Distributed Storage

Wesley Maness
Zheng Ma
Hong Ge

Outline of today

- ***Overview of a distributed storage system (Wesley)***
- Routing in such system and DHT (Zheng)
- Distributed File System (Hong)

Where are we heading?

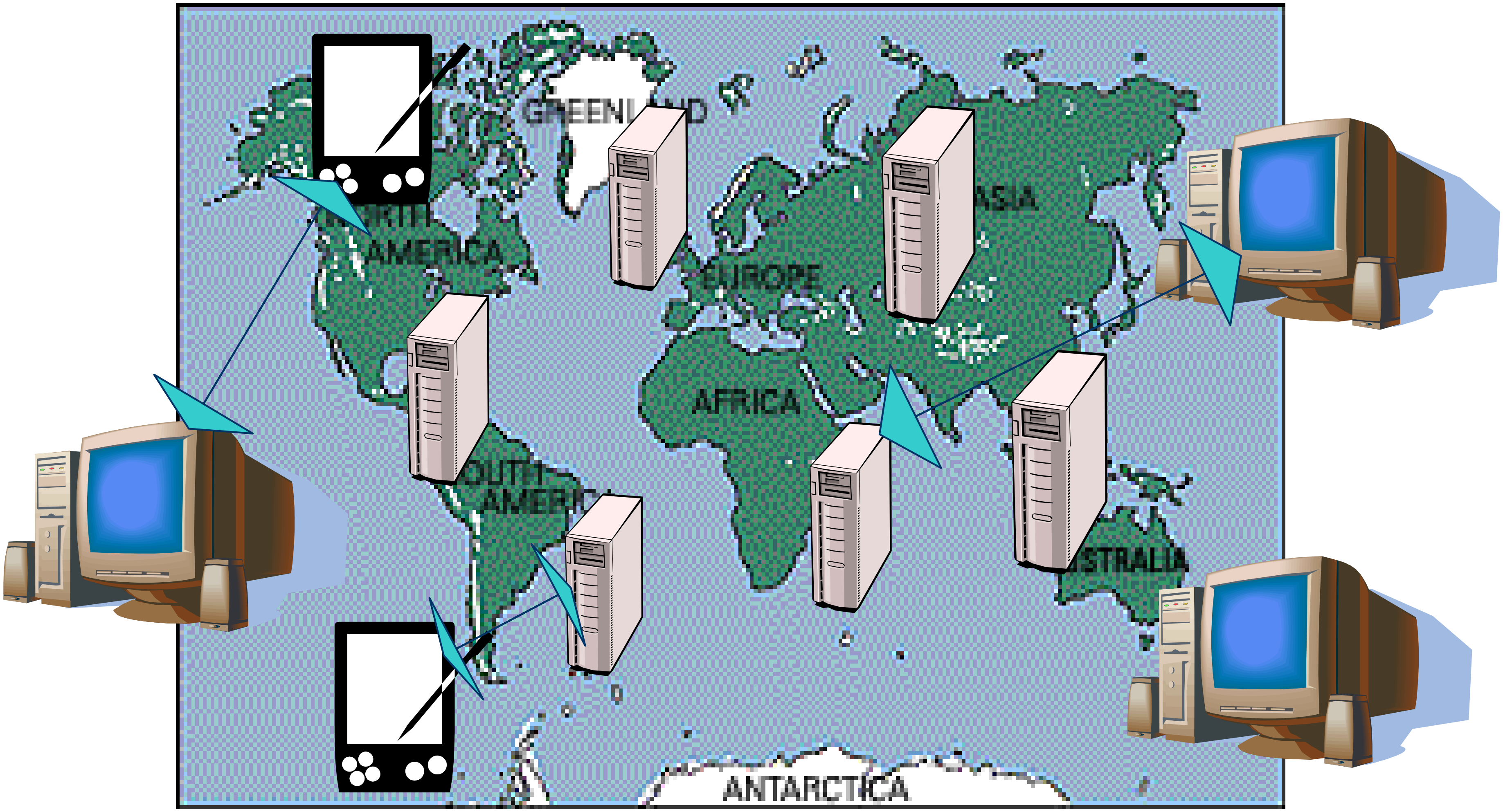
- Exploiting ubiquitous computing
  - Small devices, sensors, smart materials, cars, etc
  - Are we there? Cell-phone, watch, pen, smart-jacket, etc.
- Planetary-scale Information Utilities
    - Infrastructure is *transparent* and always active
    - Extensive use of redundancy of hardware and data
  - Devices that negotiate their interfaces automatically
  - Elements that tune, repair, and maintain themselves

So what does this mean?

- Personal Information Mgmt is the Killer App
- Time to move beyond the Desktop
- Information Technology as a Utility

- *Some people think OceanStore is the answer*

OceanStore: An Architecture of Global-Scale Persistent Storage



OceanStore: ~ a Utility Infrastructure

- You want storage but without the issues of backup, loss, secure
- [is there a need?] Outsourcing of storage is already common
- [basic idea] to pay your monthly bill and your data is always there
  - One company, one bill, simple pay structure

OceanStore: ~ desired properties

- Automatic maintenance
  - Adapt to failure, repair itself, changes
- How long should information be guaranteed?
- Divorce information from location…
  - System not disabled from natural disasters -> how do you solve this?
  - Adopts in changes in demands and regional outages

Assumptions

- Untrusted Infrastructure
  - Untrusted components, only ciphertext in infrastructure
- (Responsible) Entity
  - Storage Provider would guarantee the durability and consistency of data
  - Only trusted with integrity not content of data
- Well Connected
  - Producers and consumers most of time connected to high-bandwidth network
- Promiscuous Caching (data that can flow anywhere is referred to as *nomadic data*) (difference between NFS/AFS)
  - Data can be cached anytime, anywhere
- Optimistic Concurrency via Conflict Resolution (CVS)
  - Avoid locking in wide area!

Underlying Technology

- Access Control
- Data Update
  - Primary Replica
  - Archival Storage
  - Secondary Replica
- Data Read
- Data Location & Routing *;Tapestry*

Access Control

- Reader Restriction
  - Encrypt All Data
  - Distribute Encryption Key to Users with Read Permission

- Writer Restriction
  - Access Control List (ACL) for an Object
  - All Writes be Signed so that Well-behaved Servers and Clients Verify them based on the ACL

Underlying Technology

- Access Control
- Data Update
  - Primary Replica
  - Archival Storage
  - Secondary Replica
- Data Read
- Data Location & Routing (Tapestry)

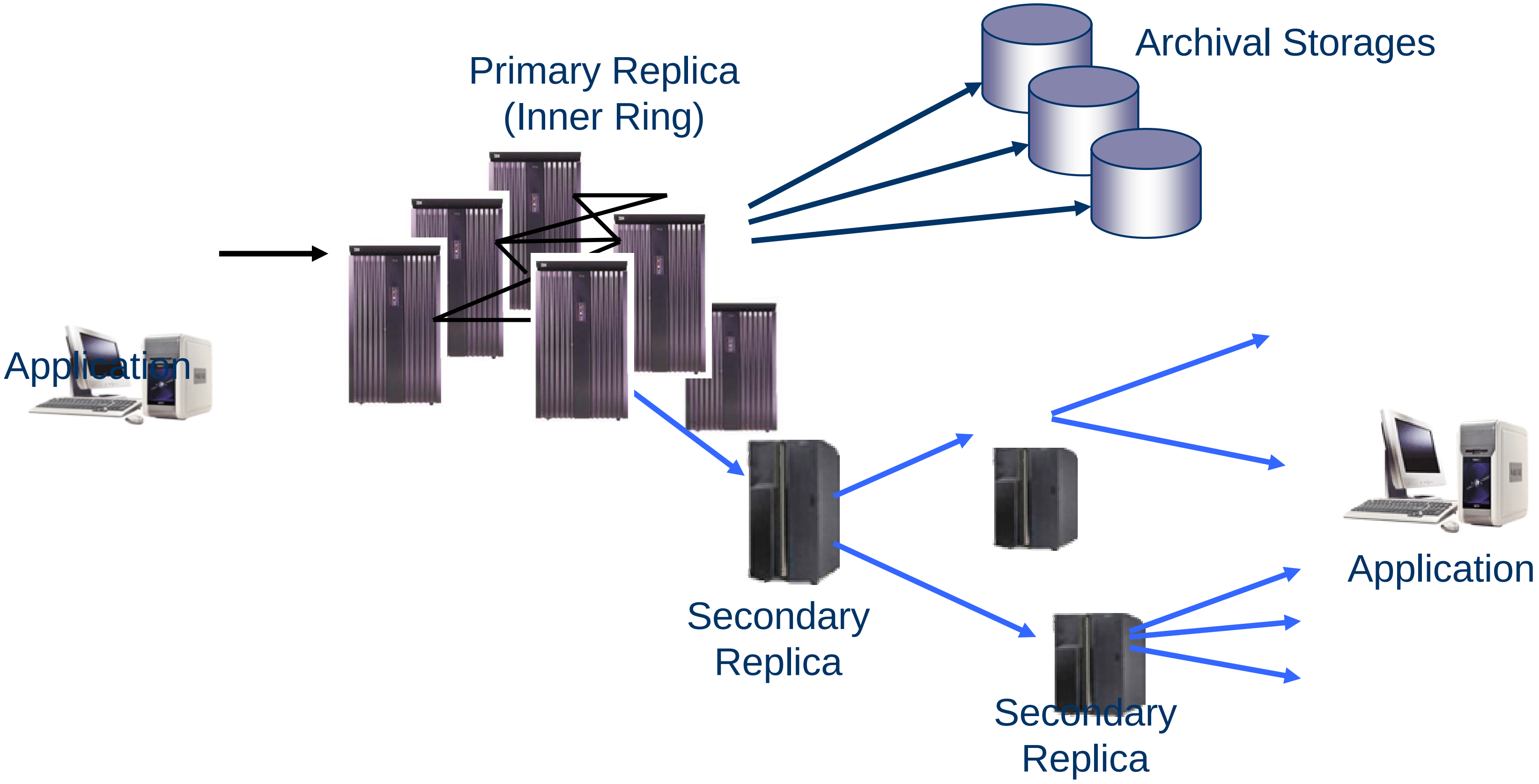
< Update Message Format >

Timestamp
Client ID
<Predicate 1, Action 1>
<Predicate 2, Action 2>
. . .
<Predicate N, Action N>
Client Signature

Data Update (1/2)

- Adding a New Version to the Head of Version Stream
- Array of Potential Actions each Guarded by a Predicate
  - *Predicate* Examples
    - Checking Latest Version_Num, Comparing a Region of Bytes to an Expected Value, etc.
  - *Action* Examples
    - Replacing a Set of Bytes, Appending New Data, Truncating the Object, etc.

Data Update (2/2)



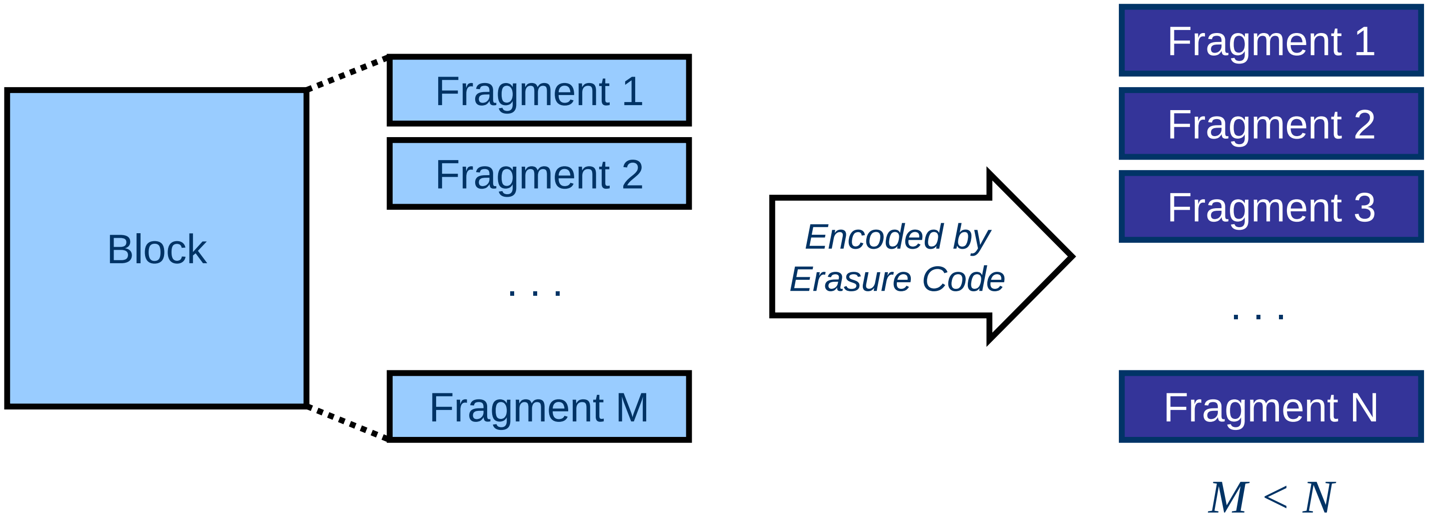
< OceanStore Update Path >

Primary Replica

- Inner Ring
  - A Set of Servers that Implement Object’s Primary Replica
  - Applies Updates and Creates New Versions
    - Serialization
    - Access Control
    - Create Archival Fragments
  - Update Agreements
    - *Byzantine Agreement Protocol*
      - Distributed Decision Process in which All Non-faulty Participants Reach the Same Decision for a Group of Size 3f+1, no more than f Faulty Servers

