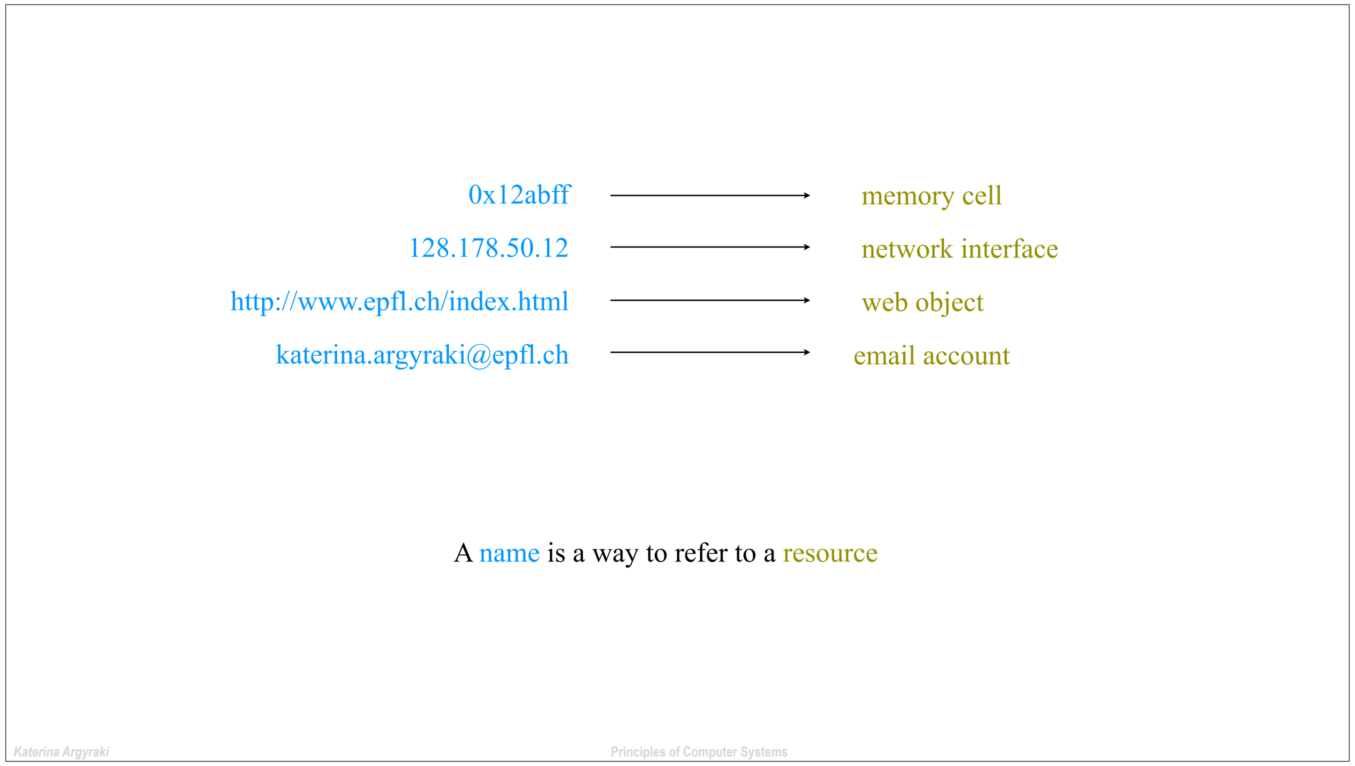
Principles of Computer Systems: Naming

Prof. Katerina Argyraki
School of Computer & Communication Sciences

Introduction



0x12abff → memory cell

128.178.50.12 → network interface

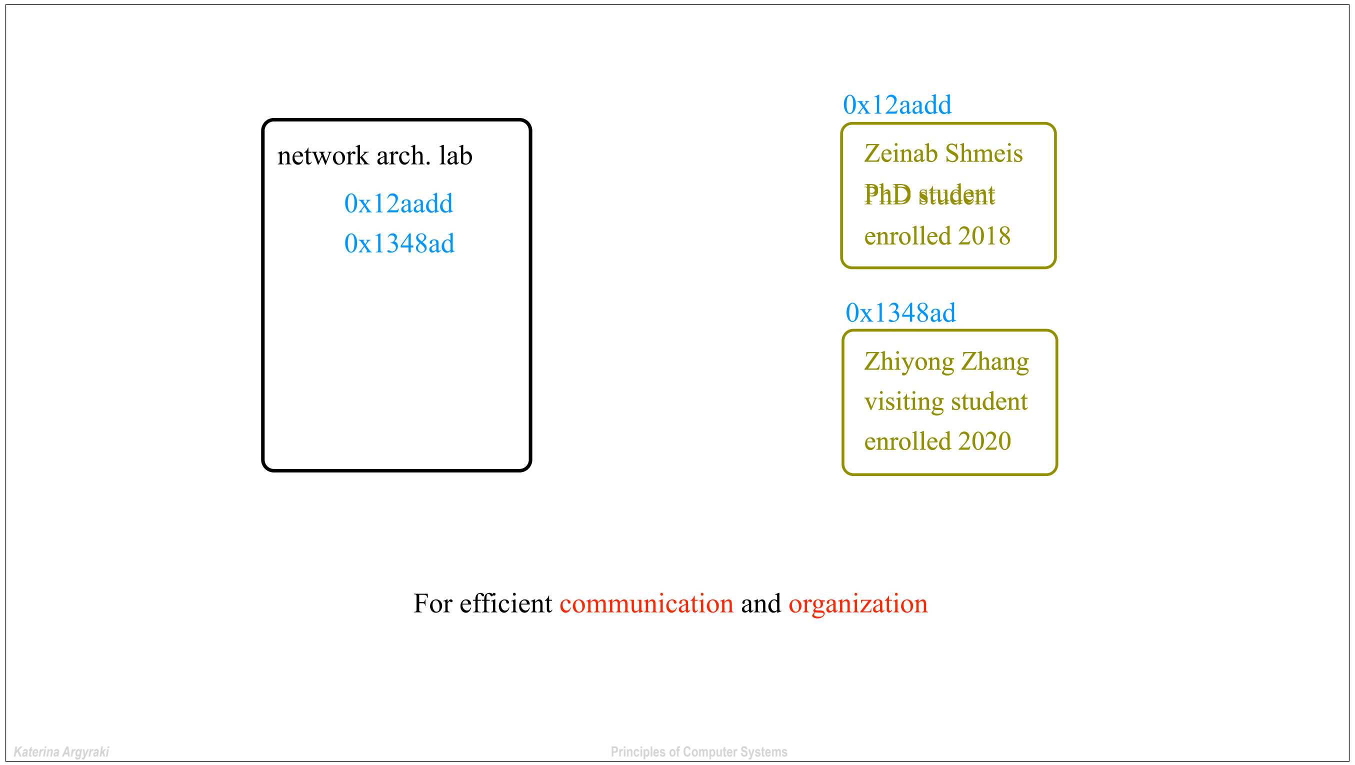
http://www.epfl.ch/index.html → web object

katerina.argyraki@epfl.ch → email account

A name is a way to refer to a resource

Let’s look at a few examples of names:

- A memory address is a name, which refers to a memory cell.
- An IP address is a name, which refers to a network interface.
- A web URL is a name, which refers to a web object.
- And an email address is a name, which refers to an email account.



network arch. lab

0x12aadd
0x1348ad

0x12aadd

Zeinab Shmeis
PhD student
enrolled 2018

0x1348ad

Zhiyong Zhang
visiting student
enrolled 2020

For efficient communication and organization

Why do systems use names?

One obvious use is for specifying which resource to read from or write to,
but there other, more subtle uses:

Suppose we have an object stored in memory,
which stores information about the network architecture lab,

and we also have objects that store information about different members of the lab.

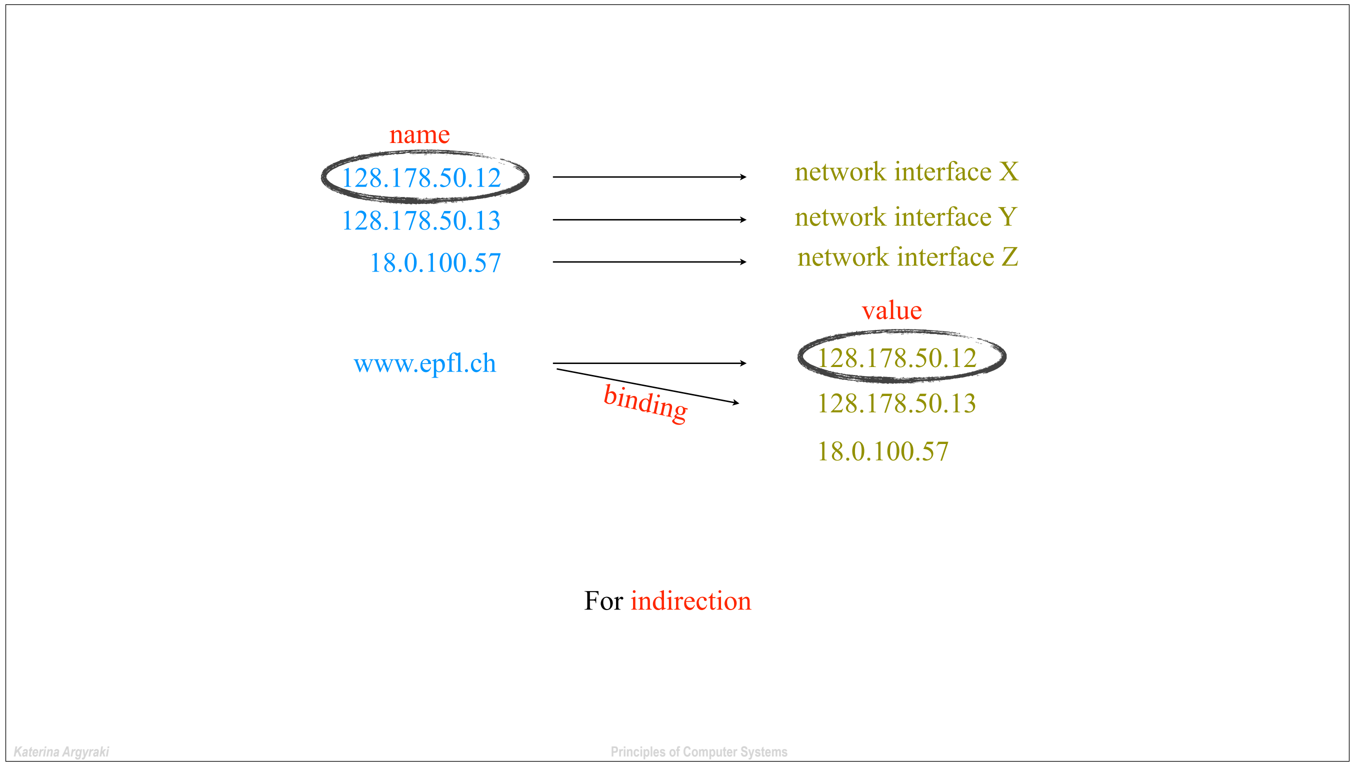
These objects need to communicate,
in particular, the lab object on the left needs to access the content of the member objects on the right.

One way to do that is to copy the contents of the member objects into the lab object.

Another way is to assign names to the member objects
and store only these names into the lab object.
Whenever the lab object needs to access the member objects,
it uses their names to find them.

To those of you who program a lot,
this example is akin to passing function arguments by value versus by reference.
And, btw, a C++ pointer is a *name* for the object it is pointing to.

So, systems use names for efficient communication and organization.



name

128.178.50.12 → network interface X

128.178.50.13 → network interface Y

18.0.100.57 → network interface Z

value

www.epfl.ch → 128.178.50.12

binding

128.178.50.13

18.0.100.57

For indirection

We said earlier that an IP address is a name that refers to a network interface.

But there exists another way to refer to a network interface: a DNS name.

However, a DNS name does not point directly to a network interface.

It points to an IP address, which then points to a network interface.

So, an IP address can be a name that points to a network interface,
and at the same time the value of a DNS name.

