THE UNIVERSITY OF
CHICAGO

Class 4: Naming

MPCS: 52040
Distributed Systems

Kyle Chard

chard@uchicago.edu

Course schedule (* things might change a little)

Date	Lecture discussion topic	Class	Assessment
January 10	1. Introduction to Distributed Systems	Intro/Logistics	
January 17	2 Distributed architectures 3 Processes and virtualization	1 Docker	Homework 1 due
January 24	4 Networks and Communication	2 RPC/ZMQ/MPI	Homework 2 due
January 31	5 Naming	3 DNS/LDAP	Homework 3 due
February 7	6 Coordination and Synchronization		**Mid term exam**
February 14	7 Fault tolerance and consensus	Raft Project description	Homework 4 due (Project released)
February 21	8 Consistency and replication	4 FaaS	
February 28	9 Distributed data	5 Distributed data	
March 7	10 Data-intensive computing	6 Streaming	**Project due (March 9)** **Final exam (March 14)**

Naming

- Part 1: Names and addresses
- Part 2: Flat naming
- Part 3: Structured naming
- Part 4: Attribute-based naming

Naming

Names play a critical role in all computer systems

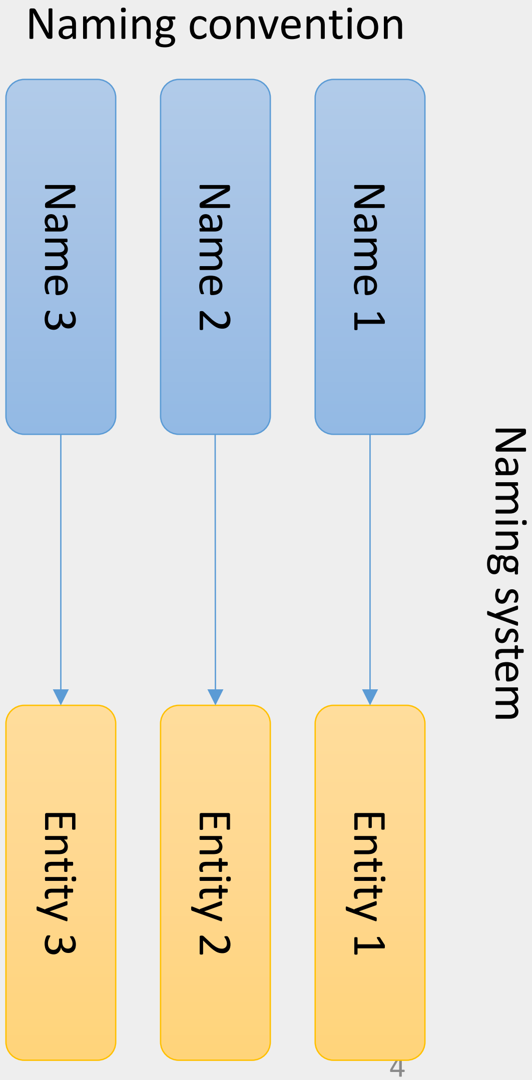
- Share resources, uniquely identify entities, refer to locations, provide human readability

To access an entity you must resolve the name to find the entity's address

- Name resolution is performed by a naming system

Distributed systems rely on distributed naming systems

- How we do this directly affects efficiency and scalability



What is a name?

String of characters used to refer to an entity:

- Kyle (user)
- Chard (cnetid)
- chard@uchicago.edu (email)
- 0000-0002-7370-4805 (ORCID)
- www.uchicago.edu (website)
- JCL362-Chinkapin-NPI27E65 (printer)
- homework1.pdf (file)
- Kyle-laptop (hostname)

Why don’t we just use addresses?

We want location independence: a name that is independent from its address

Why? flexibility and ease of use

- E.g., addressing, migration, replacement, etc.

Central theme: resolve names to addresses

In principle, a naming system maintains name-address bindings

- Simply a table (name, address)
- Will this work in a distributed system?
  - Unlikely when you consider spanning large and wide networks with many resources

What we aim to do is distribute:

- the name-address table, and
- the mechanism by which name-address are resolved

Identifiers and pure names

A true identifier is a name that has the following properties

1. An identifier refers to at most one entity
2. Each entity is referred to by at most one identifier
3. An identifier always refers to the same entity (i.e., it is never reused)

A pure name is a name that has no meaning (just a string)

- E.g., a MAC address provides no indication of where it is: **00-14-22-01-23-45**
- A DOI (should) provide no notion of what the name is: 10.1109/MCC.2014.52

Much easier to unambiguously refer to an entity

Naming

- Part 1: Names and addresses
- Part 2: Flat naming
  1. Simple solutions
  2. Home-based solutions
  3. DHTs
  4. Hierarchical approaches
- Part 3: Structured naming
- Part 4: Attribute-based naming

Part 2: Flat naming

Unstructured sequence of characters

- No information is encoded in the name that can help us locate the entity to which it refers
- E.g., SSN, MAC address

How do we resolve flat names? Simple solution: Broadcasting

If we happened to have efficient broadcast (e.g., in a LAN where all machines are connected to a single cable/switch) we could:

- Send message to each machine with the desired name
- Look up local name to see if the message is for me
- Only reply if I can offer an access point to that entity
- (The Address Resolution Protocol works like this)

Unfortunately, broadcasting is not efficient as network sizes grow

- Network bandwidth wasted, hosts interrupted by unnecessary requests, no global knowledge of entire system

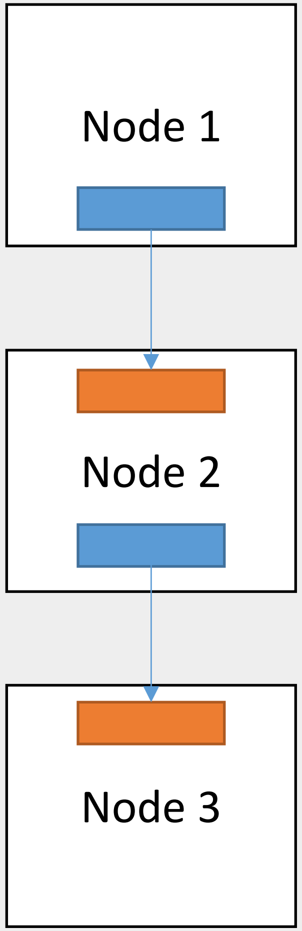
How do we resolve flat names?
Simple solution: forwarding pointers

When an entity moves from A to B what should we do?

- Look up address each time
- Leave behind a reference to its new location

How would this work with remote objects (using RPC)

- Each forwarding pointer is implemented as a (client stub, server stub) pair
- Server stub is either local reference to entity, or a reference to a remote client stub
- Whenever an object moves it leaves behind a client stub on A and installs a server stub on B



Discussion: pros and cons of this approach

Pros:

1. Simplicity: as soon as the entity is located a client can look up the current address by following the chain of references

Cons:

1. Chains can become long and expensive to resolve
2. Intermediaries must maintain their part of the chain as long as needed
3. Vulnerability to broken links as one missing link will mean the entity cannot be located

Home-based approaches

When mobility is common, a popular approach in large-scale systems is to introduce a home location

The home location tracks the current location of an entity

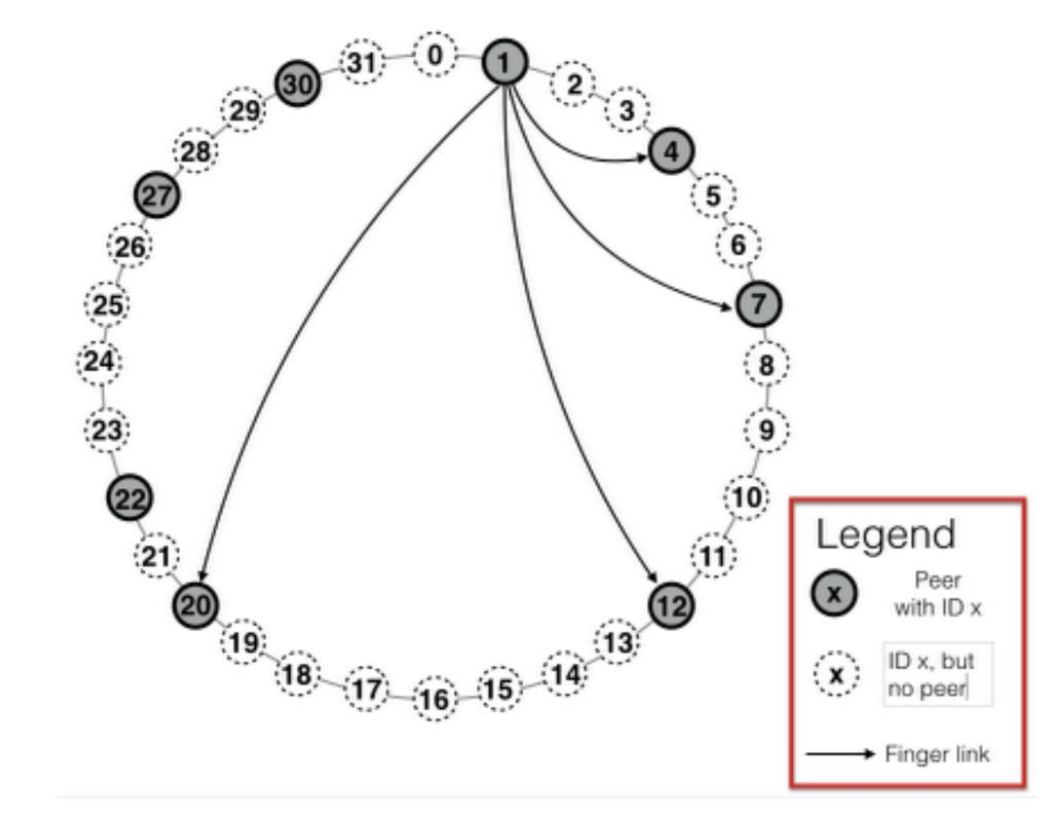
We often select the place where an entity was created as its home location

How can we efficiently distribute name resolution? Distributed Hash Tables

- Distributed system that provides lookup like a hash table
  - Stores key-value pairs, can be efficiently added/retrieved
- P2P model (all processes are equal)
  - Nodes are processes; edges are comms channels
  - Nodes can be added/removed with minimal effort and content redistribution
- Typically a structured overlay network (e.g., ring)
- Nodes are assigned random m-bit identifiers
- We now need to decide on which node to store a piece of data so that we can find it..
  - Each data item hashed to an m-bit key