Archival Storage

- Simple Replication
  - Tolerance of One Failure for an Addition 100% Storage Cost
- Erasure Codes
  - Efficient and Stable Storage for Archival Copies
  - Storage Cost by a Factor of N/M
  - Original Block can be Reconstructed from Any M Fragments



Secondary Replica

- *Whole-block Caching to Avoid Erasure Codes on Frequently-read Objects*

- Push-based Update
  - Every Time the Primary Replica Applies an Update

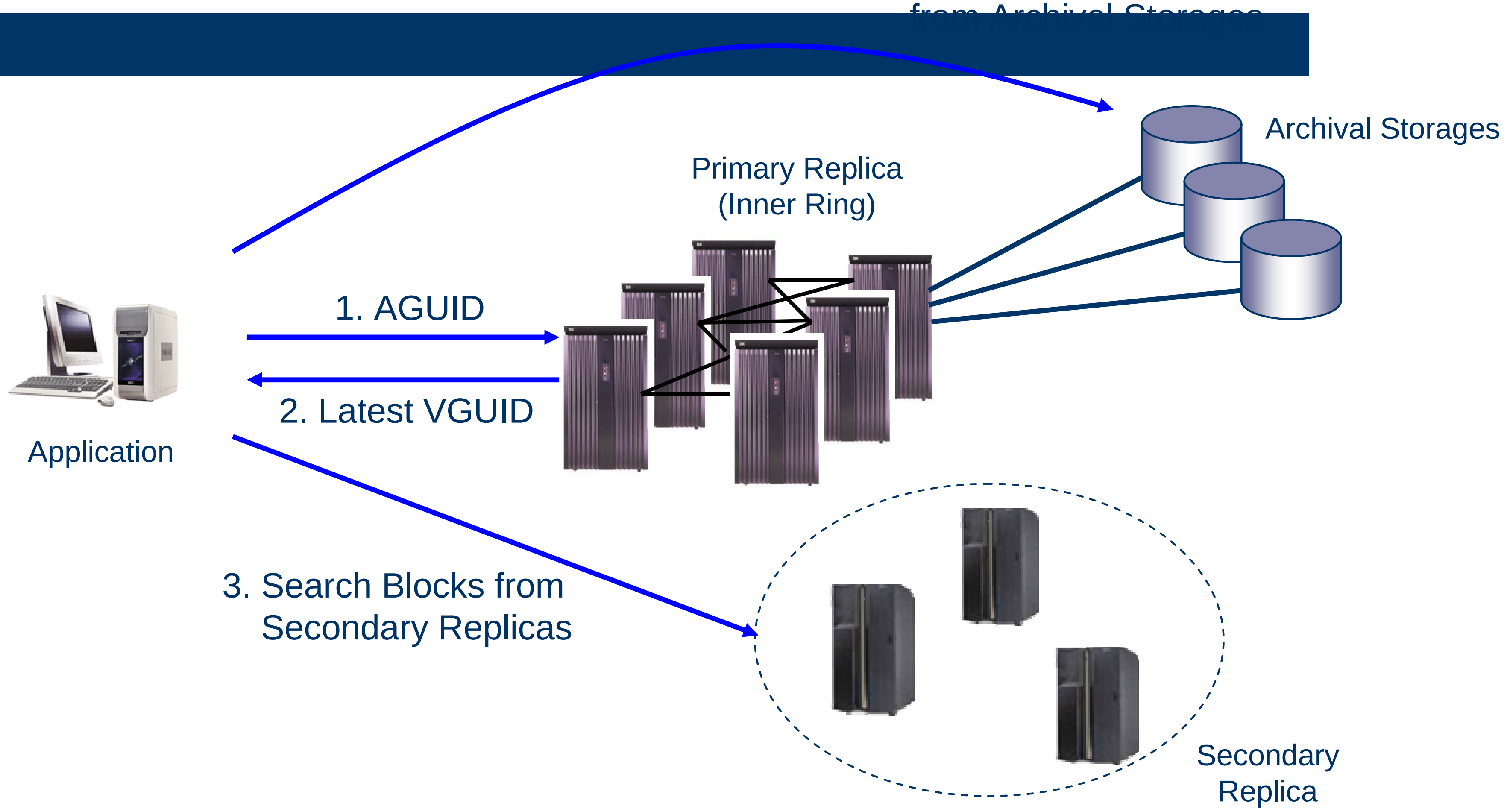
- *Dissemination Tree*
  - Application-level Multicast Tree
  - Rooted at Primary Replica
  - Parent Nodes are Pre-existing Replicas to Serve Objects

Underlying Technology

- Access Control
- Data Update
  - Primary Replica
  - Archival Storage
  - Secondary Replica
- Data Read
- Data Location & Routing (Tapestry)

Data Read

4. Search enough Fragments from Archival Storages



Archival Storages

Primary Replica (Inner Ring)

1. AGUID

2. Latest VGUID

Application

3. Search Blocks from Secondary Replicas

Secondary Replica

Introspective Optimization

- Mimics adaptation in biological systems
- Optimization of Plaxton mesh – (cluster reorganization, attempts to identify and group closely related files) (which is Tapestry, more robust, etc.)
- Replica Management – adjusts the number and location of floating replicas in order to service access requests more efficiently

OceanStore Conclusions

- OceanStore: another utility provider
  - Global Utility model for persistent data storage
- OceanStore assumptions:
  - Untrusted infrastructure with a responsible party
  - Mostly connected with conflict resolution
  - Continuous on-line optimization
- OceanStore properties:
  - Provides security, privacy, and integrity
  - Provides extreme durability
  - Lower maintenance cost through redundancy, continuous adaptation, self-diagnosis and repair
  - Large scale system has good statistical properties
- (Pond is Next) hopefully a better idea of conflict resolution and encryption

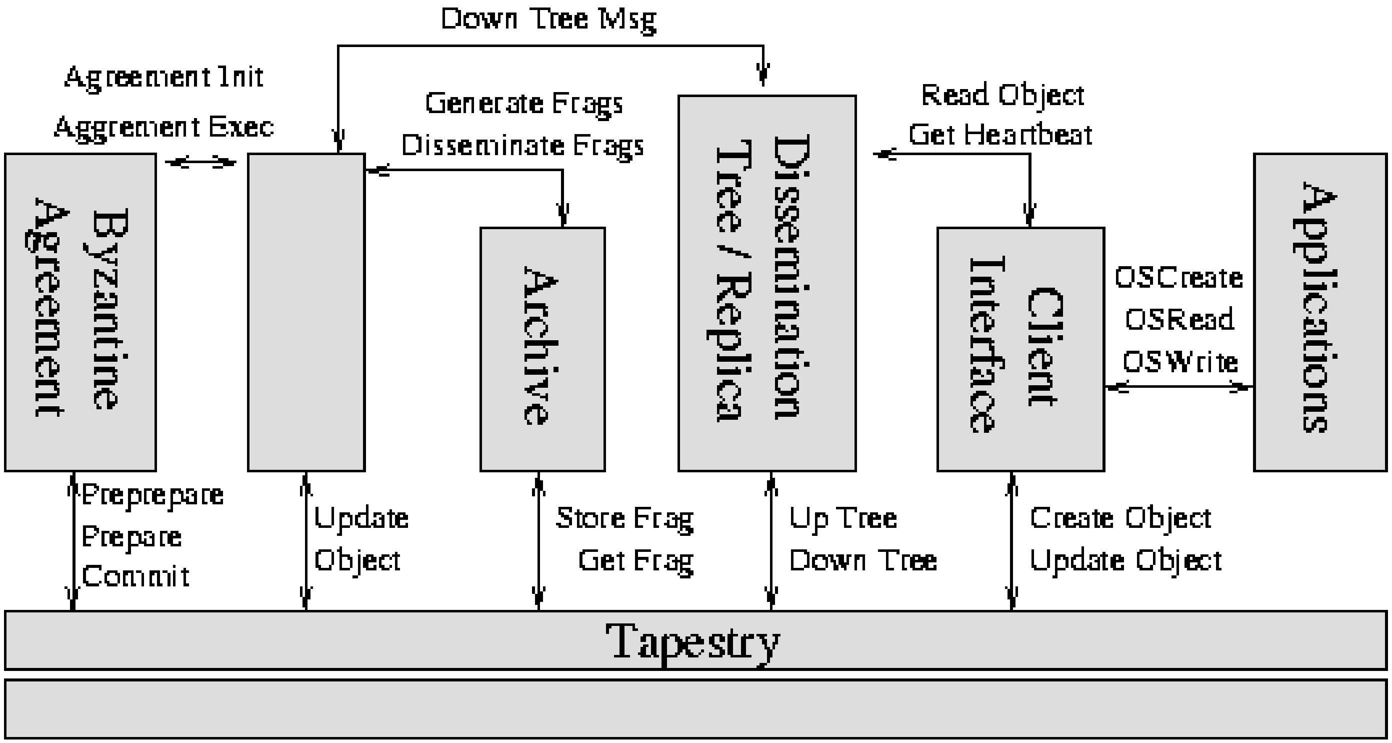
Pond

- Java Implementation of OceanStore proposal
- Included Components
  - ✓ Initial floating replica design
    - Conflict resolution and Byzantine agreement
  - ✓ Routing facility (Tapestry)
    - Bloom Filter location algorithm
    - Plaxton-based locate and route data structures
  - ✓ Introspective gathering of tacit info and adaptation
  - ✓ Initial archival facilities
    - Interleaved Reed-Solomon codes for fragmentation
    - Methods for signing and validating fragments
- Target Applications
  - – Email application, proxy for web caches, streaming multimedia applications

Pond ~ current status

- Subsystems operational
  - Fault-tolerant inner ring - only inner ring can apply updates – access control, serialization
  - Self-organizing second tier (allows for faster fetching, reads)
  - Erasure-coding archive (deep-archival)

Pond



JNI for crypto, SEDA stages, 280+kLOC Java

Pond ~ Testing & Results

- Ran 500 virtual nodes on PlanetLab
  - Inner Ring in SF Bay Area
  - Replicas clustered in 7 largest P-Lab sites
- Streams updates to all replicas
  - One writer - *content creator* – repeatedly appends to data object
  - Others read new versions as they arrive
  - Measure network resource consumption
- (next slide)

Results of ‘NFS vs. OceanStore’

	LAN	(local cluster)		WAN	(PL: NFS UW, IR in UCB, S, UW)	
	Linux	OceanStore		Linux	OceanStore	
Phase	NFS	512	1024	NFS	512	1024
I(w)	0	1.9	4.3	0.9	2.8	6.6
II(w)	0.3	11	24	9.4	16.8	40.4
III(r)	1.1	1.8	1.9	8.3	1.8	1.9
IV(r)	0.5	1.5	1.6	6.9	1.5	1.5
V(r+w)	2.6	21	42.2	21.5	32	70
Total	4.5	37.2	73.9	47	54.9	120.3

All experiments are run with the archive disabled using 512 or 1024-bit keys, as indicated by the column headers. Times are in seconds, and each data point is the average over at least three trials. The standard deviation for all points was less than 7.5% of the mean.

Future Research areas

- The removal of bottlenecks in updates and redundancy propagation
- Improve stability in global distributed environment, e.g. better load balancing techniques
- Data Structure Improvement
- Management of replicas
- Archival Repair

Outline of today

- Overview of a distributed storage system (Wesley)
- ***Routing in such system and DHT (Zheng)***
- Distributed File System (Hong)

Preface: From Tapestry to Chord and beyond

- Who am I:
  - 3rd Year PhD student in system group
  - http://www.cs.yale.edu/~zhengma
- What will I present:
  - Distributed file sharing and P2P system
  - Routing algorithms for DHT

Talk Outline of this part

- ***Motivation for OceanStore and Tapestry***
- Tapestry overview and details (optional)
- Motivation for P2P system and DHT
- Chord overview and details (optional)
- Ongoing work / Open problems

Challenges in the Wide-area

- Trends:
  - Exponential growth in CPU, storage
  - Network expanding in reach and b/w
- Can applications leverage new resources?
  - Scalability: increasing users, requests, traffic
  - Resilience: more components → more failures
  - Management: intermittent resource availability → complex management schemes
- Proposal: an infrastructure that solves these issues and passes benefits onto applications

Driving Applications

- Leverage of cheap & plentiful resources: CPU cycles, storage, network bandwidth
- Global applications share distributed resources
  - Shared computation:
    - SETI, Entropia
  - **Shared storage (Today's focus)**
    - **OceanStore, Gnutella**
  - Shared bandwidth
    - Application-level multicast, content distribution networks
- **Question**: Are they really in large demand? Vague future or not? What else? Killer app?

Answers: my 3 cents

- End 2 End arguments in network community
  - Implement a feature on upper layer as much as we can to have easier deployment for Internet
- Fast development of applications
  - Moore law in computer hardware
- Relatively slow change in Internet core
  - Not too many industrial researchers who work on core networking. (http://www.icir.org/floyd/talks/NSF-Jan03.pdf)

Key problem: Location and Routing

- Hard problem in a system like this:
  - Locating and messaging to resources and data
- Goals for a wide-area overlay infrastructure
  - Easy to deploy
  - Scalable to millions of nodes, billions of objects
  - Available in presence of routine faults
  - Self-configuring, adaptive to network changes
  - Localize effects of operations/failures

Talk Outline

- Motivation for OceanStore and Tapestry
- ***Tapestry overview and details (optional)***
- Motivation for P2P system and DHT
- Chord overview and details (optional)
- Ongoing work / Open problems

What is Tapestry?

