CprE 450/550X
Distributed Systems and Middleware

Distributed File Systems

Yong Guan
3216 Coover
Tel: (515) 294-8378
Email: guan@ee.iastate.edu

May 1, 2003

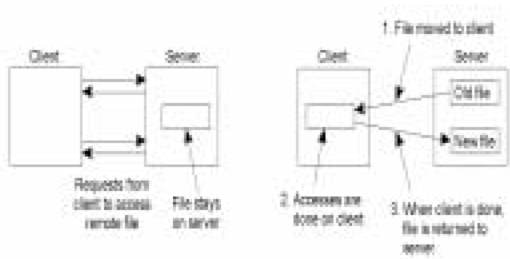
2

Readings for Today's Lecture

- References
  - Chapter 10 of "Distributed Systems: Principles and Paradigms"
  - Paper list on Peer-to-Peer systems on the course page.

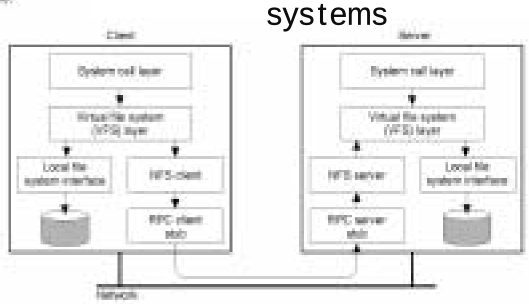
3

NFS Architecture



a) The remote access model.
b) The upload/download model

The basic NFS architecture for UNIX systems



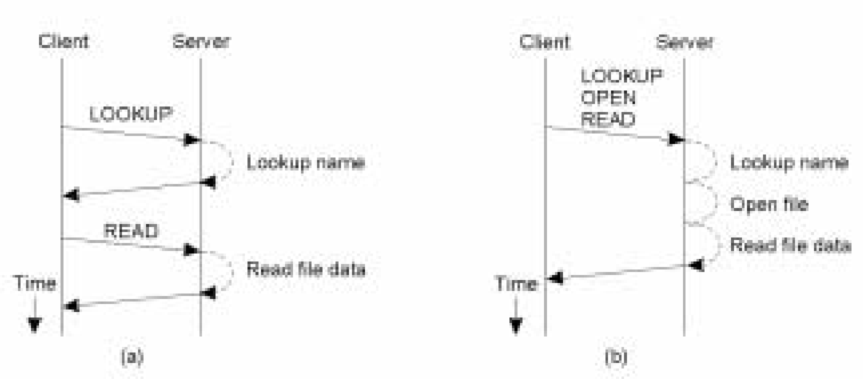
4

File System Model

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Rmdir	Yes	No	Remove an empty subdirectory from a directory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Read the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

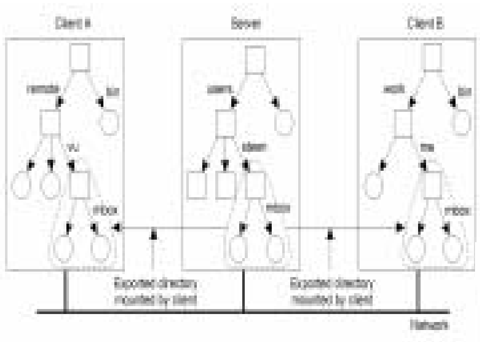
An incomplete list of file system operations supported by NFS.

Communication



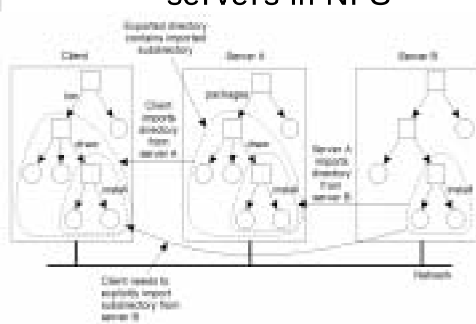
a) Reading data from a file in NFS version 3.
b) Reading data using a compound procedure in version 4.