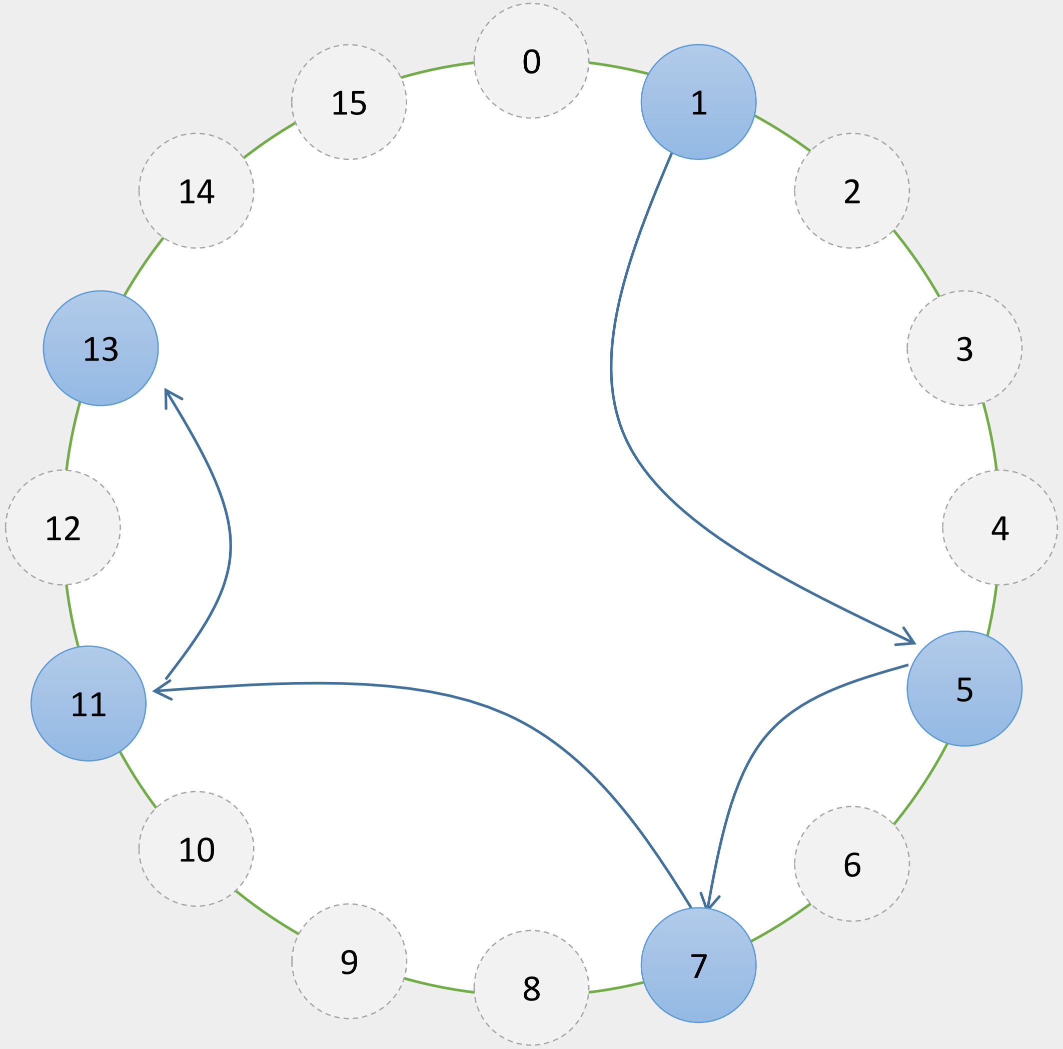
key(data) = hash(data value)

  - We store data with key *k* on node with smallest *id >= k*
  - Call this node the successor of k and denoted succ(k)



Name resolution in Chord

- Boils down to: how can we efficiently resolve key *k* to the address *succ(k)?*
- Easy solution: each node *p* should keep track of the successor node *succ(p+1)* and its predecessor *pred(p)*
- We can now apply a linear search:
  - When receiving a request to resolve a key it will be forwarded to one of its two neighbors (unless of course the key belongs to that node)



Finger tables

- Each node maintains a finger table (shortcuts) of size *s*
  - Where *s* <= *m* entries

$$FT_p[i] = succ(p + 2^{i-1})$$

  - Intuition: the *i*-th entry points to the first node succeeding *p* by at least 2^{i-1}

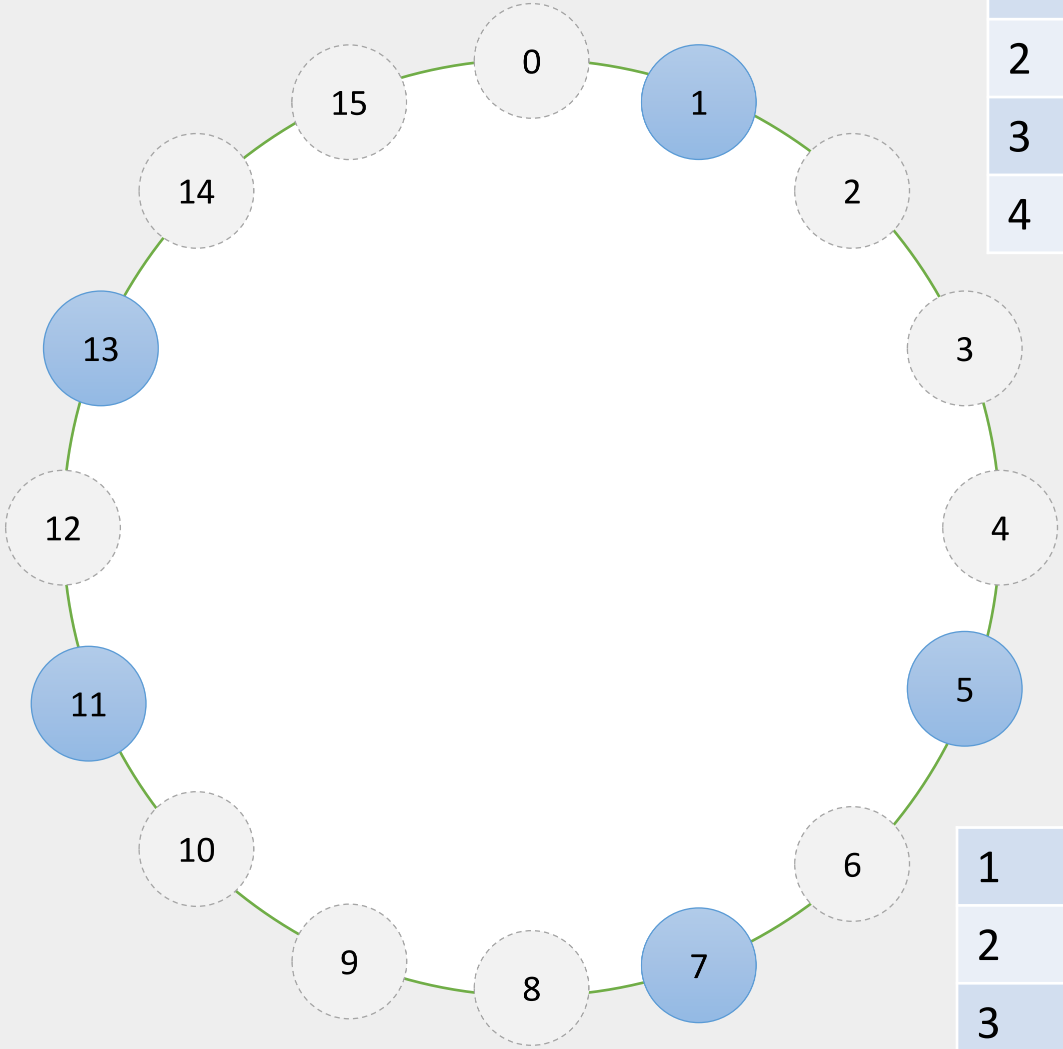
- To look up a key *k*, node *p* will immediately forward the request to a node *q* with index *j* in *p*'s finger table where

$$q = FT_p[j] < k < FT_p\,[j + 1]$$

Node 10 FT (s=16)

1	11 (10 + $2^{1\text{-}1}$)
2	12 (10 + $2^{2\text{-}1}$)
3	14 (10 + $2^{3\text{-}1}$)
4	2 (10 + $2^{4\text{-}1}$) % 16