- A prototype of a *decentralized, scalable, fault-tolerant, adaptive* location and routing infrastructure
  (Zhao, Kubiatowicz, Joseph et al. U.C. Berkeley)
- Network layer of OceanStore
- Routing: Suffix-based hypercube
  - Similar to Plaxton, Rajamaran, Richa (SPAA97)
- Decentralized location:
  - Virtual hierarchy per object with cached location references
- Core API:
  - *publishObject(ObjectID, [serverID])*
  - *routeMsgToObject(ObjectID)*
  - *routeMsgToNode(NodeID)*

Tapestry details (optional)

- Namespace (nodes and objects)
  - 160 bits → 2^{80} names before name collision
  - Each object has its own hierarchy rooted at *Root*
    f(ObjectID) = RootID, via a dynamic mapping function
- Suffix routing from A to B
  - At h^{th} hop, arrive at nearest node hop(h) s.t.
    hop(h) shares suffix with B of length *h* digits
  - Example: 5324 routes to 0629 via
    5324 → 2349 → 1429 → 7629 → 0629
- Object location:
  - Root responsible for storing object's location
  - Publish / search both route incrementally to root

Publish / Lookup (optional)

- Publish object with ObjectID:

  // route towards "virtual root," ID=ObjectID

  For (i=0, i<Log_2(N), i+=j) { //Define hierarchy

  - j is # of bits in digit size, (i.e. for hex digits, j = 4)
  - Insert entry into nearest node that matches on last i bits
  - If no matches found, deterministically choose alternative
  - Found real root node, when no external routes left

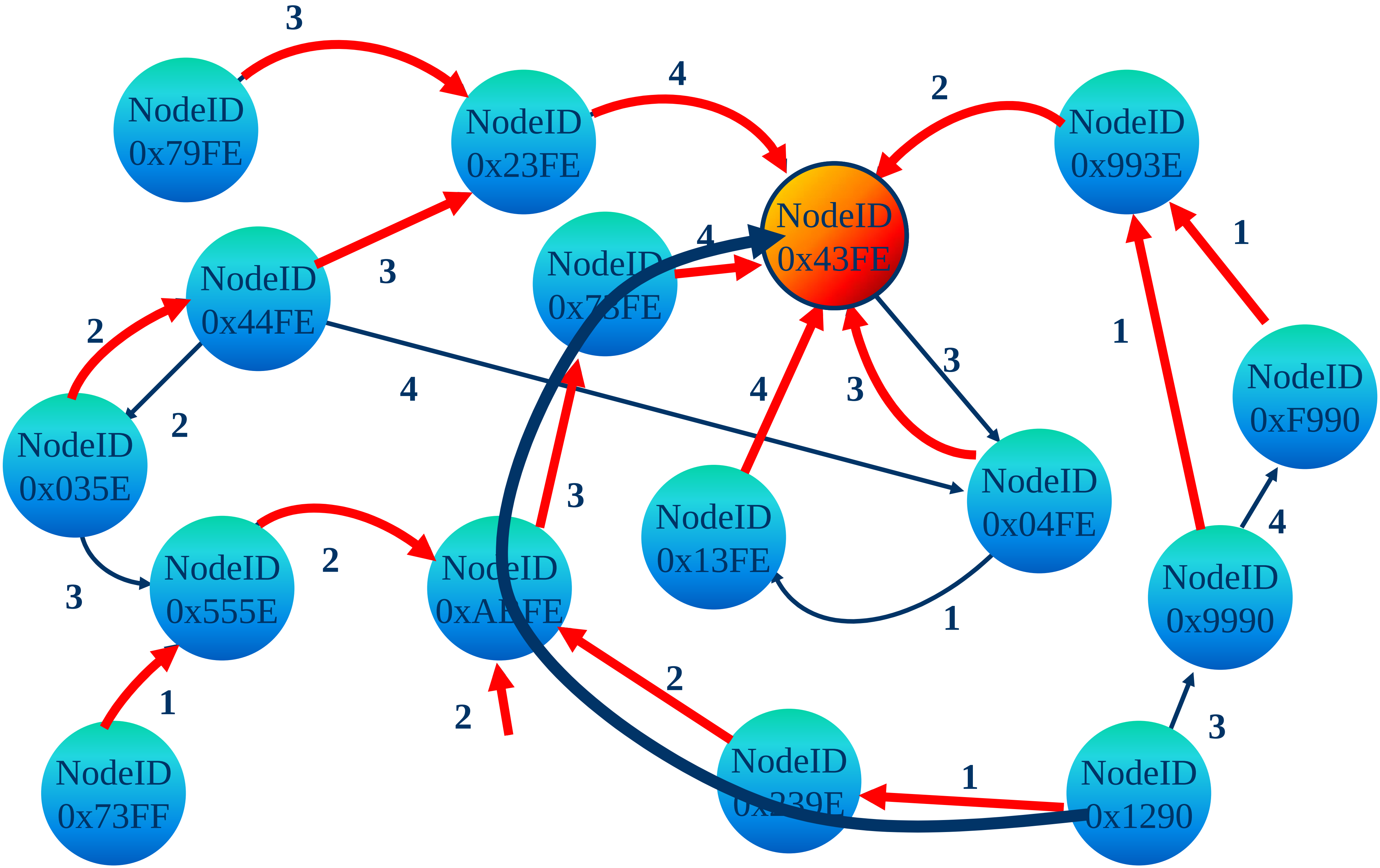
- Lookup object

  Traverse same path to root as publish, except search for entry at each node

  For (i=0, i<Log_2(N), i+=j) {

  - Search for cached object location
  - Once found, route via IP or Tapestry to object

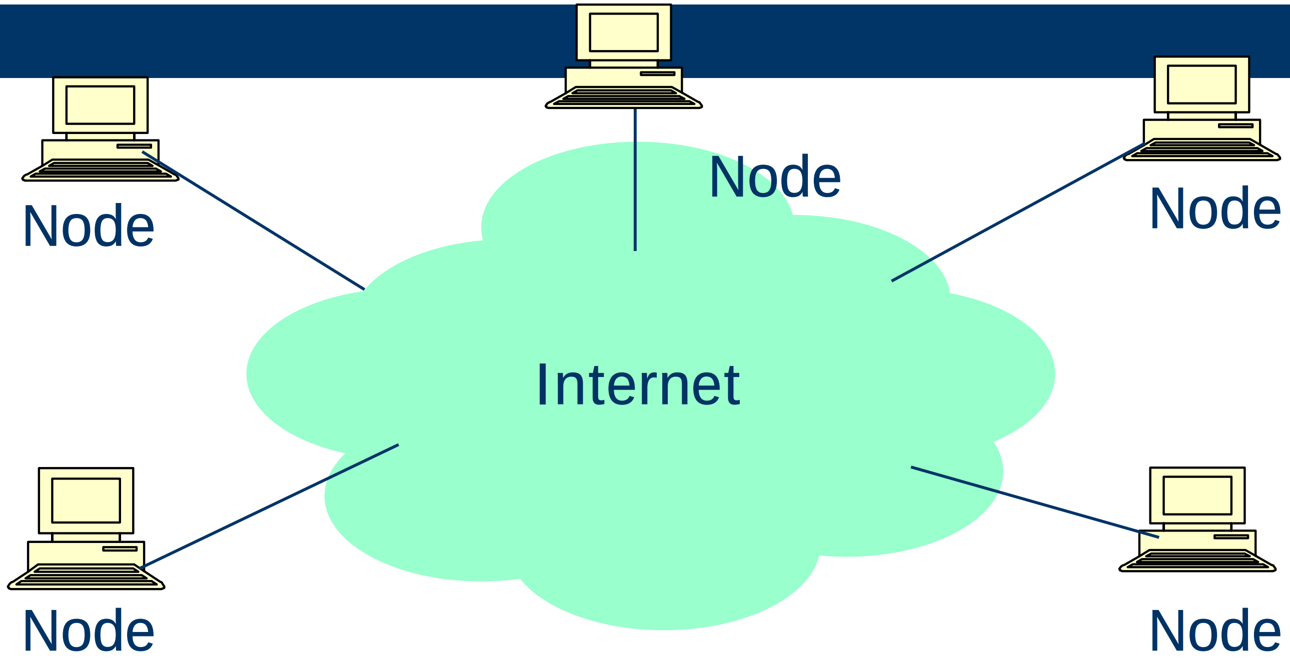
Tapestry Mesh (optional)



Talk Outline

- Motivation for OceanStore and Tapestry
- Tapestry overview and details (optional)
- ***Motivation for P2P system and DHT***
- Chord overview and details (optional)
- Ongoing work / Open problems

What is a P2P system?



- A distributed system architecture:
  - No centralized control
  - Nodes are symmetric in function
- Larger number of unreliable nodes
- Enabled by technology improvements

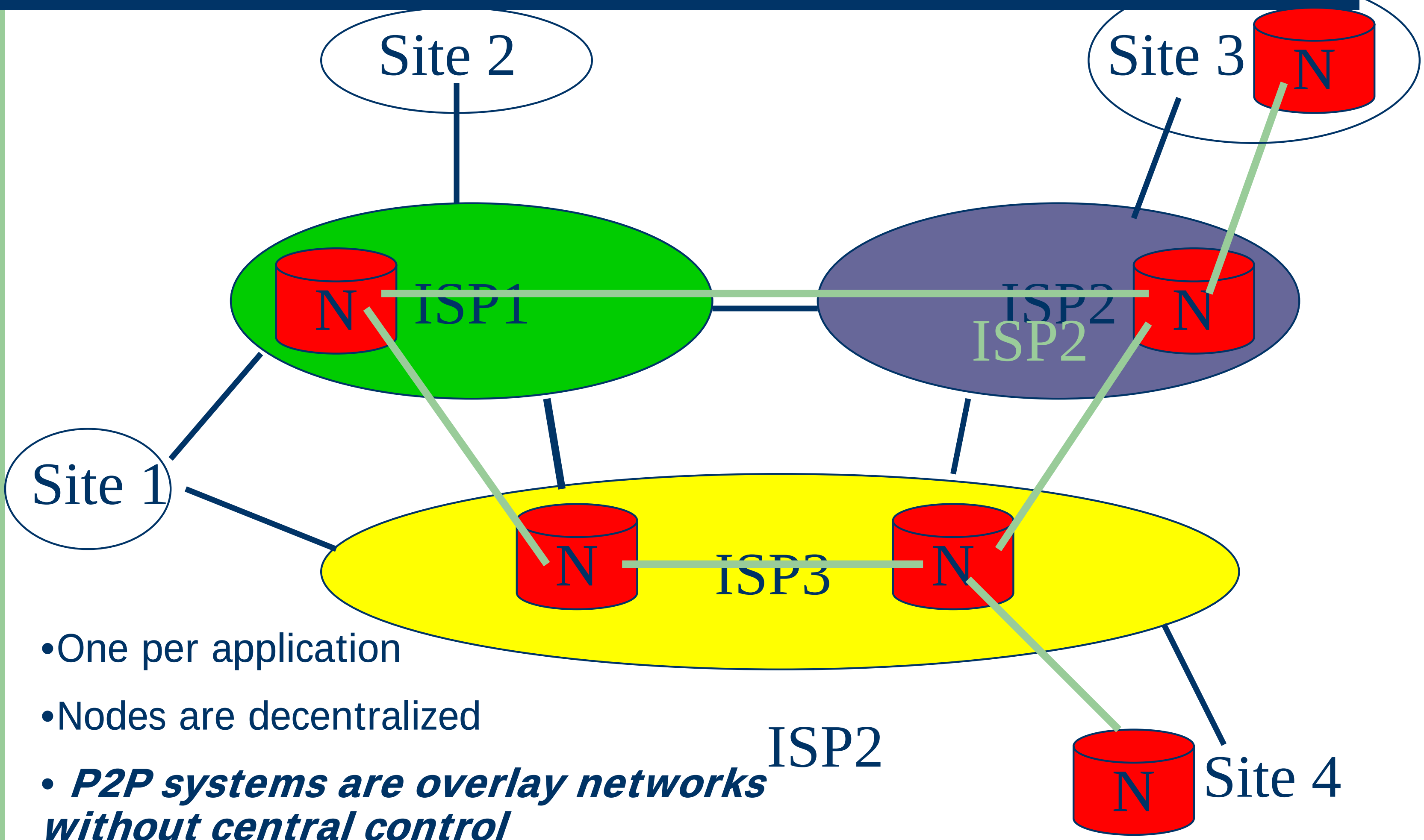
How did it start?

- Killer app: Napster – ***free*** music sharing over the Internet
  - Will this survive from the legal issues?
- Key idea: share the storage *and* bandwidth of individual (home) users
  - From Economic perspective: **merchandise exchange economy** -- willing to give because of willing to get.