NFS: Naming

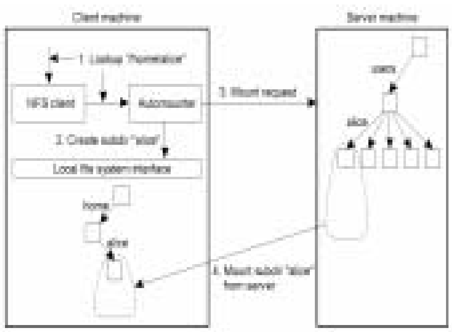


Mounting (part of) a remote file system in NFS.

Mounting nested directories from multiple servers in NFS

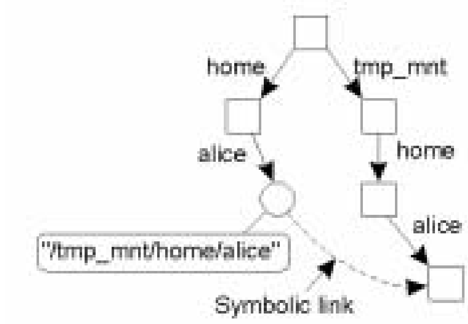


NFS: Automounting



A simple automounter for NFS

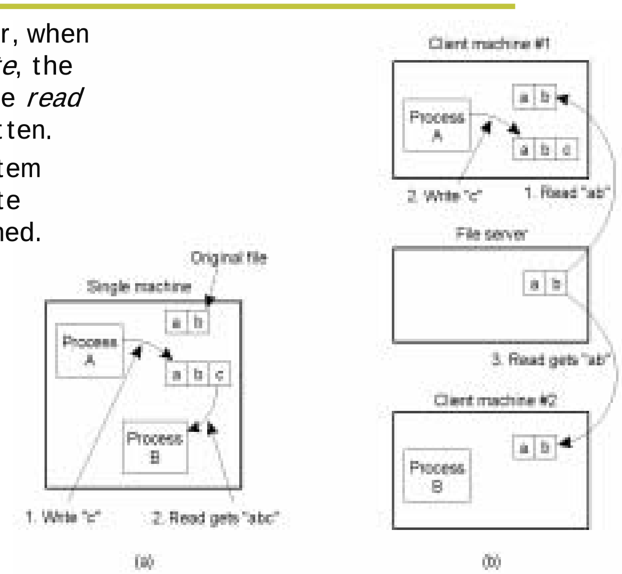
Using symbolic links with automounting



NFS: Semantics of File Sharing

a) On a single processor, when a *read* follows a *write*, the value returned by the *read* is the value just written.

b) In a distributed system with caching, obsolete values may be returned.



9

NFS: Semantics of File Sharing

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transaction	All changes occur atomically

Four ways of dealing with the shared files in a distributed system.

10

NFS: File Locking in NFS

Request access / **Current file denial state**

	NONE	READ	WRITE	BOTH
READ	Succeed	Fail	Succeed	Succeed
WRITE	Succeed	Succeed	Fail	Succeed
BOTH	Succeed	Succeed	Succeed	Fail

(a)

Current access state / **Requested file denial state**

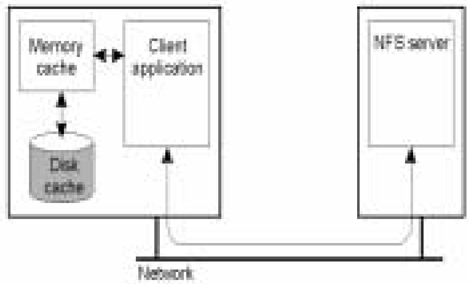
	NONE	READ	WRITE	BOTH
READ	Succeed	Fail	Succeed	Succeed
WRITE	Succeed	Succeed	Fail	Succeed
BOTH	Succeed	Succeed	Succeed	Fail

(b)

The result of an *open* operation with share reservations in NFS.

