University of Bologna
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Engineering Bologna Campus

Class of Computer Networks M or Infrastructures for Cloud Computing and Big Data

ONs and Advanced Filesystems

Antonio Corradi

Academic year 2016/2017

OVERLAY NETWORKS

There are many situations where you want to organize a **logical connection between different entities that reside in different locations and networks**

The solution is an **Overlay Network (ON) at the application level that connects all those entities to be considered together in an ON**

Overlay networks may *be very different and also enforced in different ways*, but their importance is paramount in many situations because **they answer to efficiency and scalability**

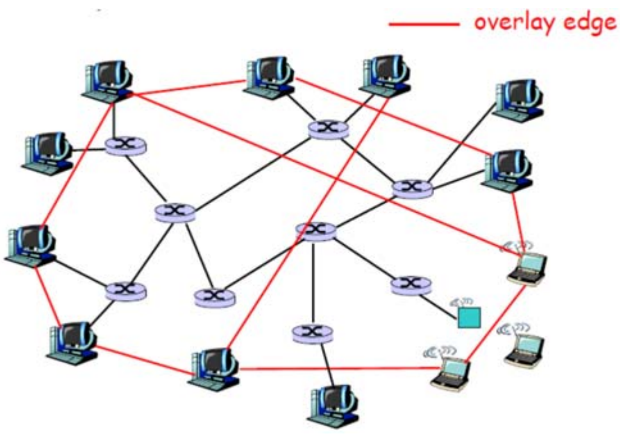
One main point very important is not only **organizing** it, but also to **grant QoS** and to **respect an agreed SLA**

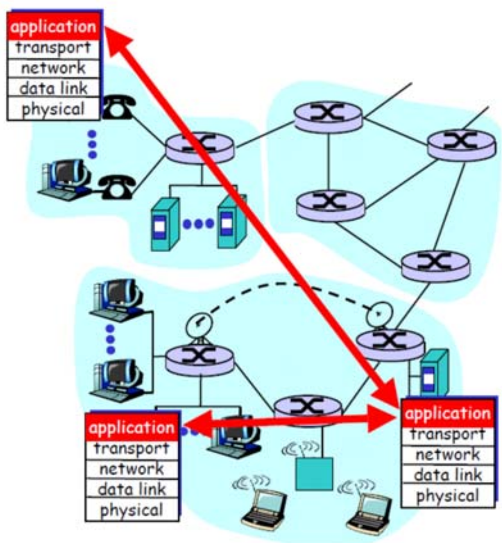
That is the reason why there are many different solutions for different cases, and also many different solutions and tools embodying these requirements

OVERLAY NETWORKS

The main point is to create a **new network at the application level** and to maintain it with **specified requirements**

All participants become part of it and **can freely communicate** (the same as if they were in a real network connection), by using an **application neighborhood**





CLASSIFICATION of OVERLAY NETWORKS

There are two main different kinds:

- **Unstructured overlays**
- **Structured overlays**

By focusing on new nodes arriving and entering the ON,

in ***Unstructured overlays***, new **nodes choose randomly the neighbor to use to access** to the ON

in ***Structured overlays***, there is **a precise strategy to let nodes in** and to *organize the architectures, maintained also to react to discontinuities and failures*

ONs propose solutions for P2P applications, but also for MOMs (even if **statically-oriented)**

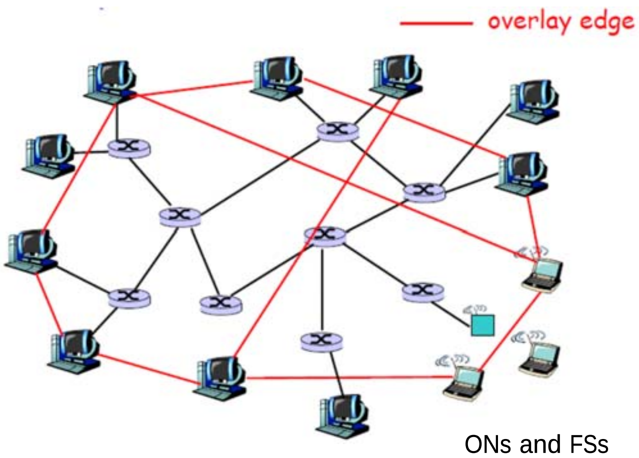
P2P	Napster, Gnutella, Kazaa, BitTorrent
Support	Chord, Pastry/Tapestry, CAN
Social Nets	MSN, Skype, Social Networking Support

OVERLAY NETWORKS: USAGE

A good **overlay network** has the goal making **efficient the operations among the group of current participants** obeying the **specific requirements**

All participants in an overlay have **a common goal of exchanging information**, for instance…

They tend to **exchange data**: files in a P2P application, messages in social nets, specific application protocols in other environments, etc.



SYSTEM AND APPLICATION KEY ISSUES

ONs should organize the communication support and also enable the application level management

Lookup

- To find out very fast the appropriate user information (content/resource) on the ON

Guaranteed Throughput

- To communicate over an ON need support for content distribution/dissemination
- To replicate content … fast, efficiently, reliably

Management

- To maintain efficiently the ON under a high rate of connections/disconnections and intermittent failures in load balanced approach
- To guarantee both application reliability and availability (maybe very difficult): a self-organizing approach is typically followed

ON MANAGEMENT PROPERTIES

Overlay networks imply many challenges to cope with

- **Maintaining** the edge links (via pointers to IP addresses?)
- **Favoring** the insertion in the neighborhood
- **Checking** link liveness
- **Identifying** problems and faults
- **Recovering** edges
- **Overcoming** nodes going down and their unavailability
- **Re-organizing the overlay**, when some nodes leave the network and other nodes get in
- **Keeping the structure**, despite mobile nodes intermittent presence (and eventual crashes or leaving)
- **Creating a robust connection**, independently of omissions and crashes (QoS?)

NAPSTER: A PIONEER P2P

A non-structured approach for file retrieving

Centralized Lookup

Centralized directory services deal with nodes entering

Any node connects to a Napster server

Any node uploads list of files to server

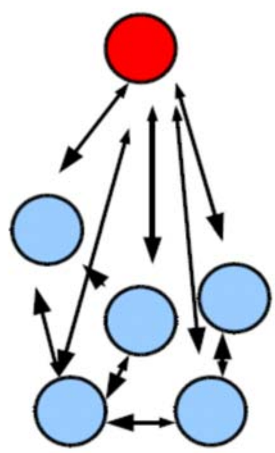
Any node gives servers keywords to search the full list with

File exchange peer to peer

Lookup is centralized from servers,
but files copied P2P

Select "best" of correct answers
(announce by ping messages)

