Peer to Peer File Storage Systems

CS 699

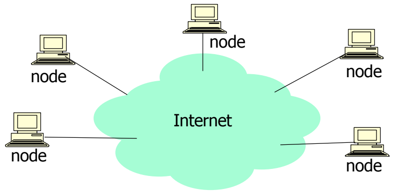
Acknowledgements

Some of the followings slides are borrowed from a talk by Robert Morris (MIT)

P2P File Systems

- File Sharing is one of the most popular P2P applications
  - E.g. Music sharing ala Napster, Gnutella, etc.
  - Anonymous storage
- Many P2P file systems built on top of DHTs
  - Freenet (anonymity)
  - PAST
    - Whole file storage
    - On top of PASTRY
  - CFS
    - Block-oriented
    - On top of Chord

Target Uses



- Serving data with inexpensive hosts:
  - open-source distributions
  - off-site backups
  - tech report archive
  - efficient sharing of music

How to mirror open-source distributions?

- Multiple independent distributions
  - Each has high peak load, low average
- Individual servers are wasteful
- Solution: aggregate
  - Option 1: single powerful server
  - Option 2: distributed service
    - But how do you find the data?

Some Design Challenges

- Scalability
- Avoid hot spots
- Spread storage burden evenly
- High availability
  - Tolerate unreliable participants
- Anonymity
- Security

Freenet: A Distributed Anonymous Information Storage & Retrieval System

Ian Clarke et al

Freenet: Design Goals

- Location-independent (wide area) distributed file system
- Goals
  - Anonymity for producers and consumers of information
  - Deniability for storers of information
  - Resistance to attempts by third parties to deny access to information
  - Efficient dynamic storage and routing of information
  - Decentralization of all network functions

Architecture

- Overlay network of nodes that store files
- Files are identified by location-independent keys
- Queries routed via steepest-ascent hill-climbing search with backtracking
- Transparent lazy replication
- Files are encrypted for deniability
- Anonymity: requesters and inserters of files cannot be identified since a node in a request path cannot tell whether its predecessor initiated the request or is forwarding it

Keys and Searching (1)

- Each file has a unique file key obtained by using the 160-bit SHA-1 hash function
- Three types of keys
  - Keyword-signed Key (KSK):
    - generate public/private key pair from descriptive text, e.g. gmu/cs/it818/lec2; apply SHA-1 to public key to get file key; private half of key used to sign the file
    - Problems: flat global name space, “key-squatting”
  - Signed-Subspace Key (SSK)
    - User randomly generates a public/private key pair for her namespace, hashes public namespace key and descriptive string separately, XORs them, hashes the result to obtain file key; private key used to sign file
    - Knowing public key of namespace and descriptive text enable users to compute file key
    - Can create a hierarchical file structure using SSKs

Keys and Searching (2)

- Third type of key
  - Content-hash key (CHK)
    - Derived by hashing the contents of the file
- CHKs used in conjunction with SSKs for implementing updating and splitting of files
  - User stores a file under its CHK
  - User stores an indirect file (with a SSK) whose contents are the CHK
  - File can be retrieved in two steps if SSK is known
  - The indirect file is updated by the owner with the new CHK if the file is updated
  - The indirect file can contain the CHKs of the parts of a large file that are split up and stored separately

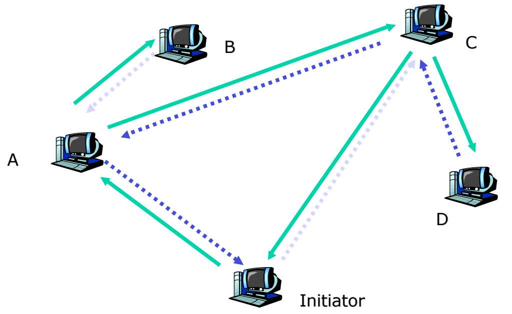
Retrieving Data

- File request = Key Request
- Each node maintains a routing table with (key, node) entries

Node receiving request checks its own store for key

1. if key found, returns file + note saying it is the data source
2. If key not found, looks up “lexicographically closest” key in routing table to requested key and forwards the request to the corresponding node
3. If data returned, passes the data back to the original requestor, caches a copy of the data, creates a new entry in routing table associating data source with requested key
4. If node cannot forward request because downstream node is down, the second nearest key is tried and so on… when it runs out of candidates, it reports a failure to upstream neighbor, who then tries its second choice, etc.
5. If hops to live exceeded, failure is reported