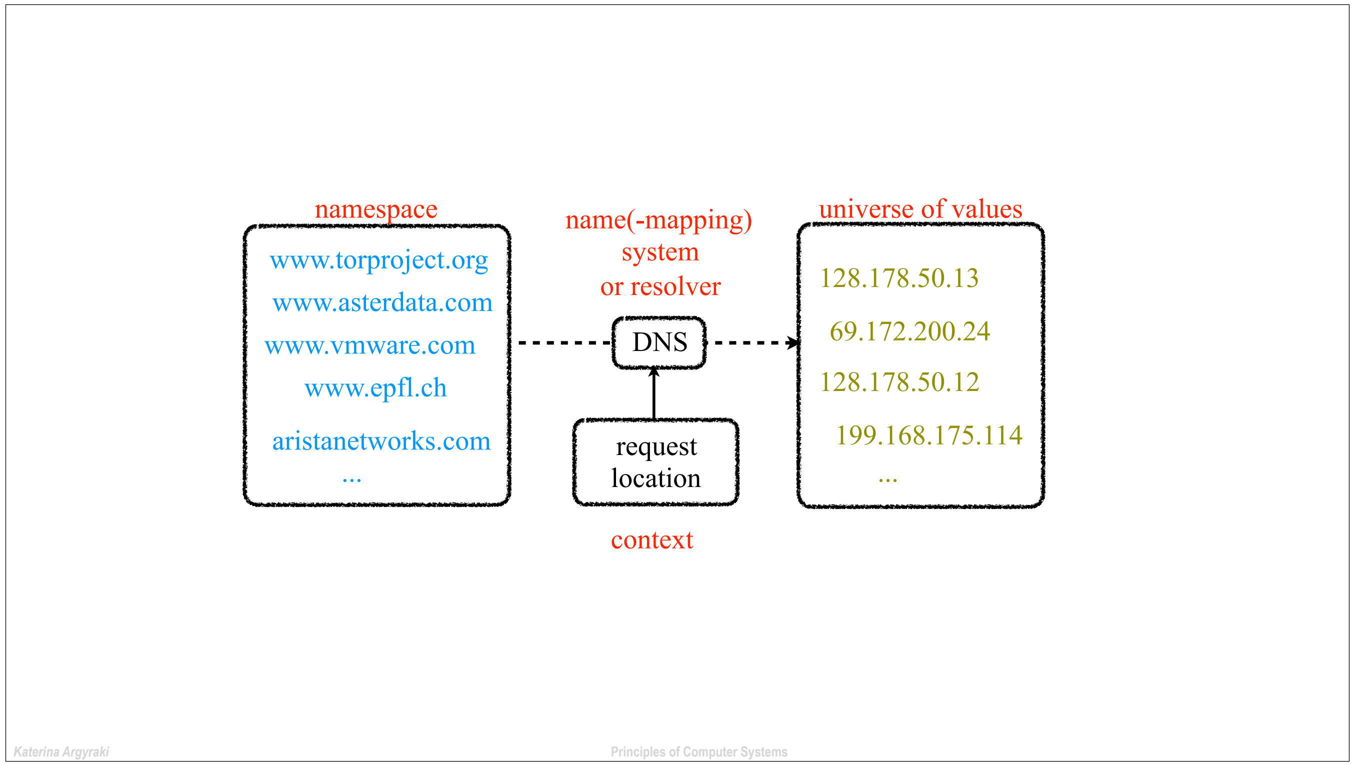
In this scenario, we are using a name -- in particular, an IP address --
as an indirection tool.

Why is this useful?

For one thing,
it allows us to dynamically change the network interface that www.epfl.ch refers to
by changing only the IP address that it is mapped to.

This is the basic idea behind dynamic DNS.
It changes DNS name-to-value mappings so as to optimize network performance.

In this situation, the act of choosing one of all possible values
and mapping a name to it is called binding.



namespace

www.torproject.org
www.asterdata.com
www.vmware.com
www.epfl.ch
aristanetworks.com
...

name(-mapping)
system
or resolver

DNS

request
location

context

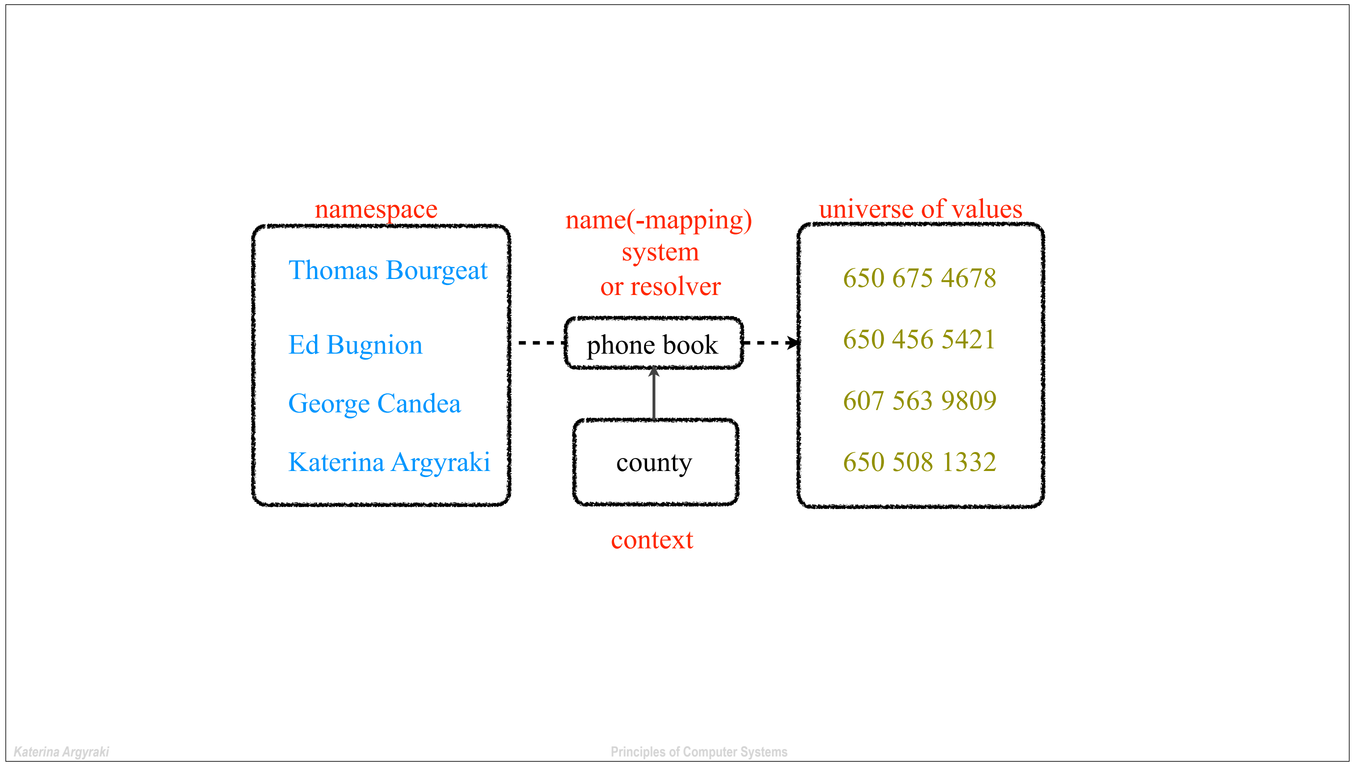
universe of values

128.178.50.13
69.172.200.24
128.178.50.12
199.168.175.114
...

Now consider the set of all the possible DNS names on the left,
the set of all the possible IP addresses on the right,
and the DNS system that maps names to IP addresses in the middle.

We call the set on the left a namespace,
the set on the right a universe of values,
and the system that does the mapping a name-mapping algorithm or resolver.

In the case of dynamic DNS, to decide which value to map a name to,
the DNS system needs extra information, e.g., the location of the requester.
We call this extra information, “context.”



namespace

Thomas Bourgeat

Ed Bugnion

George Candea

Katerina Argyraki

name(-mapping)
system
or resolver

phone book

county

context

universe of values

650 675 4678

650 456 5421

607 563 9809

650 508 1332

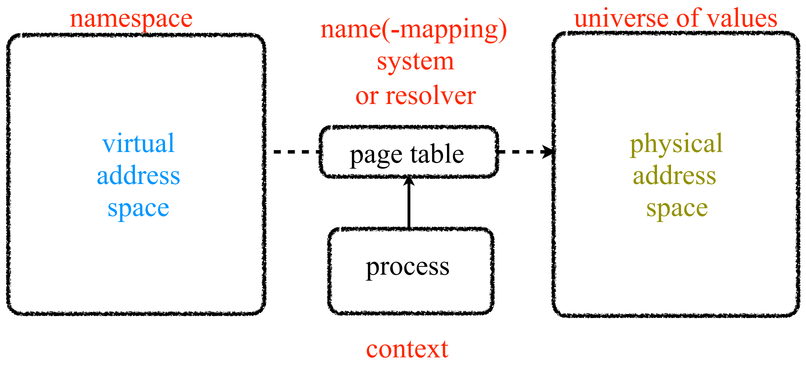
Katerina Argyraki

Principles of Computer Systems

Let’s look at a couple more examples of naming schemes:

A phone book maps human names to phone numbers.

In the US at least, there is a different phone book per county,
so each county is a different context.



namespace

virtual address space

name(-mapping) system or resolver

page table

process

context

universe of values

physical address space

To get back to computer systems:

A page table maps virtual memory addresses
to physical memory addresses.

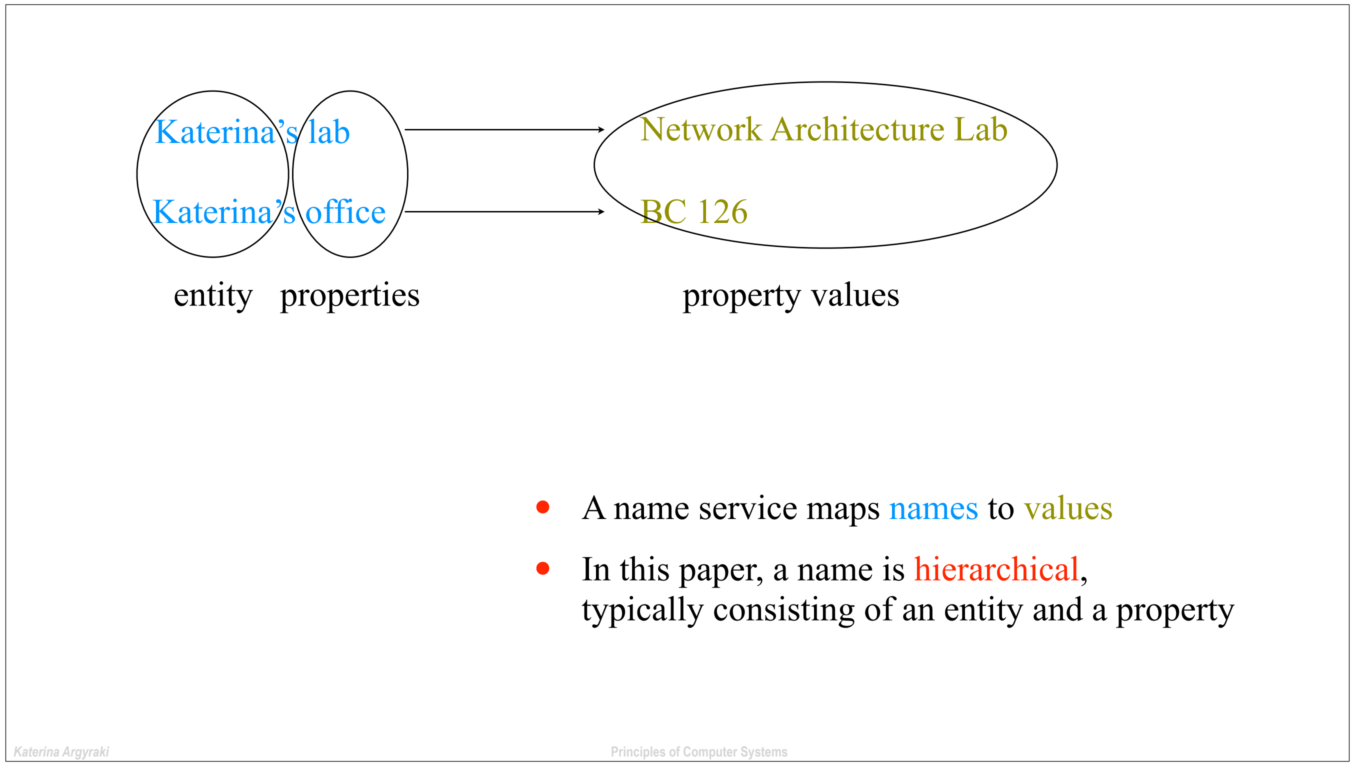
And there is a typically a different page table per process,
so each process is a different context.