Performance Bottleneck and Low scalability



GNUTELLA

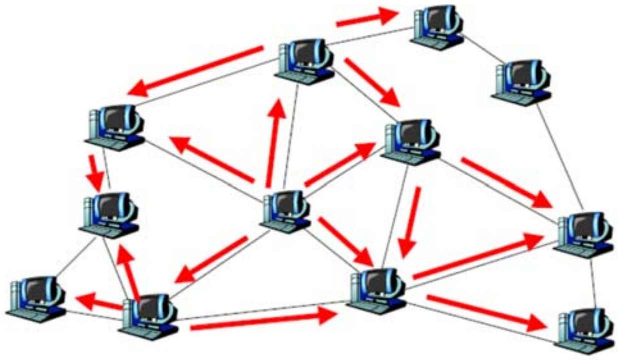
GNUTELLA is the main representative of **unstructured ONs,** by providing a **distributed approach** in **file retrieval**

Fully decentralized organization and **lookup** for files

There are nodes with different degrees of connections and availability (from high-degree nodes to low-degree ones)

High-degree nodes may receive and control even more links

Flooding based lookup, obviously **inefficient** in terms of scalability and bandwidth



GNUTELLA: NEIGHBOOR SCENARIO

Step 0: Join the network

Step 1: Determining who is on the network

- **"Ping"** packet is used to announce your presence on the network.
- Other peers respond with a **"Pong"** packet and Ping connected peers
- A Pong packet also contains:
  - *IP address*, *port number*, *amount of data* that peer share
  - Pong packets come back via same route of Ping

Step 2: Searching

- Gnutella "**Query**" ask other peers (**N** usually **7**) for desired files
- A Query packet might ask, ***"Do you have any matching content with the string 'Volare'"?***
- Peers check to see if they have matches & respond (if they have any match) & send packet to connected peers if not (**N** usually **7**)
- It continues for **TTL** (**T** specifies the hops a packet can traverse before dying, typically **10**)

Step 3: Downloading

- Peers respond with a "**QueryHit**" (it contains contact info)
- File transfers via direct connection using **HTTP** protocol's **GET** method

GNUTELLA REACHABILITY

An analytical estimation of reachable users (T and N)

T: TTL, *N*: Neighbors for Query

	T=1	*T=2*	*T=3*	*T=4*	*T=5*	*T=6*	*T=7*
N=2	2	4	6	8	10	12	14
N=3	3	9	21	45	93	189	381
N=4	4	16	52	160	484	1,456	4,372
N=5	5	25	105	425	1,705	6,825	**27,305**
N=6	6	36	186	936	4,686	**23,436**	**117,186**
N=7	7	49	301	1,813	**10,885**	**65,317**	**391,909**
N=8	8	64	456	3,200	**22,408**	**156,864**	**1,098,056**

GNUTELLA SEARCH

GNUTELLA different versions have adopted different scalability protocols

Flooding based search is extremely wasteful with bandwidth

- Enormous number of **redundant messages** (not efficient)
- A large (linear) part of the network is covered irrespective of hits found, **without taking into account needs**
- All users do **searches in parallel**: local load grows linearly with size

Taking advantage of the unstructured network, some more efficient protocols started appearing

- **Controlling topology** for better search
  Random walk, **Degree-biased Random Walk**
- **Controlling placement of objects**
  Replication

GNUTELLA RANDOM WALK

A Scale-Free graph is a graph whose degree of distribution follows a **power law** or an **exponential law: a few highly connected nodes** and **many low connected ones**

Basic strategy based on **high degree** nodes

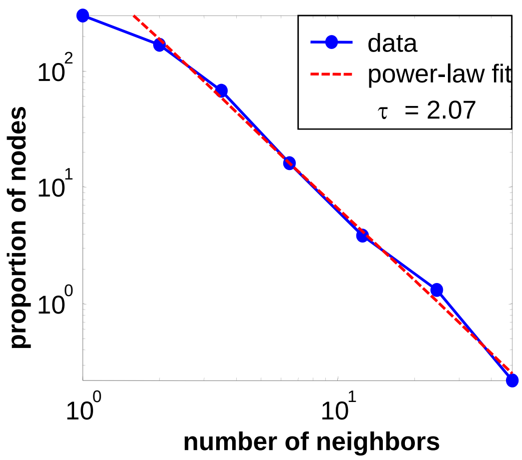
- **High degree nodes** can store the index about a large portion of the network and are easier to find by (**biased**) **random walk** in a **scale-free graph** in a scenario of random offer of files
- **High degree nodes** have a **neighborhood** of **low-degree ones**

Random walk

Moves random to avoid to visit always already last visited node

Degree-biased random walk

- Select **highest degree nodes** that have not been visited
- Walk **first climbs** to highest degree nodes, then **climbs down** on the degree sequence
- Optimal coverage can be formally proved



GNUTELLA REPLICATION

The main idea is to **spread copies of objects to peers** so that **more popular objects can be found easier** and **also launch more walks in parallel to more likely find them**

Replication is both in sense of **more copies of data** and also in terms of **more walkers to launch in parallel**

Replication strategies

Replicate with i when $\boldsymbol{q_i}$ *is the* ***number of queries*** *for object i*

Owner replication

- Produce replicas in proportion to q_i

Path replication

- Produce replicas over the path with replication as square root to q_i

Random replication

- Same as path replication to q_i, only using the given number of random nodes, not the path

…but it is still difficult to find rare objects

UNSTRUCTURED VS STRUCTURED

To go deep into ON organization…

- **Unstructured P2P networks** allow resources to be placed at any node spontaneously
  The network topology is arbitrary and the growth is free but some worst cases and bottlenecks
- **Structured P2P networks** simplify **resource location and load balancing** by defining a **topology** and **rules for resource placement** to obtain **efficient search for rare objects**

Which strategies and rules???

↓

Distributed Hash Table (DHT)

HASH TABLES

Distributed Hash Tables use **Hash principles** toward a **better retrieval** of data **content and value**

Store **arbitrary keys and connected data** (value)

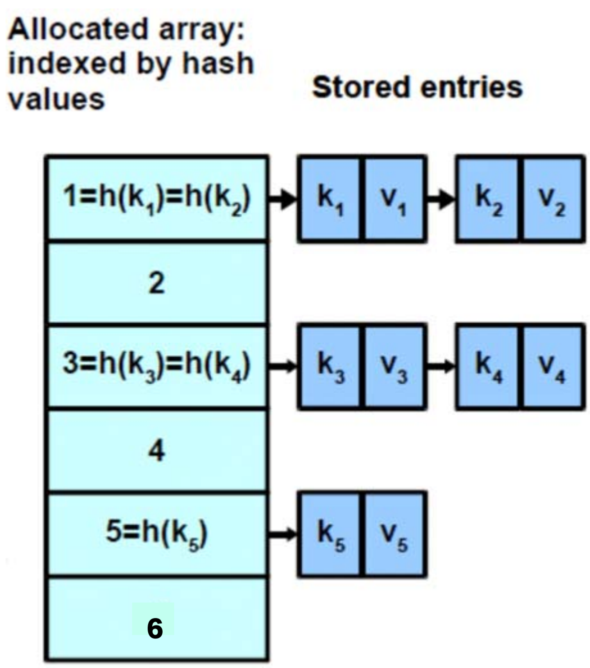
- **put** (key, value)
- value = **get**(key)