The promise of P2P computing

- Reliability: no central point of failure
  - Many replicas
  - Geographic distribution
- High capacity through parallelism:
  - Many disks
  - Many network connections
  - Many CPUs
- Automatic configuration
- Useful in public and proprietary settings

No lower layer support from Internet: Application-level overlays



- One per application
- Nodes are decentralized
- ***P2P systems are overlay networks without central control***

Routing in P2P Systems:

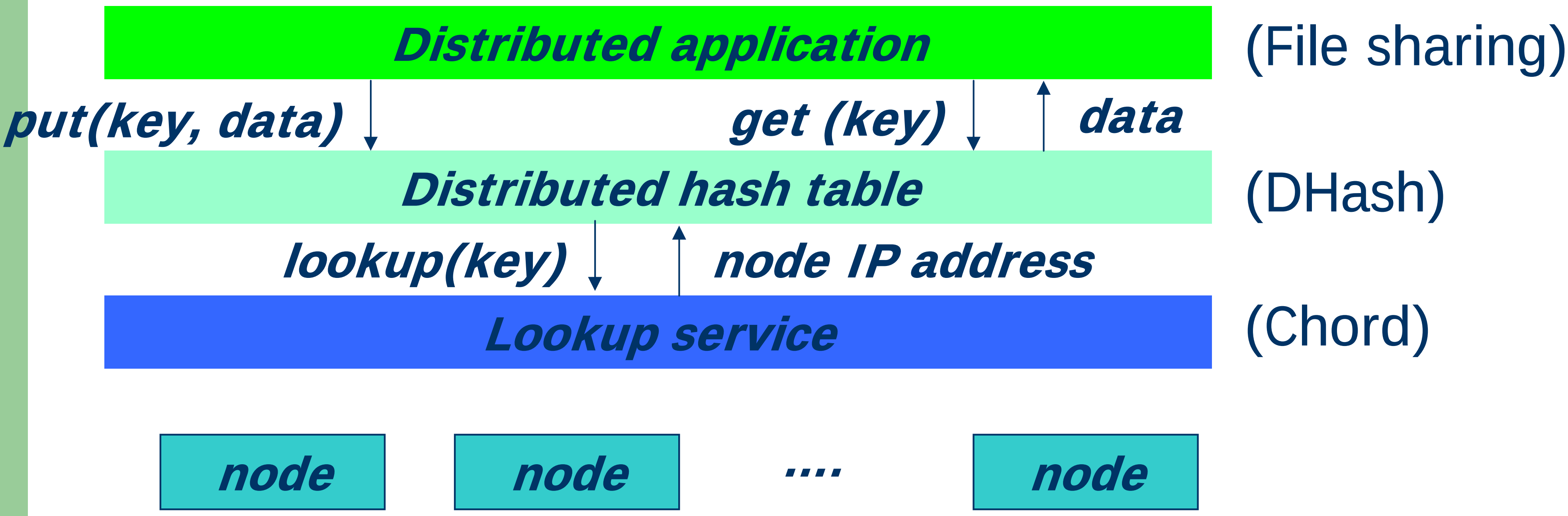
- Data centric routing instead of node centric
  - Need mapping from Data to its location in the network; then use direct application connection to the node
- All links refer to TCP/UDP connection from the applications

Evolution of routing in p2p

- Centralized server: Napster
- Flooding: Gnutella
- DHT based: Tapestry, Chord, CAN, …

Scheme	Gnutella	Tapestry	Chord	CAN
Neighbors	1(const)	Log N	Log N	d
Path length	Log N	Log N	Log N	$dN^{1/d}$
Message	N	Log N	Log N	$N^{1/d}$

Distributed hash table (DHT)



- Application may be distributed over many nodes
- DHT distributes data storage over many nodes

DHT interface

- Put(key, value) and get(key) → value
  - Simple interface!
- API supports a wide range of applications
  - DHT imposes no structure/meaning on keys
- Key/value pairs are persistent and global
  - Can store keys in other DHT values
  - And thus build complex data structures

A DHT makes a good *shared* infrastructure

- Many applications can share one DHT service
  - Much as applications share the Internet
- Eases deployment of new applications
- Pools resources from many participants
  - Efficient due to statistical multiplexing
  - Fault-tolerant due to geographic distribution

DHT implementation challenges

1. Scalable lookup
2. Balance load (flash crowds)
3. Handling failures
4. Coping with systems in flux
5. Network-awareness for performance
6. Robustness with untrusted participants
7. Programming abstraction
8. Heterogeneity
9. Anonymity
10. Indexing

Chord

Goal: simple, provably-good algorithms

Talk Outline

- Motivations for OceanStore and Tapestry
- Tapestry overview and details (optional)
- Motivations for P2P system and DHT
- ***Chord overview and details (optional)***
- Ongoing work / Open problems

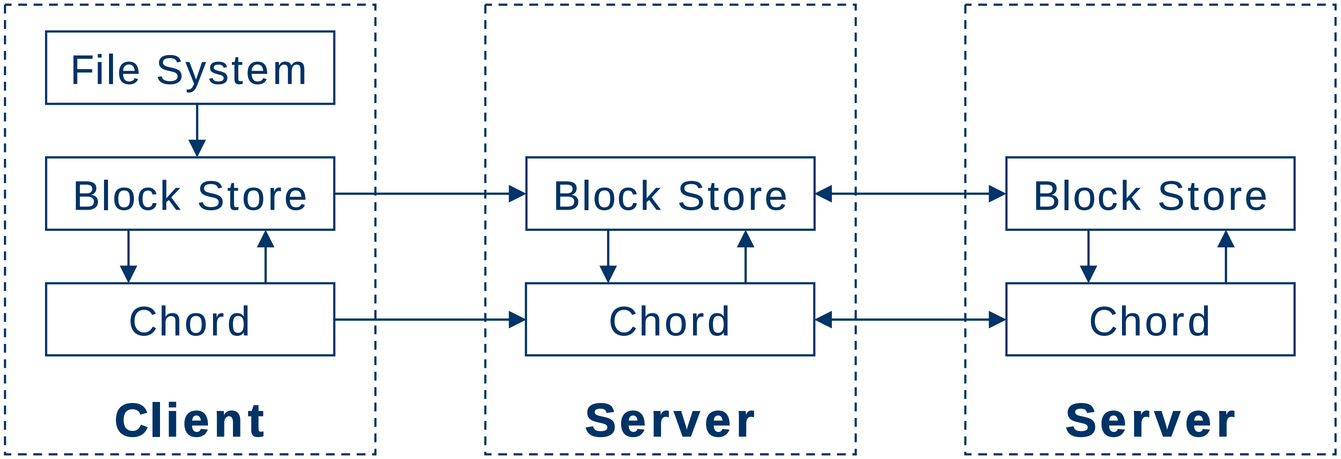
What is Chord? What does it do?

- In short: a peer-to-peer lookup system
- Given a key (data item), it maps the key onto a node (peer).
- Uses consistent hashing to assign keys to nodes .
- Solves problem of locating key in a collection of distributed nodes.
- Maintains routing information as nodes join and leave the system

Chord – addressed problems

- **Load balance**: distributed hash function, spreading keys evenly over nodes
- **Decentralization**: chord is fully distributed, no node more important than other, improves robustness
- **Scalability**: logarithmic growth of lookup costs with number of nodes in network, even very large systems are feasible
- **Availability**: chord automatically adjusts its internal tables to ensure that the node responsible for a key can always be found

Example Application



- Highest layer provides a file-like interface to user including user-friendly naming and authentication
- This file systems maps operations to lower-level block operations
- Block storage uses Chord to identify responsible node for storing a block and then talk to the block storage server on that node

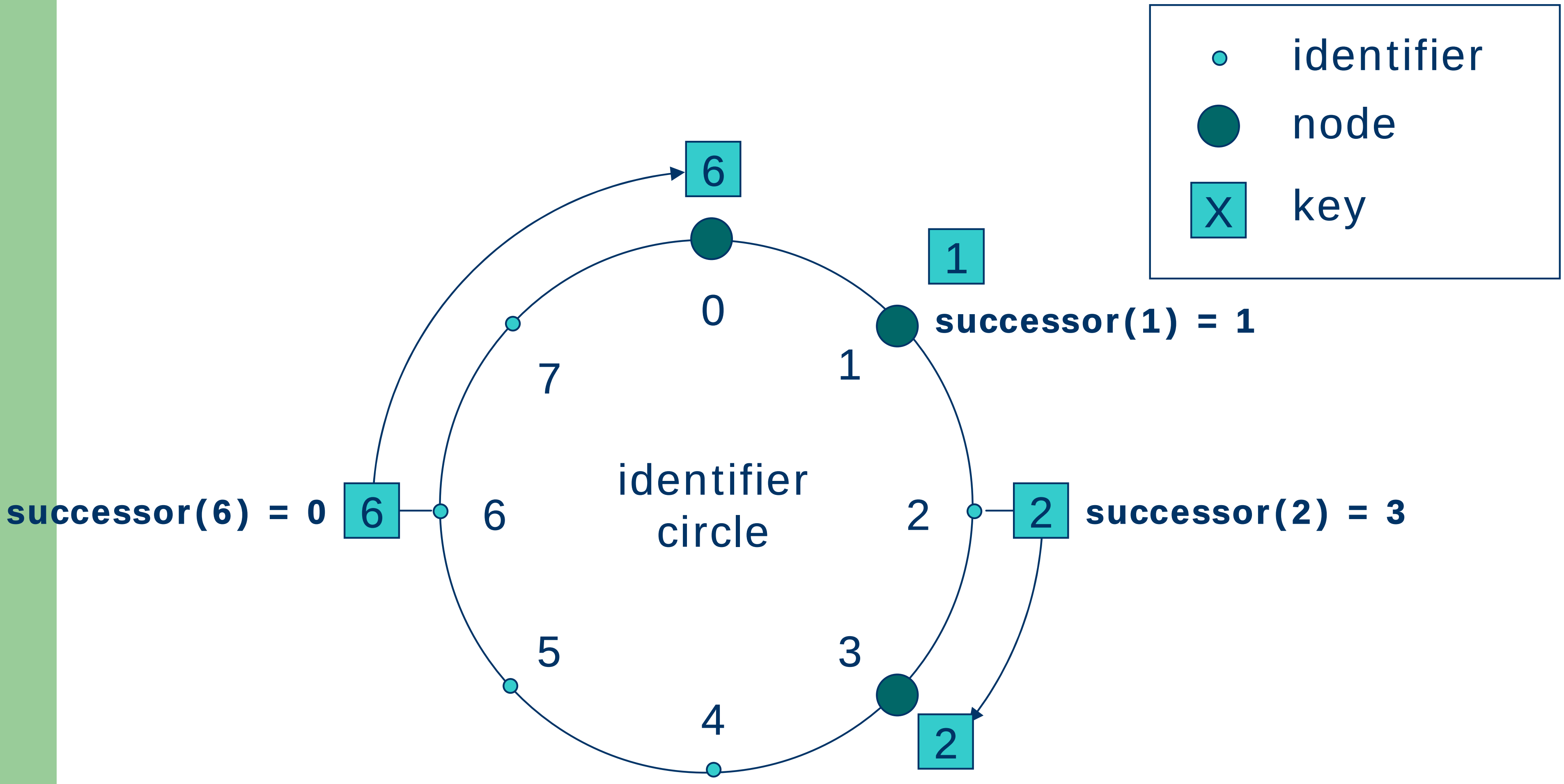
Chord details (optional)

- **Consistent hash** function assigns each node and key an m-bit *identifier.*
- SHA-1 is used as a base hash function.
- A node's identifier is defined by hashing the node's IP address.
- A key identifier is produced by hashing the key (chord doesn't define this. Depends on the application).
  - ID(node) = hash(IP, Port)
  - ID(key) = hash(key)

Chord details (optional)

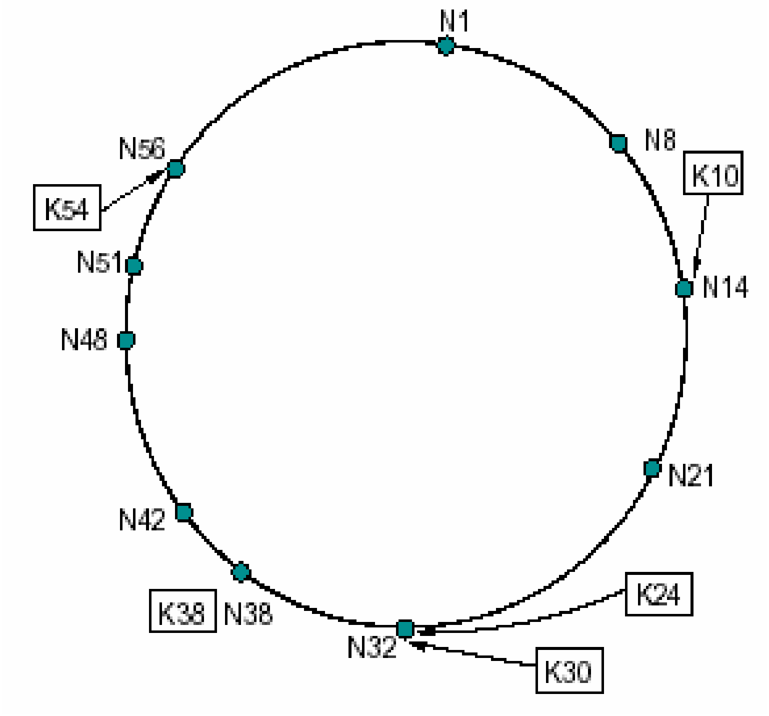
- In an m-bit identifier space, there are 2^m identifiers.
- Identifiers are ordered on an *identifier circle* modulo 2^m.
- The identifier ring is called *Chord ring*.
- Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space.
- This node is the successor node of key k, denoted by *successor(k)*.