Name types

- Private: unique within a context
  - *e.g., a private IP address is unique within an organization*
- Global: unique across contexts
  - *e.g., a global IP address is unique within the Internet*

- Hierarchical: name relationship implies value relationship
  - *e.g., two IP addresses sharing the same prefix*
- Flat: name relationship implies nothing
  - *e.g., content IDs in Peer-to-Peer networks*

Designing a Global Name Service



Katerina's lab

Katerina's office

entity properties

Network Architecture Lab

BC 126

property values

- A name service maps names to values
- In this paper, a name is hierarchical, typically consisting of an entity and a property

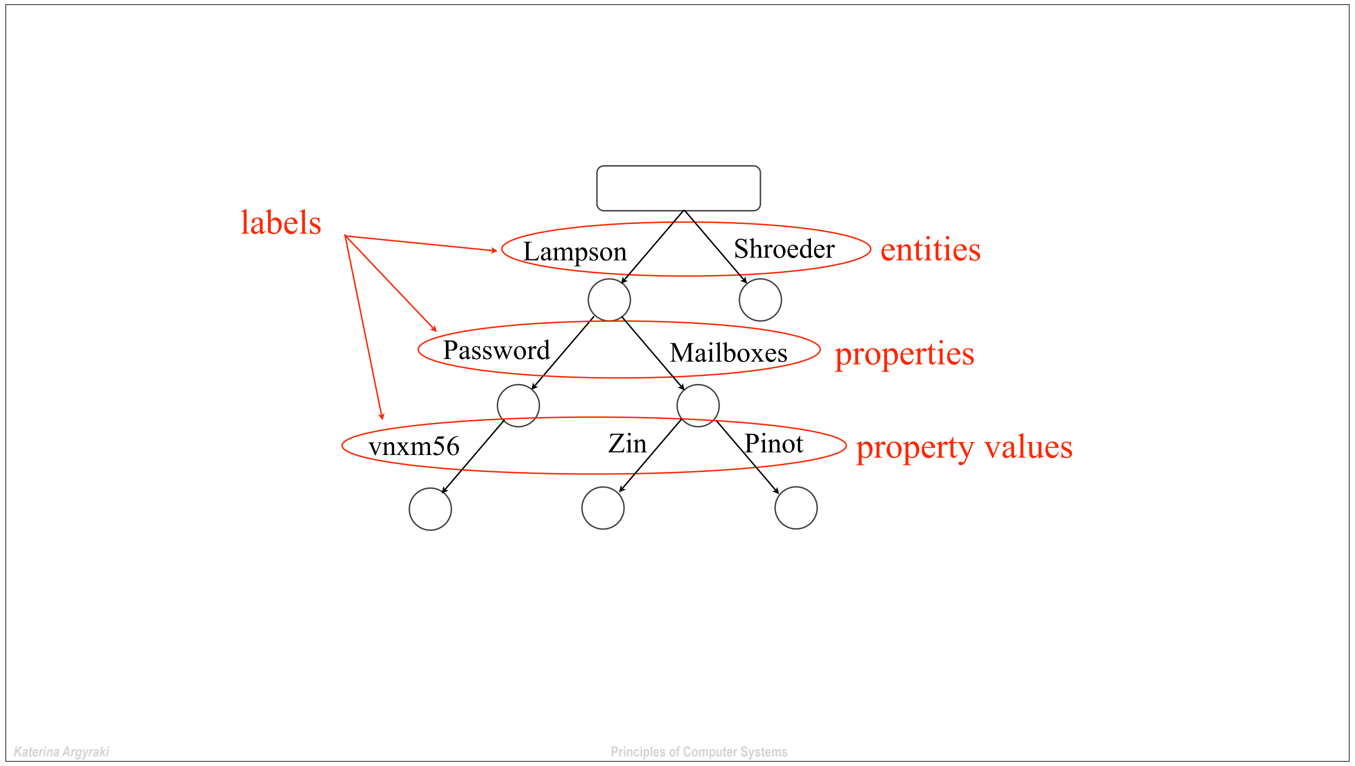
The role of a name service -- also called a directory service --
is to map names to values.

For example, …

BL had a particular kind of name in mind when he did this paper:
one that consists of an entity and a property.

For example, …

Because he had this particular type of hierarchical name in mind, he proposed to organize data in directories that look like

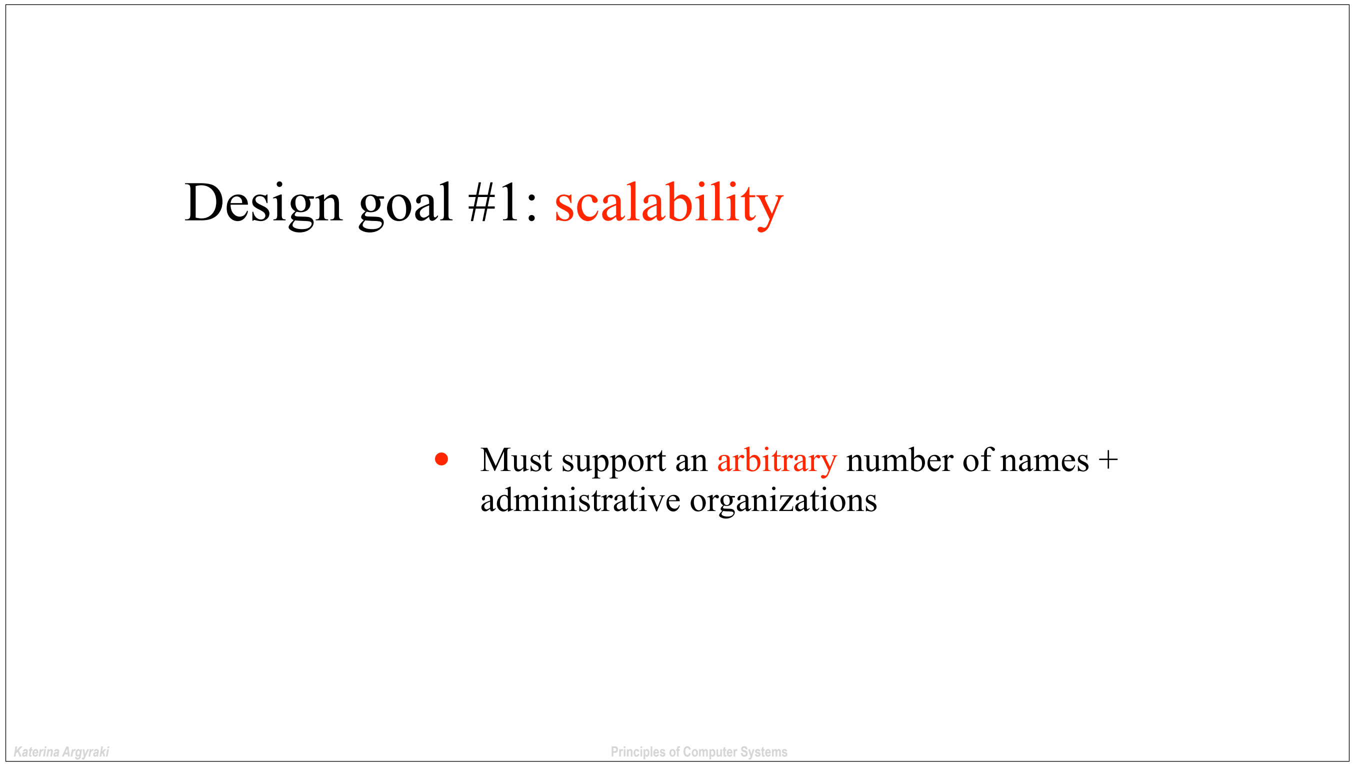


trees (because trees are good for representing hierarchy).

Each branch of the tree has a label associated with it.
- At the top of the tree we have the root.
- The branches of the root correspond to entities.
- The branches at the next level down correspond to properties.
- And the branches at the bottom level correspond to values.

For example, Lampson is an entity, Password is a property of Lampson, and vnxm56 is the value of Lampson’s password.

—> Which are the desired properties of a name service?



Design goal #1: scalability

- Must support an arbitrary number of names + administrative organizations

It should have scalability.

—> What is scalability?

- The ability to grow.
    - (1) Have many users, have many components.
    - (2) Its properties change with the number of users or components in a “manageable” way. E.g., user-perceived throughput or latency remain stable with the number of users/components.
- When we talk about scalability, we should be clear about which particular property we are talking about — number of users, throughput, latency, per-node memory and compute requirements…
- Ideally, we can give a mathematical formula that precisely describes scalability.
- The more precisely we can describe it, the better.

—> What kind of scalability does BL want?

- Max number of names: arbitrary.
- Max number of administrative organizations: arbitrary.

—> Why is it hard to support an arbitrary number of names?
Arbitrary number of names = arbitrary amount of resources
(at the very least memory to store all the mappings).

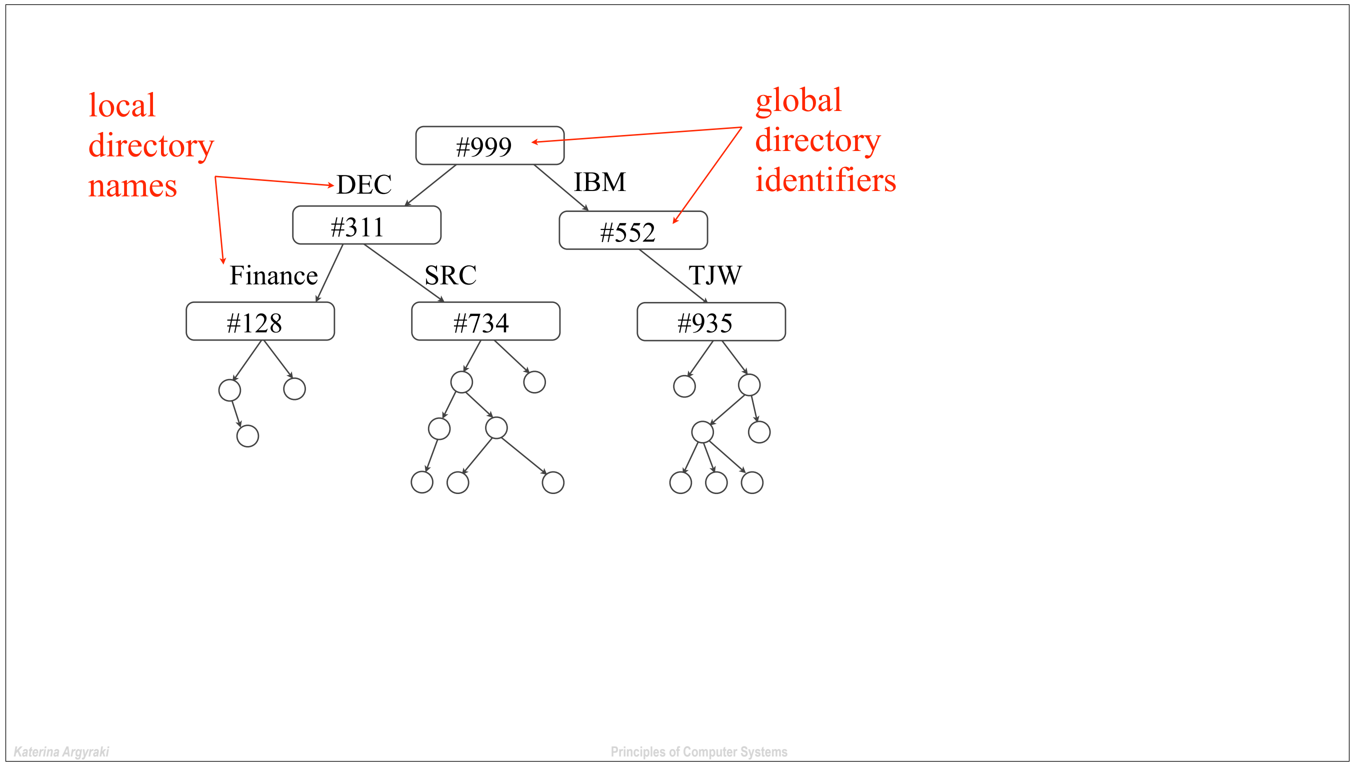
—> What are examples of systems that require an arbitrary amount of resources?

DNS, P2P systems like BitTorrent, search engines…

—> How do these systems do it?

- DNS and P2P are decentralized systems, where multiple resource owners collaborate.
  Every entity that introduces new names/content (so, new load) also contributes resources.
- Search engines are typically owned by a single organization. Every entity that introduces new load also contributes revenue through ads.

—> How does GNS do it?
He goes for a decentralized design, where we have…



local directory names

global directory identifiers

multiple directories,
each one potentially owned by a different organization,
themselves organized as a tree.