Lookup must be fast

- Calculate hash function h() on key that returns a storage cell

Chained hash table

- Store keys in the overflow chain (together with optional value)



DISTRIBUTED HASH TABLES

Hash table functions in an ON is typically P2P: **lookup of data indexed by keys can be very efficient and fast** (**find the nodes where the data are kept**)

Key-hash → node mapping

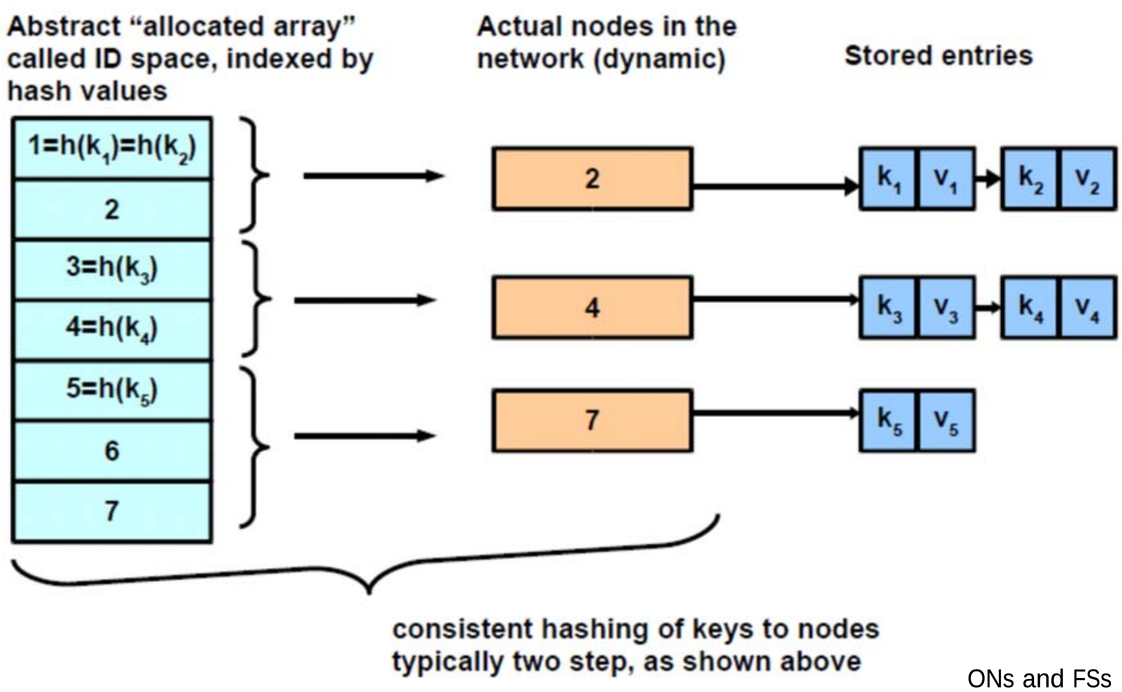
- Assign a **unique live node** to any key
- **Find this node quickly** and cheaply in the overlay network

Support **maintenance of the ON** and **optimization of its current organization of nodes**

- **Load balancing**: maybe even change the key-hash wen the nodes change → necessity of node mapping on the fly
- **Replicate entries on more nodes** to increase availability

DISTRIBUTED HASH TABLES

Find the **best node allocation** depending on **existing nodes where nodes can enter and leave the ON**



STRUCTURED HASH TABLES

Many examples of tools for supporting Distributed Hash Tables - DHT

- **Chord**

  Consistent hashing **ring-based** structure

- **Pastry**

  Uses an ID space concept similar to Chord

  but exploits the concept of a **nested group** toward **acceleration**

Also many other solutions

- CAN

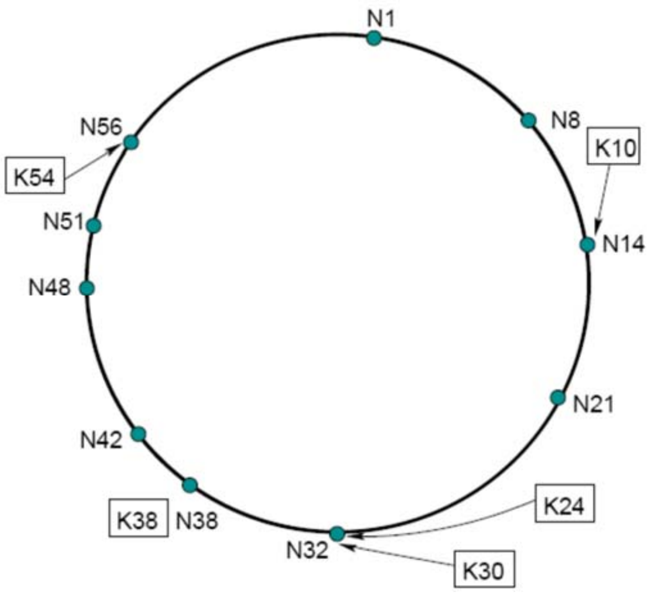
  Nodes/objects are mapped into a d-dimensional cartesian space

…

CHORD HASH TABLES

Hash is **applied over a dynamic ring**

- Consistent hashing based on an **ordered ring overlay of** the **nodes**
- Both keys and nodes are hashed to 160 bit IDs (SHA-1)
- Keys are assigned to nodes by using **consistent hashing**
  - **The key goes into the successor node in the ID space**



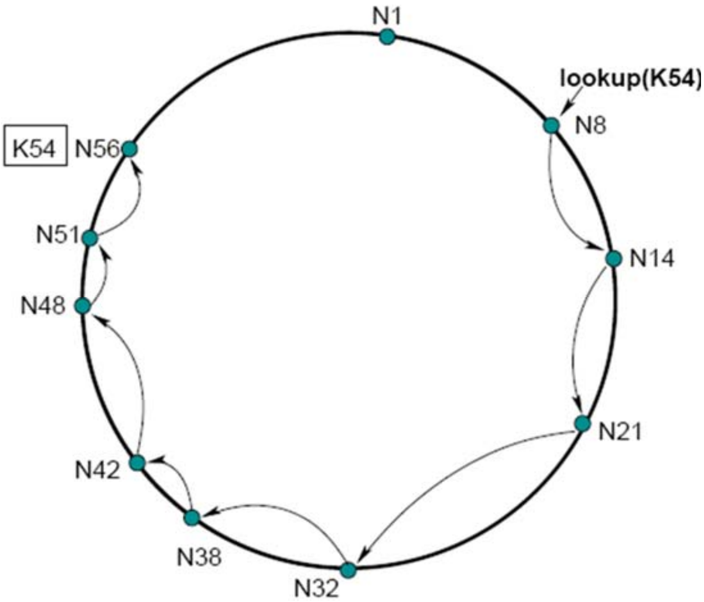
CHORD PRIMITIVE LOOKUP

Lookup query is **forwarded to the successor in one direction (one way)**

- Forward the query **around the circle**
- In the worst case, O(**N**) forwarding is required
  - In two ways, O(**N/2**)