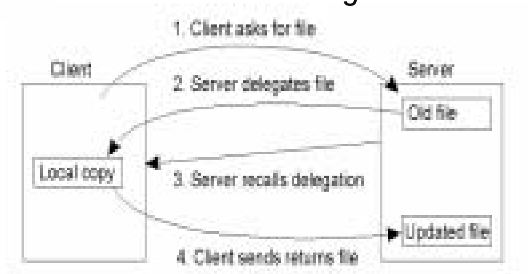
a) When the client requests shared access given the current denial state.
b) When the client requests a denial state given the current file access state.

NFS: Client Caching

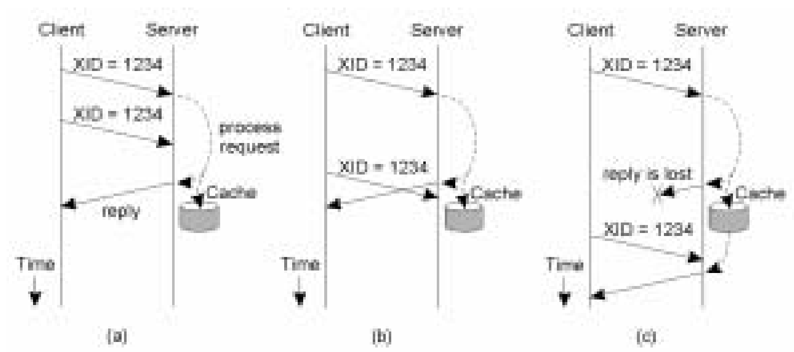


Client-side caching in NFS

Using the NFS version 4 callback mechanism to recall file delegation



NFS: RPC Failures



Three situations for handling retransmissions.

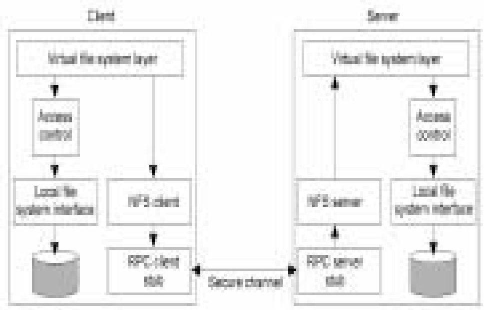
a) The request is still in progress

b) The reply has just been returned

c) The reply has been some time ago, but was lost.

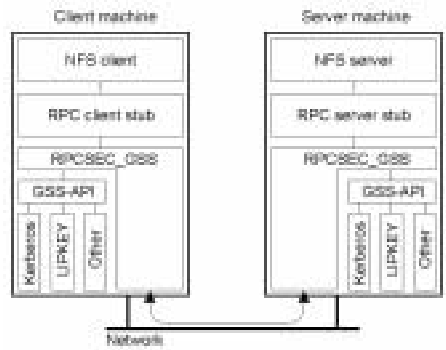
13

NFS: Security



The NFS security architecture

Secure RPC in NFS version 4



14

NFS: Access Control

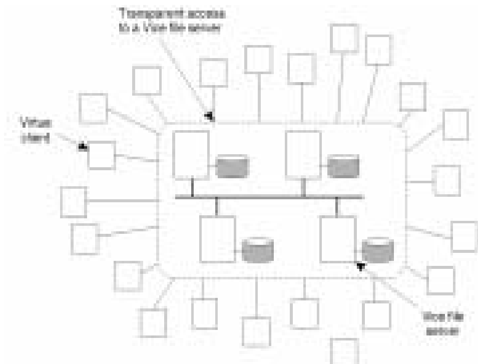
Operation	Description
Read_data	Permission to read the data contained in a file
Write_data	Permission to to modify a file's data
Append_data	Permission to to append data to a file
Execute	Permission to to execute a file
List_directory	Permission to to list the contents of a directory
Add_file	Permission to to add a new file t5o a directory
Add_subdirectory	Permission to to create a subdirectory to a directory
Delete	Permission to to delete a file
Delete_child	Permission to to delete a file or directory within a directory
Read_acl	Permission to to read the ACL
Write_acl	Permission to to write the ACL
Read_attributes	The ability to read the other basic attributes of a file
Write_attributes	Permission to to change the other basic attributes of a file
Read_named_attrs	Permission to to read the named attributes of a file
Write_named_attrs	Permission to to write the named attributes of a file
Write_owner	Permission to to change the owner
Synchronize	Permission to to access a file locally at the server with synchronous reads and writes

The classification of operations recognized by NFS with respect to access control.