Finger Tables



1	5
2	5
3	5
4	11

1	1
2	1
3	1
4	5

1	7
2	7
3	11
4	13

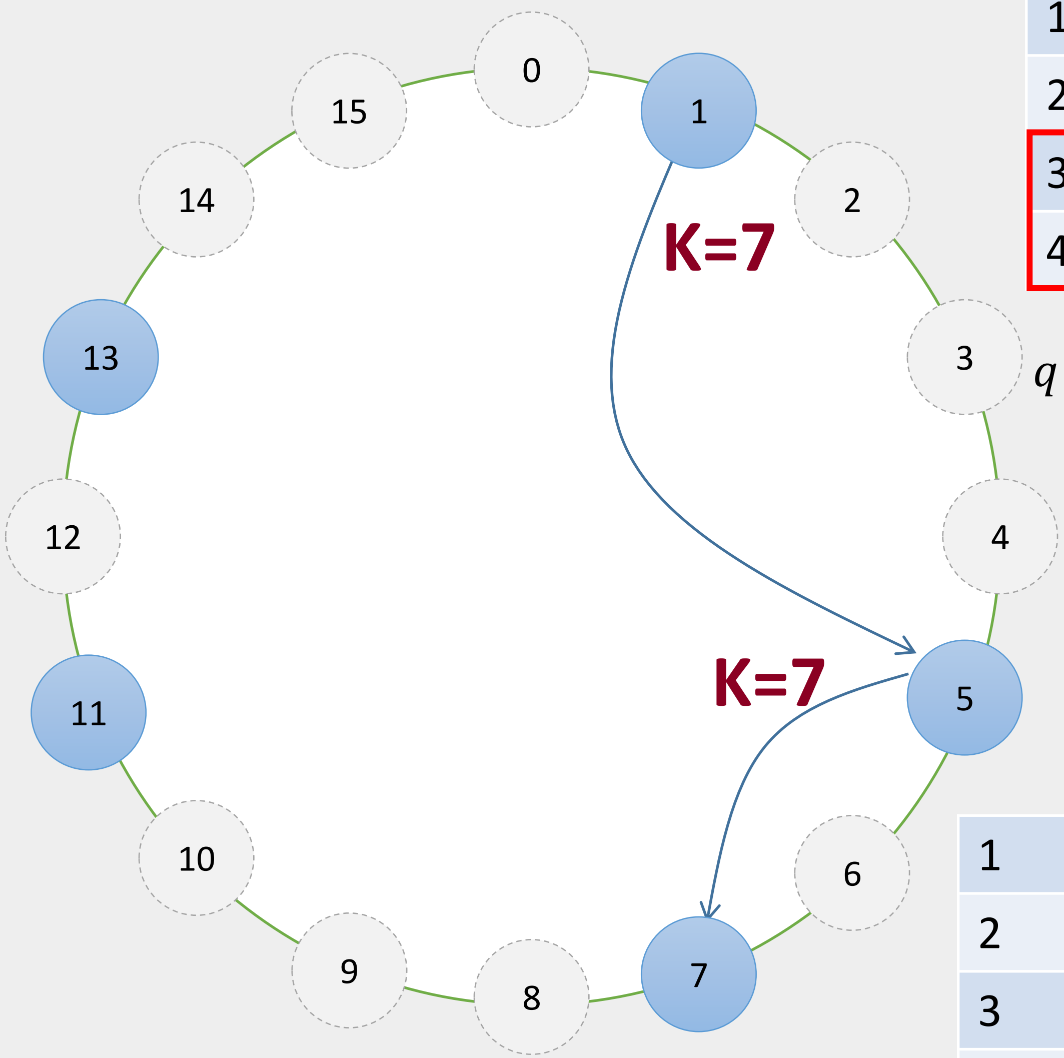
1	13
2	13
3	1
4	5

1	11
2	11
3	11
4	1

Finding a key

1	5
2	5
3	5
4	11

1	1
2	1
3	5
4	5



$$q = FT_p[i] < k < FT_p\,[j+1]$$

1	7
2	7
3	11
4	13

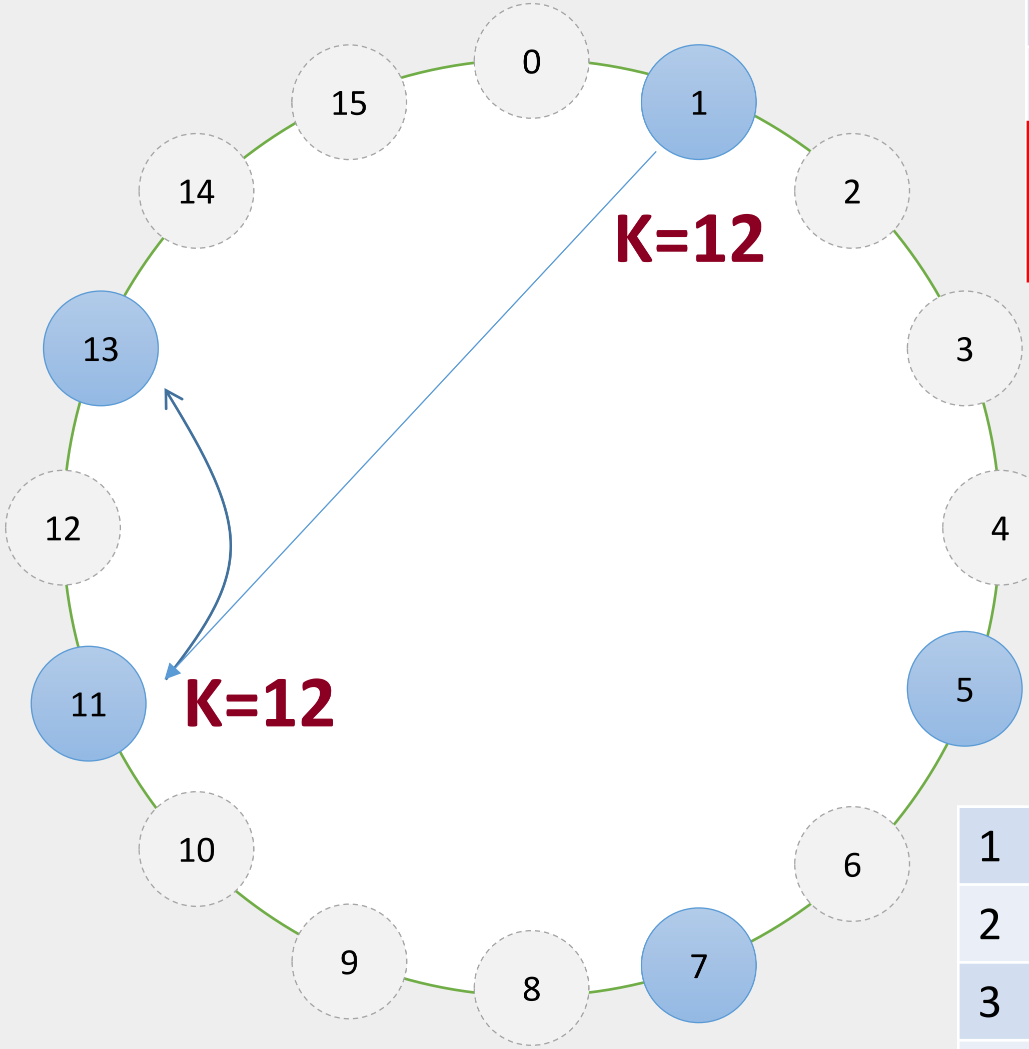
1	13
2	13
3	1
4	5

1	11
2	11
3	11
4	1

Finding a key

1	1
2	1
3	5
4	5

1	13
2	13
3	1
4	5



1	5
2	5
3	5
4	11

$$q = FT_p[i] < k < FT_p\,[j+1]$$

1	7
2	7
3	11
4	13

1	11
2	11
3	11
4	1

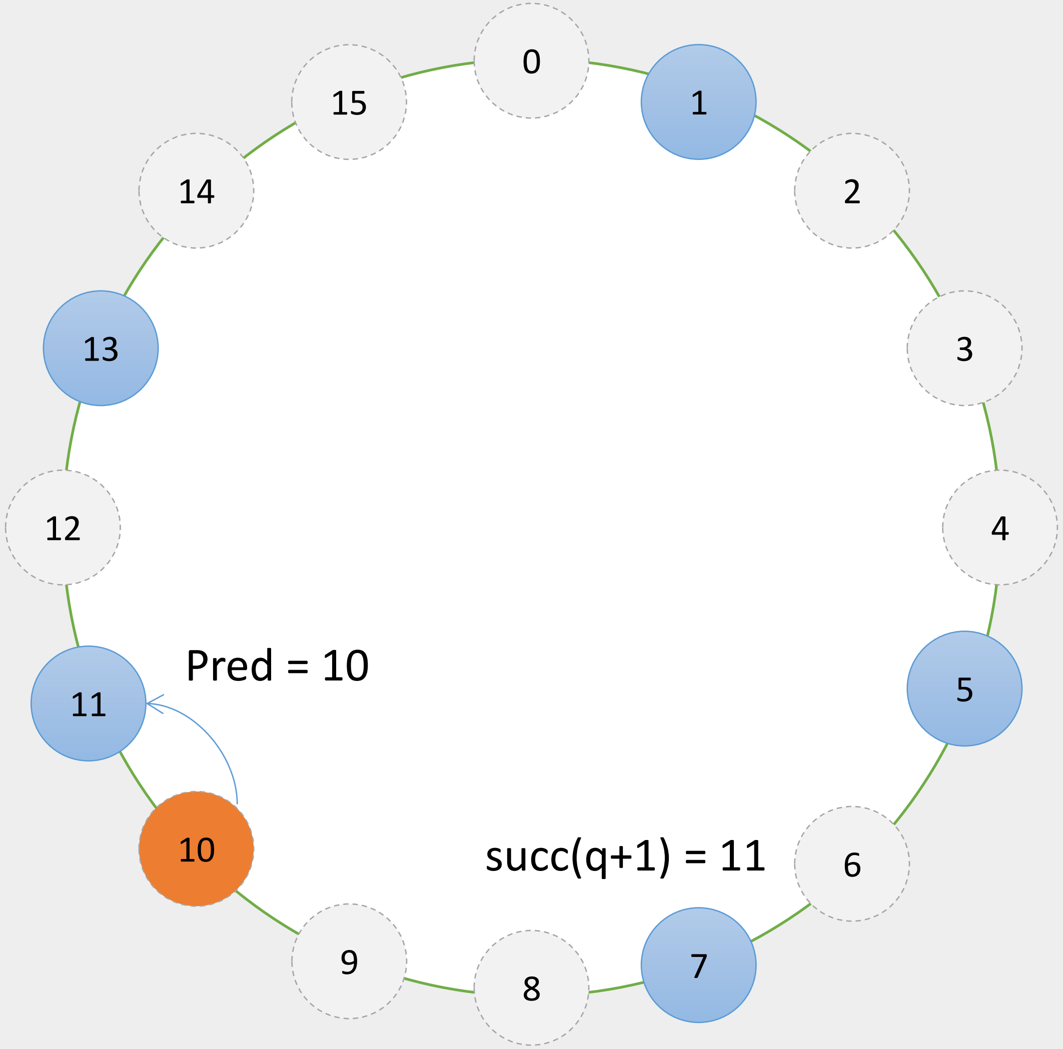
Joining the network

- In large distributed systems the collection of nodes can change frequently
  - Nodes may join or leave the network, or
  - Nodes may fail (the same as leaving) and recover (the same as joining)

- Complexity comes from keeping finger tables up-to-date
  - Importantly every node FT[1] must be correct as it refers to the next node in the ring

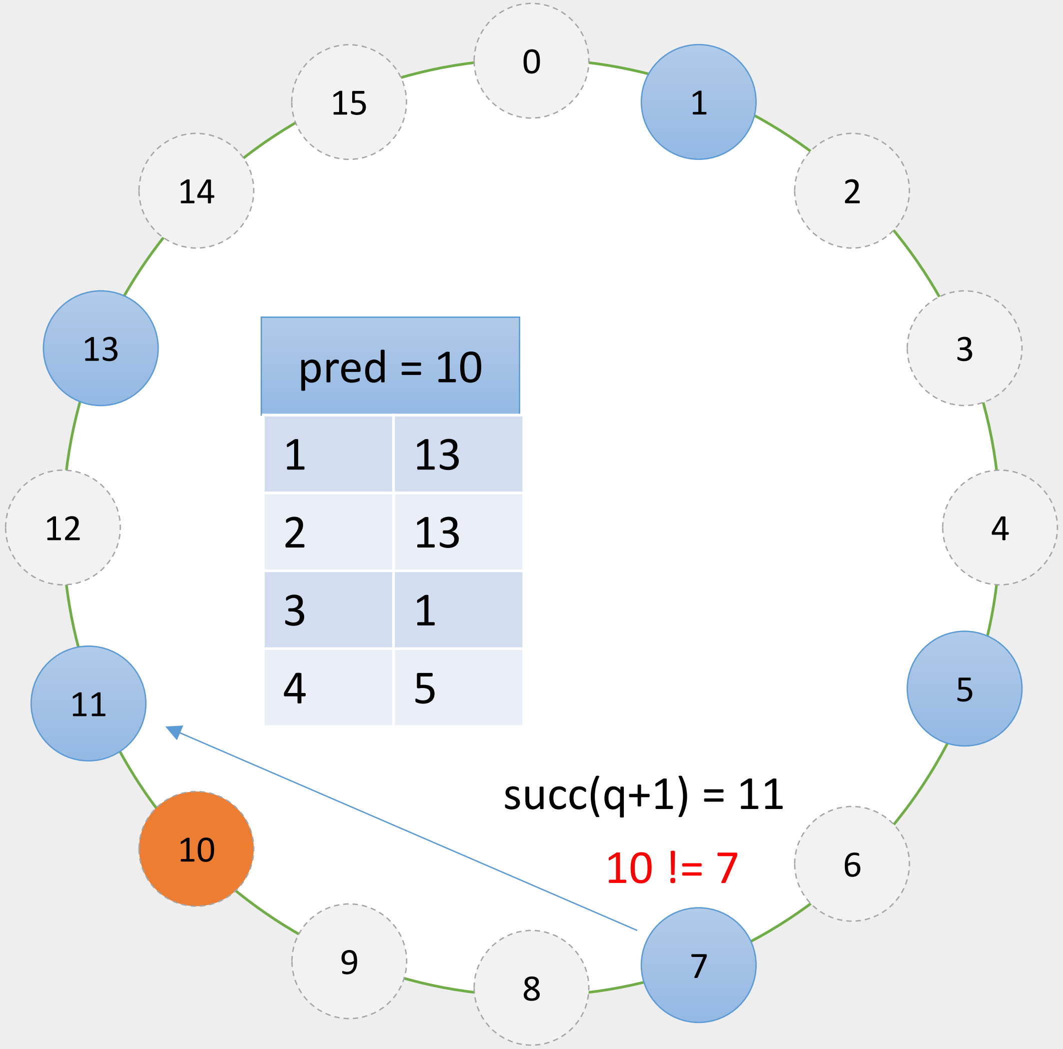
Joining the network

- Joining is easy, if node *q* wants to join the system they:
  - Contact an arbitrary node and request lookup for succ(q+1)
  - Once identified, *q* can insert itself in the network (telling succ(q+1) that it is now its predecessor)



