HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Peer-to-Peer and Grid Computing

Chapter 4: Peer-to-Peer Storage

Chapter Outline

- Using DHTs to build more complex systems
    - How DHT can help?
    - What problems DHTs solve?
    - What problems are left unsolved?
- P2P storage basics, with examples
    - Splitting into blocks (CFS)
    - Wide-scale replication (OceanStore)
    - Modifiable filesystem with logs (Ivy)
- Future of P2P filesystems

How to Use a DHT?

- Recall: DHT maps keys to values
- Applications based on DHTs must need this functionality
  - Or: Must be designed in this way!
  - Possible to design an application in several ways
- Keys and values are application specific
  - For filesystem: Value = file
  - For email: Value = email message
  - For distributed DB: Value = contents of entry, etc.
- Application stores values in DHT and uses them
  - Simple, but a powerful tool

Problems Solved by DHT

- DHT solves the problem of mapping keys to values in the distributed hash table
- Efficient storage and retrieval of values
- Efficient routing
  - Robust against many failures
  - Efficient in terms of network usage

- Provides hash table-like abstraction to application

Problems NOT Solved by DHT

- Everything else except what is on previous slide…
- In particular, the following problems
- Robustness
  - No guarantees against big failures
  - Threat models for DHTs not well-understood yet
- Availability
  - Data not guaranteed to be available
  - Only probabilistic guarantees (but possible to get high prob.)
- Consistency
  - No support for consistency
  - Data in DHT often highly replicated, consistency is a problem
- Version management
  - No support for version management
  - Might be possible to support this to some degree

P2P FS: Introduction

- P2P filesystems (FS) or P2P storage systems were the first applications of DHTs
- Fundamental principle:

  Key = filename, Value = file contents
- Different kinds of systems
  - Storage for read-only objects
  - Read-write support
  - Stand-alone storage systems
  - Systems with links to standard filesystems (e.g., NFS)

P2P FS: Current State

- Only examples of P2P filesystems come from research
- Research prototypes exist for many systems
- No wide-area deployment
  - Experiments on research testbeds
  - No examples of real deployment and real usage in wide-area
- After initial work, no recent advances?
  - At least, not visible advances

- Three examples:
  - Cooperative File System, CFS
  - OceanStore
  - Ivy

P2P FS: Why?

- Why build P2P filesystems?
- Light-weight, reliable, wide-area storage
  - At least in principle…

- Distributed filesystems not widely deployed either…
  - Were studied already long time ago

- Gain experience with DHT and how DHTs could be used in real applications
  - DHT abstraction is powerful, but it has limitations
  - Understanding of the limitations is valuable

P2P FS: Basic Techniques

- Three fundamental basic techniques for building distributed storage systems

1. Splitting files into blocks
2. Replicating files (or blocks!)
3. Using logs to allow modifications

- For now: Simple analysis of advantages and disadvantages and three examples
- Detailed performance analysis in Chapter 5
  - For blocks and replication

Splitting Files into Blocks

- Why: Files are of different sizes and peers storing large files have to serve more data

Pro:

- Dividing files into equal-sized blocks and storing blocks on different peers can achieve load balance
- If different files share blocks, we can save on storage

Con:

- Instead of needing one peer online, all peers with all blocks must be online (see below)
- Need metadata about blocks to be stored somewhere
- Granularity tradeoff: Small blocks -> Good load balance, but lots of overhead and vice versa

Replication

- Why: If file (or block) is stored only on one peer and that peer is offline, data is not available
- Replicating content to multiple peers significantly increases content availability

Pro:

- High availability and reliability
  - But only probabilistic guarantees

Con:

- How to coordinate lots of replicas?
  - Especially important if content can change
- Unreliable network requires high degree of replication for decent availability
  - “Wastes” storage space

Logs

- Why: If we want to change the stored files, we need to modify every stored replica
- Keep a log for every file (user, …) which gives information about the latest version

Pro:

- Changes concentrated in one place
- Anyone can figure out what is the latest version

Con:

- How to keep the log available?
  - By replicating it? ;-)

P2P FS: Overview

- Three examples of P2P filesystems
- CFS (blocks and replication)
  - Basic, read-only system
  - Based on Chord
- OceanStore (replication)
  - Vision for a global storage system
  - Based on Tapestry
- Ivy (logs)
  - Read-write, provide NFS semantics
  - Based on Chord