Routing Algorithm: search or insert



Routing algorithm

- Each node makes a copy of data file
  - Transparent data replication
  - File cached close to requestors – locality
- Nodes make new entries for previously unknown nodes that supply files
  - Increases connectivity
- Nodes will tend to become specialized in clusters of files with similar keys
  - Improves efficiency of future requests
  - Lexicographic similarity has no correlation to descriptive strings of a file or to file contents

Inserts

- Inserts follow same algorithm as searches
- If a node receiving an insert request has an old version of the file (with the same key), it returns the pre-existing file as if a request was made – this enables detection of collisions
- Once path for insert established, inserter sends the data which is propagated along path and also stored at each node along the way

Security Issues

- Anonymity:
  - Any node along the way for an insert or search can replace the data source field to claim itself or any arbitrary node as the source
  - Messages do not automatically terminate after hops-to-live = 1 but are forwarded with finite probability
- Deniability:
  - All files are encrypted; storer does not know encryption key

Performance

- ❑ Simulation study
- ❑ Metrics
  - Network convergence: how much time for the pathlengths to come down to acceptable levels?
  - Scalability: how does pathlength grow as network size increases
  - Fault tolerance: how does pathlength evolve as nodes fail
  - Small-world model applicable?

PAST: A Large-scale, persistent peer-to-peer storage utility

A. Rowstron & P. Druschel
SOSP 2001

Overview

- ❑ Peer-to-peer storage utility
  - Similar to Oceanstore, Freenet, CFS
- ❑ PAST nodes form a self-organizing overlay network
  - PASTRY used as routing layer (similar to Tapestry)
- ❑ Nodes can insert or retrieve files and (optionally) contribute storage
- ❑ Replication used for additional reliability; caching and locality for performance

---

PASTRY

- ❑ Based on Plaxton mesh like Tapestry with some differences
- ❑ Prefix routing with one digit resolved at each step
  - NodeIds and fileIds are sequences of digits with base 2^b
  - $O(\log_{2^b} N)$ steps
  - Routing table
    - $\log_{2^b} N$ levels each with 2^b-1 entries
    - Leaf set – L numerically closest nodeids (L/2 larger, L/2 smaller)
    - Neighborhood set – L closest nodes based on proximity metric - (this set is valuable in obtaining routing info from nodes that are closeby)

PAST Operations

1. Fileid = Insert(name, owner-credentials, k, file)
   - Fileid (160 bits) is SHA-1 hash of file name, owner's public key, and a random salt
   - Stores a file at k nodes in the PAST network whose nodeids are numerically closest to the 128 most significant bits of fileId
     - PAST nodes have 128-bit ids generated by a hash of the node's public key or IP address
   - Owner credentials = file certificate containing file metadata signed with owner's private key
   - Once all k nodes closest to fileid have accepted the file, an acknowledgement returned to client to which each of the k nodes attaches a store receipt
2. File = lookup(fileid)
   - Retrieves the file from one of the k nodes with a copy (normally from the closest such node)
3. Reclaim(fileId,owner-credentials)
   - Reclaims the storage occupied by k copies of the file
   - Weaker semantics than delete – does not guarantee that the file is no longer available

Security

- ❑ Each PAST node and user hold a smart card with a public/private key pair
- ❑ Smart cards generate and verify certificates ensuring the integrity of fileid and nodeid assignments
  - Assume smart cards are tamper-proof
- ❑ Store receipts, file certificates, reclaim certificates ensure integrity and authenticity of stored content
- ❑ Pastry routing scheme
  - All messages are signed, preventing forged entries
  - Routing is redundant, etc….

Storage Management

- ❑ Storage imbalance arises from
  - Statistical variation in assignment of nodeids and fileids
  - Size distribution of inserted files may have large variance
  - Storage capacity of individual nodes differs
    - Assume no more than two orders of magnitude difference
- ❑ Goals
  - Balance the remaining free storage space among nodes as storage space utilization nears 100%
  - Maintain the invariant that copies of the file are stored at the k nodes closest to fileId
- ❑ Techniques
  - Replica diversion
  - File diversion

Replica Diversion

- ❑ If one of the k closest nodes (say A) cannot accommodate a copy of a file, one of the nodes in its leaf set (say B) that is not among the k closest is used to store a replica
  - A makes an entry in its file table with a pointer to B, and returns a store receipt as usual
- ❑ Need to ensure that
  - Failure of node B causes the creation of a new replica
    - Under the PASTRY protocol, nodes keep track of live and failed nodes in their leaf set
  - Failure of node A does not render replica on B inaccessible
    - Achieved by entering a pointer to B on C, the k+1 th closest nodeid to fileid