15

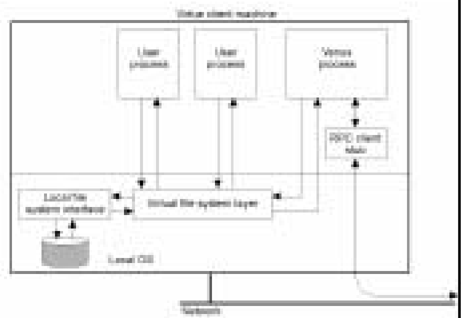
16

Coda File System

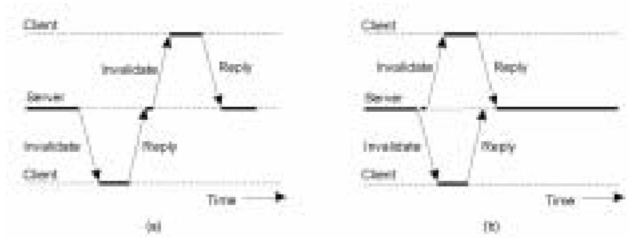


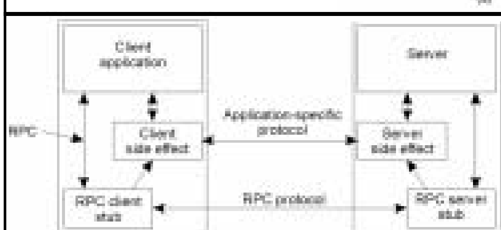
The overall organization of AFS

The internal organization of a Virtue workstation



Coda File System (cont.)

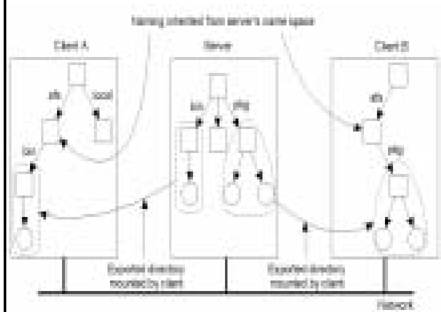




Side effects in Coda's RPC2 system

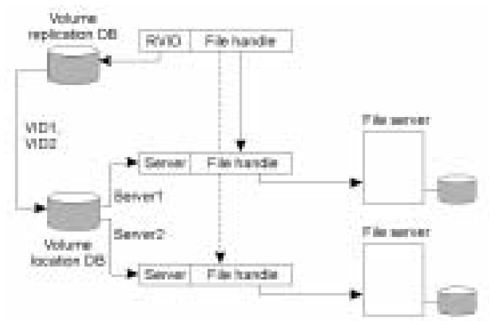
a) Sending an invalidation message one at a time.
b) Sending invalidation messages in parallel.

Coda File System (cont.)