CFS

- CFS = Cooperative File System
- Developed at MIT, by same people as Chord

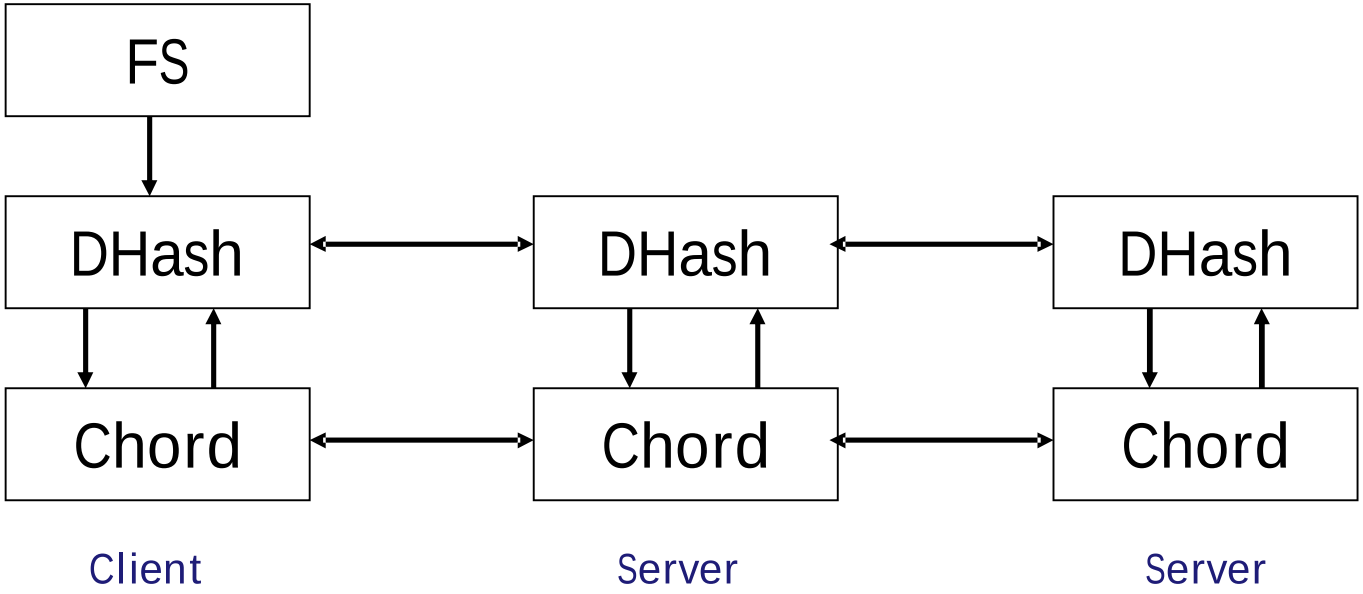
- CFS based on the Chord DHT
- Read-only system, only 1 publisher

- CFS stores blocks instead of whole files
  - Part of CFS is a generic block storage layer
- Features:
  - Load balancing
  - Performance equal to FTP
  - Efficiency and fast recovery from failures

CFS Properties

- Decentralized control
- Scalability (comes from Chord)
- Availability
  - In absence of catastrophic failures, of course…
- Load balance
  - Load balanced according to peers’ capabilities
- Persistence
  - Data will be stored for as long as agreed
- Quotas
  - Possibility of per-user quotas
- Efficiency
  - As fast as common FTP in wide area

CFS: Layers, Clients, and Servers



- FS: provide filesystem API, interpret blocks as files
- DHash: Provides block storage
- Chord: DHT layer, slightly modified
- Clients access files, servers just provide storage

Chord Layer in CFS

- Chord layer is slightly modified from basic Chord
- Instead of 1 successor, each node keeps track of *r* successors
- Finger tables as before

- Also, try to reduce lookup latency
  - Nodes measure latencies to other nodes
  - Report measured latencies to other nodes

DHash Layer

- DHash stores blocks as opposed to whole files
  - Better load balancing
  - More network query traffic, but not really significant
- Each block replicated *k* times, with $r \geq k$
- Two kinds of blocks:
  - Content block, addressed by hash of contents
  - Signed blocks (= root blocks), addressed by public key
    - Signed block is the root of the filesystem
    - One filesystem per publisher
- Blocks are cached in network
  - Most of caching near the responsible node
  - Blocks in local cache replaced according to least-recently-used
  - Consistency not a problem, blocks addressed by content
    - Root blocks different, may get old (but consistent) data