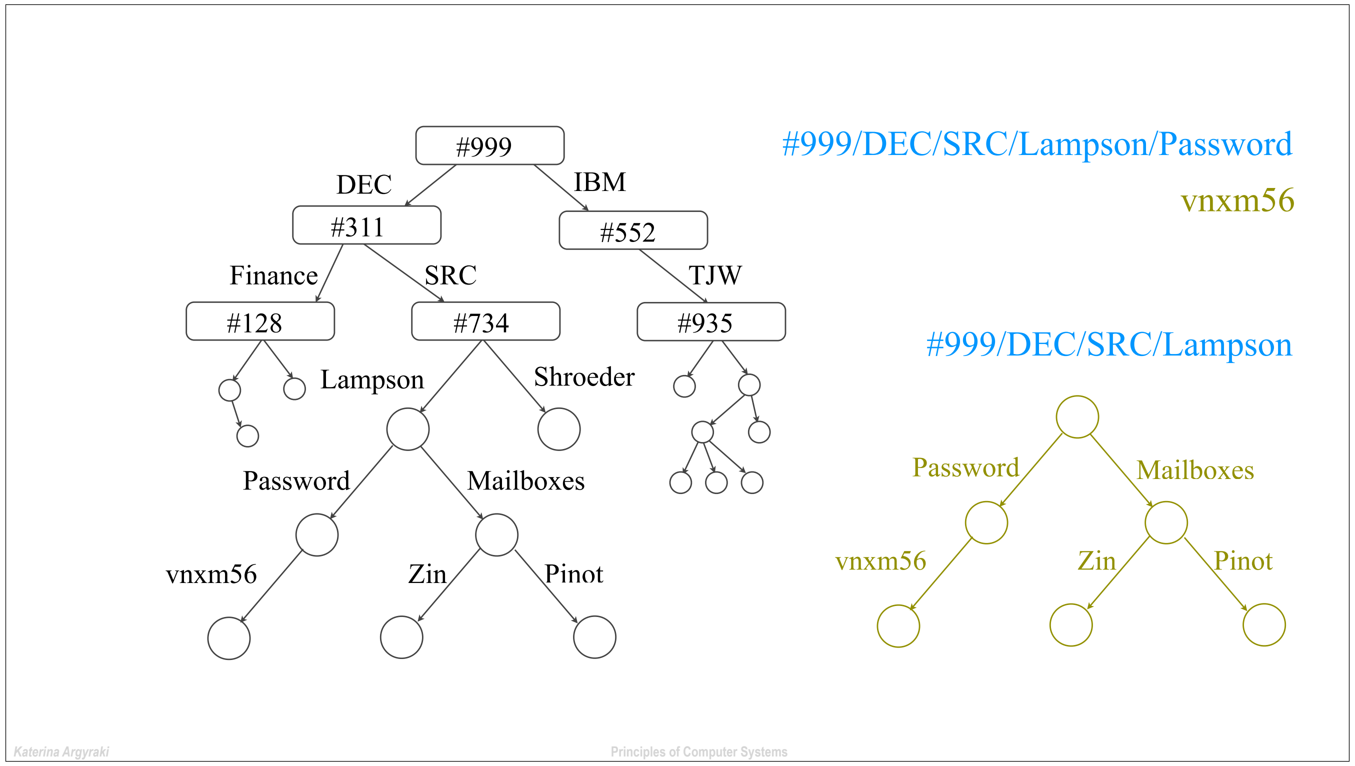
Since there are multiple directories, we need a way to tell them apart,
and for that we use:

- directory identifiers that are globally unique;
- and directory names that are unique locally, within their parent directory.

Do not be confused by the fact that one is called a directory *identifier* whereas the other is called a directory *name*.
Both of these are names -- they are different ways to refer to directories.
The fundamental difference between a directory identifier like #311 and a directory name like SRC is that

- the former is globally unique,
- whereas the latter is unique only within its parent directory.

For example, IBM could also have a child directory named SRC.



#999/DEC/SRC/Lampson/Password

vnxm56

#999/DEC/SRC/Lampson

—> Are the labels of the tree hanging from each directory private or global?

Private.

Examples of names…

—> How does this design achieve scalability?

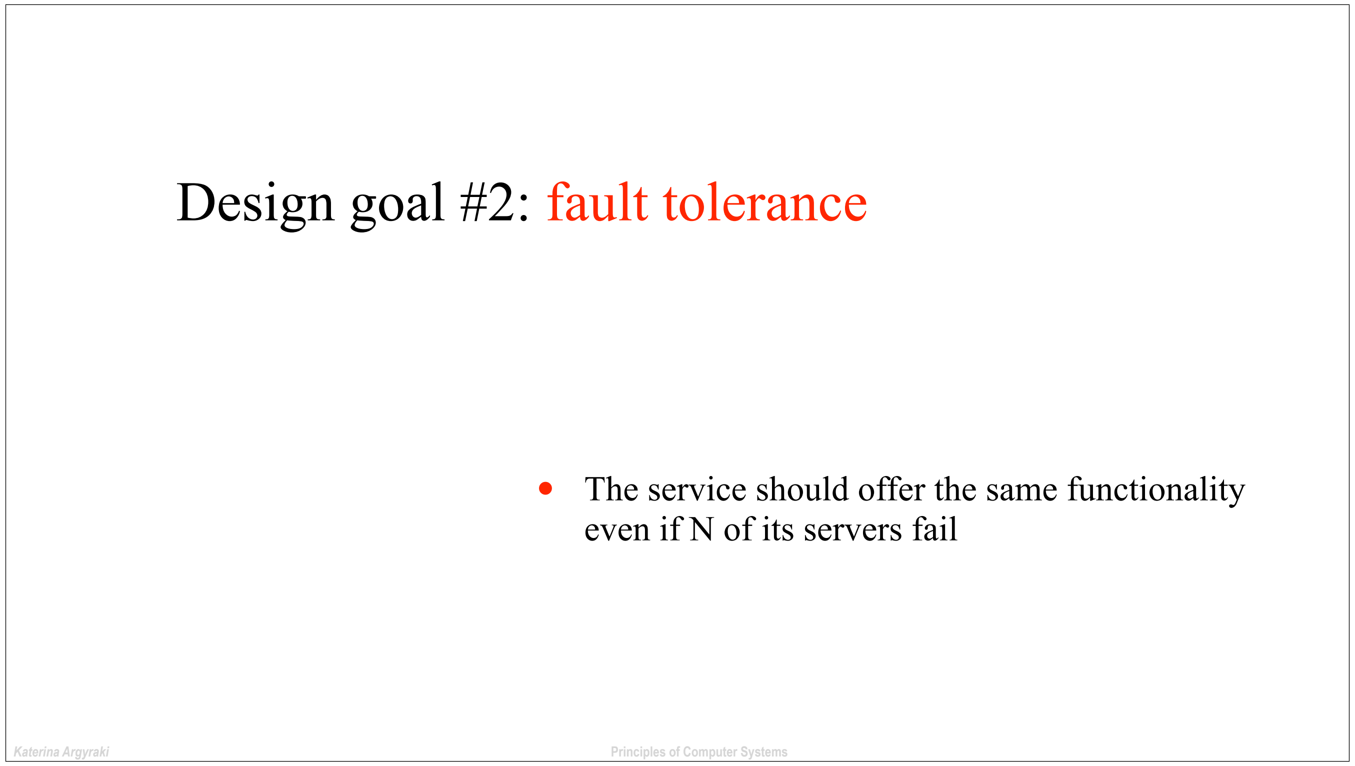
A tree can have an arbitrary number of nodes = names and organizations.

The fact that each organisation can own and manage its own directory makes the resource cost manageable.

The fact that each directory has its own namespace makes the cost of human effort manageable: operators do not have to check with each other before creating a new subdirectory.

Design goal #1: scalability

- Achieved through a hierarchy of directories, each potentially owned by a different entity, each with a private namespace



Design goal #2: fault tolerance

- The service should offer the same functionality even if N of its servers fail

A good name service should also have fault tolerance.

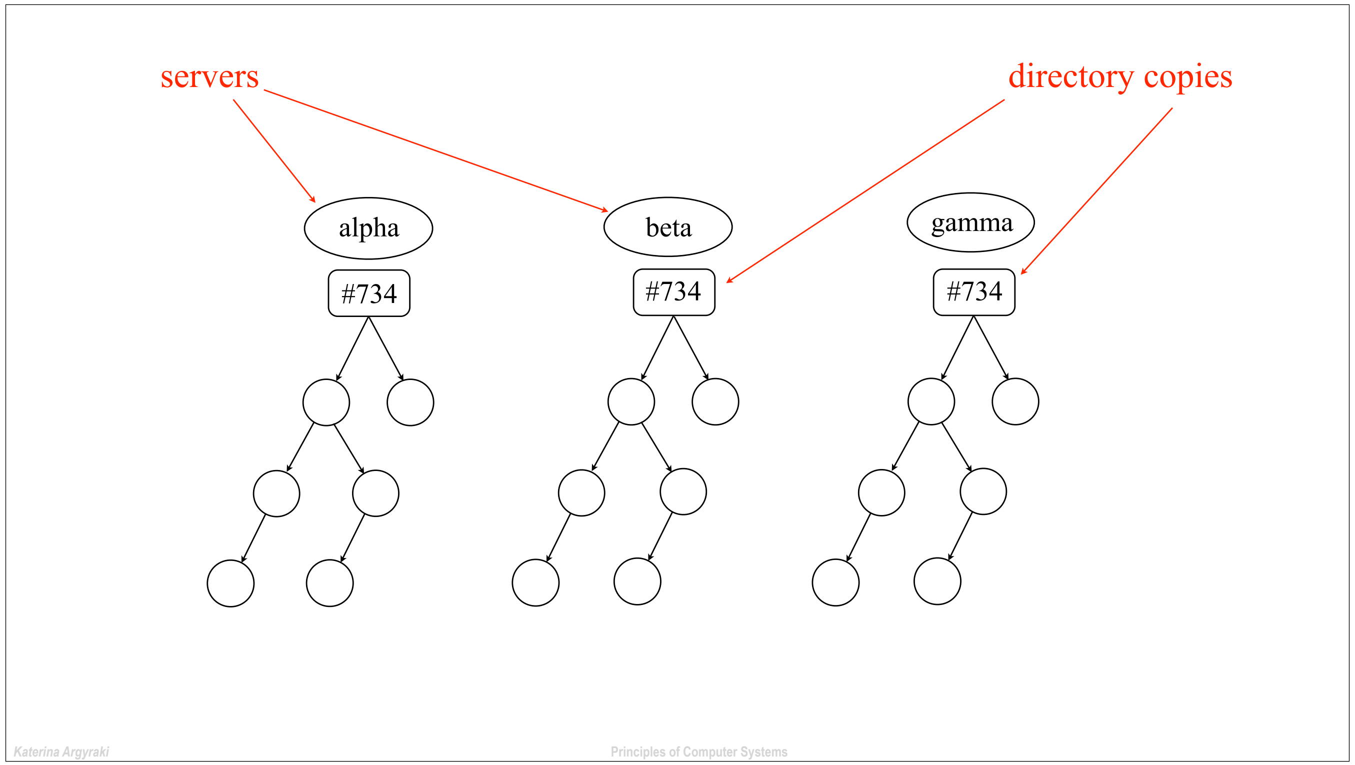
—> What is fault-tolerance?

- The ability to operate successfully in the face of infrastructure failures.
- When we talk about fault-tolerance, we should be clear about how many and what kind of infrastructure failures we are talking about.

—> What kind of fault-tolerance does BL want?

Service operation should be unaffected if up to N of its servers fail.

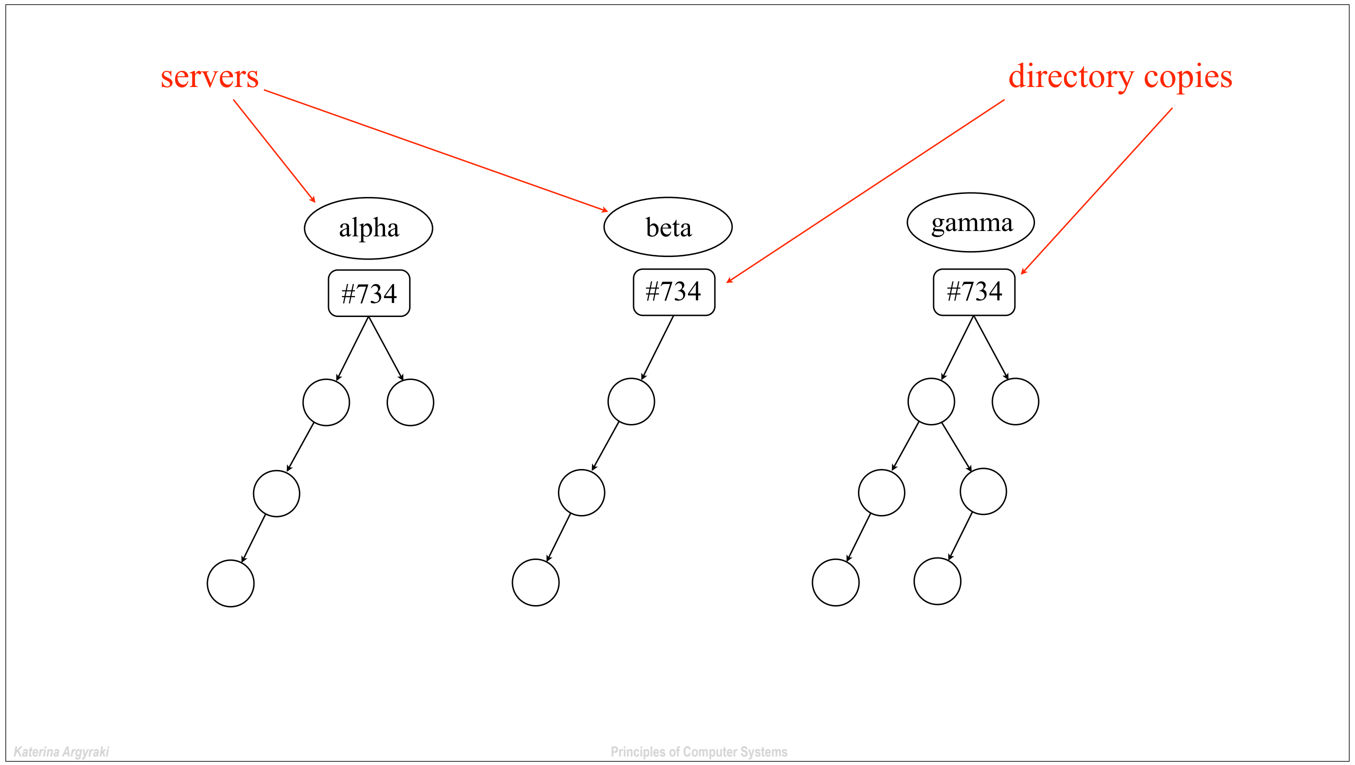
—> How does GNS achieve this?
It employs…



redundancy:

- it has multiple servers,
- and each directory is copied in at least N+1 of them.