Consistent Hashing :Successor Nodes (opt)



Consistent Hashing (opt)

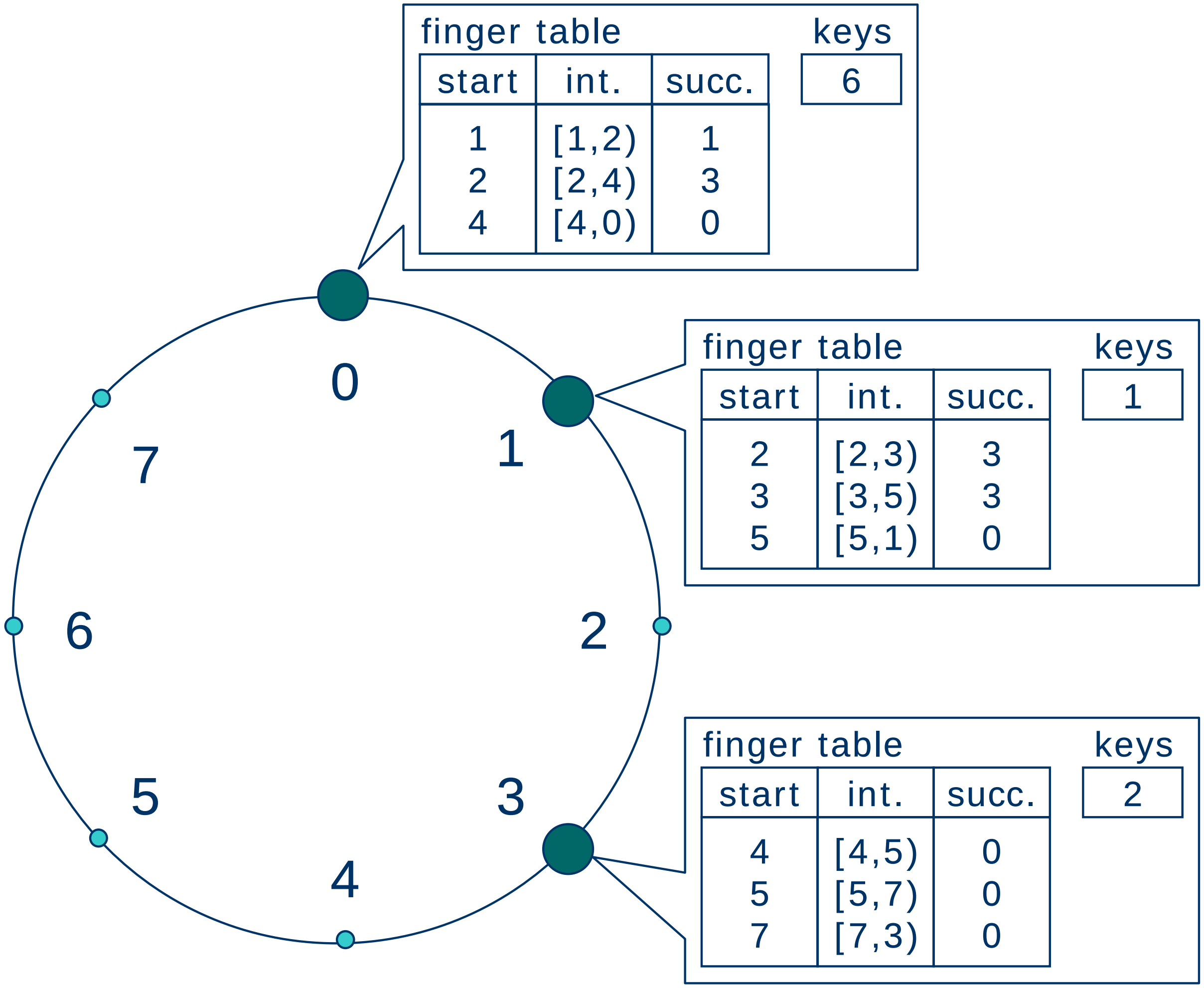
- For m = 6, # of identifiers is 64.
- The following Chord ring has 10 nodes and stores 5 keys.
- The successor of key 10 is node 14.



Acceleration of Lookups (optional)

- Lookups are accelerated by maintaining additional routing information
- Each node maintains a routing table with (at most) *m* entries (where $N=2^m$) called the **finger table**
- i^{th} entry in the table at node *n* contains the identity of the first node, *s*, that succeeds *n* by at least 2^{i-1} on the identifier circle (clarification on next slide)
- *s* = successor(n + 2^{i-1}) (all arithmetic mod 2)
- *s* is called the i^{th} *finger* of node *n*, denoted by *n.finger(i).node*

Finger Tables (1) (optional)



finger table

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	0

keys

6

finger table

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	0

keys

1

finger table

start	int.	succ.
4	[4,5)	0
5	[5,7)	0
7	[7,3)	0

keys

2

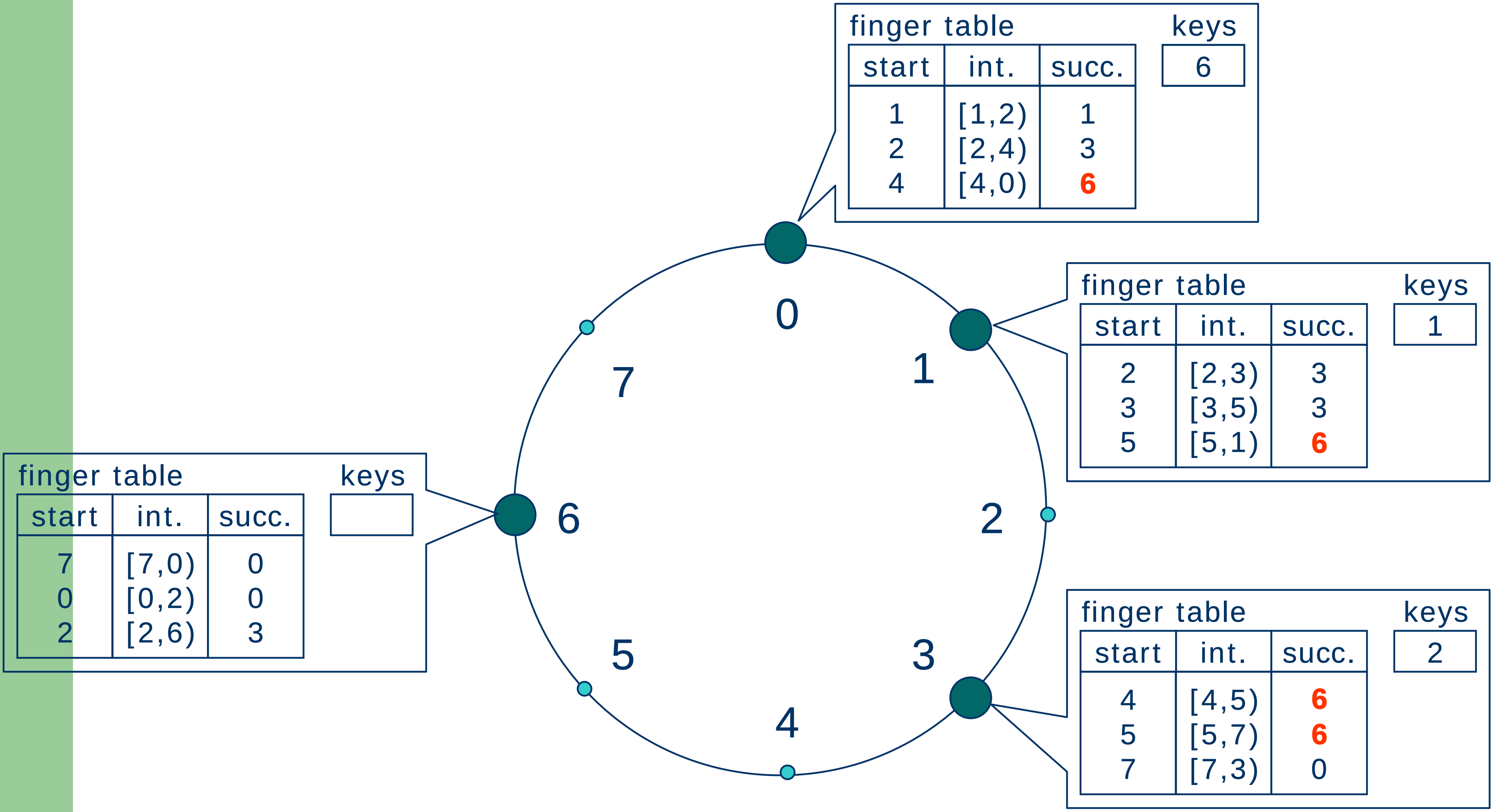
Finger Tables (2) - characteristics

- Each node stores information about only a small number of other nodes, and knows more about nodes *closely* following it than about nodes *farther away*

- A node’s finger table generally does not contain enough information to determine the successor of an arbitrary key *k*

- Repetitive queries to nodes that immediately precede the given key will lead to the key’s successor eventually

Node Joins – with Finger Tables



finger table

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	**6**

keys: 6

finger table

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	**6**

keys: 1

finger table

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

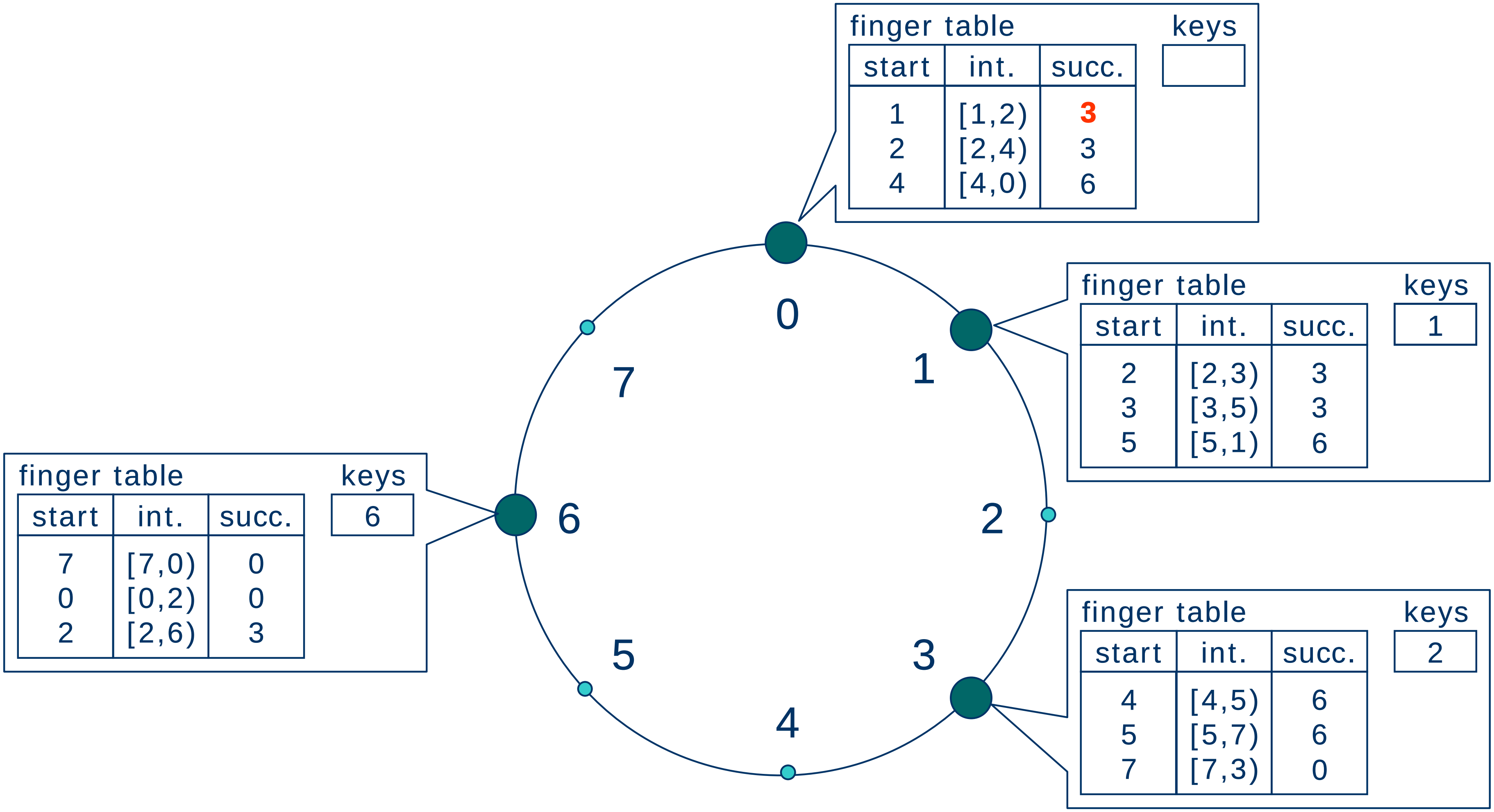
keys

finger table

start	int.	succ.
4	[4,5)	**6**
5	[5,7)	**6**
7	[7,3)	0

keys: 2

Node Departures – with Finger Tables



finger table

start	int.	succ.
1	[1,2)	**3**
2	[2,4)	3
4	[4,0)	6

keys

finger table

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	6

keys
1

finger table

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

keys
6

finger table

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

keys
2

Chord – The Math (optional)

- Every node is responsible for about *K/N keys* (N nodes, K keys)
- When a node joins or leaves an N-node network, only *O(K/N)* keys change hands (and only to and from joining or leaving node)
- Lookups need O(log N) messages
- To reestablish routing invariants and finger tables after node joining or leaving, only O($\log^2$N) messages are required

Talk Outline

- Motivations for OceanStore and Tapestry
- Tapestry overview and details (optional)
- Motivations for P2P system and DHT
- Chord overview and details (optional)
- ***Ongoing work / Open problems***

Many recent DHT-based projects

- File sharing [CFS, OceanStore, PAST, Ivy, …]
- Web cache [Squirrel, ..]
- Backup store [Pastiche]
- Censor-resistant stores [Eternity, FreeNet,..]
- DB query and indexing [Hellerstein, …]
- Event notification [Scribe]
- Naming systems [ChordDNS, Twine, ..]
- Communication primitives [I3, …]

Some open problems

- http://www.cs.rice.edu/Conferences/IPTPS02
- O(log n) path lengths with O(1) neighbors
- Trade off when combining with other properties
- Routing hop spots
- Incorporating geography (neighbor selection/proximity routing)
- Exploit the heterogeneity in p2p system

My 2 cents

- What can we really do with p2p system?
  - File Sharing (legal issues)
  - P2P service in education (http://chronicle.com/prm/daily/2004/01/2004012606n.htm)
  - Video streaming
  - Spam watch (Middleware2003)
- Security:
  - Possibility of attacks the p2p system.
  - Privacy.
- Thanks !