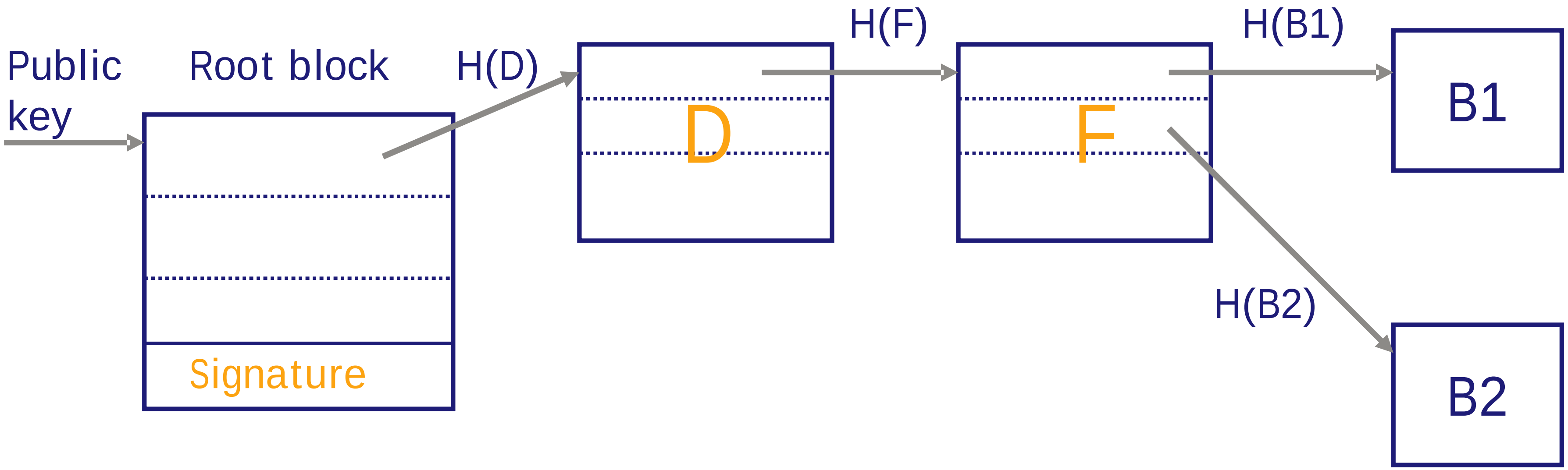
DHash API

- DHash provides following API to clients:

Put_h(block)	Store block
Put_s(block, pubkey)	Store or update signed block
Get(key)	Fetch block associated with key

- Filesystem starts with root (= signed) block
  - Block under publisher's public key and signed with private key
  - For clients, read-only, publisher can modify filesystem by inserting a new root block
  - Root block has pointers to other blocks
    - Either piece of a file or filesystem metadata (e.g., directory)
- Data stored for an agreed-upon finite interval

CFS Filesystem Structure



- Root block identified by publisher’s public key
  - Each publisher has its own filesystem
  - Different publishers are on separate filesystems
- Other blocks identified based on hash of contents
- Other blocks can be metadata or pieces of file

Load Balancing and Quotas

- Different servers have different capabilities
- One real server can run several virtual servers
- Number of virtual servers depends on the capabilities
  - “Big” servers run more virtual servers
- CFS operates at virtual server level
  - Virtual nodes on a single real node know each other
  - Possible to use short cuts in routing

- Quotas can be used to limit storage for each node
- Quotas set on a per-IP address basis
  - Better quotas require central administration
  - Some systems implement “better” quotas, e.g., PAST

Updates in CFS

- Only publisher can change data in filesystem
- CFS will store any block under its content hash
  - Highly unlikely to find two blocks with same SHA-1 hash
  - No explicit protection for content blocks needed
- Root block is signed by publisher
  - Publisher must keep private key secret

- No explicit delete operation
  - Data stored only for agreed-upon period
  - Publisher must refresh periodically if persistence is needed

OceanStore

- OceanStore developed at UC Berkeley
- Runs on Tapestry DHT
- Supports object modification

- Vision of ubiquitous computing:
  - Intelligent devices, transparently in the environment
  - Highly dynamic, untrusted environment
- Question: Where does persistent information reside?
- OceanStore aims to be the answer
- OceanStore's target:
  - 10^{10} users, each with 10000 files, i.e., 10^{14} files total

OceanStore: Basics and Goals

- Users pay for the storage service
  - Several companies can provide services together
- Two goals:

1. Untrusted infrastructure
   - Everything is encrypted, infrastructure unreliable
   - However, assume that "most servers are working correctly most of the time"
   - One class of servers trusted to follow protocol (but not trusted with data)
2. Nomadic data
   - Anytime, anywhere
   - *Introspection* used to tune system at run-time

OceanStore Applications

- OceanStore suitable for many kinds of applications
- Storage and sharing of large amounts of data
  - Data follows users dynamically
- Groupware applications
  - Concurrent updates from many people
  - For example, calendars, contact lists, etc.
  - In particular, email and other communication applications
- Streaming applications
  - Also for sensor networks and dissemination

- Here we concentrate on storage

System Overview

- Each object has globally unique identifier (GUID)
- Objects replicated and migrated on the fly
- Replicas located in two ways
  - Probabilistic, fast algorithm tried first
  - Slower, but deterministic algorithm used if first one fails

- Objects exist in two forms, active and archival
  - Active is the latest version with handle for updates
  - Archival is a permanent, read-only version
    - Archive versions encoded with erasure codes with lot of redundancy
    - Only a global disaster can make data disappear