Updating pointers and finger tables

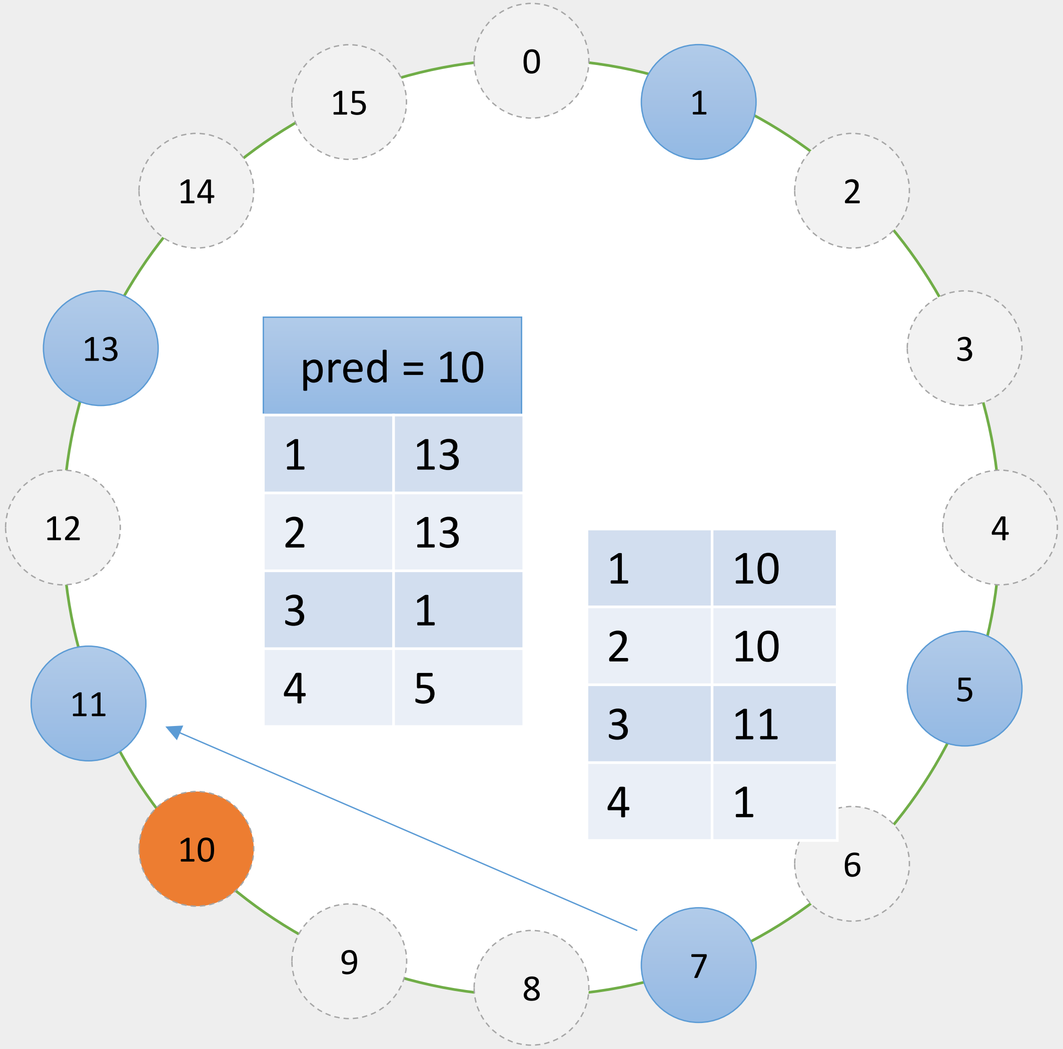
- Each node frequently runs the following procedure
  - succ(p + 1) and requests that node return pred(succ(p+1))
  - If p = pred(seq(p+1)) then we know information is consistent
  - If not, then p’s successor has updated its predecessor and so a new node has entered the system..
    - p can update its finger table to point to the new node



pred = 10	
1	13
2	13
3	1
4	5

Joining and updating finger tables

- Each node frequently runs the following procedure
  - succ(p + 1) and requests that node return pred(succ(p+1))
  - If p = pred(seq(p+1)) then we know information is consistent
  - If not, then p’s successor has updated its predecessor and so a new node has entered the system..
    - p can update its finger table to point to the new node

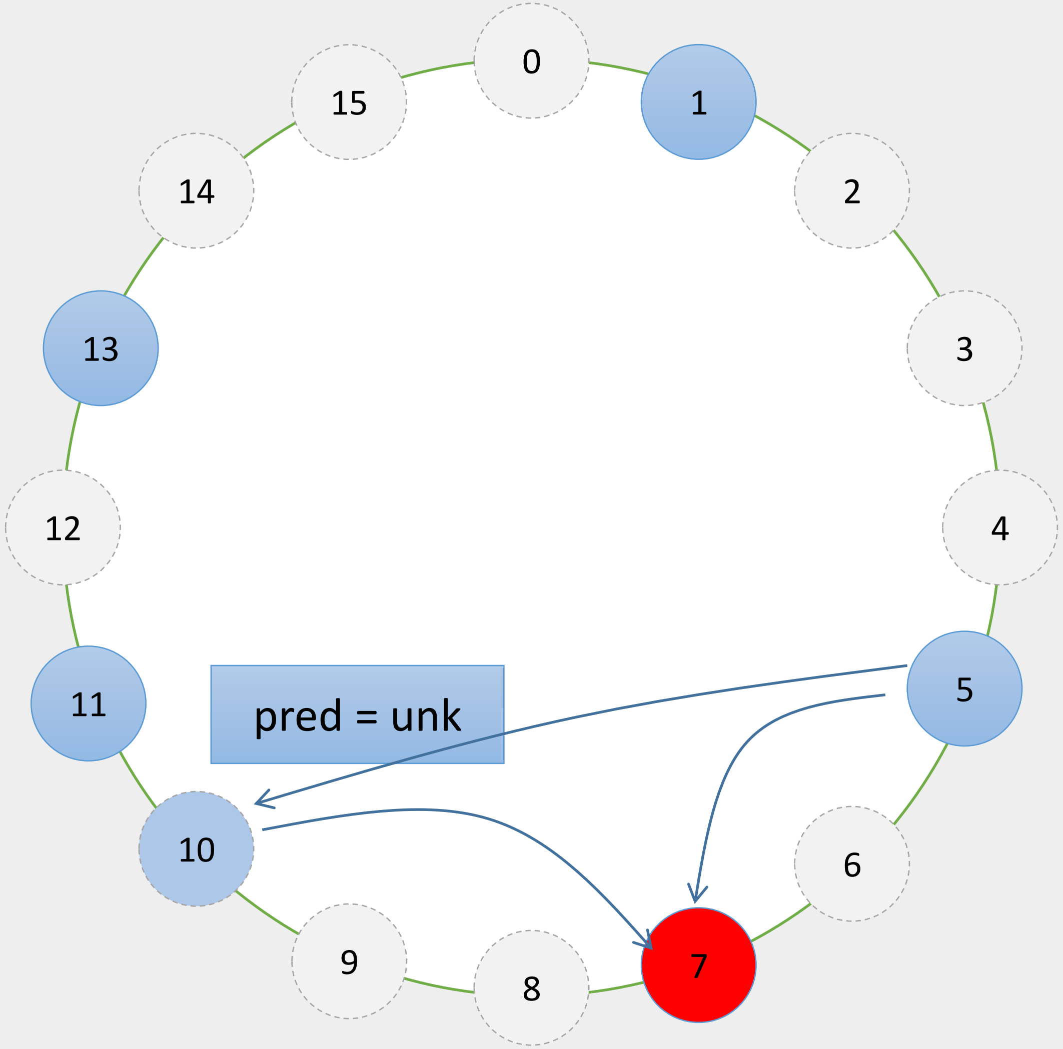


pred = 10	
1	13
2	13
3	1
4	5

1	10
2	10
3	11
4	1

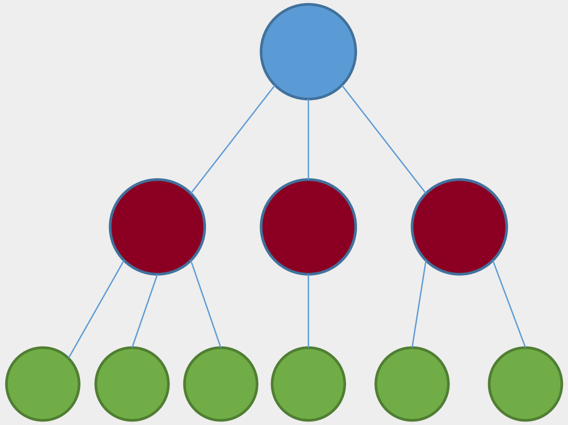
Leaving the network

- Also easy, no need to do anything when leaving ☺
- Each node will regularly check if its predecessor is alive
    - If it doesn't receive anything back then it sets *pred(p)* = 'unknown'
    - When node checks its link to the next node it won't reply
    - Then check *succ(p+1)* where predecessor is unknown then it will notify *succ(p+1)* that it suspects it to be the predecessor



Hierarchical approaches (for Flat names)

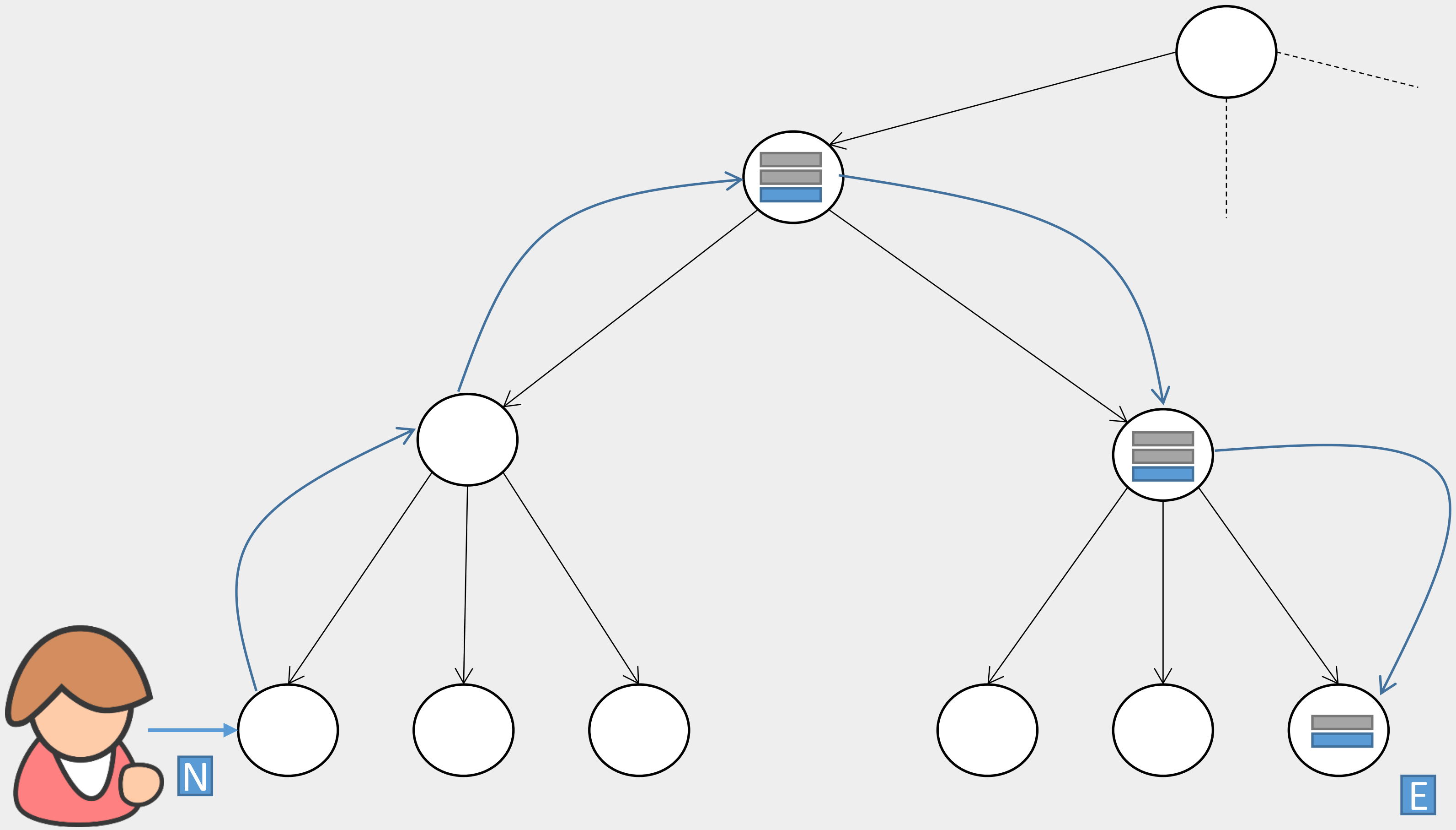
- A good solution to deal with scalability concerns is to structure name resolution into a hierarchy
  - E.g., file systems, domain name system, object stores