Efficiency

- CHORD may keep **finger tables** to identify faster the node (finger tables as caches for the successors)

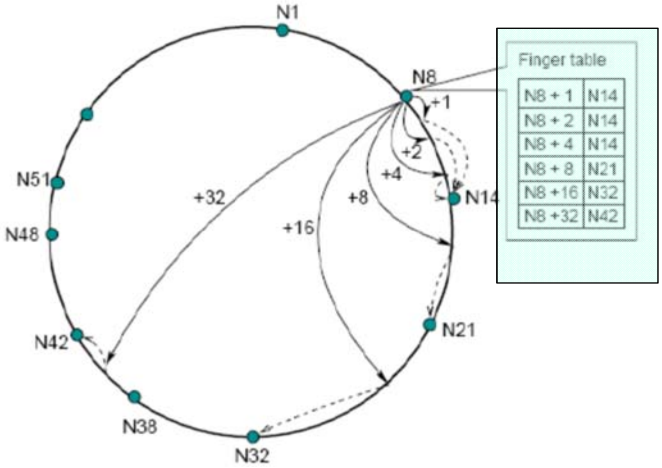


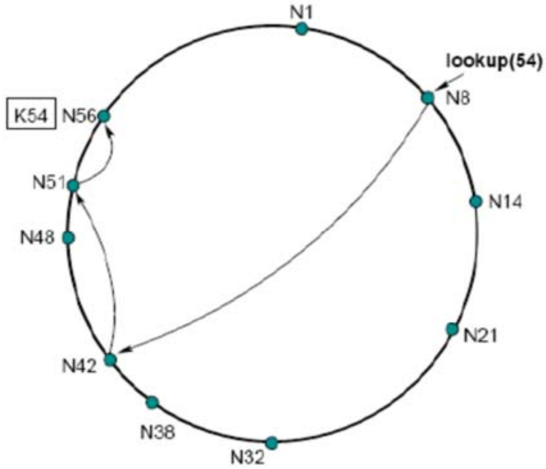
CHORD CONSISTENT HASHING

CHORD works on the idea of making **operations easier**

- **Consistent hashing**
  - **Randomized**
    - All nodes receive roughly an equal share of load
  - **Local**
    - Adding or removing a node involves an O(1/N) fraction of the keys getting new locations
- **Actual lookup**
  - Chord needs to know only **O(log N)** nodes in addition to successor and predecessor to achieve **O(log N)** message complexity for lookup

CHORD SCALABLE LOOKUP





The i_{th}-entry of a **finger table** points the **successor of the key** (nodeID + 2^i)

A finger table has **O(log N)** entries and the scalable lookup is bounded to **O(log N)**

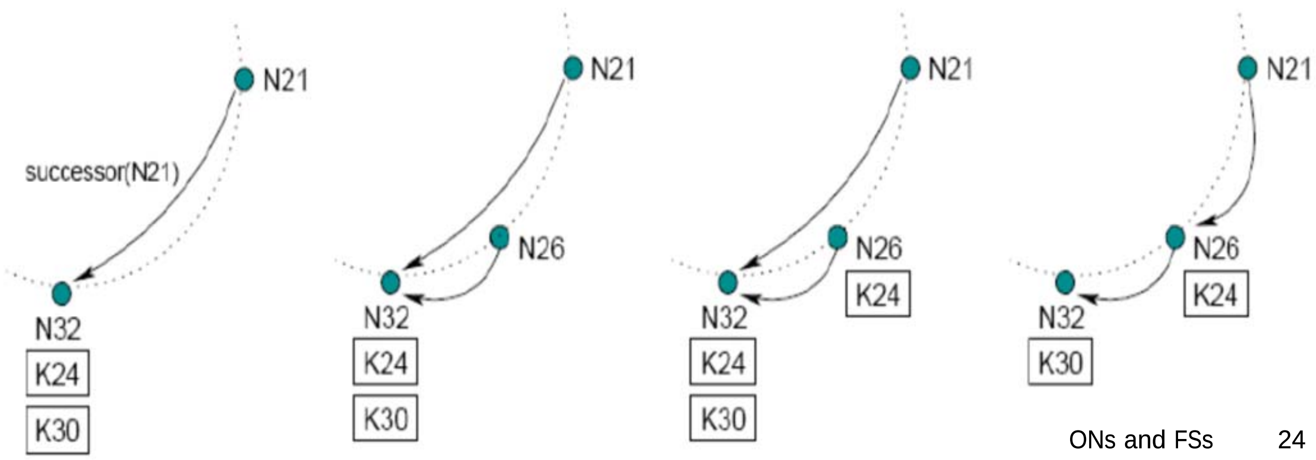
CHORD NODE JOIN

A new node has to

- Fill its own **successor, predecessor** and **fingers**
- Notify other nodes of which it can be a **successor**, **predecessor** and **fingers**

Simple way: **find its successor, then stabilize**

- Join immediately the ring (lookup works), then modify the structure organization – we will optimize **lazely** and **lately**



CHORD STABILIZATION

If the ring is correct, then routing is correct, and **fingers are needed for the speed only**

Stabilization

The support monitors the structure and organizes itself by controlling the ON freshness

- Each node periodically runs the stabilization routine
- Each node refreshes all fingers by periodically calling find_successor($n+2^i-1$) for a random i
- Periodic cost is O(logN) per node due to finger refresh

CHORD FAILURE HANDLING

The failure of nodes is handled by

- **Replication**: instead of one successor, we keep a number of **R successors**
  - **More robust to node failure** (one can find new successor if the old one failed)
- **Alternate paths while routing**
  - If a finger does not respond, take the previous finger, or the replicas, if close enough

In robust DHT, keys replicate on the **R successor nodes**

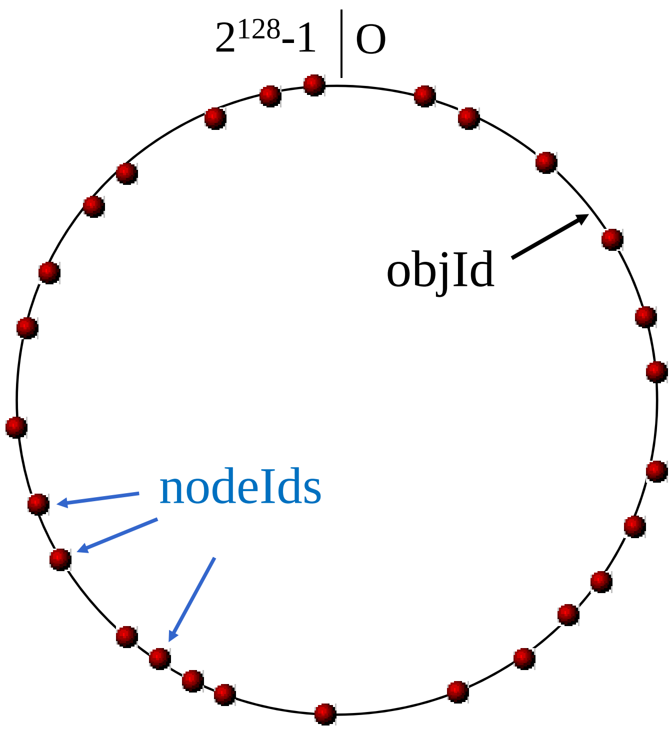
- The stored data become equally more robust