Naming and Access Control

Object naming

- Self-certifying object names
  - Hash of the object’s owners key and human readable name
- Allows directories
- System has no root, but user can select her own root(s)

Access control

- Restricting readers
  - All data is encrypted, can restrict readers by not giving key
  - Revoke read permission by re-encrypting everything
- Restricting writers
  - All writes must be signed and compared against ACL

Locating Objects

- OceanStore uses two mechanisms for locating objects

1. Probabilistic algorithm
    - Frequently accessed objects likely to be nearby and easily found
    - This algorithm finds them fast
    - Uses attenuated Bloom filters
    - See below for more details
2. Deterministic algorithm
    - OceanStore based on Tapestry
    - Deterministic routing is Tapestry’s routing
    - Guaranteed to find the object
    - See Chapter 3 for the details

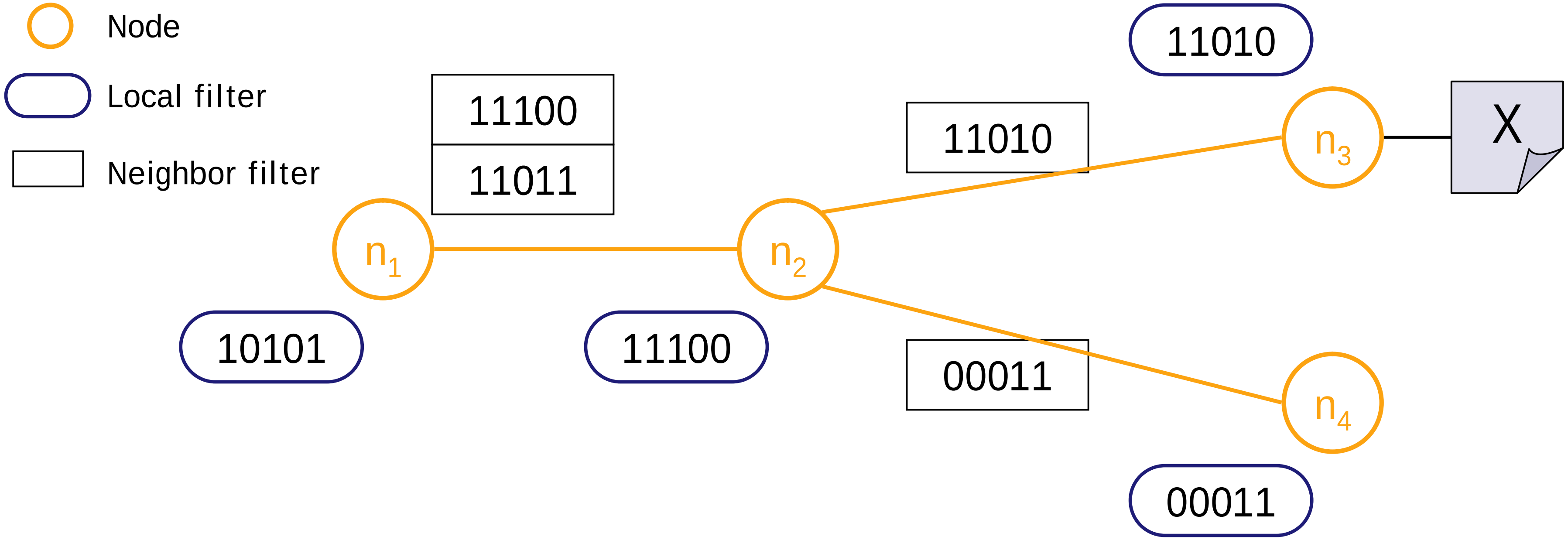
Sidenote: Bloom Filters

- Bloom filters are "*a space-efficient probabilistic data structure that is used to test whether or not an element is a member of a set*"
    - False positives are possible
    - False negatives are NOT possible
- Bloom filter is an array of *k* bits
    - Also need *m* different hash functions, each maps key to a bit
- To insert, calculate all *m* hash functions and set bits to 1
- To check, calculate all *m* hash functions and if all bits are 1, key is "probably" in the set
    - If any bit is 0, then it is definitely not in

Sidenote: Bloom Filters

- To insert or check for an item, Bloom filters take average $O(m)$ time
    - Fixed constant! Independent of number of entries
    - No other data structure allows for this (hash table is close)
- Attenuated Bloom filter of depth D is same as an array of D normal Bloom filters
    - First filter is for locally stored objects at current node
    - The i^{th} Bloom filter is the union of all Bloom filters at distance i through any path from current node
    - Attenuated Bloom filter for each network edge
    - Queries routed on the edge where the distance to object is shortest

Probabilistic Query Process



- Node n_1 wants to find object X, X hashes to bits 0, 1, 3
- Node n_1 does not have 0, 1, and 3 in local filter
  - Neighbor filter for n_2 has them, forward query to n_2
- Node n_2 does not have them in local filter
  - Filter for neighbor n_3 has them, forward to n_3
- Node n_3 has object