Outline of today

- Overview of a distributed storage system (Wesley)
- Routing in such system and DHT (Zheng)
- ***Distributed File System (Hong)***

Motivation

- Sharing of data in distributed systems
- Each user in a distributed system is potentially a creator as well as consumer of data
  - User may use/update information at a remote site
  - Physical movement of a user may require his data to be accessible elsewhere
- Goal: provide ease of data sharing in a secure, reliable, efficient, and usable manner that is independent of the size and complexity of the distributed system

Main Issues

- Data Consistency
  - A mechanism must be provided in order to ensure that each user can see changes that others are making to their copies of data
  - Lock is used as concurrency control to ensure consistency
  - Things become more complex when replication is implemented for high availability and data persistence, since different replica may be inconsistent because of server failure, etc

Main Issues (cont.)

- Location Transparency
  - The name of a file is devoid of location information. An explicit file location mechanism dynamically maps file names to storage sites
  - A uniform name space is provided to users
- Security
  - DFS must provide authentication and authorization (once users are authenticated, the system must ensure that the performed operations are permitted on the resources accessed)
  - Encryption becomes an indispensable building block

Main Issues (cont.)

- Availability
  - System should be available despite server crash or network partition
  - Replication, the basic technique used to achieve high availability, introduces complication of its own (how to propagate changes in a consistent and efficient manner?)
- Data Persistence
  - The loss or destruction of a device does not lead to lost data
  - Replication is also useful for this purpose

Main Issues (cont.)

- Performance
  - The network is considerably slower than the internal buses. Therefore, the less clients have to access servers, the more performance can be achieved
  - Caching can lower network load
  - Store hints information at client
    - A hint is a piece of information that can substantially improve performance if correct but has no semantically negative consequence if erroneous. (e.g. file location information)
  - Transferring data in bulk reduces protocol processing overhead

Case Study 1. NFS

- Sun Microsystems Network File System, first released by Sun in 1985
- The most used DFS on networks of workstations
- Design Consideration: portability and heterogeneity
  - Sun made a careful distinction between the NFS protocol, and a specific implementation of an NFS server or client (by other vendors)
  - NFS has been ported to almost all existing operating systems like MVS, MacOS, OS/2 and MS-DOS

NFS (cont.)

- Stateless Protocol
  - Server don’t store information about the state of client access to its files
  - Each RPC request from a client contains all the information needed to satisfy the request
  - Simplify crash recovery on servers
  - Sacrifice functionality and Unix compatibility: NFS doesn’t support locks and therefore doesn’t assure consistency

NFS (cont.)

- Naming and Location
  - NFS clients are usually configured so that each sees a Unix file name space with a private root
  - The name space on each client can be different. It’s the job of system administrator to determine how each client will view the directory structure
  - Location transparency is obtained by convention, rather than being a basic architectural feature of NFS
  - Name-to-site bindings are static.

NFS (cont.)

- Caching
  - NFS clients cache individual pages of remote files and directories in their main memory
  - When a client caches any block of a file, it also caches a timestamp indicating when the file was last modified on the server
  - A validation check is always performed when a file is opened and when the server is contacted to satisfy a cache miss. After a check, cached blocks are assumed valid for a finite interval of time
  - If a cached page is modified, it is marked as dirty ad scheduled to be flushed to the server. The actual flushing will occur after some delay. However, all dirty pages will be flushed to the server before a close operation on the file completes

NFS (cont.)

- Replication
  - As originally specified, NFS did not support data replication
  - More recent versions of NFS support replication via a mechanism called Automounter. (Automounter allows remote mount points to be specified using a set of servers rather than a single server. However, propagation of modifications to replicas has to be done manually)
  - This replication mechanism is intended primarily for READ-ONLY files (frequently read but rarely modified)

NFS (cont.)

- Security
  - NFS uses the underlying Unix file protection mechanism on servers for access checks
  - In the early versions of NFS, mutual trust was assumed among all participating machines. The identity of a user was determined by a client machine and accepted without further validation by a server
  - More recent versions of NFS use DES-based mutual authentication to provide a higher level of security. However, since file data in RPC packets is not encrypted, NFS is still vulnerable

Case Study 2. AFS

- Andrew File System, started in 1983 at CMU
- Design Consideration: scalability and security
  - Many design decisions in Andrew are influenced by its anticipated final size of 5000 to 10000 nodes
  - Scale renders security a serious concern, since it has to be enforced rather than left to the good will of the user community

AFS (cont.)

- Naming and Location
  - The file name space on an Andrew workstation is partitioned into a shared and a local name space
  - The shared name space is local transparent and is identical on all workstations. It is partitioned into disjoint sub trees, and each sub tree is assigned to a single server, called its custodian. Each server contains a copy of a fully replicated location database that maps files to custodians
  - The local name space is unique to each workstation and is relatively small. It only contains temporary files or files needed for workstation initialization

AFS (cont.)

- Caching
  - Files in the shared name space are cached on demand on the local disks of workstations. A cache manager, called Venus, runs on each workstation
  - When a file is opened, Venus checks the cache for the presence of a valid copy. Read and write operations on an open file are directed to the cached copy. No network traffic is generated by such requests. If a cached file is modified, it is copied back to the custodian when the file is closed
  - Cache consistency is maintained by the mechanism called callback. When a file is cached from a server, the latter makes a note of this fact and promises to inform the client if the file is updated by someone else

AFS (cont.)

- Replication
  - Replication of READ-ONLY data (frequently read but rarely modified)
  - Subtrees that contain such data may have read-only replicas at multiple servers. Propagation of changes to the read-only replicas is done by an explicit operational procedure

AFS (cont.)

- Concurrency Control
  - Provided by emulation of the Unix flock system call.
  - Lock and unlock operations on a file are performed directly to its custodian

AFS (cont.)

- Security
  - Servers are physically secure, are accessible only to trusted operators, and run only trusted system software. Neither the network nor workstations are trusted by servers
  - AFS uses Kerberos protocol for mutual authentication between client and server. Kerberos protocol is a two-step authentication scheme. When a user logs in to a workstation, his password is used to establish a communication channel to an authentication server. An authentication ticket is obtained from the authentication server and saved for future use

Case Study 3. CODA

- Coda File System, developed since 1987 at CMU
- A distributed file system with its origin in AFS2
- Design Consideration: availability
  - Coda's goal is to provide the highest degree of availability in the face of all realistic failures, without significant loss of usability, performance, or security

CODA (cont.)

- Server Replication
  - The unit of replication in Coda is volume. A volume is a collection of files that are stored on one server and form a partial subtree of the shared file name space
  - The set of servers that contain replicas of a volume is its volume storage group (VSG). For each volume from which it has cached data, Venus keeps track of the subset of the VSG that is currently accessible. This subset is reffered to as the accessible volume storage group (AVSG)

CODA (cont.)

- Server Replication (cont.)
  - The replication strategy is a variant of the read-one, write-all approach. When a file is closed after modification, it is transferred to all members of the AVSG
  - When servicing a cache miss, a client obtains data from one member of its AVSG called the preffered server. Although data is transferred only from one server, the other servers are contacted to verify that the preferred server does indeed have the latest copy of data. If not, the member of the AVSG with the latest copy is made the preferred site and the AVSG is notified that some of its members have stale replicas

CODA (cont.)

- Disconnected Operation
  - Disconnected operation offers possibility of accessing distributed file system files without being connected to the network at all
  - Disconnected operation begins when no member of a VSG is accessible. But it only provides access to data that was cached at the client at the start of disconnected operation. When disconnected operation ends, modified files and directories are propagated to the AVSG. Should conflicts occur, CODA provides some tools for the user to decide which update must prevail

CODA (cont.)

- Disconnected Operation (cont.)
  - Coda allows a user to specify a prioritized list of files and directories that Venus should strive to retain in the cache. Once each 10 minutes, a process is initiated in order to bring to the local disk all files with larger priorities

NFS vs. AFS vs. CODA

	NFS	AFS	CODA
Client Cache Location	Main memory	Local disk	Local disk
Replication	POOR. Just for read-only directories	POOR. Just for read-only directories	GOOD. Overhead is distributed among clients
Consistency	POOR. Concurrent access generates unpredictable results	FAIR. Session semantics	POOR. Session semantics weakened by server replication
Scalability	POOR. Server saturate rapidly	EXCELLENT. Ideal for wide area networks with low degree of file sharing	EXCELLENT. Ideal for wide area networks with low degree of file sharing
Performance	POOR. Inefficient protocol	FAIR. Large latency on non-cached files, though	GOOD. Looks for the “closest” replica
Data Persistence	POOR. Delayed writes may cause loss of data	FAIR. Automatic backup tools	GOOD
Availability	POOR	POOR	EXCELLENT. Server replication. Disconnected operations
Security	POOR. Server trust on clients	GOOD. Access control lists. Kerberos authentication between client and server	GOOD. Access control lists. Kerberos authentication between client and server

Case Study 4. GFS

- Google File System, developed at Google
- A scalable distributed file system for large distributed data-intensive applications
- GFS provides fault tolerance while running on inexpensive commodity hardware, and delivers high aggregate performance to a large number of clients.

GFS (cont.)

- GFS vs. Traditional FS
  - component failures are the norm rather than the exception
  - files are huge by traditional standards
  - most files are mutated by appending new data rather than overwriting existing data
  - co-designing the applications and the file system API

GFS (cont.)

- Architecture

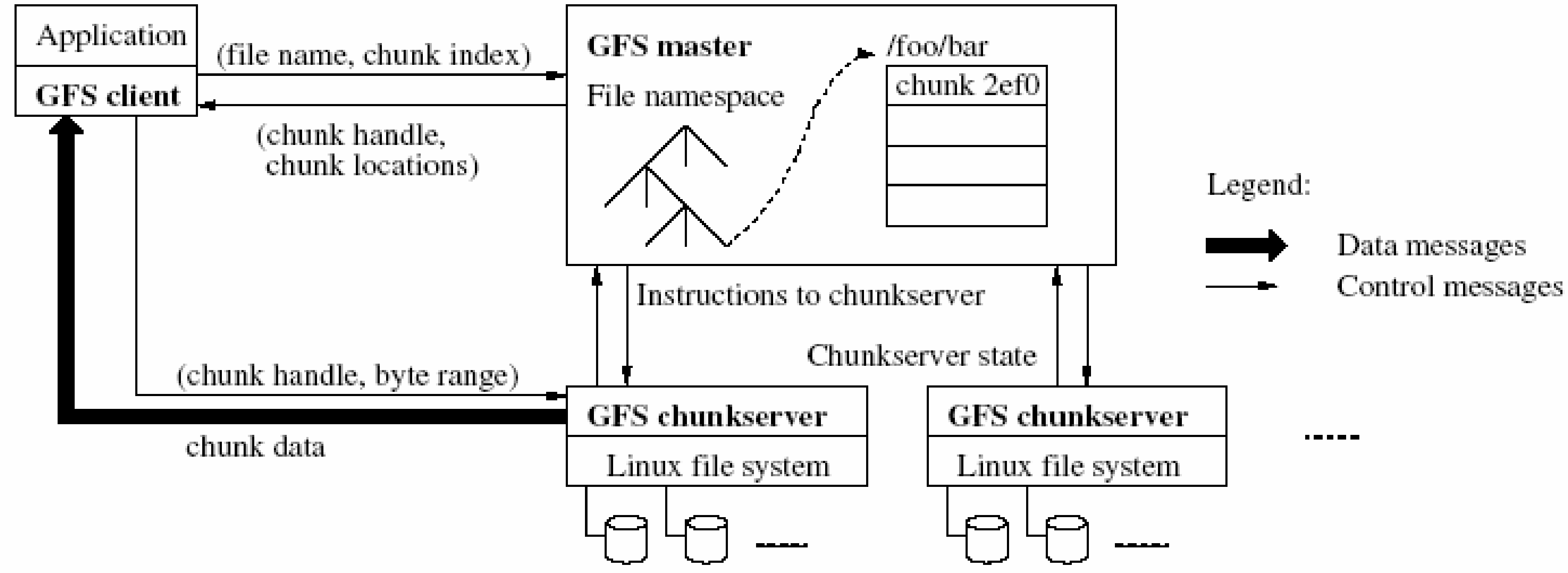


Figure 1: GFS Architecture

GFS (cont.)

- Clients cache metadata but don’t cache file data
- The systems maintains a number of replicas for each chunk to ensure data persistence
- Master controls concurrent access to files and directories
- GFS doesn’t scale. Its single master is a bottleneck

GFS (cont.)

- High performance achieved by very specific design and optimization aiming at Google’s environment
- Fast recovery of master as well as master replication ensures high availability
  - Logs are used in recovery of master
- GFS is a successful system. But it brings few new concepts in DFS design and implementation. Its lack of generality determines that it cannot have wide application

Open Problems

- High availability
  - CODA's goal is to provide highest degree of availability without significant loss of performance. However, it sacrifices consistency
  - Consistency, availability and performance seem to be mutually contradictory in a distributed system. Is there a way to achieve high availability without loss of consistency and performance?

Open Problems (cont.)

- Scalability
  - AFS-like systems take scalability as a dominant design consideration. Such systems give users in different continents the possibility of sharing files
  - With rapid growth of Internet, we need global scale distributed file system with infinite scalability

Open Problems (cont.)