—> What challenge does redundancy introduce?
How to keep different copies…



consistent.

The designer of the system must decide what kind of consistency the system must have.
Differently said, which are the valid states of the system that can be visible to the users?
If 3 users access directory #734, and they happen to access 3 different copies of it, is it valid that they see 3 different views?

The kind of consistency that a service provides is a key part of the abstraction that the service provides to its users.

Scenario #1:

- Any users accessing the same directory at the same point in time must get the exact same view.
- This is strong consistency.
- Differently said, the service provides to its users the abstraction of a single tree.

—> Can the system provide this abstraction, this kind of consistency?
Sure: Whenever a user updates a directory, the system does not serve any other users, until it has updated all copies of that directory.

—> Is this always a good idea?
It depends.
The designer of the system must

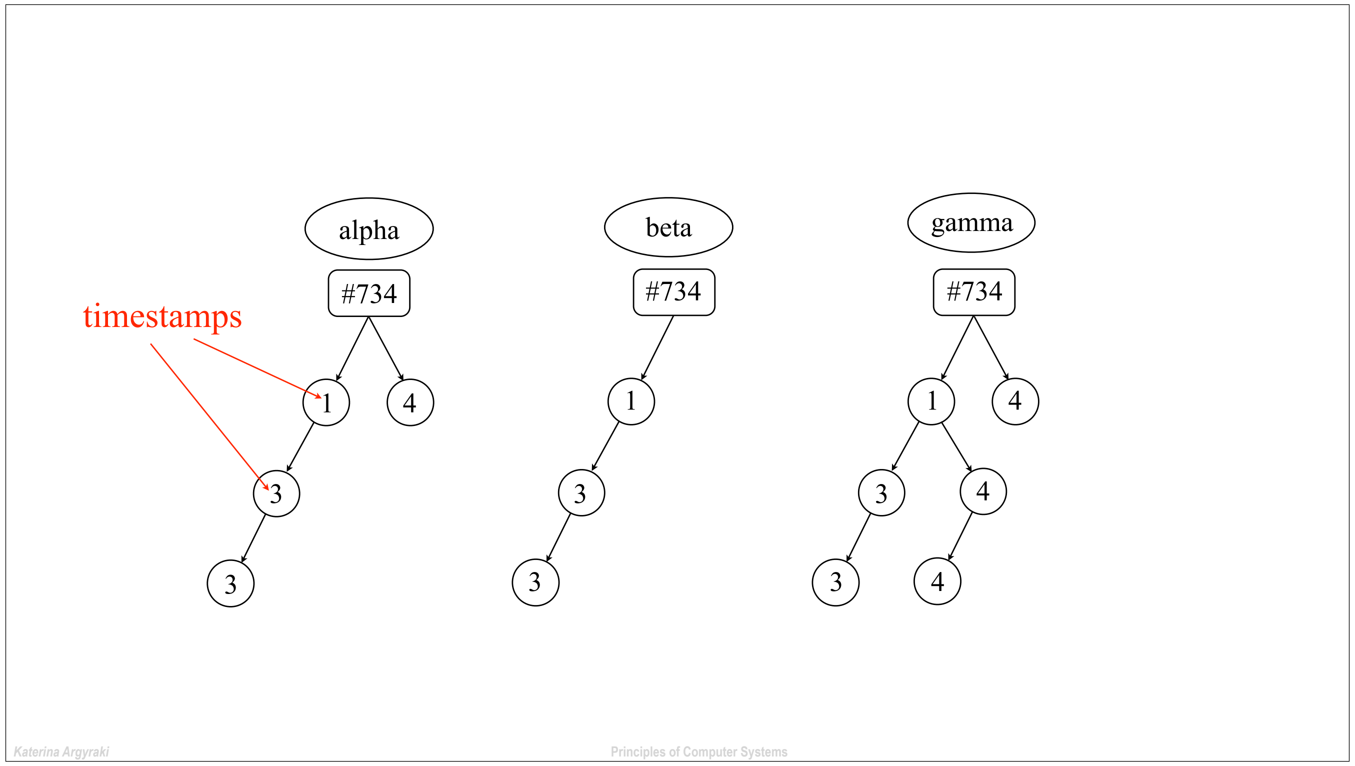
- decide what rate of updates the system must support
- and weigh that against the amount of time it takes to update all copies of a directory.

So, there is a trade-off between consistency and maximum supported update rate.

Scenario #2:

- If we freeze updates to the system, it will eventually reach a state where any users accessing the same directory will get the exact same view.
- This is eventual consistency.
- What abstraction does this service provide to its users? A collection of trees that may differ arbitrarily (!)
- Many services today provide this abstraction, because they implement it such that in practice the trees differ very little most of the time. That does not change the fact that the abstraction does not guarantee anything about how inconsistent the views of different users may be.

This is the abstraction, the kind of consistency, that BL chose.
—> How does GNS provide it?



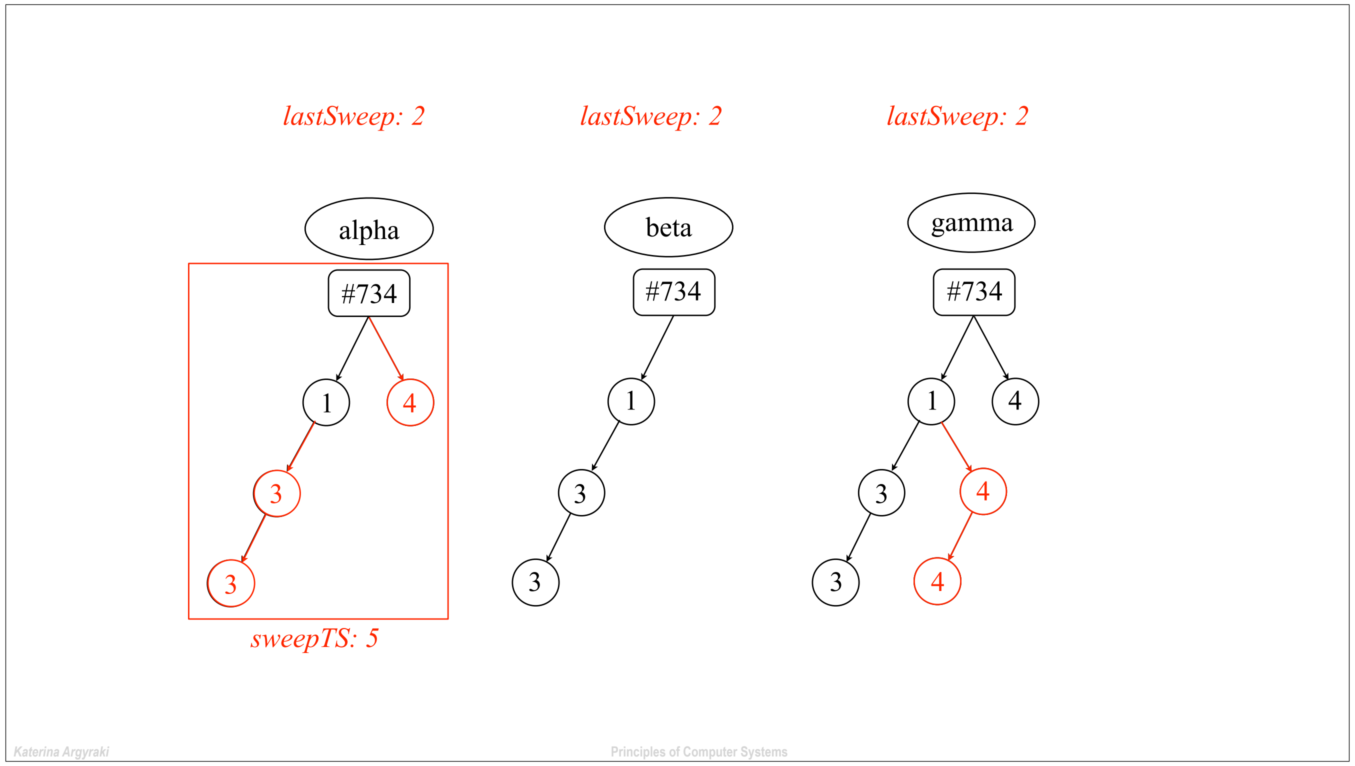
Whenever a user updates a directory, this update is immediately applied to *one* directory copy (DC).
The server that hosts the DC associates the update with a timestamp.

And then there is a special operation that is called a “sweep,”
which travels from one DC to another and synchronizes them.

The sweep occurs in two phases:

- In phase 1, it passes by each DC and reads all updates that occurred since the last sweep.
- In phase 2, it passes again by each DC and applies to the DC all updates collected in phase 1 that have not been already applied to the DC.

Let’s look at the details through…



an example.

Each DC has a “lastSweep” variable associated with it, which indicates when this DC was last synchronized through a sweep.
E.g., these three DCs were last synchronized through a sweep at time 2.

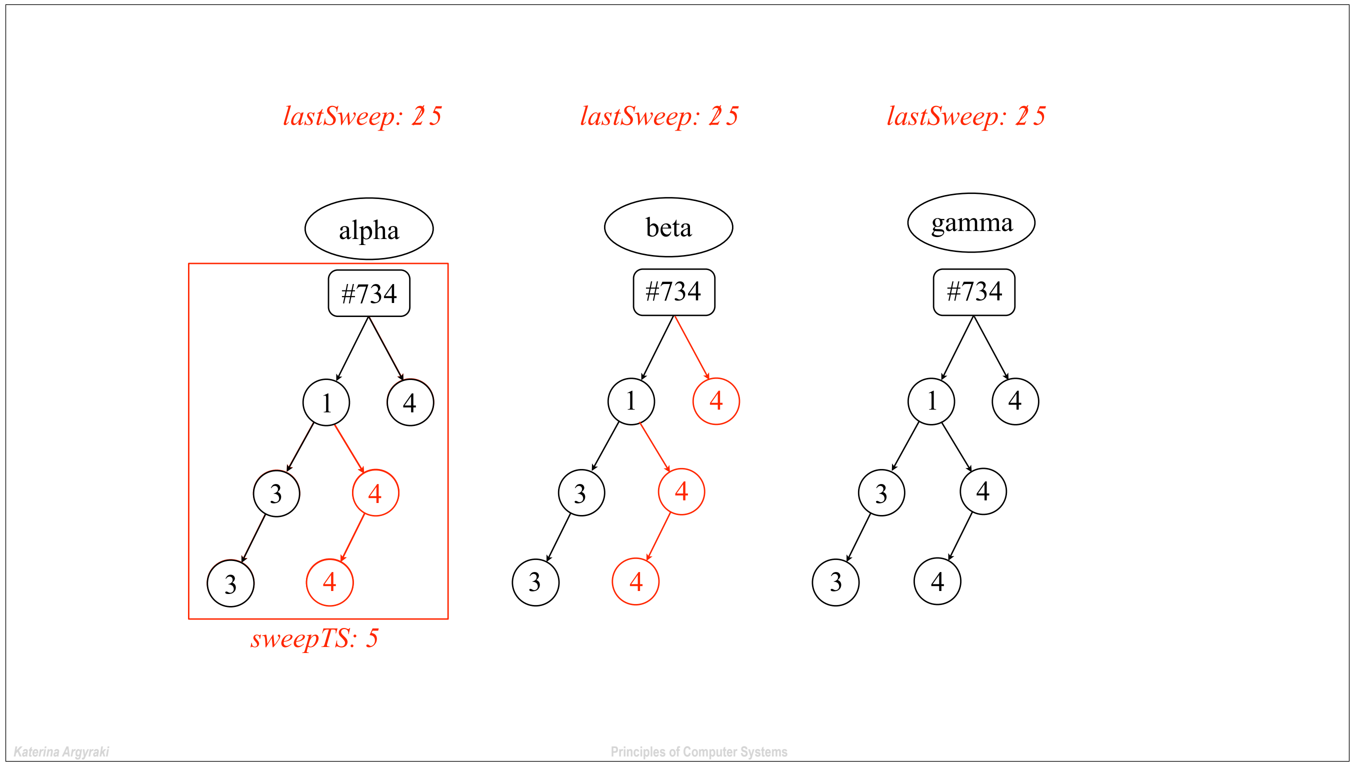
Suppose a new sweep arrives at the first DC.
Each sweep has a “sweepTS” variable associated with it.
E.g., this sweep has sweepTS 5.

The sweep checks:

- What is the value of the last sweep?
- Are there any updates that were made on this DC since that time?

Differently said, are there any branches/nodes that have timestamp 2 or higher?
Then the sweep reads these updates and moves to the next DC.