Update Model

- Update model in OceanStore based on conflict resolution
- Update semantics
  - Each update has a list of predicates with associated actions
  - Predicates evaluated in order
  - Actions for first true predicate are atomically applied (commit)
  - If no predicates are true, action aborts
  - Update is logged in both cases

Update Model: Predicates

- List of predicates is short
- Untrusted environment limits what predicates can do
- Available predicates:
  - Compare-version (metadata comparison, easy)
  - Compare-size (same as above)
  - Compare-block (easy if encryption is position-dependent block cipher)
  - Search (possible to search ciphertext, get boolean result)

Available Operations

- Four operations available
  - Replace-block
  - Insert-block
  - Delete-block
  - Append
- If position-dependent cipher, Replace-block and Append are easy operations
- For Insert-block and Delete-block:
  - Two kinds of blocks: Index and data blocks
  - Index blocks can contain pointers to other blocks
  - To insert a block, we replace old block with an index block which points to old block and new block
    - Actual blocks are appended to object
  - May be susceptible to traffic analysis

Serializing Updates

- Replicas divided into two tiers
  - Primary tier is trusted to follow protocol
  - Secondary tier is everyone else
- Primary tier cooperate in a Byzantine agreement protocol
- Secondary tier communicates with primary tier and secondary via epidemic algorithms
- Reason for two tiers:
  - Fault-tolerant protocols possible with only a small number of replicas, protocols communication-intensive
  - Primary tier is well-connected and small

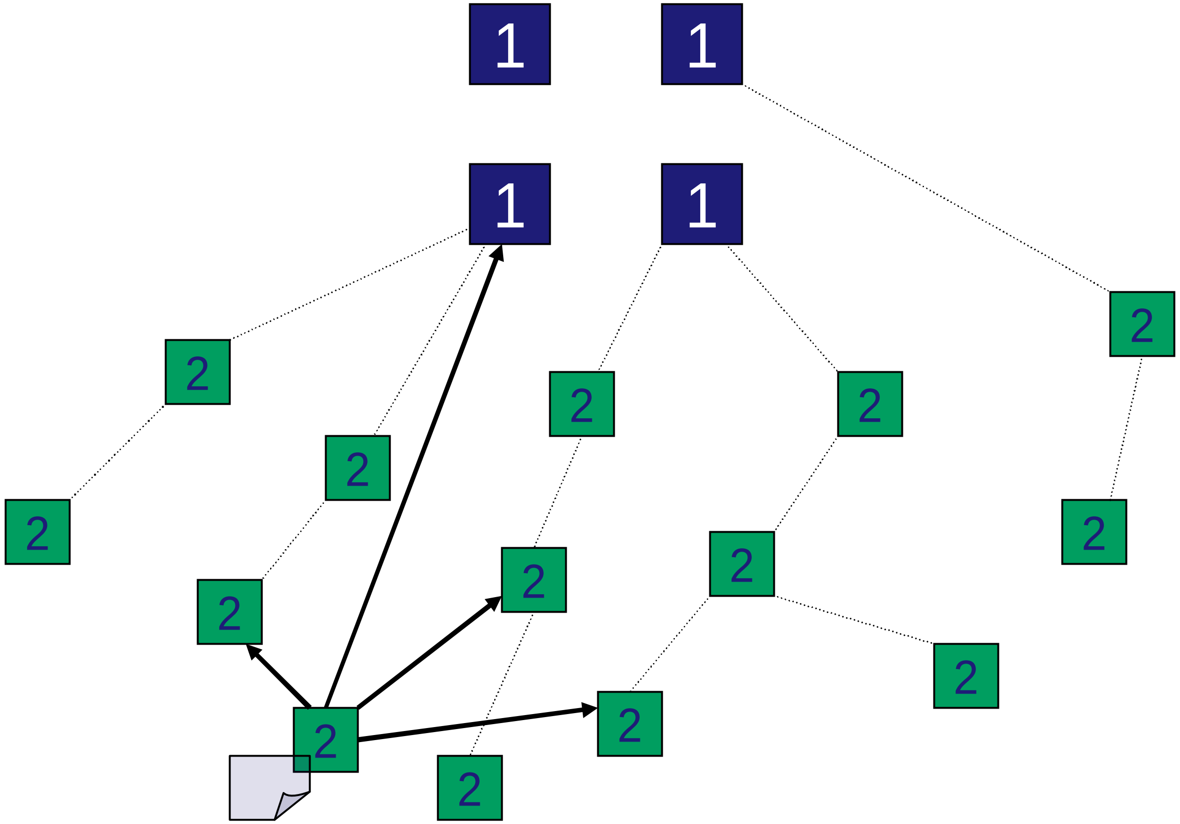
Byzantine Generals Problem

- Several divisions of the Byzantine army surround an enemy city. Each division is commanded by a general.
- The generals communicate only through messenger
  - Need to arrive at a common plan after observing the enemy
- Some of the generals may be traitors
  - Traitors can send false messages
- Required: An algorithm to guarantee that
  1. All loyal generals decide upon the same plan of action, irrespective of what the traitors do.
  2. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Solution: (from L. Lamport)