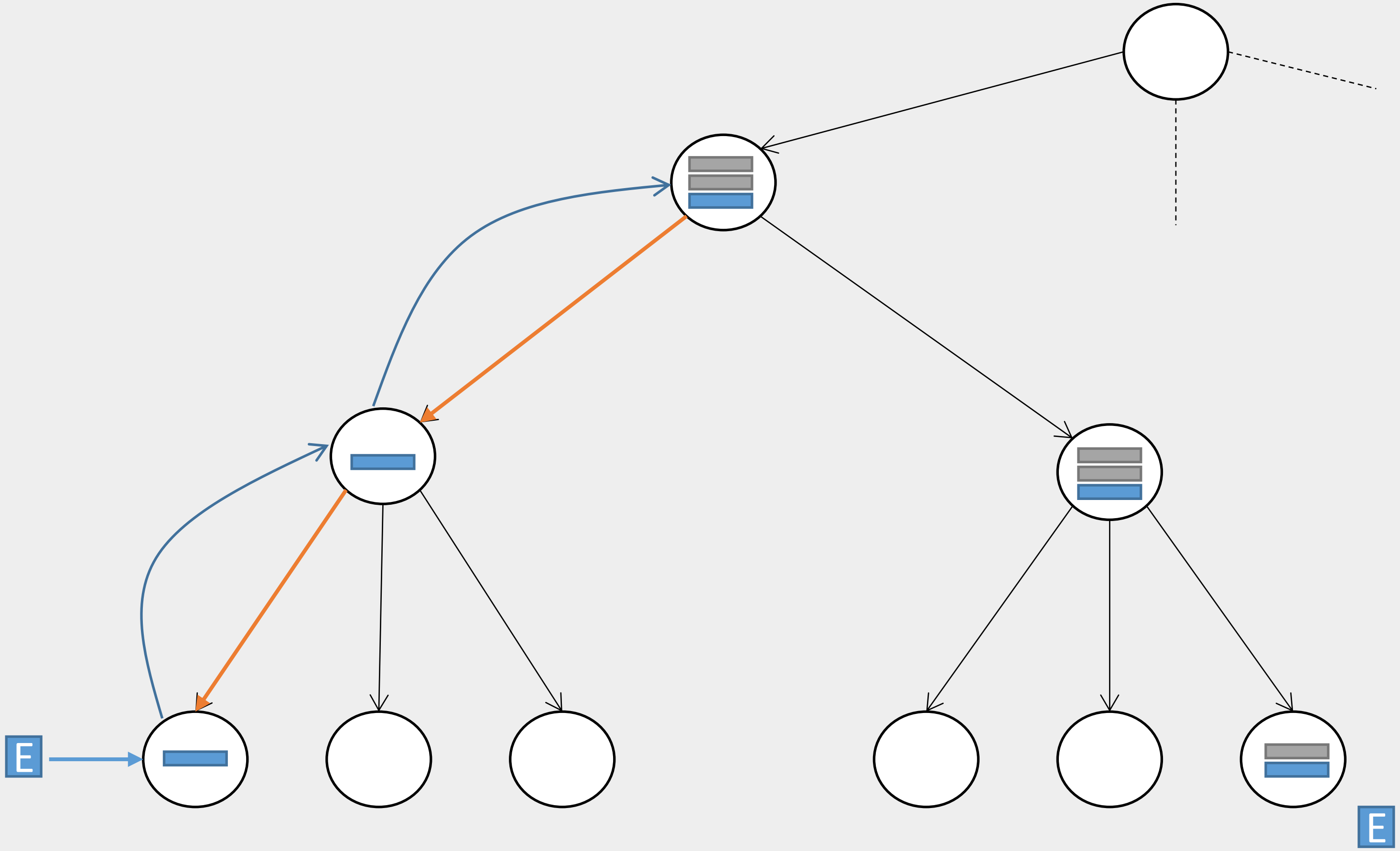
- Network divided into domains
  - Single root domain that spans the network
  - Each domain subdivided into small subdomains
  - Lowest level domain is called a leaf domain

- Each domain has an associated directory node *dir(D)* that tracks entities in the domain

Lookup



Insert



Agenda

- Part 1: Names and addresses
- Part 2: Flat naming
- Part 3: Structured naming
    1. Namespaces
    2. Name resolution
    3. Implementation
- Part 4: Attribute-based naming

Structured naming

Structured names include some information that can help us locate entities

Systems generally support structured names that are composed from simple, human-readable names

- Contain structural separators (e.g., “/”, “.”, “:”)
- Filesystem, DNS names, …

Namespaces

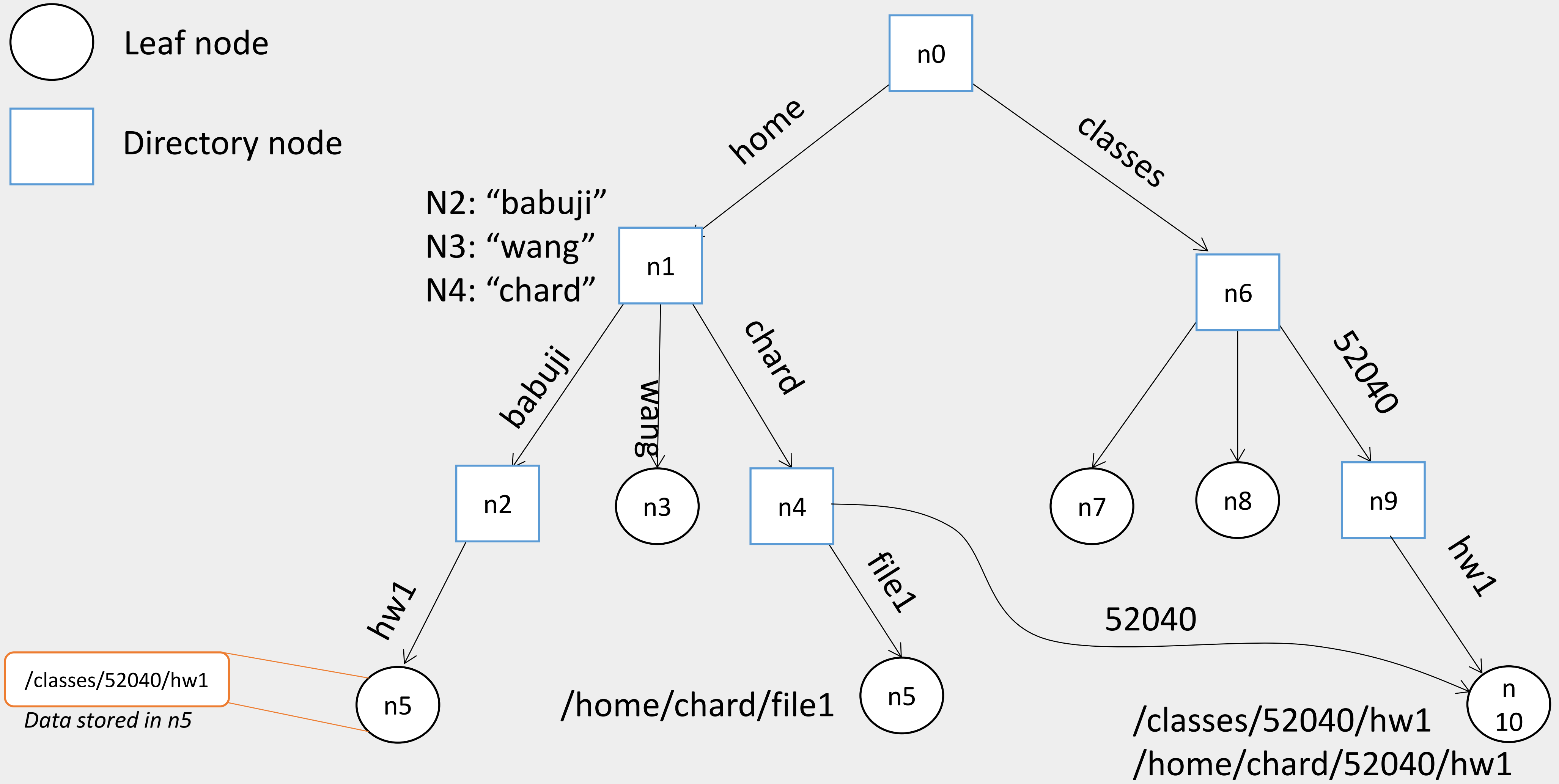
- Names are organized into namespaces

- Namespaces can be represented as a naming graph, a directed graph with two types of nodes:
  - Directory node has outgoing edges (labeled with a name)
    - Stores a directory table with outgoing edges represented as a pair (node identifier, edge label)
  - Leaf nodes represent the entity (has no outgoing edges)

Name resolution

- A path name is used to refer to a node in the naming graph
- A path name is represented by a sequence of edge labels separated by a slash
  - An absolute path name starts from the root node and begins with a slash (e.g., /home/chard/file1)
  - A relative path name does not start at the root node and does not begin with a slash (e.g., chard/file1)
- Given a path name we can lookup information in order:
  - [label1, label2, …, labeln]: start at root node, lookup label 1 in the directory table, returns identifier of node to which label 1 refers
  - Resolution continues by looking up label 2 in directory, and then 3, ..