PASTRY

PASTRY is a DHT similar to CHORD in a more organized way for efficient access

- Based on a **sorted ring** in an **ID space** (as in Chord)
  - Nodes and objects are assigned a **128**-bit identifier
- **NodeID** interpreted as a sequence of digits in **base 2^b**
  - In practice, the identifier **is viewed** in Hex (base 16)
  - **Nested groups are the replication entities**
- **The node responsible for a key** is the **numerically closest** (not the **successor**)
  - Bidirectional sequencing by using numerical distance
- **Finger-like shortcuts** can speed up lookups

PASTRY: OBJECT DISTRIBUTION



Consistent hashing of nodes and objects ID

128 bit circular id space

nodeIds (uniform random)

objIds (uniform random)

Invariant: nodes with **numerically closest nodeId** maintain objects

PASTRY

PASTRY keeps

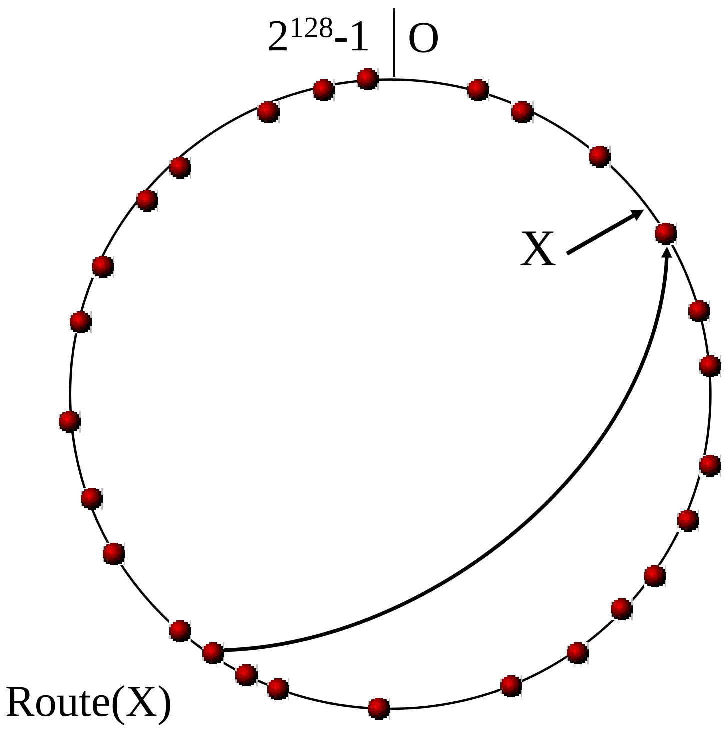
Routing tables to explore **proximity** and find **close neighbors numerically**

Leaf sets to maintain IP addresses of **nodes with closest larger and smaller nodeIds in the close neighborhood**

Generic P2P location and a **routing infrastructure**

- ***Self-organizing*** overlay network
- Lookup/insert object in < $log_{16} N$ routing steps (expected)
- *O(log N)* per-node state
- Network proximity routing

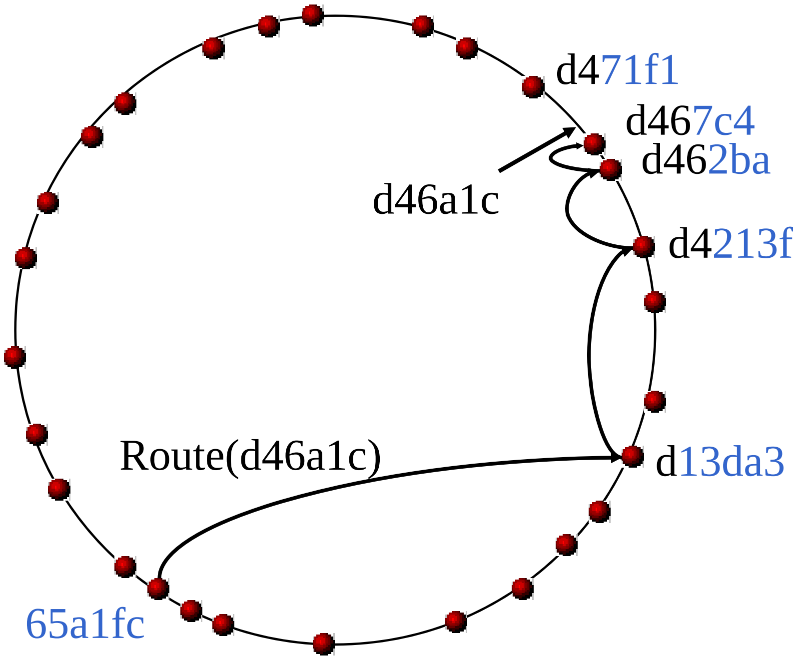
PASTRY INSERT / LOOKUP



A message with key X is routed to live nodes with nodeId closest to X

Problem: complete routing table not feasible

PASTRY ROUTING TABLES



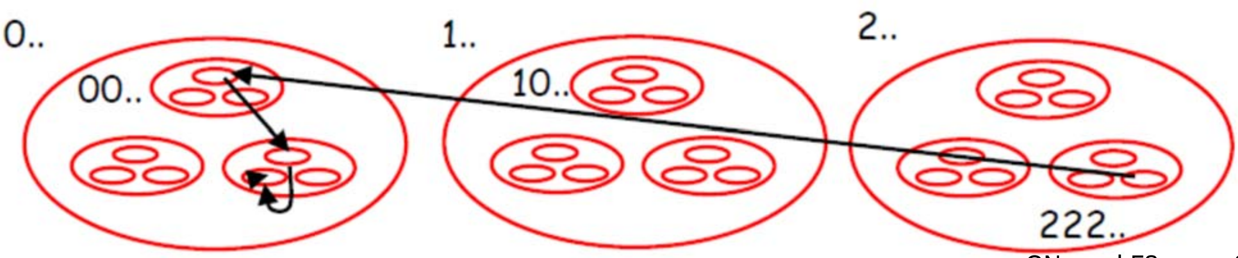
Properties

- $\log_{16}$ N steps
- O(*log N*) state

PASTRY NESTED GROUPS

Simple example: nodes & keys have n-digit **base-3** ids, e.g., 02112100101022

- There are **3** nested groups for each group
- Each node knows IP address of one delegate node in some of the other groups
- Suppose node in group **222**… wants to lookup key k= 02112100210
  - Forward query to a node in 0…, then to a node in 02…, then to a node in 021…, then so on.