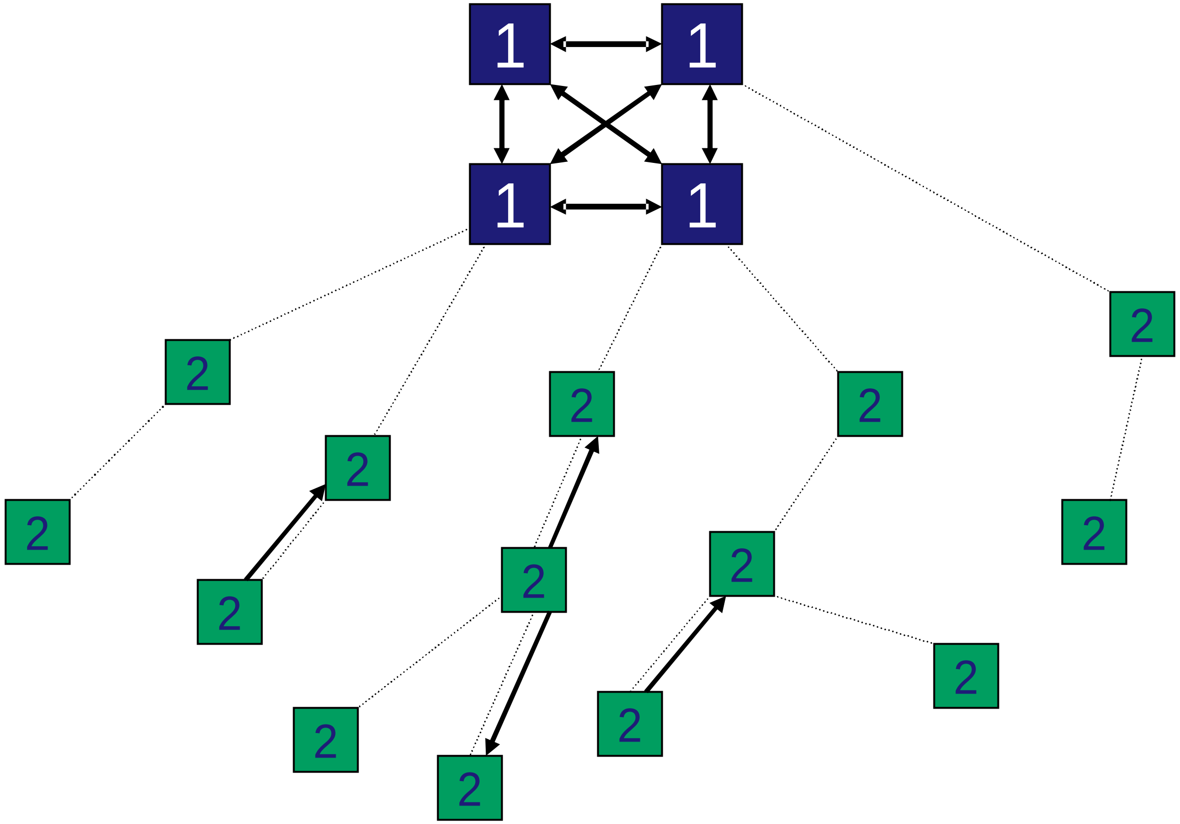
- If no more than m generals out of $n = 3m + 1$ are traitors, everybody will follow the orders