- Heterogeneity
  - It's desirable that users running different operating system could share data through a distributed file systems
  - Ubiquitous computing places requirement on heterogeneity
  - Coping with heterogeneity is inherently difficult because of the presence of multiple computational environments, each with its own notion of file naming and functionality

Open Problems (cont.)

- Multimedia Support
  - Multimedia applications deal with huge amounts of information which can currently get to terabytes of data and transfer rates of hundreds of megabytes per second
  - We need distributed file systems with high I/O bandwidth and fast response

Open Problems (cont.)

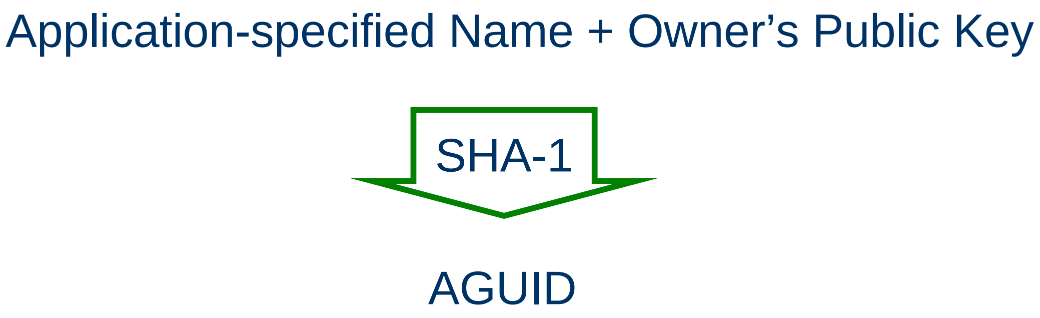
- Security
  - Security may turn out to be the bane of global scale distributed systems
  - we need to take extra measures to make sure that information is protected from prying eyes and malicious hands

Thank you! Questions?

Backup Slides

Data Model

- Data Object
  - A File in a Traditional File System
  - Named by an *Active Globally-Unique Identifier, AGUID*
    - Location Independent
    - Preventing Name Space Collisions



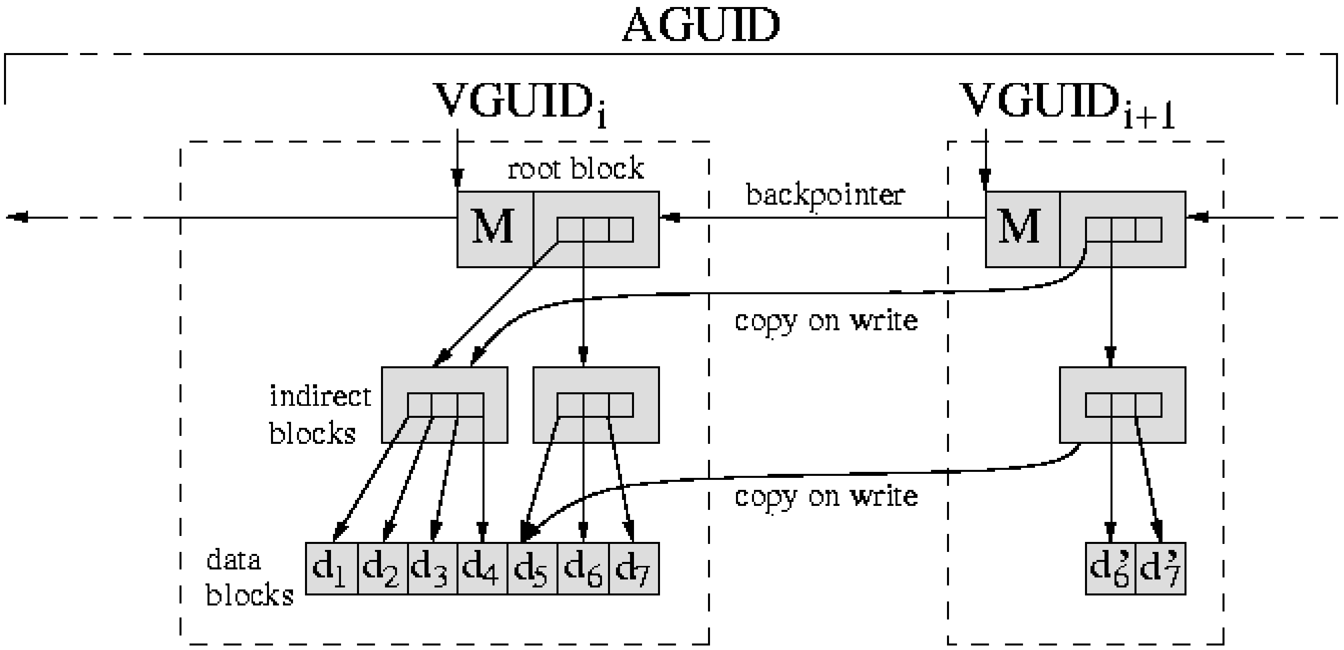
Application-specified Name + Owner's Public Key

SHA-1

AGUID

Data Model

- Data Object
  - Sequences of Read-only Versions
  - Block Reference

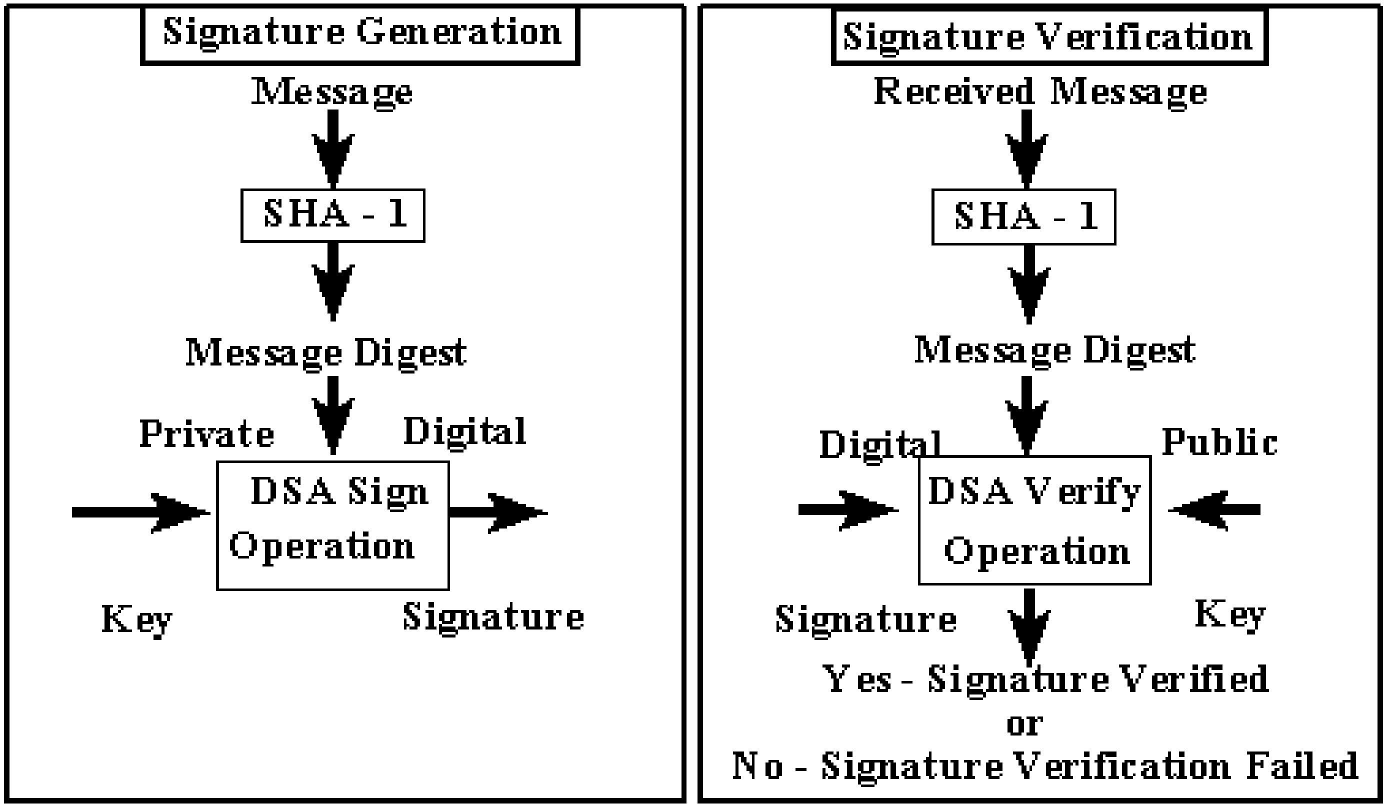


SHA-1 (http://www.itl.nist.gov/fipspubs/fip180-1.htm)

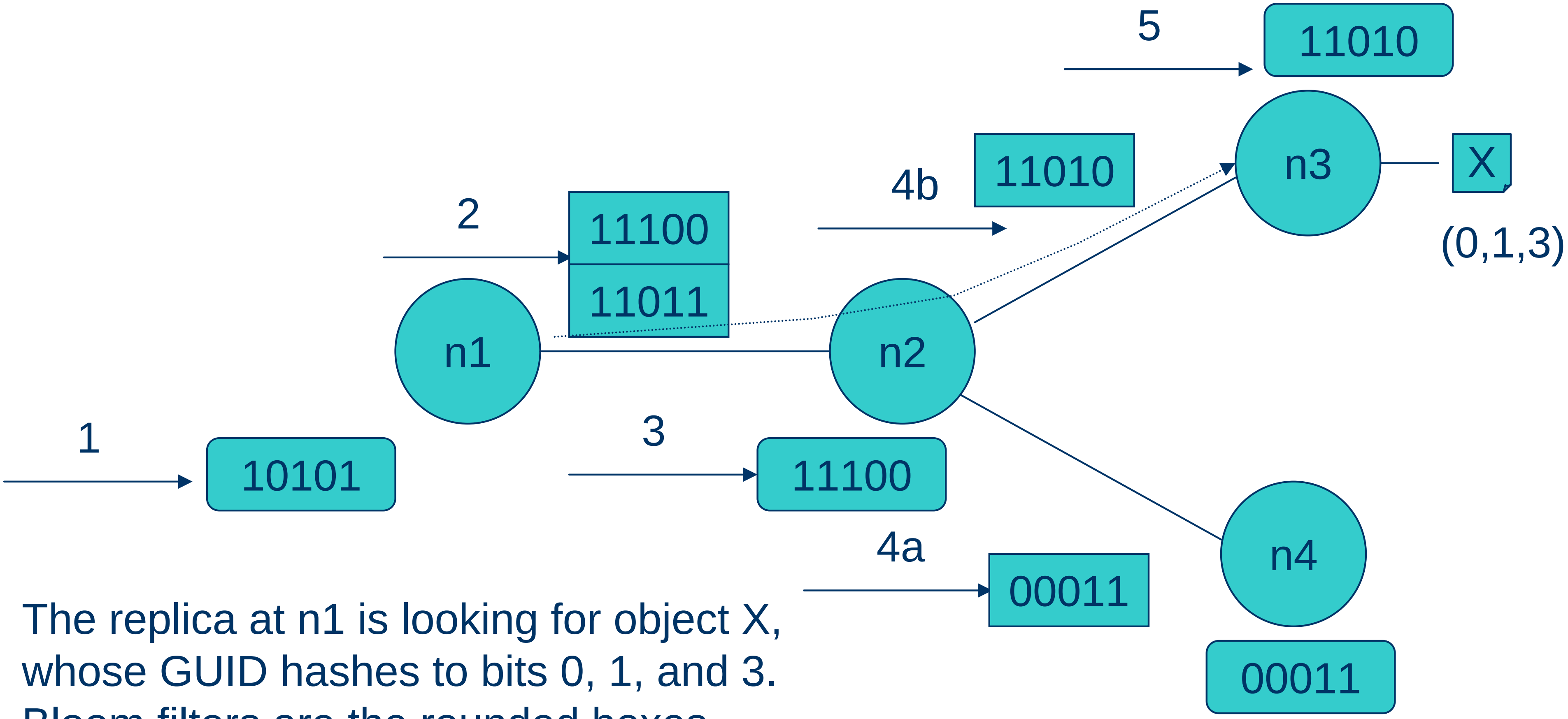
- Secure Hash Algorithm, SHA-1, for computing a condensed representation of a message or a data file. When a message of any length < 264 bits is input, the SHA-1 produces a 160-bit output called a message digest. The message digest can then be input to the Digital Signature Algorithm (DSA) which generates or verifies the signature for the message. Signing the message digest rather than the message often improves the efficiency of the process because the message digest is usually much smaller in size than the message. The same hash algorithm must be used by the verifier of a digital signature as was used by the creator of the digital signature.

  The SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify. SHA-1 is a technical revision of SHA (FIPS 180). A circular left shift operation has been added to the specifications in section 7, line b, page 9 of FIPS 180 and its equivalent in section 8, line c, page 10 of FIPS 180. This revision improves the security provided by this standard. The SHA-1 is based on principles similar to those used by Professor Ronald L. Rivest of MIT when designing the MD4 message digest algorithm ("The MD4 Message Digest Algorithm," Advances in Cryptology - CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 303-311), and is closely modelled after that algorithm.

SHA-1 (http://www.itl.nist.gov/fipspubs/fip180-1.htm)



The probabilistic query process



The replica at n1 is looking for object X, whose GUID hashes to bits 0, 1, and 3. Bloom filters are the rounded boxes where as square boxes are neighbor filters.