Replica diversion cont'd

- Policies
  1. Reject a replica if file_size/free_space > t
     - Nodes among k closest (primary replica stores) use a larger threshold t than diverted replica stores
  2. How to chose node for diverted replica?
     - Select a node with maximal remaining free space among nodes in the leaf set of a primary store node, have a nodeId that is not among k closest, and does not already have a diverted copy
  3. When to divert the entire file to another part of the nodeId space?
     - If a primary store rejects the replica, and the node it then selects for the diverted replica also rejects it, the entire file is diverted

Other issues

- File diversion
  - On failure, client generates a new fileid and tries again (three times before giving up)
- File encoding
- Caching
  - A file that is routed through a node keeps a cached copy if its size is less than a fraction c of its current cache size
  - Greedy-dual-size cache replacement

Experimental Results

- Able to achieve global storage utilization > 98%
- Failed file insertions remains below 5% at 95% utilization and biased towards large files
- Caching is effective in achieving load balancing and reduces fetch distance and network traffic

Wide-Area Cooperative Storage with CFS

Robert Morris
Frank Dabek, M. Frans Kaashoek,
David Karger, Ion Stoica

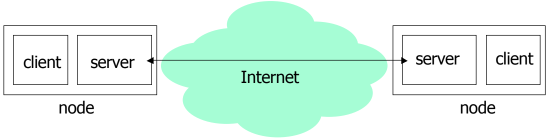
MIT and Berkeley

Design Challenges

- ❑ Avoid hot spots
- ❑ Spread storage burden evenly
- ❑ Tolerate unreliable participants
- ❑ Fetch speed comparable to whole-file TCP
- ❑ Avoid O(#participants) algorithms
  - ➢ Centralized mechanisms [Napster], broadcasts [Gnutella]

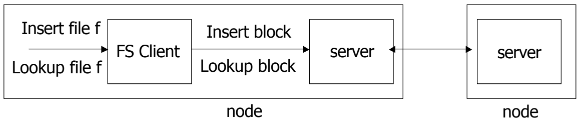
- ❑ CFS solves these challenges

CFS Architecture



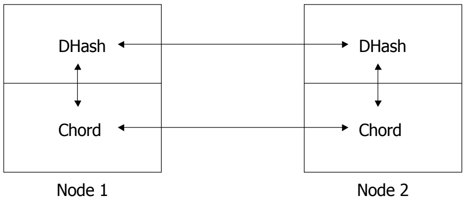
- ❑ Each node is a client and a server (like xFS)
- ❑ Clients can support different interfaces
  - ➢ File system interface
  - ➢ Music key-word search (like Napster and Gnutella)

Client-server interface

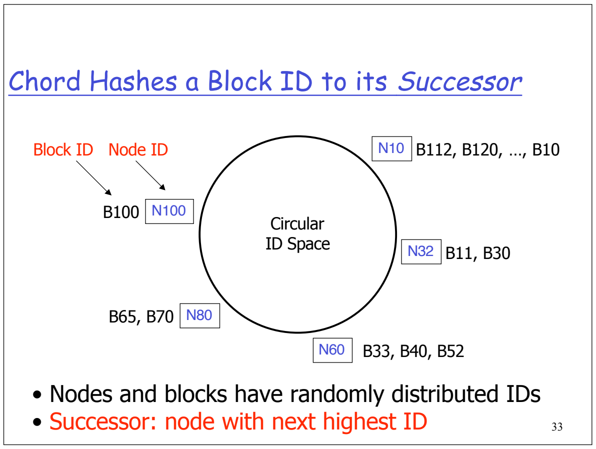


- ❑ Files have unique names
- ❑ Files are read-only (single writer, many readers)
- ❑ Publishers split files into blocks
- ❑ Clients check files for authenticity [SFSRO]