Path of an Update



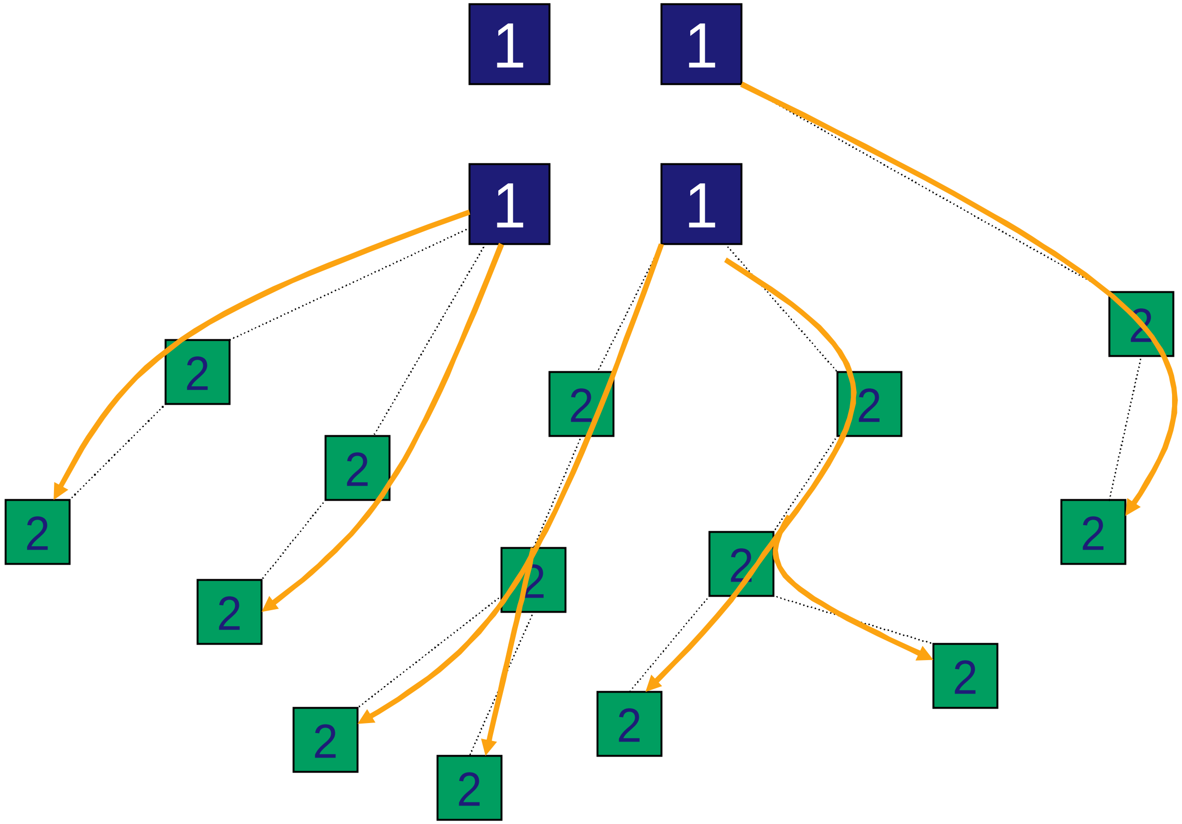
- Update is sent to primary tier and to random replicas in secondary tier for that object

Path of an Update



- Primary tier performs Byzantine agreement
- Secondary tier propagates update epidemically

Path of an Update



- When primary tier has finished agreement protocol, the update is sent over multicast to all secondary replicas

Deep Archival Storage

- Archival mechanism uses erasure codes
  - Reed-Solomon, Tornado, etc.
- Generate redundant data fragments
  - Created by the primary tier
- If there are enough fragments and they are spread widely, then it is likely that we can retrieve the data
- Archival copies are created when objects are changed
  - Every version is archived
  - Can be tuned to be done less frequently

Ivy

- Ivy developed at MIT
- Based on Chord
- Provides NFS-like semantics
    - At least for fully connected networks

- Any user can modify any file
- Ivy handles everything through logs

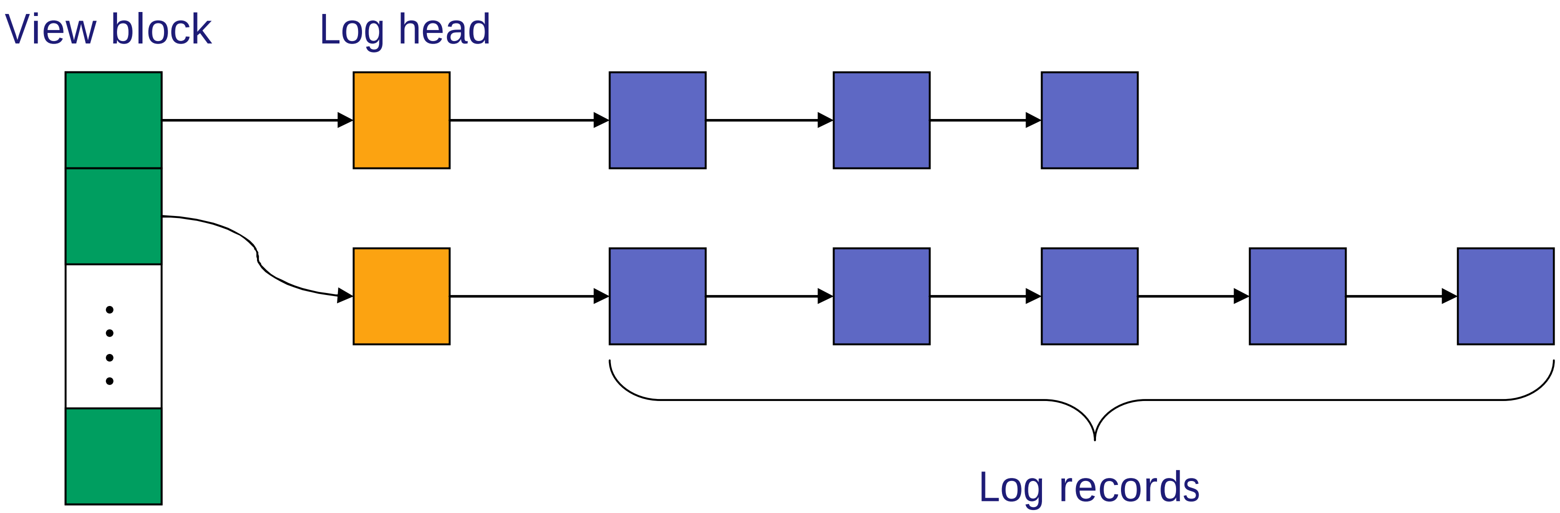
- Ivy presents a conventional filesystem interface