PASTRY ROUTING TABLE (# 65A1FC)

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

$\log_{16} N$ rows

PASTRY ROUTING TABLE ...

m=16 b=2

b=2, so node ID is base 4 (16 bits)

Node ID 10233102

Contains the nodes that are *numerically* closest to local node MUST BE UP TO DATE

Leaf set

< SMALLER		LARGER >	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing Table

02212102	1	22301203	31203203
0	11301233	12230203	13021022
10031203	10132102	2	10323302
10200230	10211302	10222302	3
10230322	10231000	10232121	3
10233001	1	10233232	
0		10233120	
		2	

m/b rows

2^b-1 entries per row

Entries in the *m^{th}* column have *m* as next digit

n^{th} digit of current node

Entries in the *n^{th}* row share the first *n* digits with current node

[common-prefix next-digit rest]

Entries with no suitable node ID are left empty

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Contains the nodes that are closest to local node according to proximity metric

PASTRY ROUTING TABLE AND LEAFSET

Leaf set

- Set of nodes that are **numerically closest to the node**, the same as Successors in Chord
  - L/2 smaller & L/2 higher
- Replication boundary
- Stop condition for lookup
- **Support reliability and consistency**

Routing table

- Provides **delegate nodes** in **nested groups**
- **Self-delegates** for the nested group where the node belongs to
- O(log N) rows
  → O(log N) lookup

Base-4 routing table

NodeId 10233102

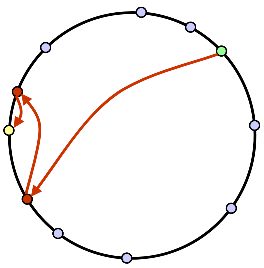
Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

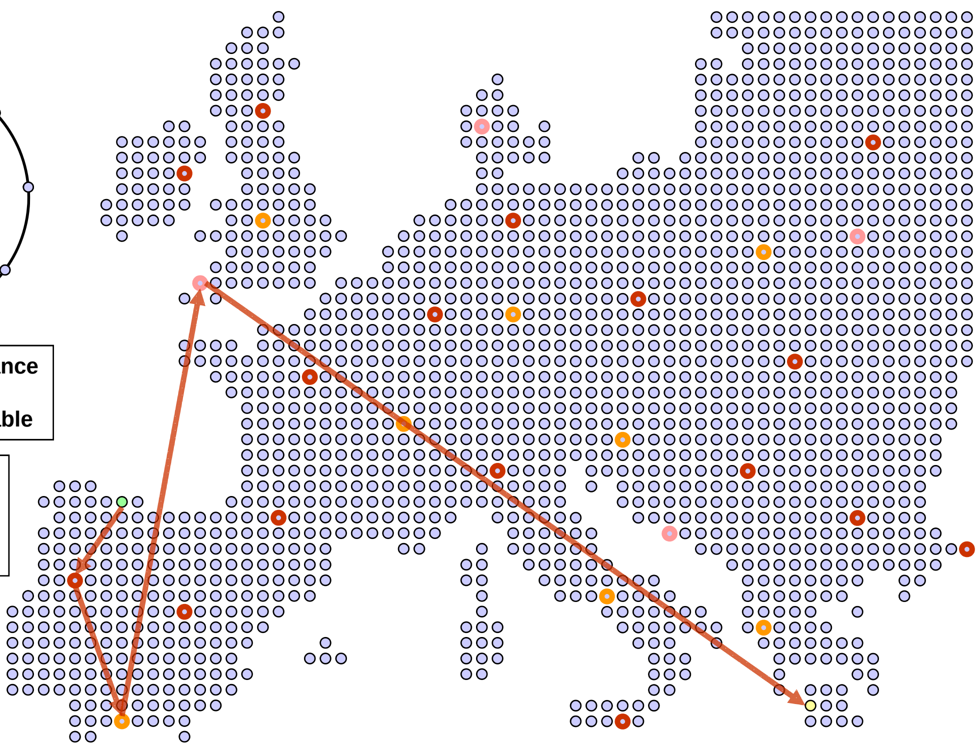
-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

PASTRY ROUTING & TOPOLOGY



Expected node distance increases with row number in routing table

Smaller and smaller numerical jumps
Bigger and bigger topological jumps



PASTRY JOIN AND FAILURES

Join

- Uses routing to find **numerically closest nodes** already in the network
- **Asks state from all nodes** on the route and initializes its own state

Error correction

- **Failed leaf node**: contact a leaf node on the side of the failed node and add an appropriate new neighbor
- **Failed table entry**: contact a live entry with same prefix of the failed entry until new live entry is found
  if none found, keep trying with longer prefix table entries

OVERLAY NETWORK USAGE

ONs are very used inside P2P systems for file exchange

P2P Napster, Gnutella, Kazaa, BitTorrent

Social networks, for instance, need to connect fast different users: **Overlay Nets** can help in preparing a support for those communications, ready to use and always available

So, inside the infrastructure you have those organizations dynamic and continually balanced

Social Nets MSN, Skype, Social Networking Support

Also in case of Cloud large infrastructure, to find parts of the support, when in need of finding new zones and copies

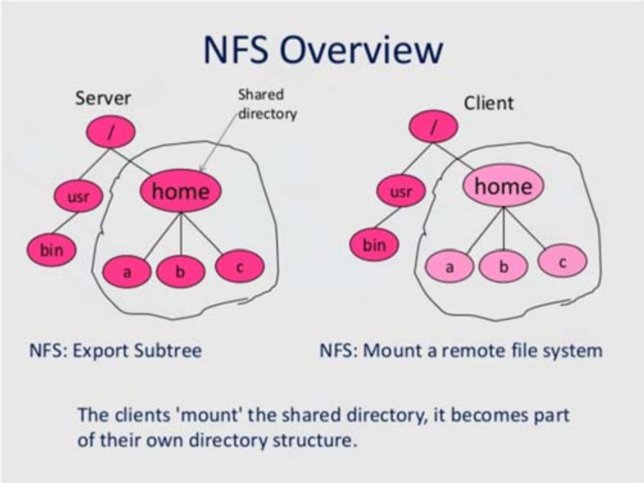
Cloud for internal and federated discovery

…

DISTRIBUTED FILE SYSTEMS

Network File System or **NFS** is the pioneer C/S file system and the most diffused network file system

It is based on the idea of **client machines that interacts with server machines where files reside**



The implementation is **transparent** after the mounting of the file systems in the client

NFS is stateless and efficient: there is no heavy weight on the server machines, and the load is on the client, connections are UDP, etc.

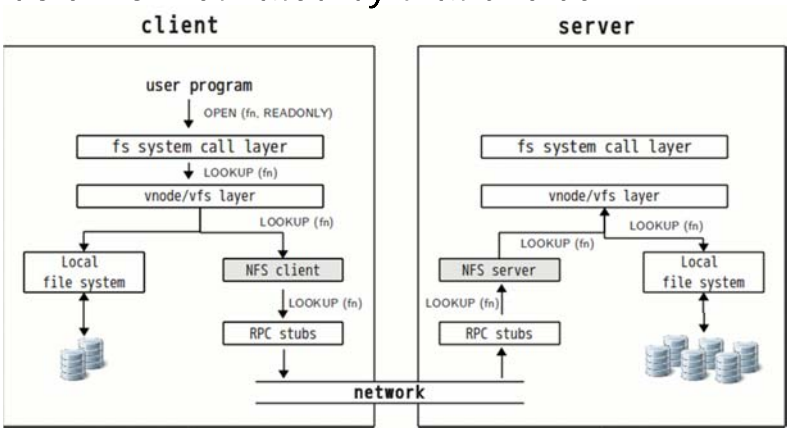
There are many variations based on TCP connections, optimizations, etc.