Naming graph



Aliases

- An **Alias** is another name for the same entity

- Several ways to implement aliases
  - Hard link: allow multiple **absolute path** names to refer to the same node in a graph
  - Soft (symbolic) link: allow **a leaf node** to contain a name of another node
    - First resolve N’s name, read contents of N (yielding M), resolve M

Closure mechanism

- Challenge: resolution only possible if we know how and where to start

- A closure mechanism to select the implicit context from which to start name resolution
  - www.uchicago.edu: start at a DNS name server
  - /home/chard/homework: start at the local file
  - +1 773 702 4107: dial a phone number
  - 77.167.55.6: route message to a specific IP address

Agenda

- Part 1: Names and addresses
- Part 2: Flat naming
- Part 3: Structured naming
    1. Namespaces
    2. Name resolution
    3. Implementation
- Part 4: Attribute-based naming

Namespace implementation

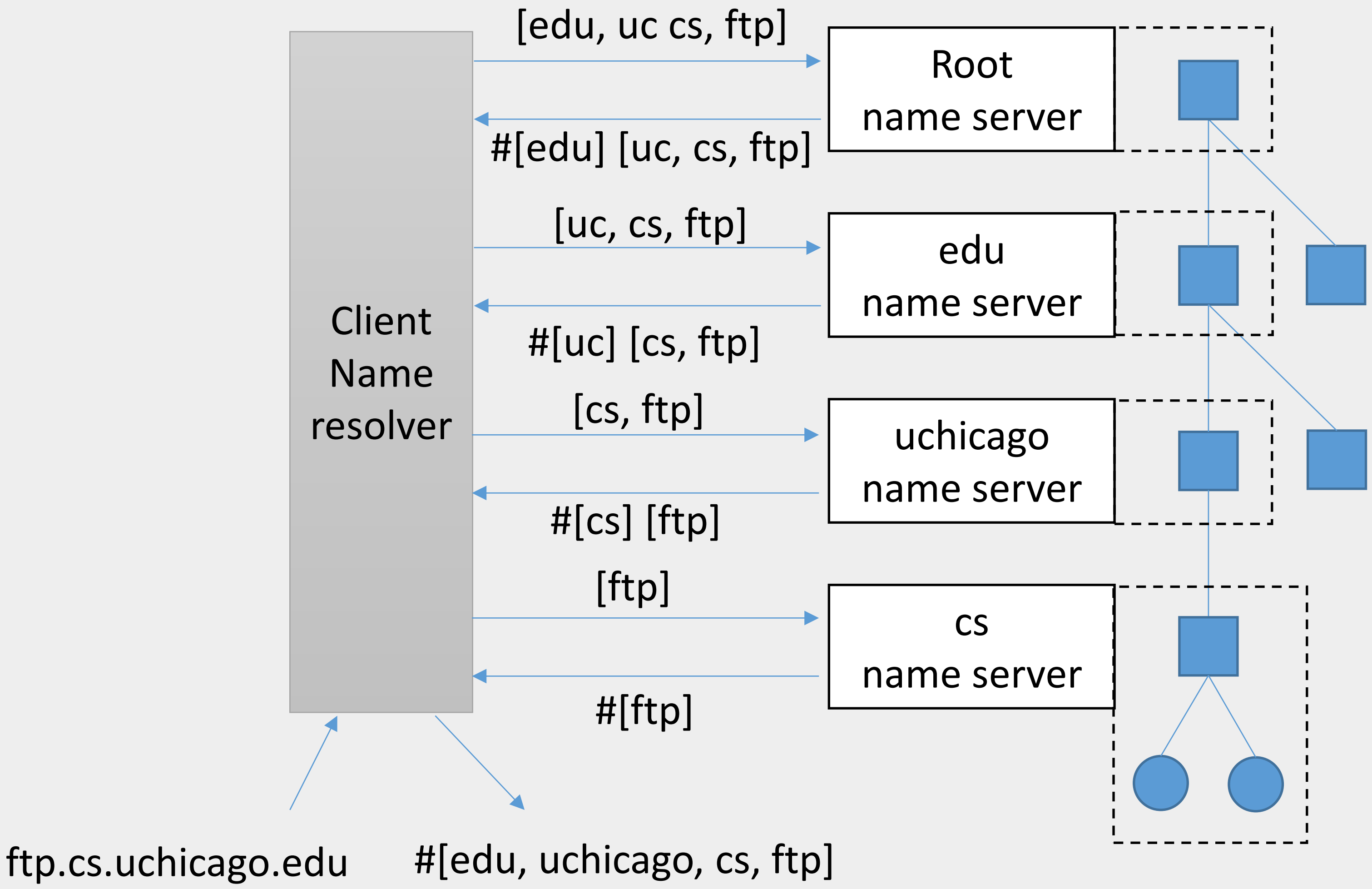
- Implementing the naming model is easy. How do we implement the naming system?

- Aim: distribute the name resolution process (and namespace management) across several machines by distributing nodes of the naming graph

- Question here is how to implement the name resolution process
  - Two basic approaches: iterative and recursive

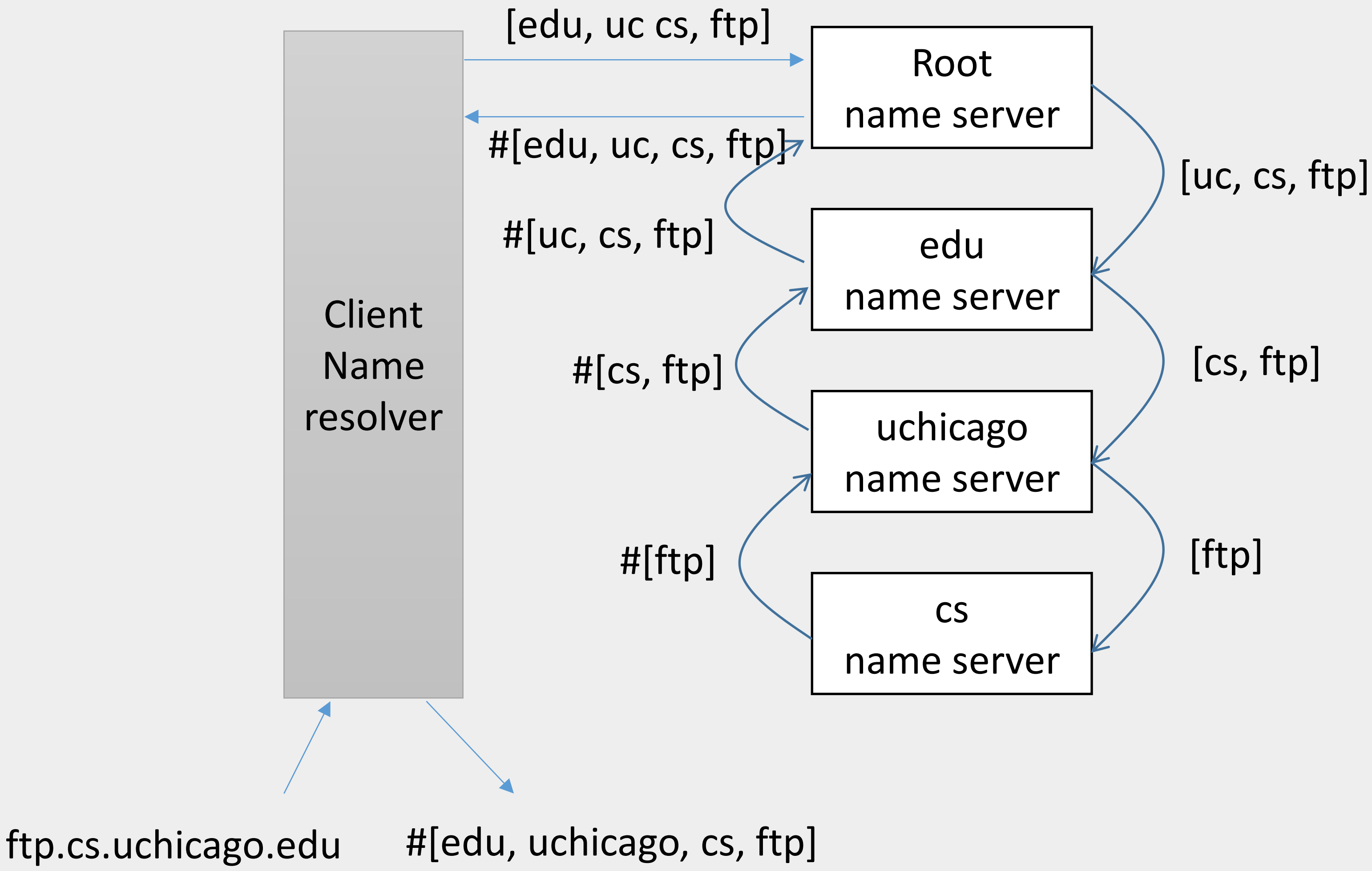
Iterative name resolution

- Resolver hands over complete name to the root name server (using well-known address)
- Root server will resolve path as far as it can and return to the client
- Client will then contact the next name server



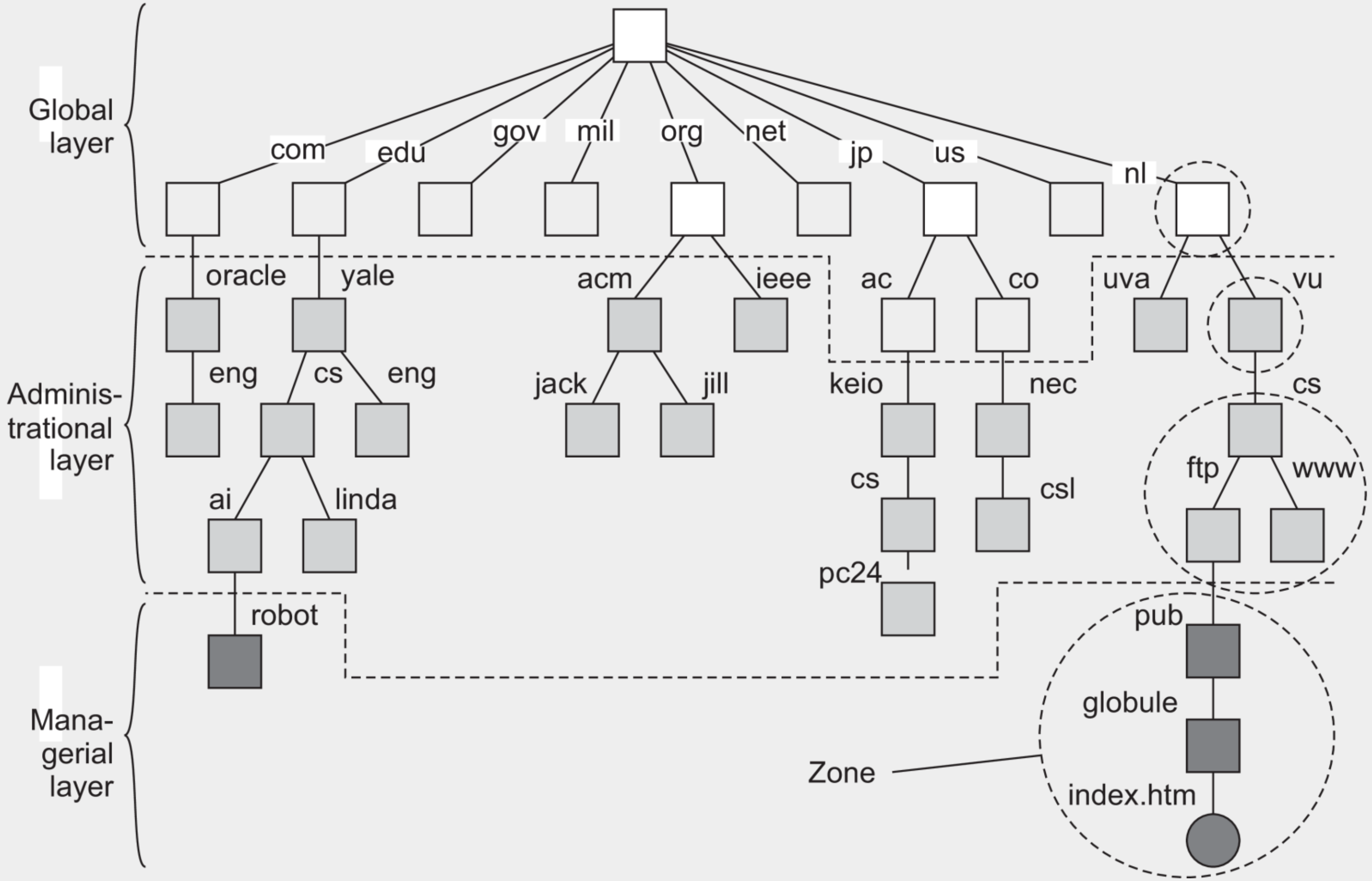
Recursive name resolution

- Client requests resolution from the root name server
- Root name server then passes the request to the next name server
- Name servers will return the result following the same approach
- Advantage: caching is more effective, communication costs reduced
- Disadvantage: additional load on name servers