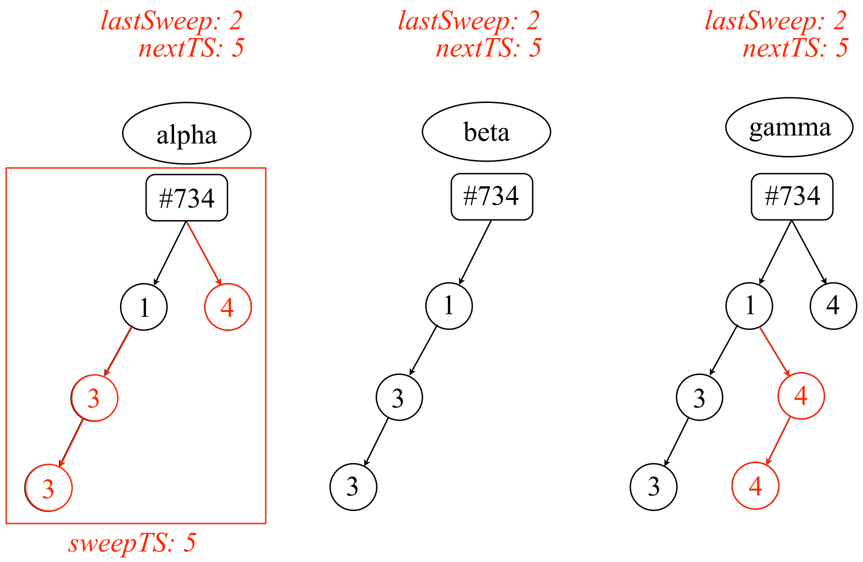
The same thing happens at every DC that the sweep reaches:
it checks for updates that have occurred since the last sweep and reads them.



Then there is a second phase.

When the sweep arrives for the second time at a DC, it checks: Have I collected during phase 1 any updates that are not present in this DC? If yes, it applies the missing updates to the DC, then moves to the next DC.

—> Is this scheme correct? Could an update reach a subset of the DCs?

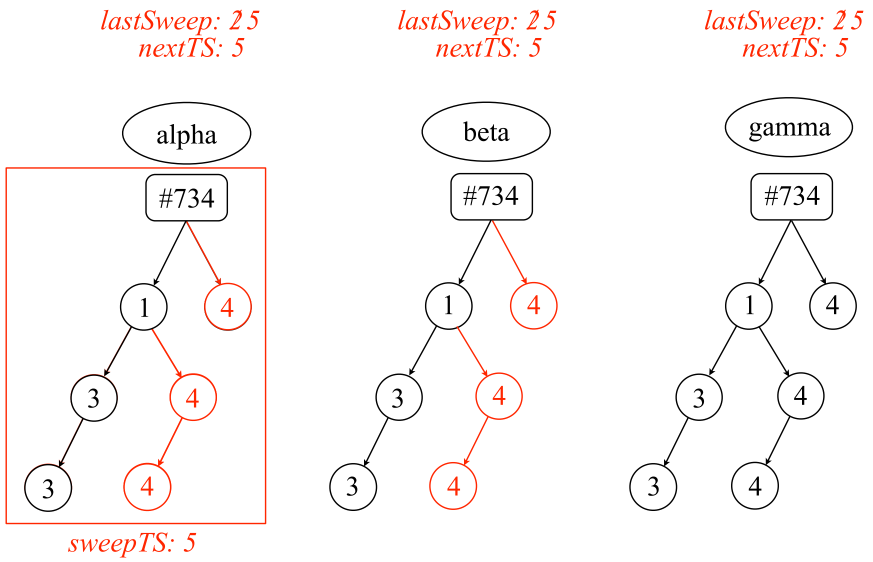


lastSweep: 2
nextTS: 5

lastSweep: 2
nextTS: 5

lastSweep: 2
nextTS: 5

sweepTS: 5



lastSweep: ~~2~~ 5
nextTS: 5

lastSweep: ~~2~~ 5
nextTS: 5

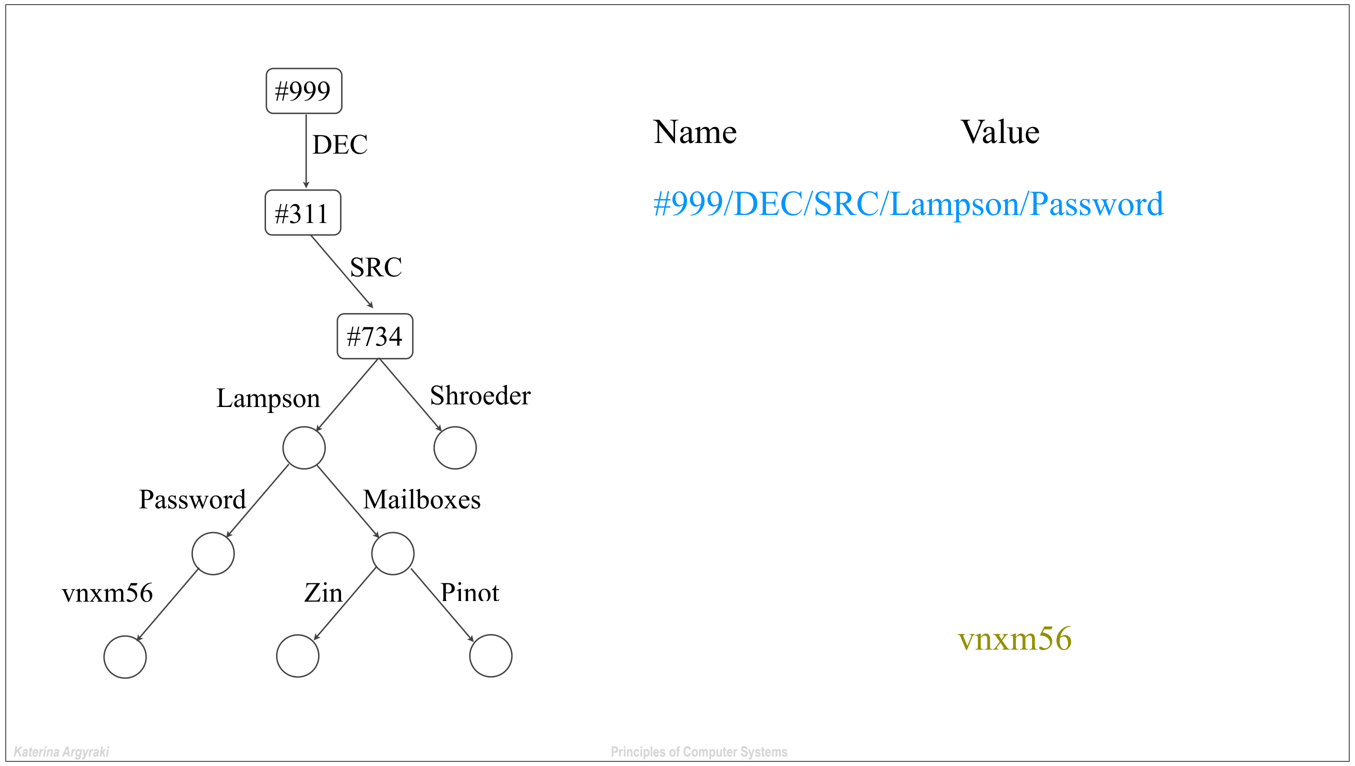
lastSweep: ~~2~~ 5
nextTS: 5

sweepTS: 5

Design goal #2: fault tolerance

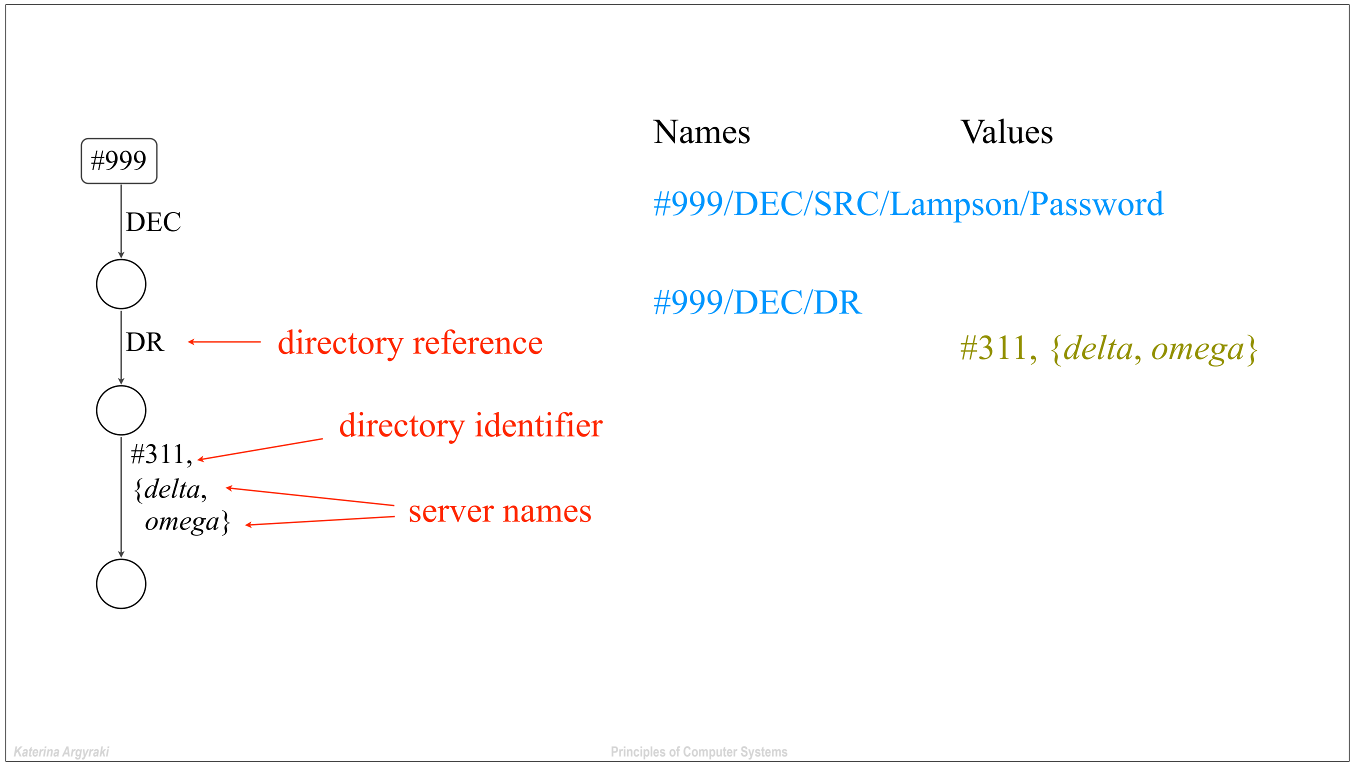
- Achieved through redundancy
  - *redundant directory copies on different servers*
- Combined with eventual consistency
  - *copies periodically synchronized through sweep*

Name lookup



Name	Value
#999/DEC/SRC/Lampson/Password	vnxm56

Suppose we have this directory,
and a client wants to lookup (know the value of) this name.



Names	Values
#999/DEC/SRC/Lampson/Password	
#999/DEC/DR	#311, {*delta*, *omega*}

The one piece of information that every client must have is the identifier of a server that stores a copy of the root directory.

So, the client can send a query to the root directory.
What must she ask?

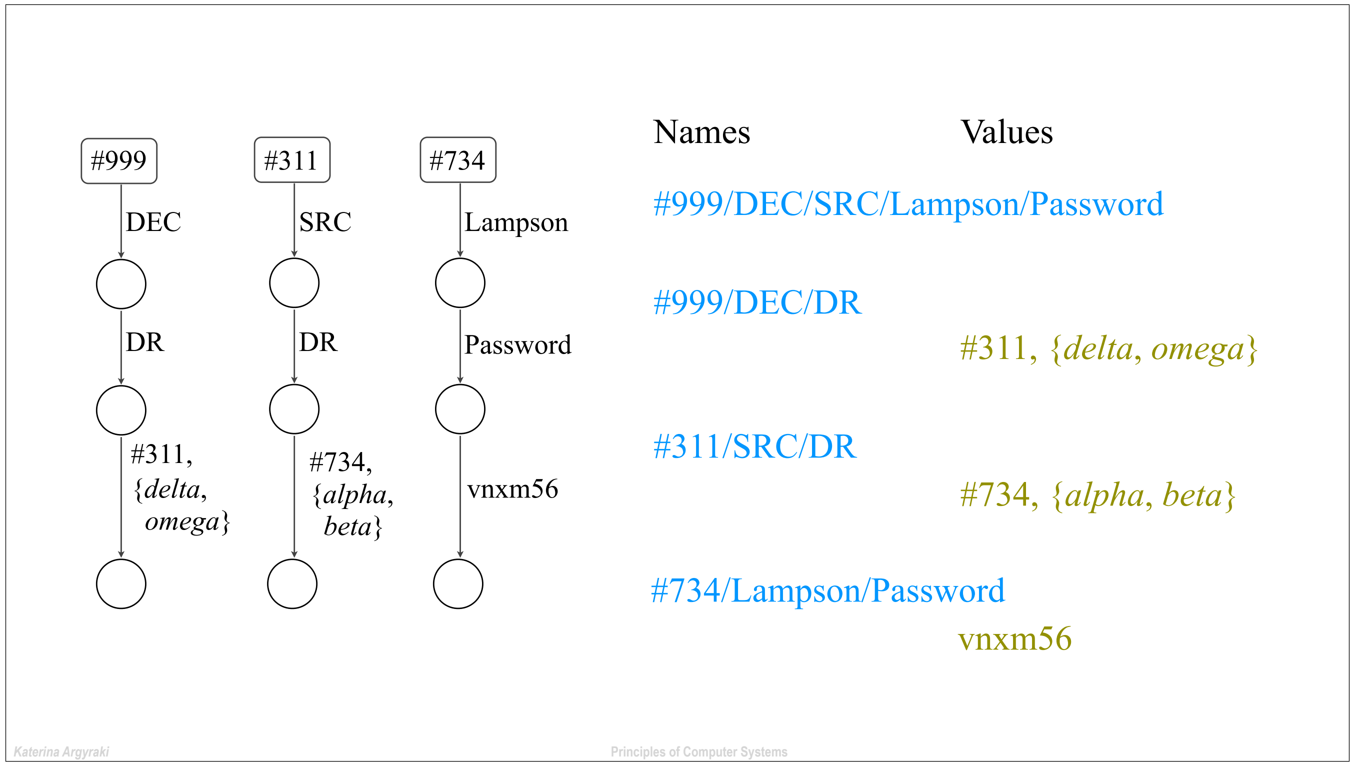
The root directory stores at least two pieces of information about the DEC directory:

- Its global identifier.
- Names of servers storing a copy of the DEC directory.