NFS lacks of any idea of Replication and QoS

DISTRIBUTED FILE SYSTEMS

Network File System had the initial goal of using **RPC for the entire communication support** so it strives for efficiency and reducing costs

The large diffusion is motivated by that choice



The implementations are optimized and the overhead very low: the diffusion was incredibly large

No replication nor QoS are granted

GLOBAL FILE SYSTEMS

Modern global systems need new tools for **data storage with the necessary quality and also with global scalability**

File systems must use **replication** and other strategies toward **quality**

Starting with traditional C/S ones to

Typically **dynamic management of data in all their parts to achieve QoS**

- **Distributed** file systems
  - Google File System for Google data GFS
  - Hadoop file system HDFS
- **Other solutions … later**

Google File System (GFS)

GFS exploits **Google** hardware, data, and application properties to improve performance of storage and search

- **Large scale**: thousands of machines with thousands of disks
- Files are **huge** (normal files have multi-GB size)
  - **Design decision: difficult to manage billions of small files**
- **File access** model is **read/append (almost no write)**
  - **Most reads are sequential**
  - **Random writes practically non-existent**
- **Component failures** are '**normal**' events
  - Hundreds of thousands of machines/disks
  - **MTBF of 3 years/disk → 100 disk failures/day**
  - Additionally other failures: **network, memory, power failures**

DESIGN CRITERIA

Detect, tolerate, and recover automatically from failures

Deal with a "**limited**" number of **large files**

- Just a few millions
- Each 100MB – multi-GB
- Few small files

- **Read-mostly** workload
  - Large **streaming reads** (multi-MB at a time)
  - Large **sequential append operations**
    - Provide atomic consistency to parallel writes with low overhead
- High-sustained **throughput** more important than **low latency**

DESIGN NOVEL STRATEGIES

Files stored as **chunks kept with their descriptions (metadata)** and stored as local files on Linux file system

- Reliability through **replication** (at least 3+ replicas)
- **Single master** coordinates access and keeps **metadata**
  - **Simple centralized design** (one master per GFS cluster)
  - Global knowledge to optimize **chunk placement and replication decisions** using **no caching**
  - **Large data set/streaming reads** render **caching useless**
  - Clients **cache meta-data** (e.g., chunk location)
  - Linux buffer **cache allows keeping interesting data in memory for fast access**

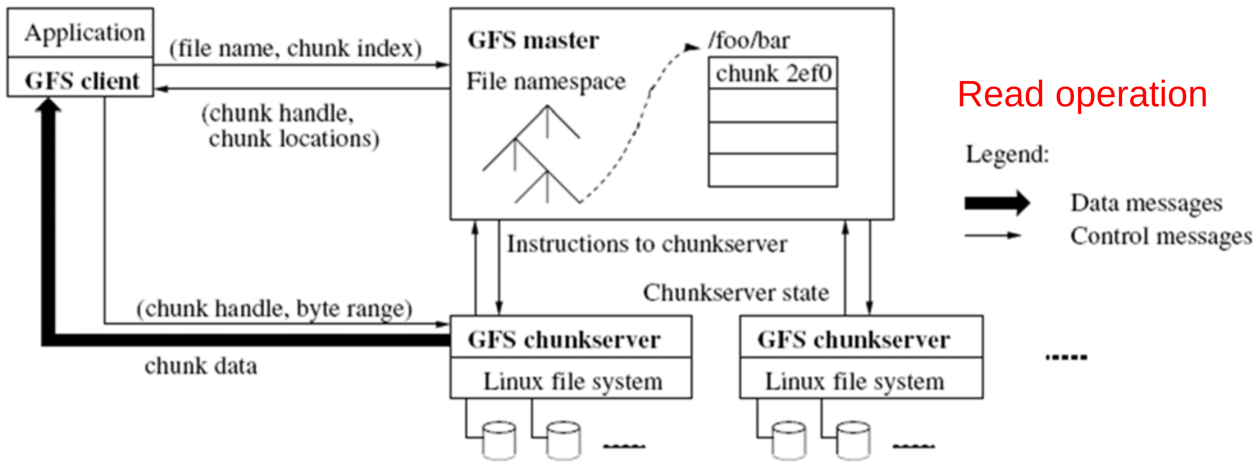
GFS ARCHITECTURE

One master server (backups replicate its state replicated) and **many chunk servers** (100s – 1000s) over linux

- **Chunk**: 64 MB portion of file, identified by 64-bit, **globally unique IDs**
- Chunks are spread across racks for better throughput & fault tolerance

Many clients accessing files stored on the same cluster

- Data flow: client <-> chunk server (master involved just in control)



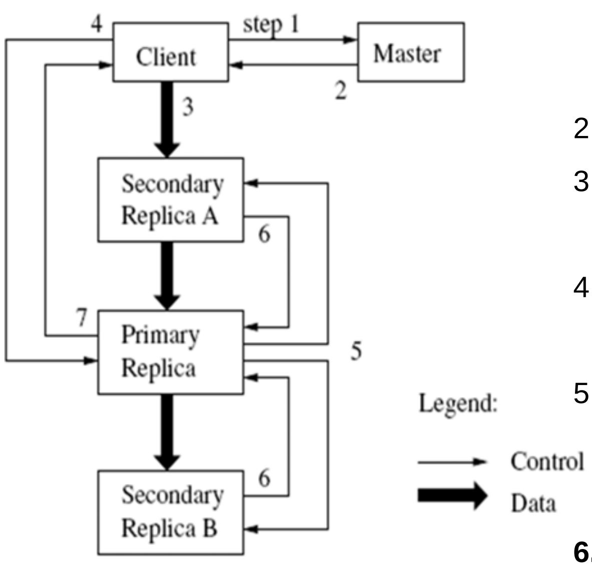
MORE ON METADATA & CHUNKS

- **Metadata (the file description)**
  - **3 types**: **file/chunk namespaces**, **file-to-chunk mappings**, **location of replicas of any chunk**
  - **All in memory** (< 64 bytes per chunk) with **GFS capacity limitation**
- **Large chunk have many advantages**
  - Fewer **client-master** interactions and **reduced size of metadata**
  - Enable persistent **TCP connection** between **clients and chunk servers**

MUTATIONS, LEASES, VERSION NUMBERS

- **Mutation**: operation that changes either the **contents** (write, append) or **metadata** (create, delete) of a chunk
- **Lease**: mechanism used to maintain **consistent mutation order across replicas**
  - Master grants a **chunk lease** to one replica (primary chunk server)
  - Primary picks a **serial order to all mutations to the chunk** (many clients can access chunk concurrently)
  - All replicas follow **this order when applying mutations**
- **Chunks** have **version numbers** to distinguish between **up-to-date** and **stale replicas**
  - Stored on disk at master and chunk servers
  - Each time master grants new lease, increments version & informs all replicas