Exercise Domain Name System

Domain Name System



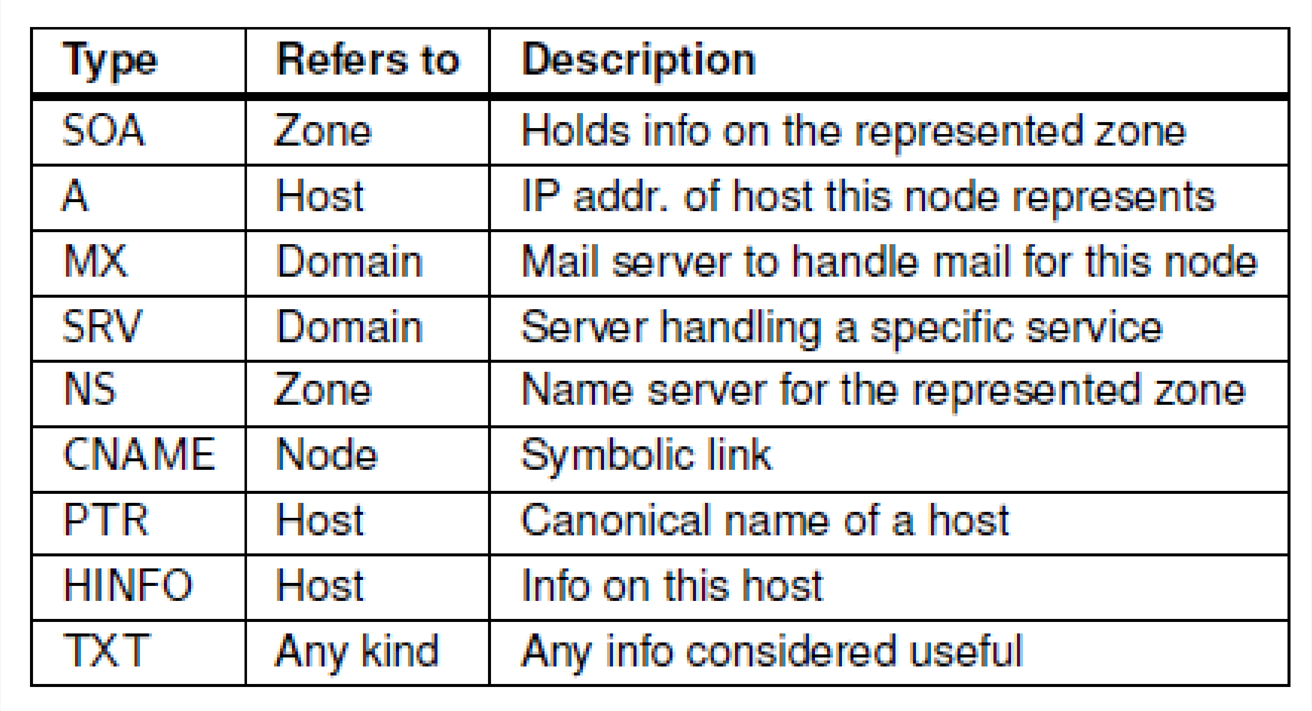
Domain Name System

- DNS
  - Hierarchically organized name space with each node having exactly one incoming edge
  - Labels are case-insensitive string of max length 63 chars
    - Longest path name is 255 characters
  - Labels separated by dots (root represented by a dot – uchicago.edu.)
  - A subtree is called a domain (com, edu, uchicago.edu)
  - A path name to a domain's root node is called a domain name

- Designed 30-40 years ago, but still works well

DNS resource records

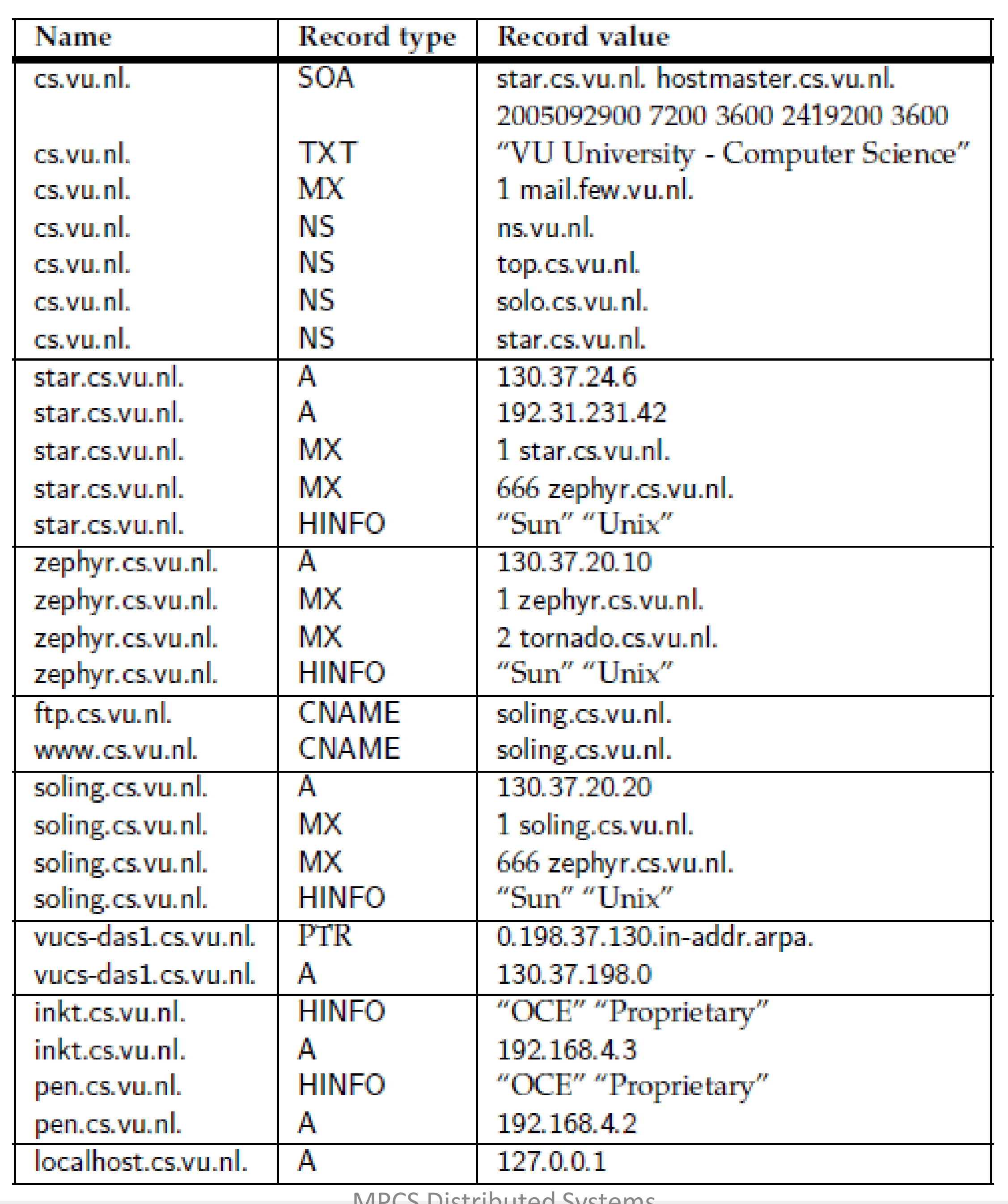
- Contents of a node is formed by a collection of resource records
  - Nodes most often represent several entities at the same time



Type	Refers to	Description
SOA	Zone	Holds info on the represented zone
A	Host	IP addr. of host this node represents
MX	Domain	Mail server to handle mail for this node
SRV	Domain	Server handling a specific service
NS	Zone	Name server for the represented zone
CNAME	Node	Symbolic link
PTR	Host	Canonical name of a host
HINFO	Host	Info on this host
TXT	Any kind	Any info considered useful

DNS Implementation

- Each zone is implemented by a name server
    - Typically replicated for availability
    - Updates for the zone are normally handled by a primary name server
        - Hosts a local database
    - Secondary (replicated) name servers request the primary server to transfer its content (called a zone transfer)
- DNS database implemented as a collection of files, one of which, contains all the resource records for all nodes in that zone



Name	Record type	Record value
cs.vu.nl.	SOA	star.cs.vu.nl. hostmaster.cs.vu.nl. 2005092900 7200 3600 2419200 3600
cs.vu.nl.	TXT	"VU University - Computer Science"
cs.vu.nl.	MX	1 mail.few.vu.nl.
cs.vu.nl.	NS	ns.vu.nl.
cs.vu.nl.	NS	top.cs.vu.nl.
cs.vu.nl.	NS	solo.cs.vu.nl.
cs.vu.nl.	NS	star.cs.vu.nl.
star.cs.vu.nl.	A	130.37.24.6
star.cs.vu.nl.	A	192.31.231.42
star.cs.vu.nl.	MX	1 star.cs.vu.nl.
star.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.
star.cs.vu.nl.	HINFO	"Sun" "Unix"
zephyr.cs.vu.nl.	A	130.37.20.10
zephyr.cs.vu.nl.	MX	1 zephyr.cs.vu.nl.
zephyr.cs.vu.nl.	MX	2 tornado.cs.vu.nl.
zephyr.cs.vu.nl.	HINFO	"Sun" "Unix"
ftp.cs.vu.nl.	CNAME	soling.cs.vu.nl.
www.cs.vu.nl.	CNAME	soling.cs.vu.nl.
soling.cs.vu.nl.	A	130.37.20.20
soling.cs.vu.nl.	MX	1 soling.cs.vu.nl.
soling.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.
soling.cs.vu.nl.	HINFO	"Sun" "Unix"
vucs-das1.cs.vu.nl.	PTR	0.198.37.130.in-addr.arpa.
vucs-das1.cs.vu.nl.	A	130.37.198.0
inkt.cs.vu.nl.	HINFO	"OCE" "Proprietary"
inkt.cs.vu.nl.	A	192.168.4.3
pen.cs.vu.nl.	HINFO	"OCE" "Proprietary"
pen.cs.vu.nl.	A	192.168.4.2
localhost.cs.vu.nl.	A	127.0.0.1