How exactly does the root directory store this information?
Well, it's a directory, and, in this system,
a directory stores data in the form of a 3-level tree (entities, properties, and values).

So, the root directory has a branch that is labeled DEC, which points to another branch labeled DR,
which points to another branch labeled with the global identifier for the DEC directory and names of servers that store copies of the DEC directory.

—> So, what is the name that the client will look up in the root directory?
#999/DEC/DR.



Names	Values
#999/DEC/SRC/Lampson/Password	
#999/DEC/DR	#311, {*delta*, *omega*}
#311/SRC/DR	#734, {*alpha*, *beta*}
#734/Lampson/Password	vnxm56

Now the client knows the name of a server that stores a copy of directory #311 and (once it has mapped the name to an address/identifier) the client can send a query to the server. What name must the client query?

The #311 directory has a branch that is labeled SRC,
which points to another branch labeled DR,
which points to another branch labeled with the global identifier for the /SRC directory and identifiers of servers that store copies of the /SRC directory.

—> So, what name must the client look up in the #311 directory?
#311/SRC/DR.

Now, finally, the client knows the identifier of a server that stores a copy of directory #734,
which is the directory that contains the entity and property she is interested in.

—> So, what name must the client look up in the #734 directory?
#734/Lampson/Password.

—> Does the client need to know which part of the name is directories and which part is not?

- Yes, because they need to put together the correct name lookups.
- This is why we say that the abstraction exposed to the client is a tree of directories + a tree for each directory.
- So, the interface between a service and its clients determines the abstraction that the service exposes to its clients.

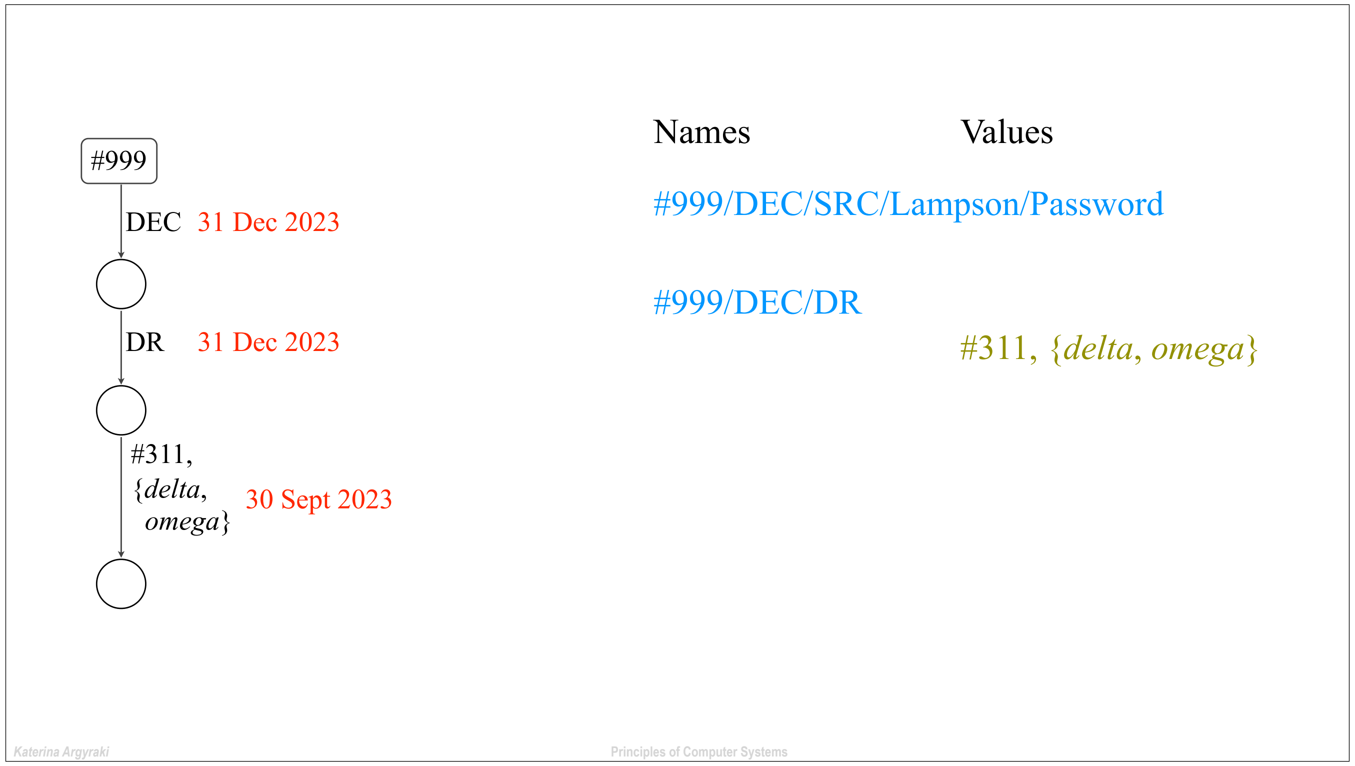
—> Is there anything that is bothering you about this picture?
The client must query three directories in order to resolve this one name.

Performance may suffer.

—> How can we overcome this problem?
With caching: any client of the system that completes a successful lookup can cache the result for some period of time.

—> What is the challenge with caching?
Maintaining consistency between the true and the cached name/value mappings.
What if a client caches Lampson’s password for an hour, but, in the meantime, the value of the password has changed?

—> How can we deal with this challenge?



#999

DEC 31 Dec 2023

DR 31 Dec 2023

#311, {*delta*, *omega*} 30 Sept 2023

Names	Values
#999/DEC/SRC/Lampson/Password	
#999/DEC/DR	
	#311, {*delta*, *omega*}

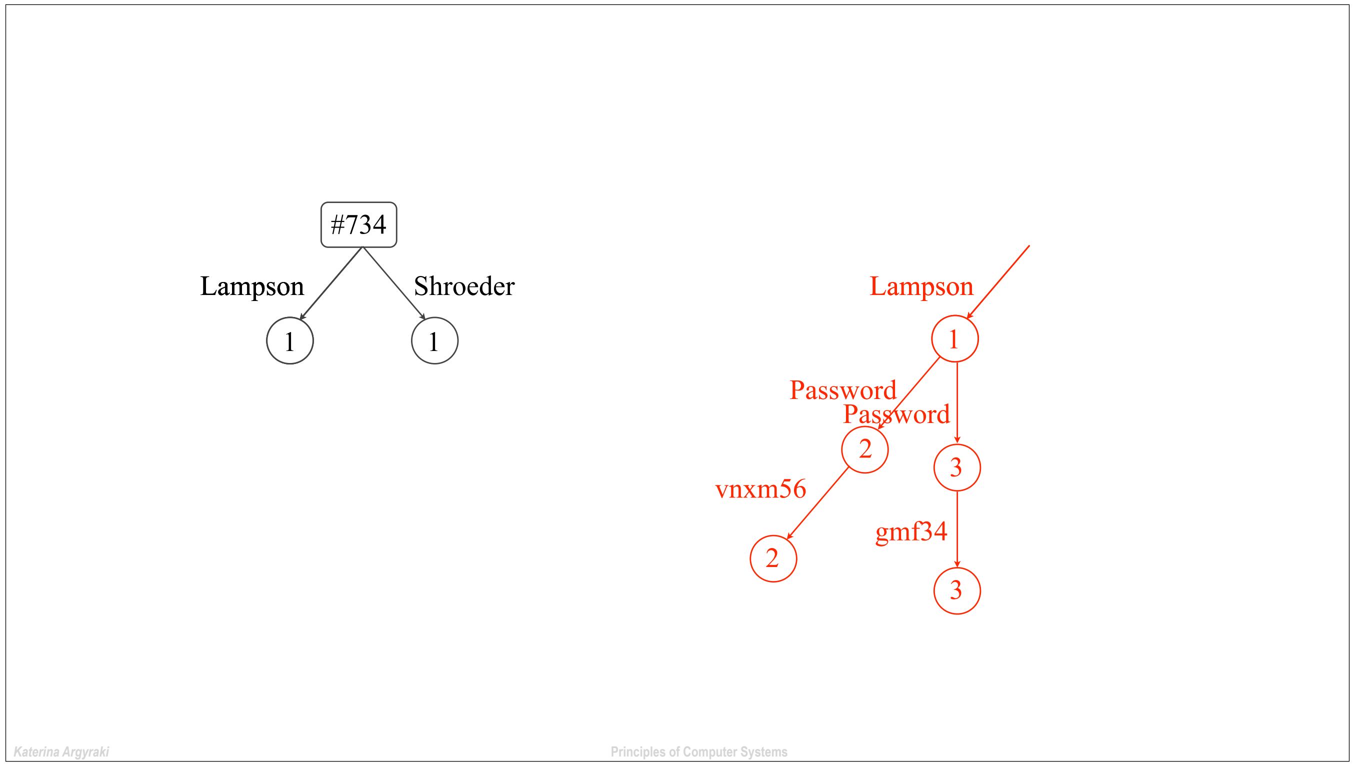
With expiration times:
The system associates with every branch of the tree an expiration time.
Hence, a client can cache a name/value pair until the earliest expiration time of any of the involved components.

—> What constraint do expiration times introduce?
They constrain the rate at which the system can be updated.
The further away the expiration time, the longer clients can cache a mapping,
but the longer it takes until that mapping can be updated.

Name lookup

- Clients can cache mappings to reduce latency
- Stale mappings avoided through expiration times

Updates



Suppose we have the directory on the left,
and a user submits to it the update on the right
(a value for Lampson’s password).

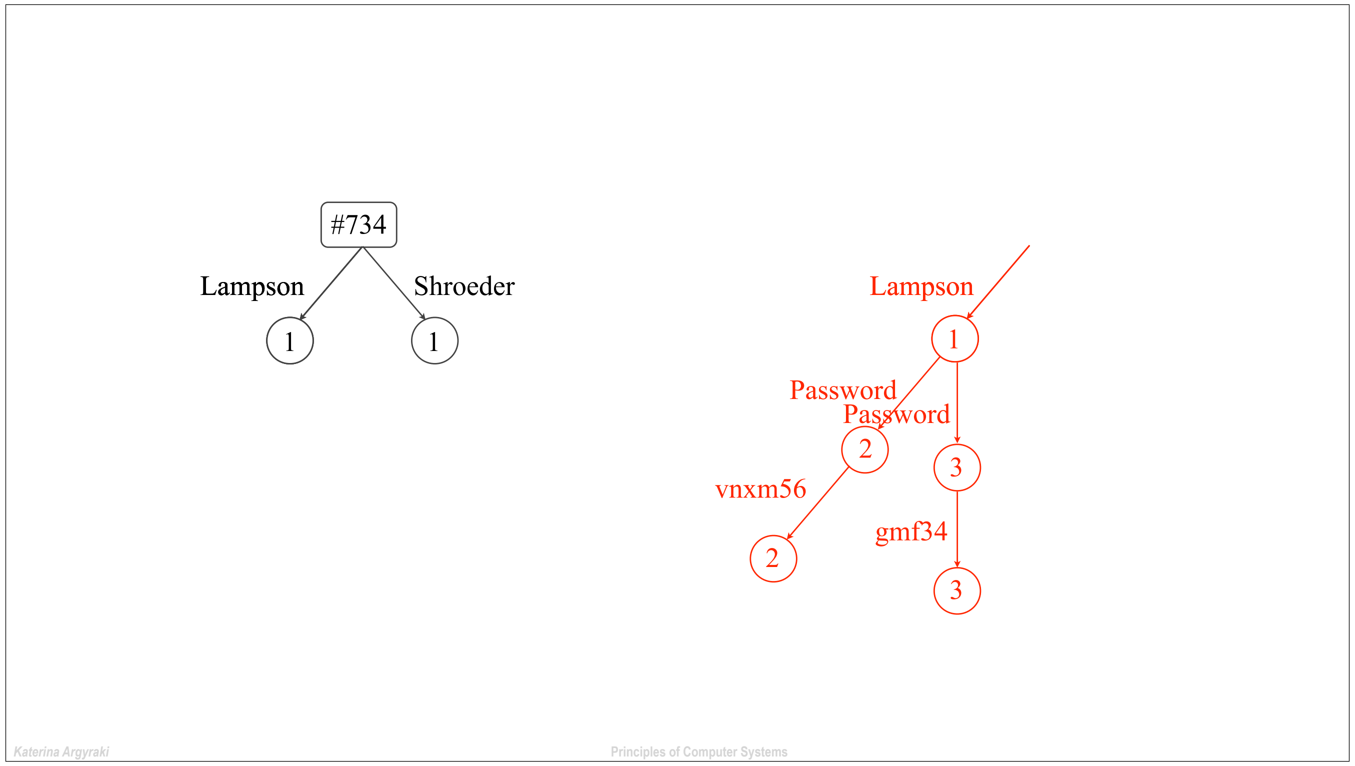
The system:

- Identifies the biggest part of the directory that is a prefix of the update.
- Discards the part of the update that is equal to that prefix.
- Appends the remaining update to the prefix.