Server Structure

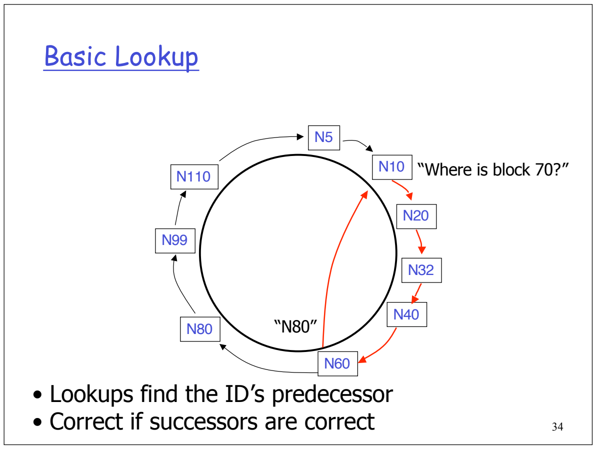


- DHash stores, balances, replicates, caches blocks
- DHash uses Chord [SIGCOMM 2001] to locate blocks



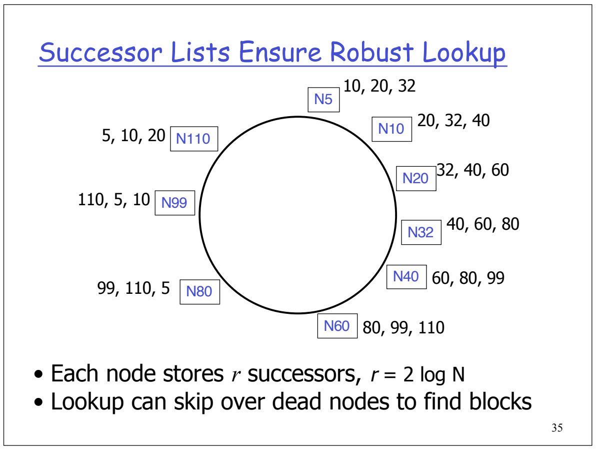
Chord Hashes a Block ID to its *Successor*

- Nodes and blocks have randomly distributed IDs
- Successor: node with next highest ID



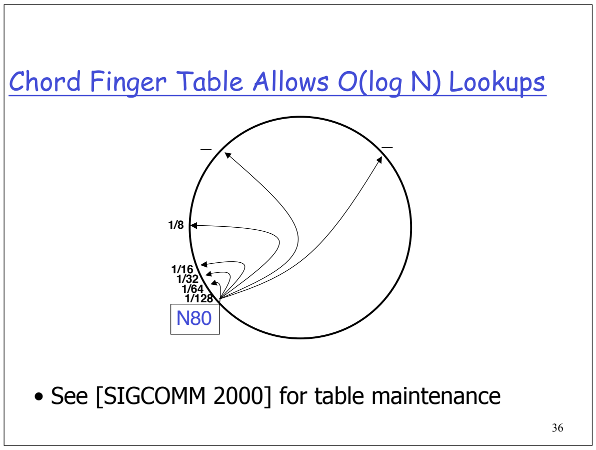
Basic Lookup

- Lookups find the ID's predecessor
- Correct if successors are correct



Successor Lists Ensure Robust Lookup

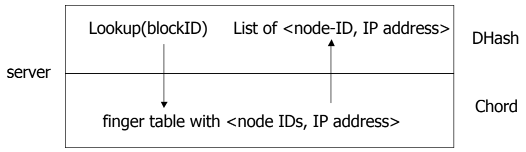
- Each node stores *r* successors, *r* = 2 log N
- Lookup can skip over dead nodes to find blocks



Chord Finger Table Allows O(log N) Lookups

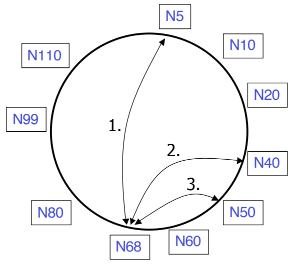
- See [SIGCOMM 2000] for table maintenance

DHash/Chord Interface



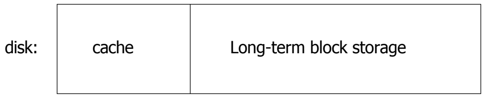
- *lookup()* returns list with node IDs closer in ID space to block ID
  - Sorted, closest first

DHash Uses Other Nodes to Locate Blocks



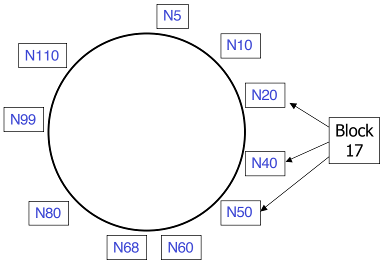
Lookup(BlockID=45)