Problems for Distributed Read/Write

1. Multiple distributed writers make it difficult to maintain consistent metadata
2. Unreliable participants make locking unattractive
    - Locking could help maintain consistency
3. Participants cannot be trusted
    - Machines may be compromised
    - Need to be able to undo
4. Distributed filesystem may become partitioned
    - System must remain operational during partitions
    - Help applications repair conflicting updates made during partitions

Solution: Logs

- Each participant maintains log of its changes
  - Logs maintain filesystem data and metadata
  - Participant has private snapshot of logs of others
- Logs stored in DHash (see under CFS for details)
- Participant writes to its own log, reads all others
  - Log-head points to most recent log record
- Version vectors impose order on log records from multiple logs



Ivy: Views

- Each user writing to a filesystem has its own log
- Participants agree on a view
  - View is a set of logs that comprise the filesystem
- View block is immutable (“root”)
- View block has log heads for all participants

Combining Logs

- How to determine order of modifications from logs?
  - Order should obey causality
  - All participants should agree on the order
- Each new log record has a sequence number and a version vector
  - Sequence number increasing (managed locally)
  - Version vector has sequence numbers from other logs in view
  - Version vector summarizes knowledge about log
- Log records ordered by comparing version vectors
  - Vectors u and v comparable if $u < v$, $v < u$, or $v = u$
  - Otherwise concurrent
- Simultaneous operations result in equal or concurrent vectors
  - Ordered by public keys of participants
  - May need special actions to return to consistency (overlapping modifications)

Ivy: Snapshots

- Private snapshots avoid traversing whole filesystem
- Snapshot contains the entire state
- Each participant has own snapshot
  - Contents of snapshots mostly the same for all participants
  - DHash will store them only once
- To create snapshot, node:
  - Gets all logs recent than current snapshot
  - Write new snapshot
- New user must either build from scratch or take a trusted snapshot

P2P FS: General Problems

- Built on top of unreliable nodes and network
- How to achieve reliability?
  - Replication gives reliability (see Chapter 5)
  - Replication makes maintaining consistency difficult
- Nodes cannot be trusted in the general case
  - Must encrypt all data
  - Hard to do content-based conflict resolution (e.g., diff)
- Performance
  - Distributed filesystems have much lower performance

P2P FS: Future

- What does future hold for P2P filesystems?
- What is right area of application?
- Intranet?
  - Trusted environment, high bandwidth
  - Possibly easy to deploy?
    - Need to make a product first?
- Global Internet?
  - Lots of untrusted peers, widely distributed
  - All the problems from above
  - Can take off as a “hobby” project?
- Nowhere?
  - P2P filesystems are a total waste of time?

P2P FS: Future

- Do we need a P2P FS to build useful applications?
- Yes, they allow efficient distributed storage
  - Working storage system is the basic building block of a useful application
- No, DHT is enough
  - Several examples of P2P applications directly on top of a DHT

- Fully reliable P2P FS would make network into one virtual computer
  - Modulo performance issues
  - Building P2P apps would be like building normal apps

P2P FS: Real-World Examples

- Microsoft has done lot of research in this area
  - Even built prototypes
- Why no products on the market?
  - Below some wild speculation

- P2P FS would compete with traditional file servers?
  - P2P needs to be built in to the OS, would kill server market?
- Impossible to build a “good enough” P2P FS?
  - Theoretically doable, but too slow and weak in practice?
- P2P filesystem can be done, but has no advantages?
  - Possibly to build a useful system, but it costs as much as a server-based system?

Chapter Summary

- How to build applications with DHTs
- Basic technologies of distributed storage
- As examples, 3 P2P filesystems
  - CFS
    - Read-only
  - OceanStore
    - Modifications allowed, global vision
  - Ivy
    - NFS-like semantics, traditional filesystem interface
- Discussion about the future of P2P filesystems