Now suppose a user submits a new update to the directory that involves the same entity and property (a new value for Lampson’s password).

The system:

- Identifies the biggest part of the directory that is a prefix of the update.
- Discards the part of the update that is equal to the prefix.
- Next it should append the remaining update to the prefix, but there already exists a branch named Lampson. The current Lampson branch has timestamp 2, whereas the new Lampson branch has timestamp 3, which is more recent, so, the update wins: The system deletes the current Lampson branch and its children and appends the update to the prefix.

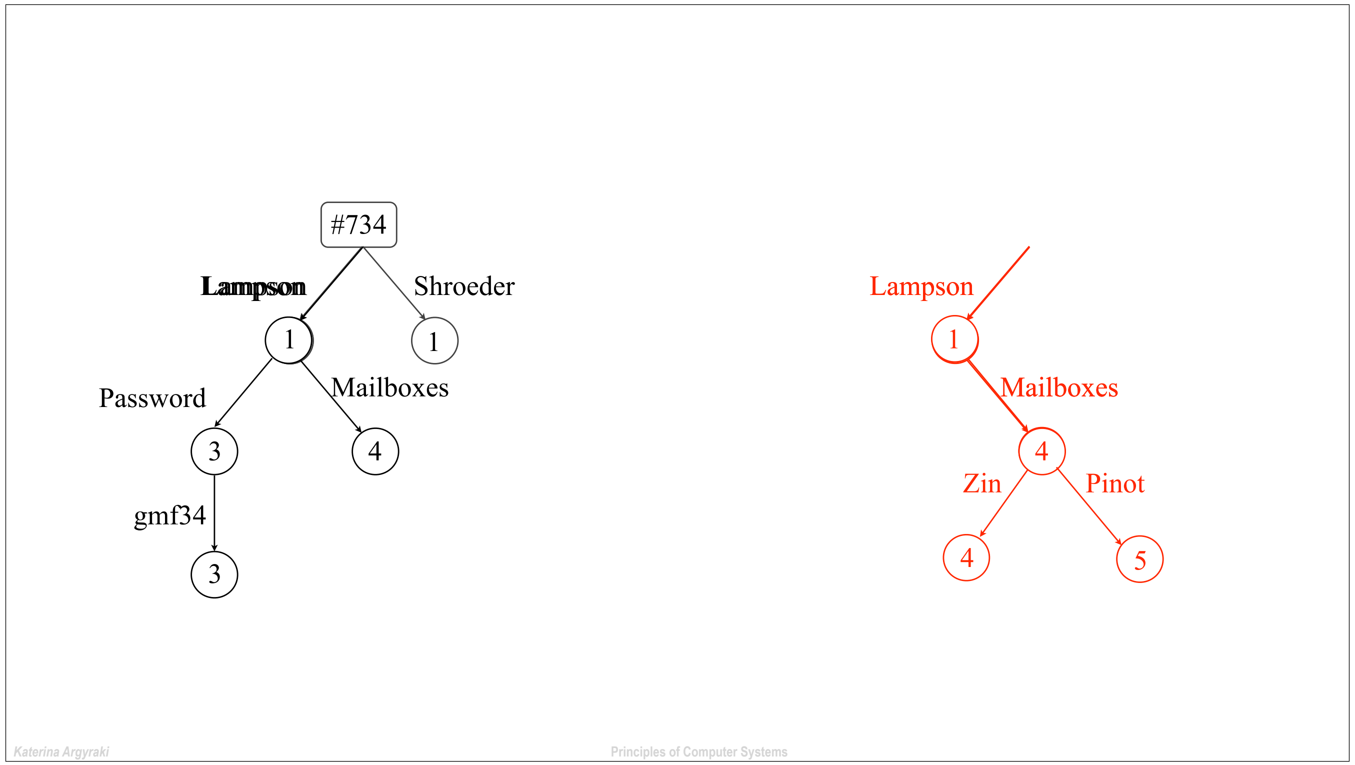


Now let’s apply the same updates, but in the opposite order: …

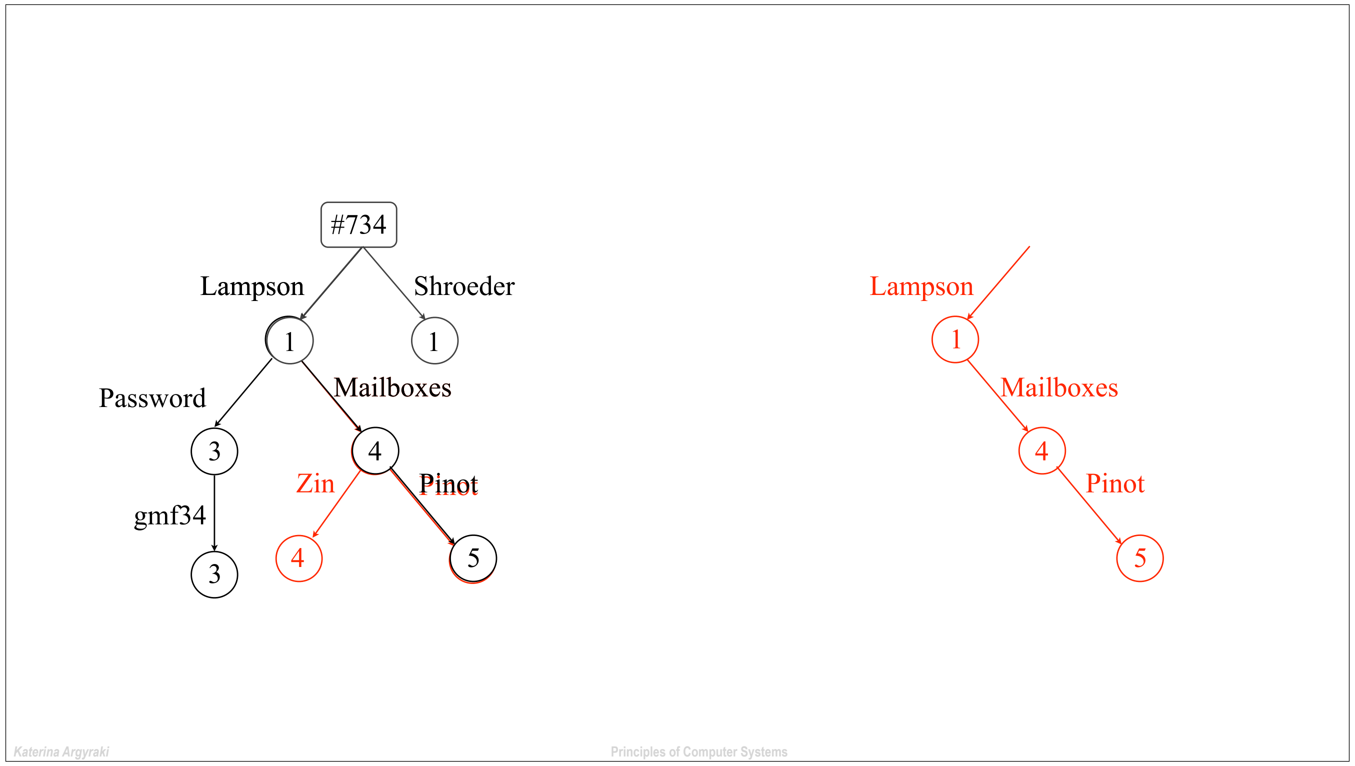
The point is that the outcome of a sequence of updates is the same, independently from the order in which they were submitted.

We call these updates commutative.

In order to have commutative updates, the system relies on the timestamps to determine which parts of an update should affect the system.



Suppose a client submits two updates to the directory …

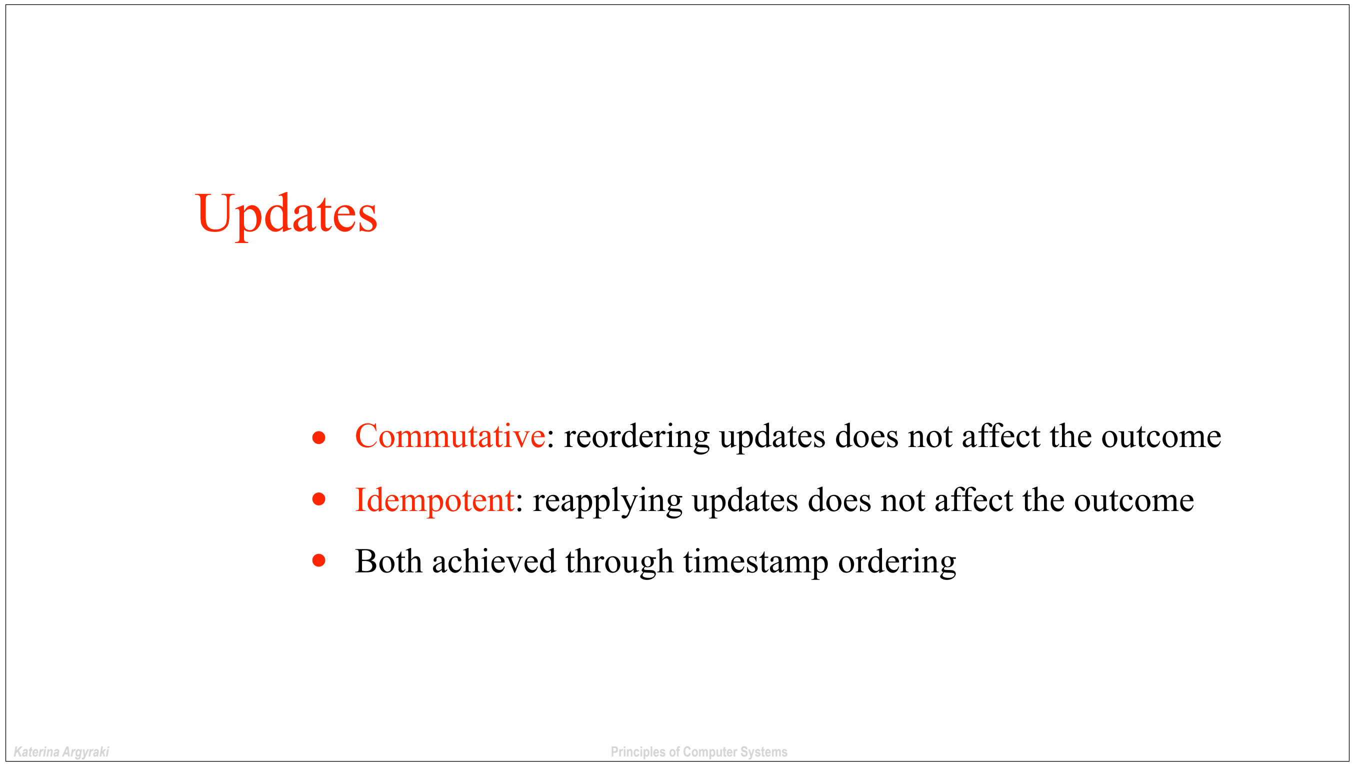


Now let’s apply the second update again…

The point is that applying the same update again has no effect on the system.

We call these updates idempotent.

In order to have idempotent updates, the system relies, again, on the timestamps.



Updates

- Commutative: reordering updates does not affect the outcome
- Idempotent: reapplying updates does not affect the outcome
- Both achieved through timestamp ordering

—> Why would a server see the same update twice, or updates with out-of-order timestamps?

Designing a Global Name Service

- Scalability: supports an arbitrary number of names and organizations
  - *achieved through hierarchy*
- Fault-tolerance: keeps functionality even when N servers fail
  - *achieved through redundancy* + *eventual consistency*
- Clients cache mappings
- Stale mappings avoided through expiration dates
- Updates are commutative and idempotent