Storing Blocks



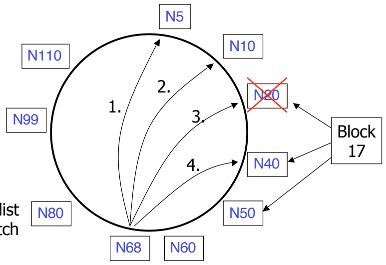
- Long-term blocks are stored for a fixed time
  - Publishers need to refresh periodically
- Cache uses LRU

Replicate blocks at *r* successors



- Node IDs are SHA-1 of IP Address
- Ensures independent replica failure

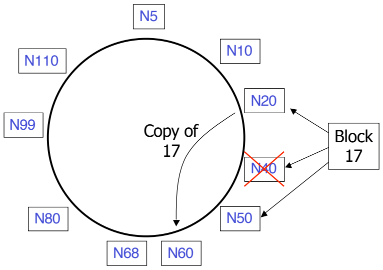
Lookups find replicas



RPCs:
1. Lookup step
2. Get successor list
3. Failed block fetch
4. Block fetch

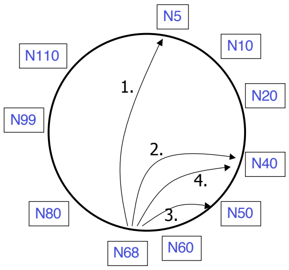
Lookup(BlockID=17)

First Live Successor Manages Replicas



- Node can locally determine that it is the first live successor

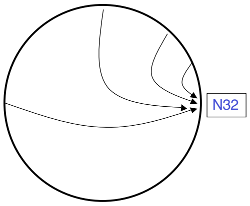
DHash Copies to Caches Along Lookup Path



RPCs:
1. Chord lookup
2. Chord lookup
3. Block fetch
4. Send to cache

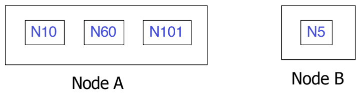
Lookup(BlockID=45)

Caching at Fingers Limits Load



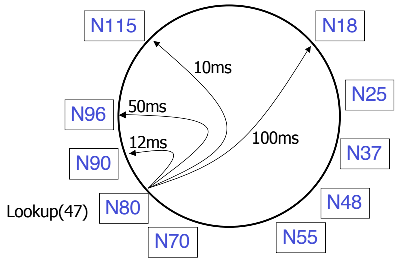
- Only O(log N) nodes have fingers pointing to N32
- This limits the single-block load on N32

Virtual Nodes Allow Heterogeneity



- Hosts may differ in disk/net capacity
- Hosts may advertise multiple IDs
  - Chosen as SHA-1(IP Address, index)
  - Each ID represents a "virtual node"
- Host load proportional to # v.n.'s
- Manually controlled

Fingers Allow Choice of Paths



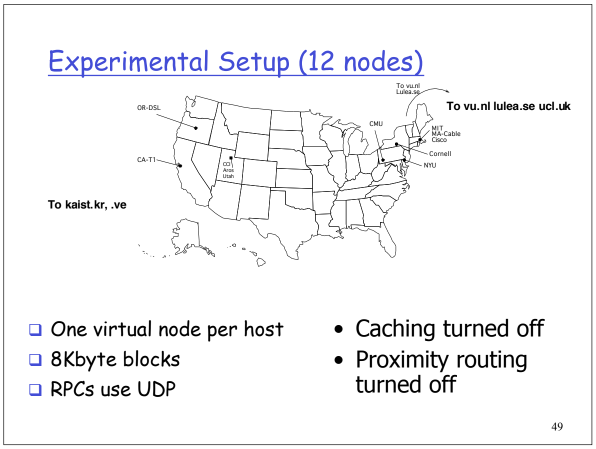
- Each node monitors RTTs to its own fingers
- Tradeoff: ID-space progress vs delay

Why Blocks Instead of Files?