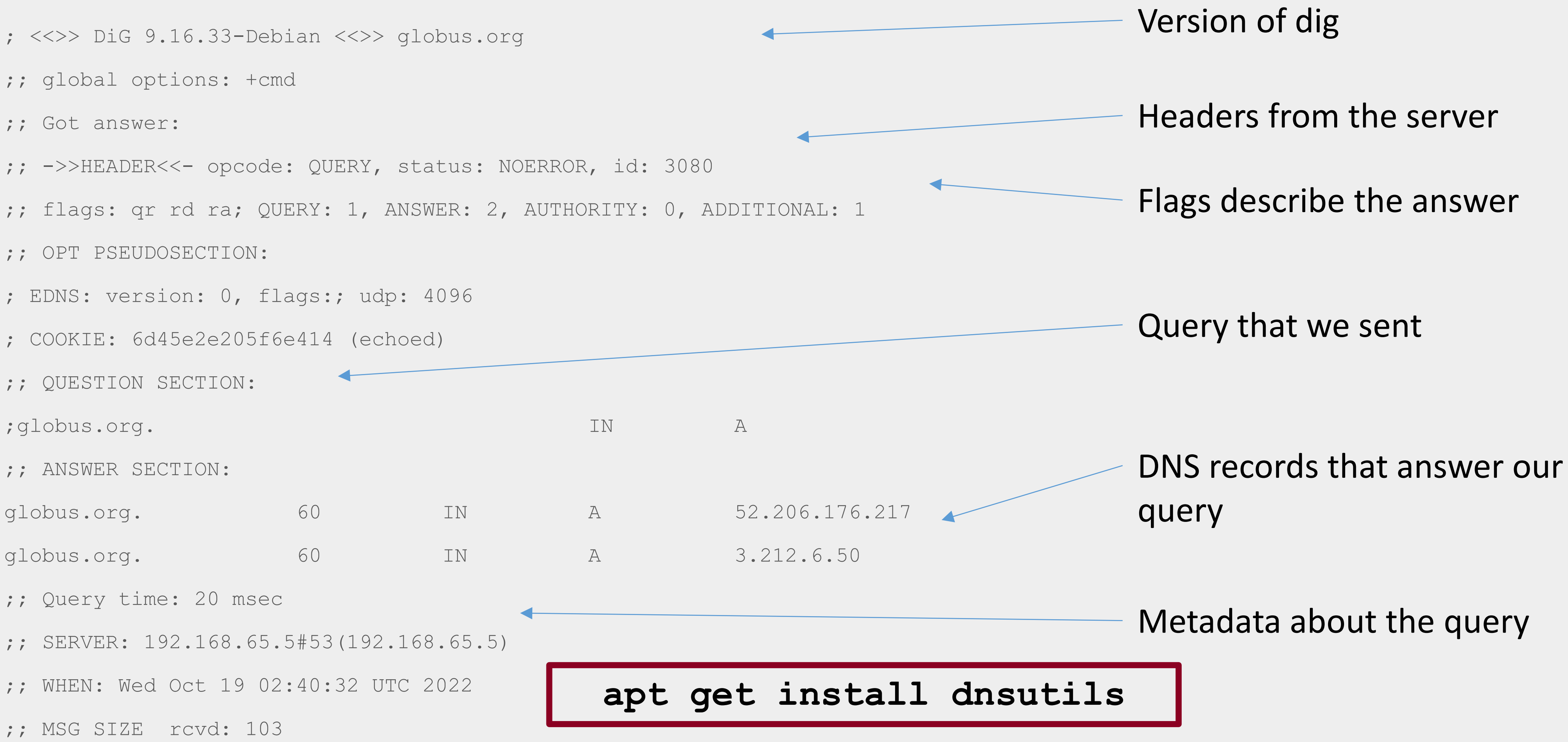
Dockerfile for exercises

```
FROM python:latest
RUN apt-get update && apt-get install -y \
   python3-dev \
   libldap2-dev \
   libssl-dev \
   ldap-utils \
   libsasl2-dev \
   dnsutils
RUN pip3 install python-ldap
RUN pip3 install dnspython
RUN pip3 install jupyterlab

CMD ["jupyter", "lab", "--ip=*", "--allow-root"]
```

- docker build -t naming-class .
- docker run -it -p 8888:8888 naming-class

Looking up DNS records: dig



```
; <<>> DiG 9.16.33-Debian <<>> globus.org
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 3080
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 6d45e2e205f6e414 (echoed)
;; QUESTION SECTION:
;globus.org.                             IN         A
;; ANSWER SECTION:
globus.org.             60         IN         A          52.206.176.217
globus.org.             60         IN         A          3.212.6.50
;; Query time: 20 msec
;; SERVER: 192.168.65.5#53(192.168.65.5)
;; WHEN: Wed Oct 19 02:40:32 UTC 2022
;; MSG SIZE  rcvd: 103
```

Version of dig

Headers from the server

Flags describe the answer

Query that we sent

DNS records that answer our query

Metadata about the query

apt get install dnsutils

DNS Exercise

1. Look up UChicago
2. Find all UChicago records (e.g., TXT, NS, …)
   - (look up dig documentation and find out how to query for all records)
3. Find UChicago CS mail server
4. Find where cname for labs.globus.org points

```
dig <domain> <type>
```

DNS Exercise

- Exercise: look up DNS information in Python (http://www.dnspython.org/)

```
Import dns.resolver as resolver
Res= resolver.resolve(‘name’, ‘record type’)
For r in res:
    Print (r)
```

- 1) Look up UChicago A record IP
- 2) Find UChicago CS mail server
  - (look up documentation and find out how to query for a particular record type)
- 3) Find where cname for labs.globus.org points

Agenda

- Part 1: Names and addresses
- Part 2: Flat naming
- Part 3: Structured naming
- Part 4: Attribute-based naming
    1. Directory services
    2. Hierarchical implementations
    3. Decentralized implementations

Attribute-based naming

- Often discovery of entities requires more than just a name
  - Important to *search* for entities using information about them
- Most common way of providing descriptions is via key-value (or attribute-value) pairs
- Attribute-based naming models assume that each entity is made up of a collection of attributes

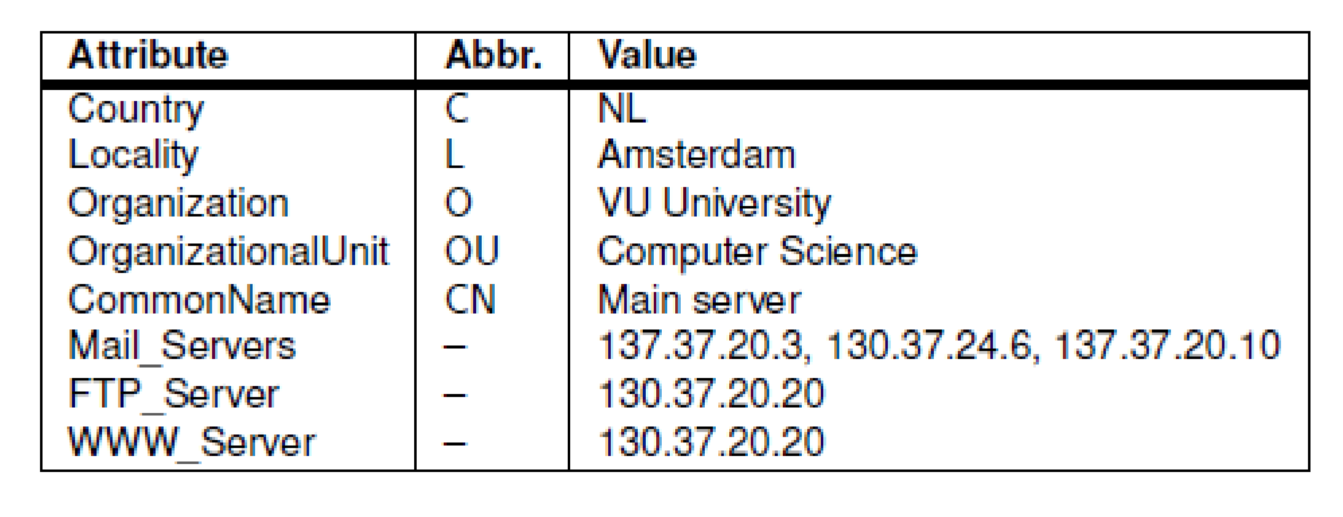
Lightweight Directory Access Protocol (LDAP)

- Common approach for distributed directory services is to combine structured naming with attribute-based naming
  - Most systems use, or build on the Lightweight Directory Access Protocol

- LDAP derived from the OSI X.500 directory service
  - Not widely deployed as discussed earlier

Conceptual model

- Contains records (or directory entries)
  - Like a resource record in DNS
- Records are made up of a collection of (attribute, value) pairs
  - Attributes have associated type
  - Single-valued vs multi-valued attributes



Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	VU University
OrganizationalUnit	OU	Computer Science
CommonName	CN	Main server
Mail_Servers	–	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	–	130.37.20.20
WWW_Server	–	130.37.20.20

Example LDAP structure following LDAP naming conventions

LDAP naming

- Each record is uniquely named so that it can be looked up
- Expressed as a sequence of naming attributes (Distinguished Name, DN)
  - Each naming attribute is called a relative distinguished name

- E.g., Country, Organization, and Organizational Unit can form a unique name
  - DN => /C=US/O=UChicago/OU=Computer Science

- Like DNS you can form a naming tree by stepping through the hierarchy

- Unlike DNS you can query:
  - search(‘‘(C=US)(O=UChicago)(OU=*)(CN=Main server)’’)

Example querying UChicago LDAP

- Install system packages (e.g., apt-get install)
    - python3-dev
    - libldap2-dev
    - libssl-dev
    - libsasl2-dev
    - ldap-utils

- Install python-ldap from PyPi

```
FROM python:latest
RUN apt-get update && apt-get install -y \
  python3-dev \
  libldap2-dev \
  libssl-dev \
    ldap-utils \
    libsasl2-dev

RUN pip3 install python-ldap
RUN pip3 install dnspython
RUN pip3 install jupyterlab

CMD ["jupyter-lab", "--allow-root", "--ip",
"0.0.0.0"]

docker build –t naming-class .
docker run  -p 8888:8888 naming-class
```

General approach

Connect

```
conn = ldap.initialize('ldap://ldap.uchicago.edu:389’)
```

(Normally) bind to our server (* not needed when anonymous)

```
conn.simple_bind_s('ldap_login', 'ldap_password’)
```

Query

```
conn.search_s(‘base dn’, Scope)
```

Simple search

-
  ```
  result = conn.search_s(
      'dc=somedomain,dc=com’,
      ldap.SCOPE_SUBTREE,
      ‘(query=query)’,
      [‘Attr1’, ‘Attr2’])
  ```

- DN – base domain to search
- ldap.SCOPE_SUBTREE — to search the object and all its descendants
- (query=query) — query to search
- [Attr1, attr2] — list of attributes to retrieve

LDAP Exercise

1. Find yourself
2. Get only your name, department, and email (note you will need to work out the right attribute
3. Find a list of (max 10) people in the department
    1. Use the DN: ou=People,dc=uchicago,dc=edu
    2. Use the asynchronous search interface
        1. id = conn.search_ext(.. sizelimit=10)
        2. Loop over conn.result(id,0)

Submitting Exercise

- naming.py (or notebook)
    - https://classroom.github.com/a/s2gbb1tM

Midterm Exam

- **During class (2/7) 10:30-12:00 (class will continue 12-1:30)**

- Covers the first 4 classes with 4 roughly equally weighted questions
  - General distributed systems/Distributed architectures
  - Processes and threads
  - Networking and communication
  - Naming
- Short answer, no real coding (perhaps some pseudocode)