Mutations step-by-step



1. Identities of **primary chunk server** holding lease and the **secondaries holding the other replicas**
2. Reply
3. Push data to **all replicas for consistency** (see next slide for details)
4. Send mutation **request to primary**, which assigns it a serial number
5. Forward mutation request **to all secondaries**, which apply it according to its serial number
6. **Ack** completion
7. **Reply** (an error in any replica results in an error code & a client retry)

DATA FLOW

Client can push **the data** to **any replica**

Data is pushed linearly along a carefully picked chain of **chunk servers**

- Each machine forwards data to "***closest***" machine in network topology that has not received it
  - Network topology is simple enough that "distances" can be accurately estimated from IP addresses
- **Pipelining**: servers receive and send data at the same time

Method introduces delay, but offers good **bandwidth** utilization

CONSISTENCY MODEL

- File **namespace mutations** (create/delete) are **atomic**
- State of a **file region** depends on
  - **Success/failure of mutations (write/append)**
  - **Existence of concurrent mutations**
- **Consistency states of replicas** and **files**:
  - **Consistent**: all clients see same data regardless of replica
  - **Defined**: consistent & client sees the mutation in its entirety
    - Example of consistent but undefined: initial record = AAAA
      concurrent writes: _B_B and CC_C;
      result = CBAC (none of the clients sees the expected result)
  - **Inconsistent**: due to a failed mutation
    - Clients see different data function of replica

UNDEFINED STATE AVOIDANCE

Traditional **random writes would require expensive synchronization** (e.g., **lock manager**)

- Serializing writes does not help (see previous slide)

Atomic *record append*: allows **multiple clients to append data to the same file concurrently**

- Serializing append operations at primary solves the problem
- The result of successful operations is well defined: **data is written at the same offset by all replica with an "at least once" semantics**
  - If one **operation fails at any replica**, the client retries; as a result, replicas may contain duplicates or fragments
  - If not enough space in chunk, add padding and return error and Client retries

RECORD APPEND SEMANTICS

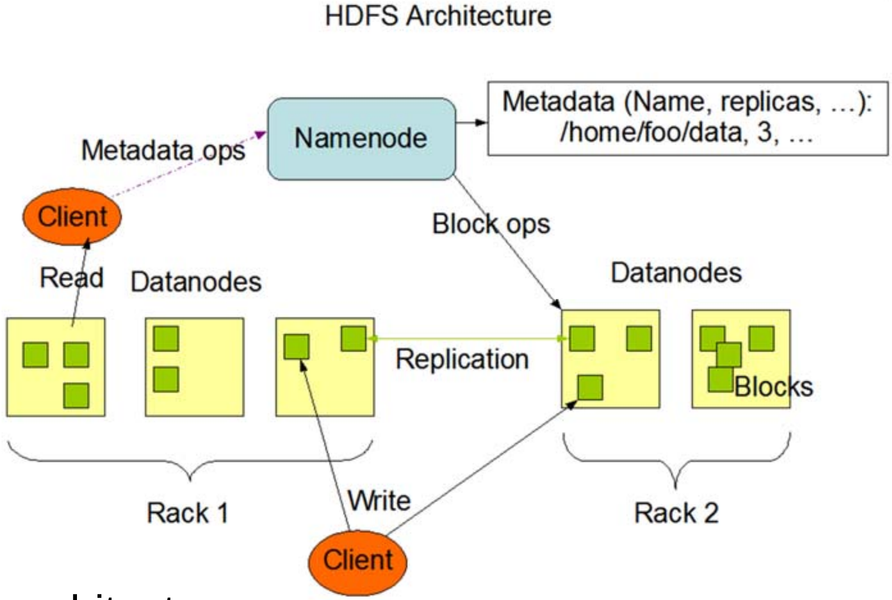
The applications must deal with **record append semantics** for specific cases

Applications using ***record append*** should include **checksums** in writing records

- Reader can **identify padding/record fragments** using **checksums**
- If application cannot tolerate duplicated records, should include unique ID in record
  - Readers can use **unique IDs** to filter duplicates

HDFS (another distributed file system)

Inspired by GFS



Master/slave architecture

- **NameNode** is master (meta-data operations, access control)
- **DataNodes** are slaves: one copy per node in the cluster
- **Files** are stored in blocks in several DataNodes

ROLES AND PRINCIPLES OF HDFS

Hadoop **D**istributed **F**ile **S**ystem is based on low cost hardware but with high fault tolerance and high availability

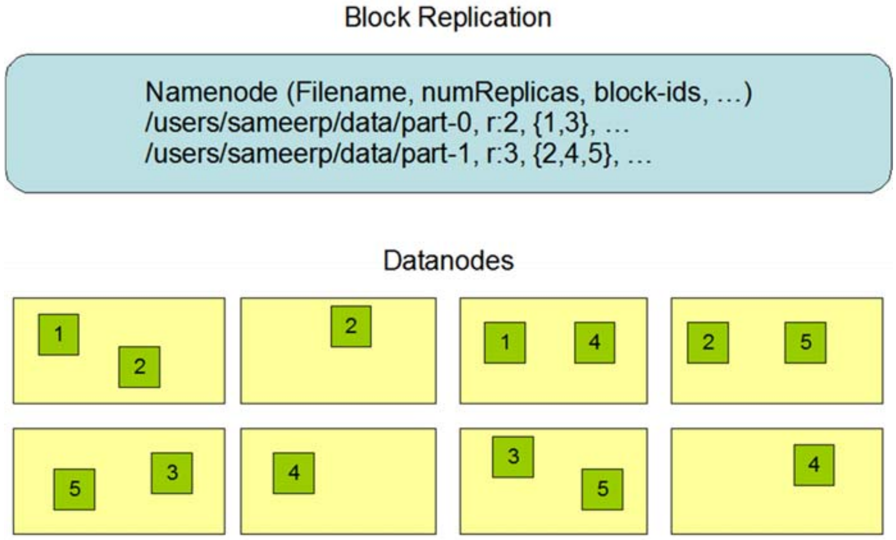
Applications access with **write-once-and-read-many** so the consistency model is similar to GFS and **computation is moved close to the related data** to operate upon

- **NameNodes** execute file system NameSpace operations like open, close, directories,…and decide on mapping
- **DataNodes** execute read write operations requested from Clients and operates on block of data

HDFS is **written in Java** and must work on **normal hardware** to store very large files on different machines so to minimize the probability of faults by using replication

Any file can decide its block size and replication degree

HDFS REPLICATION



Applications can decide **dynamically** the **replication factor**

Again master/slave architecture: **NameNode** receives **heartbeats** and **block reports** from DataNodes

- **Hearbeats** grant the **operation state** of DataNodes
- **Block reports** give the current block situations of DataNodes