- Cost: one lookup per block
  - Can tailor cost by choosing good block size
- Benefit: load balance is simple
  - For large files
  - Storage cost of large files is spread out
  - Popular files are served in parallel

CFS Project Status

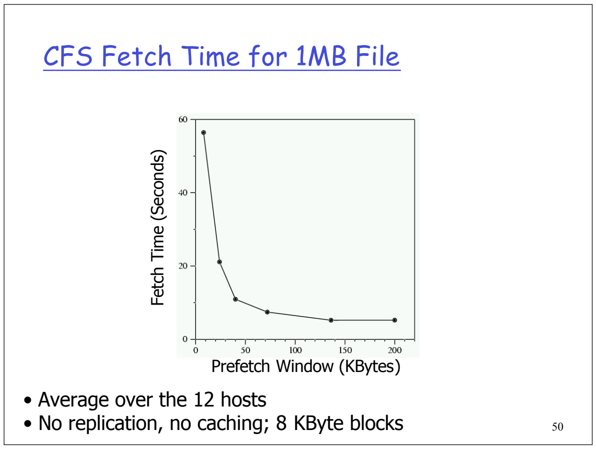
- Working prototype software
- Some abuse prevention mechanisms
- SFSRO file system client
  - Guarantees authenticity of files, updates, etc.
- Napster-like interface in the works
  - Decentralized indexing system
- Some measurements on RON testbed
- Simulation results to test scalability



Experimental Setup (12 nodes)

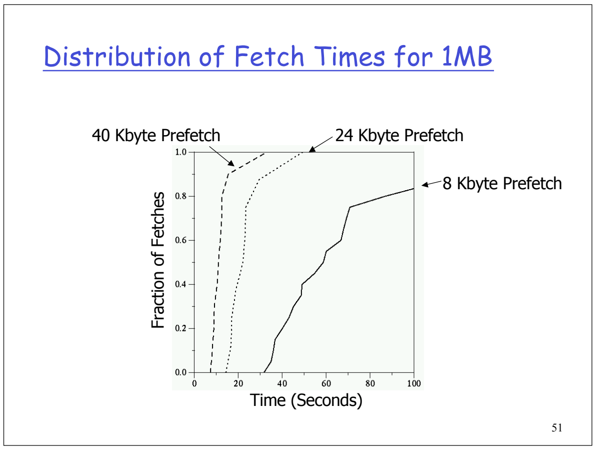
- ❑ One virtual node per host
- ❑ 8Kbyte blocks
- ❑ RPCs use UDP

- Caching turned off
- Proximity routing turned off

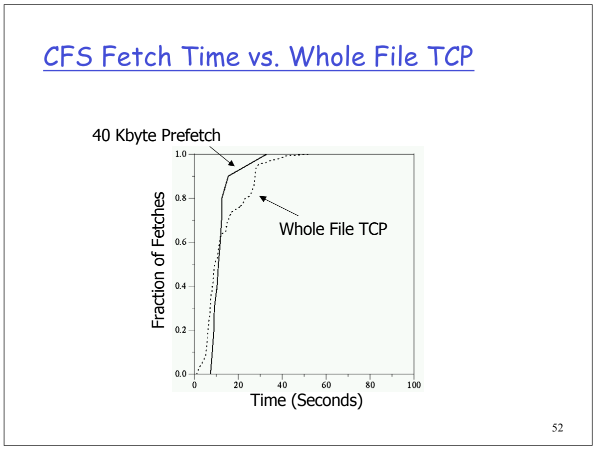


CFS Fetch Time for 1MB File

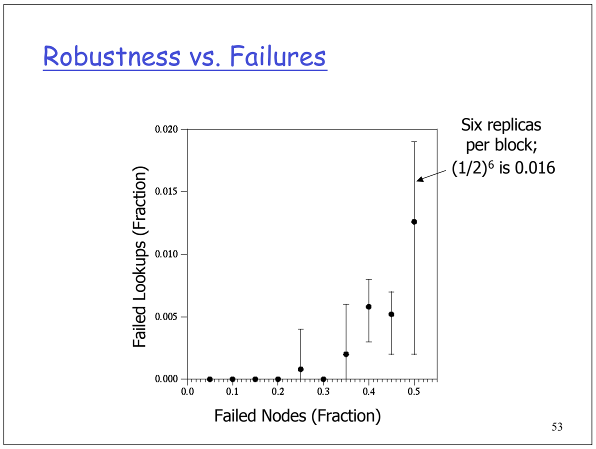
- Average over the 12 hosts
- No replication, no caching; 8 KByte blocks



Distribution of Fetch Times for 1MB



CFS Fetch Time vs. Whole File TCP



Robustness vs. Failures

Six replicas per block; $(1/2)^6$ is 0.016

CFS Summary

- CFS provides peer-to-peer r/o storage
- Structure: DHash and Chord
- It is efficient, robust, and load-balanced
- It uses block-level distribution
- The prototype is as fast as whole-file TCP

http://www.pdos.lcs.mit.edu/chord