Byzantine Agreement

- **Byzantium, 1453 AD.** The city of Constantinople, the last remnants of the hoary Roman Empire, is under siege. Powerful Ottoman battalions are camped around the city on both sides of the Bosporus, poised to launch the next, perhaps final, attack. Sitting in their respective camps, the generals are meditating. Because of the redoubtable fortifications, no battalion by itself can succeed; the attack must be carried out by several of them together or otherwise they would be thrusted back and incur heavy losses that would infuriate the Grand Sultan. Worse, that would jeopardize the prospects of a defeated general to become Vizier. The generals can agree on a common plan of action by communicating thanks to the messenger service of the Ottoman Army which can deliver messages within an hour, certifying the identity of the sender and preserving the content of the message. Some of the generals however, are secretly conspiring against the others. Their aim is to confuse their peers so that an insufficient number of generals is deceived into attacking. The resulting defeat will enhance their own status in the eyes of the Grand Sultan. The generals start shuffling messages around, the ones trying to agree on a time to launch the offensive, the others trying to split their ranks...

- **Menlo Park, 1982 AD.** The situation above describes a classical coordination problem in distributed computing known as *byzantine agreement* which was introduced in two seminal papers by Lamport, Pease and Shostak [23,30]. Broadly stated, a basic problem in distributed computing is this: Can a set of concurrent processes achieve coordination in spite of the faulty behaviour of some of them? The faults to be tolerated can be of various kinds. The most stringent requirement for a fault-tolerant protocol is to be resilient to so-called byzantine failures: a faulty process can behave in any arbitrary way, even conspire together with other faulty processes in an attempt to make the protocol work incorrectly. The identity of faulty processes is unknown, reflecting the fact that faults can (and do) happen unpredictably.

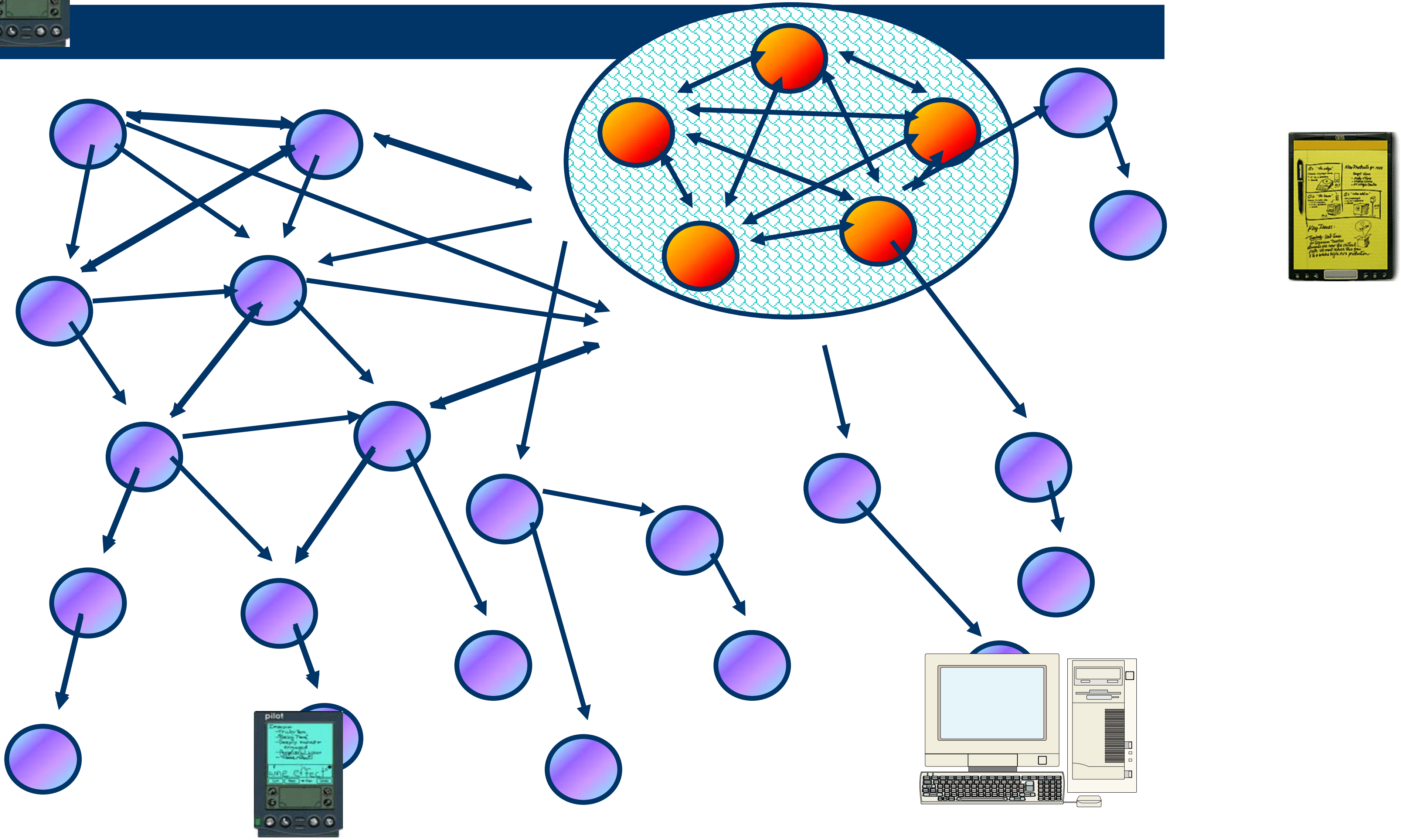
SEDA

- SEDA is an acronym for *staged event-driven architecture*, and decomposes a complex, event-driven application into a set of *stages* connected by *queues*. This design avoids the high overhead associated with thread-based concurrency models, and decouples event and thread scheduling from application logic. By performing admission control on each event queue, the service can be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. SEDA employs dynamic control to automatically tune runtime parameters (such as the scheduling parameters of each stage), as well as to manage load, for example, by performing adaptive load shedding. Decomposing services into a set of stages also enables modularity and code reuse, as well as the development of debugging tools for complex event-driven applications.

Other distributed file systems

- Freenet – storage system designed to achieve anonymity in terms of publisher and the consumer of content – document driven. Does NOT provide permanent file storage, load balancing, is not scalable
- Free Haven – decentralized, trade offs time, bandwidth, latency, to get better anonymity and robustness, no dynamic management of underlying tree structure. Focus is on persistence, lacks efficiency, but also does not guarantee long-term survivability.
- Publius – mainly focuses on availability and anonymity, distributes files as shares over n web servers. J of these shares are enough to reconstruct a file. It lacks accountability, DoS, garbage clean-up, smooth join/leave for servers.
- Mojo Nation – centralized file storage system. Uses a Central Service Broker. Breaks up files into chunks and distributes these chunks among different computers in the network. Main goals are increased band-width and load balancing. There is no long-term durability of data. Swarm distribution – is the parallel download of file fragments, reconstructed on the client. Mojos are like credits, the more your contribute, storage, network, the more you can get!
- Farsite – Logically, a single hierarchical file system is visible from all access points, but underneath files are replicated and distributed among the client machines. There is NO responsible party, thus it is possible for loss of data due to an untrusted entity.

Path of Update



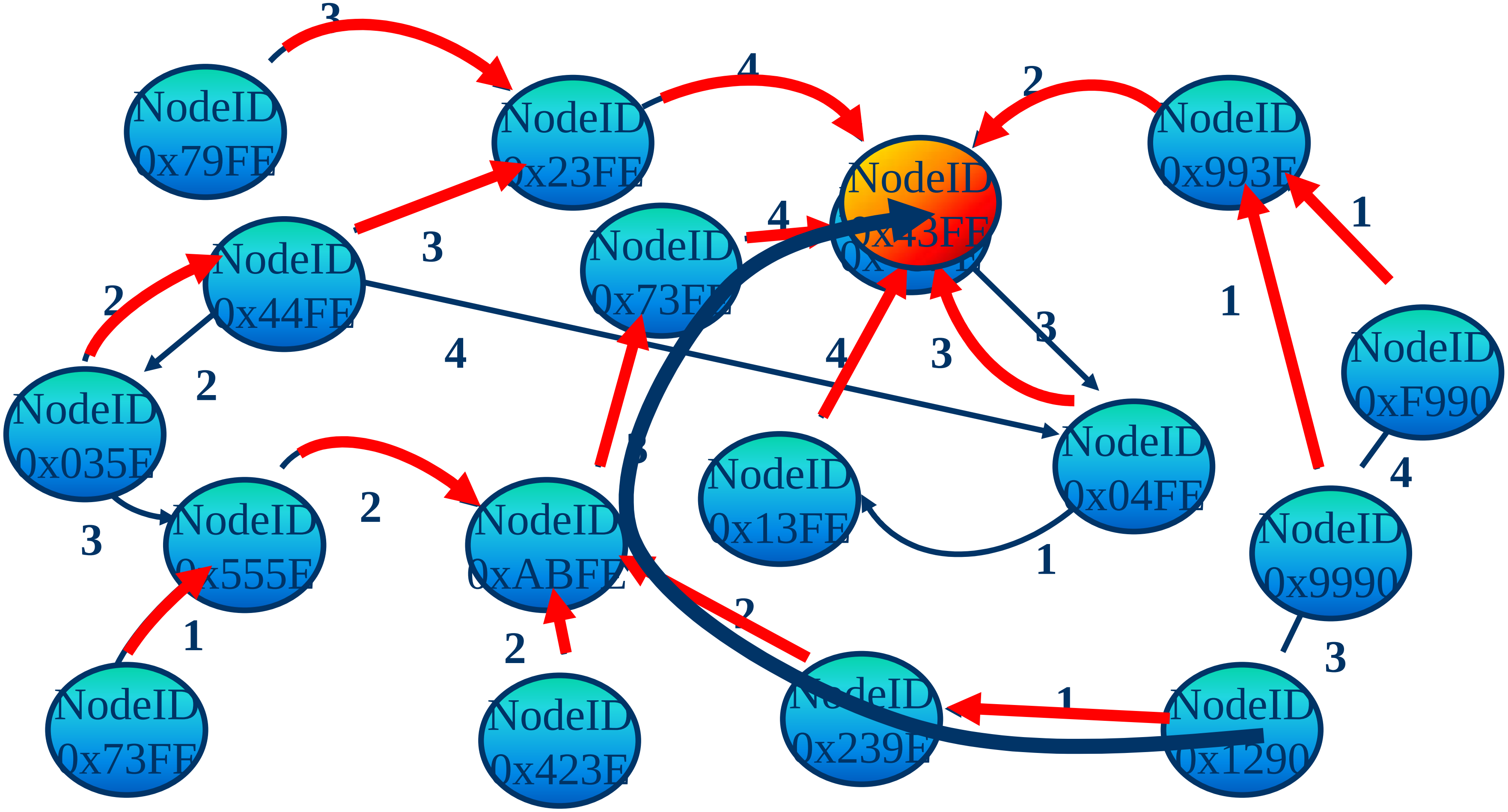
Types of data (coding) models

- Two distinct forms of data: active and archival
- *Active Data* in Floating Replicas
  - Per object virtual server
  - Logging for updates/conflict resolution
  - Interaction with other replicas to keep data consistent
  - May appear and disappear like bubbles
- *Archival Data* in Erasure-Coded Fragments
  - M-of-n coding: Like hologram
    - Data coded into *n* fragments, any *m* of which are sufficient to reconstruct (e.g m=16, n=64)
    - Coding overhead is proportional to n÷m (e.g 4)
    - *Law of large numbers advantage to fragmentation*
  - Fragments are self-verifying
  - <u>OceanStore equivalent of stable store</u>

Two levels of routing

- Fast probabilistic searching for routing cache
  - Task of routing a particular message is handled by the aggregate resources of many different nodes. By exploiting multiple routing paths to the destination, this serves to limit the power of nodes to deny service to a client, second, message route directly to their destination avoiding the multiple round-trips that a separate data location and routing process wound incur, finally the underlying infrastructure has more up-to-date information about the current location of entities than the clients.
  - Attenuated bloom filters
- Plaxton Mesh used if above fails
  - Underlying routing structure
  - Continuous adaptation
    - Network behavior
    - DoS attacks
    - Faulty servers

Basic Plaxton Mesh – an incremental suffix based routing



Plaxton Mesh use

- Tapestry (more on this later!)
- OceanStore enhancements for reliability:
  - Documents have multiple roots
  - Each node has multiple neighbor links
  - Searches proceed along multiple paths
    - Tradeoff between reliability and bandwidth?
  - Routing-level validation of query results
- Highly redundant and fault-tolerant structure that spreads data location load evenly while finding local objects quickly

Automatic Maintenance

- Byzantine Commitment for inner ring:
  - Can tolerate up to 1/3 faulty servers in inner ring
    - Bad servers can be arbitrarily bad
    - Cost ~n^2 communication
  - Continuous refresh of set of inner-ring servers

Information stored in OceanStore

- Where is persistent information stored?
- How is it protected?
- Does it last forever?
- How is it managed?
- Who owns the storage?

Applications

- OceanStore solves problems of consistency, security, privacy, wide-scale data dissemination, dynamic optimization, durable storage, and disconnected operation; this allows application developers to focus on higher-level concerns.
- (with that in mind) what are some possible uses: groupware, personal information management tools, calendars, email, contact lists, and distributed design tools. Nomadic email a user's email to migrate closer to his client, reducing the round trip to fetch messages from a remote server
- Can be used to generate very large digital libraries and repositories for scientific data, also new stream applications such as sensor data aggregations and dissemination

Pond ~ what is missing?

- Full Byzantine-fault-tolerant agreement
- Tentative update sharing
- Inner ring membership rotation
- Flexible ACL support
- Proactive replica placement