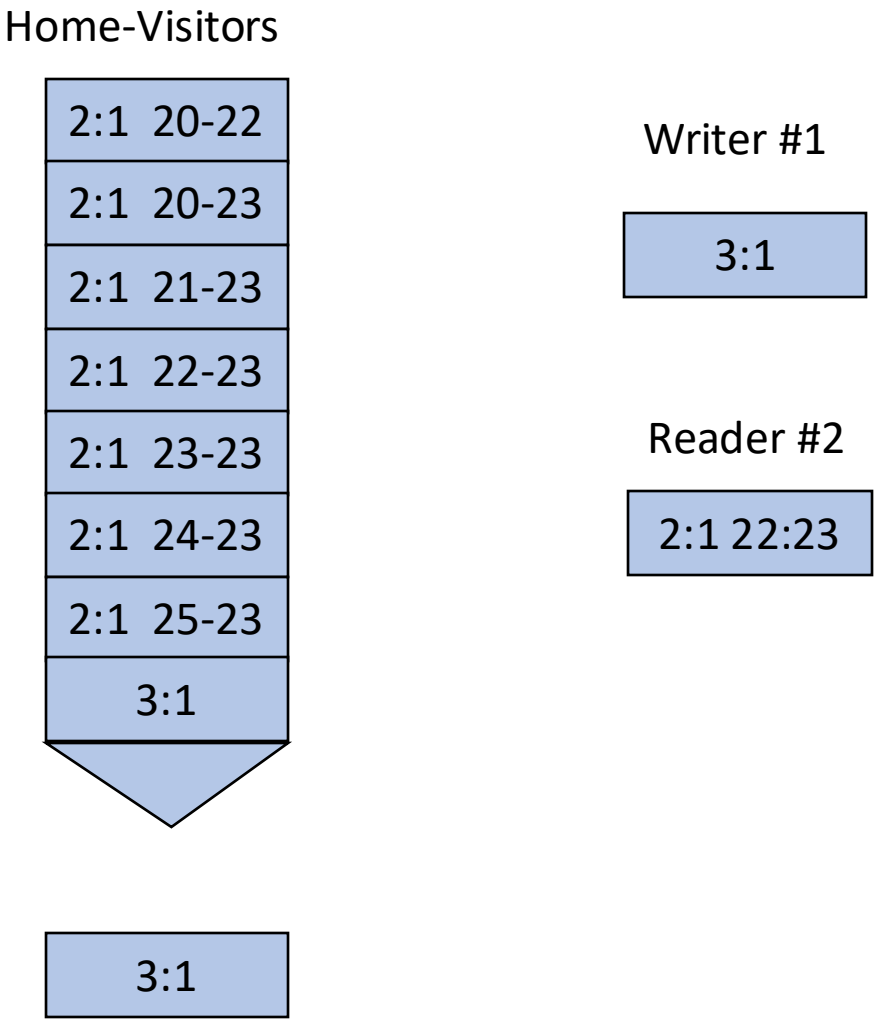
Bounded Staleness



Periodic Snapshot, continuous consistency

Guarantee: see all « old » writes. The staleness parameter denotes the allowed staleness of the system.

Read my writes



The effect of a write operation by a process on data
item x will be always seen by a successive read operation on x by the same process

Guarantee: see all writes performed by reader.
local guarantee any read by client *c* should reflect all the writes by *c* in the past. This means that it can also be local-strong-consistency, whereas for writes of other clients, reads by *c* can be eventually-consistent.

Official score keeper

Suppose visitor score

Read (visitor_score)
Write(visitor_score. Update)

Read my writes (single score keeper)

Strong consistency otherwise

Referee

4th set, home scores @24
vs=Read (visitor_score);
hs= Read(home_score);
If (hs=25) & (vs<24)
end game ;

Strong consistency

Radio Reporter

```
Do{
        vs=Read (visitor_score);
        hs= Read(home_score);
            Report vs and hs;
              Sleep (30mn);
}
```

Consistent Prefix (if reads from same replica)
Monotonic Freshness
or Bounded Staleness

Sportswriter

```
While not end of the game{
        drink beer;
}

Go out to diner;
    vs=Read (visitor_score);
    hs= Read(home_score);
    write article;
```

Eventual consistency
or Bounded Staleness

Statistician

Wait for end of game;
score Read(“home_stats”);
stat=Read (“season-runs”);
Write{“season-runs”, stat
|+score);

Strong consistency (1st read)
Read My Writes after

Supporter

Read score;
Discuss score with friends

Eventual consistency or
Strong consistency

A wide range of models

Conclusions

- Different clients want different guarantees
- One client might want different guarantees for different reads
- Several models can be applied
- Strong consistency would do but is prohibitive performance wise
- Use the lowest consistency (to the left) consistency model that is “correct” for your application