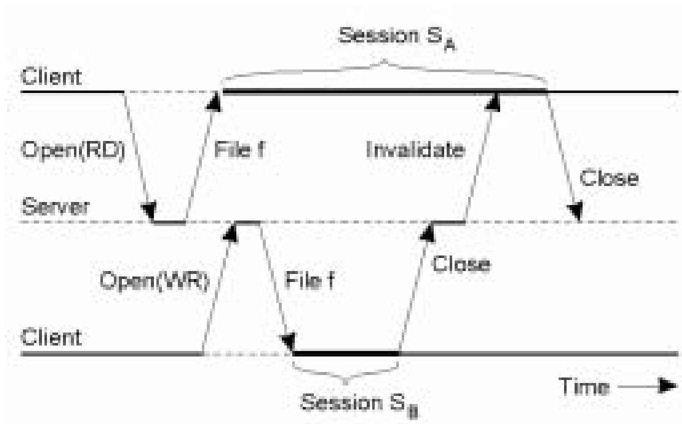


Clients in Coda have access to a single shared name space

Coda file identifier



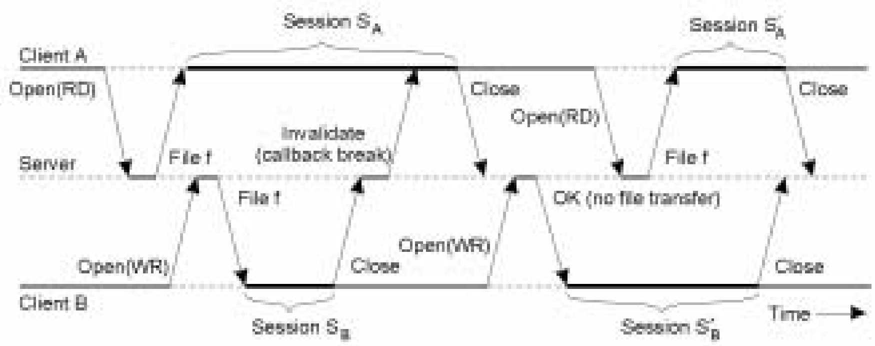
Coda File System (cont.)



The transactional behavior in sharing files in Coda.

21

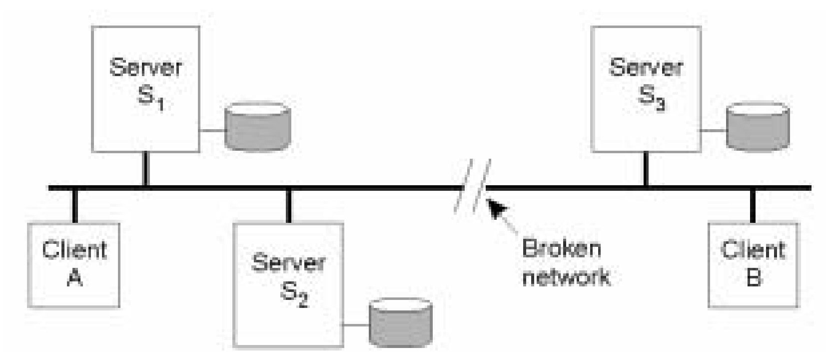
Coda File System (cont.)



Client Caching: The use of local copies when opening a session in Coda.

22

Coda File System (cont.)

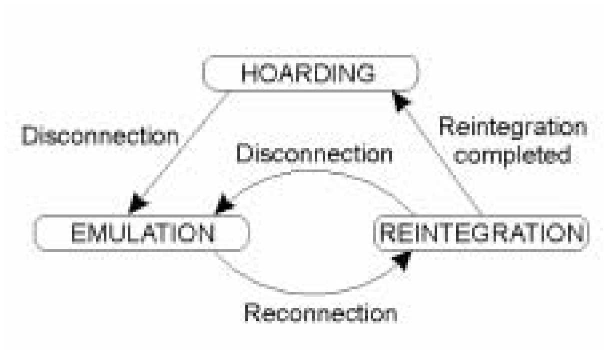


Two clients with different AVSG for the same replicated file.

23

Coda File System (cont.)

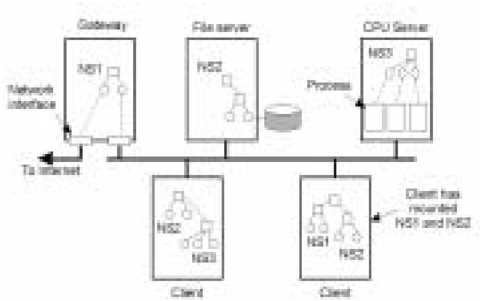
Disconnected Operation



The state-transition diagram of a Coda client with respect to a volume.

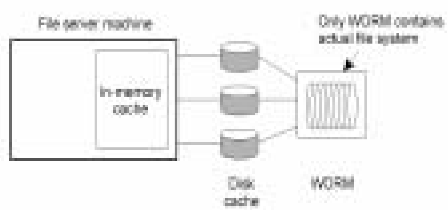
24

Plan 9: Resources Unified to Files

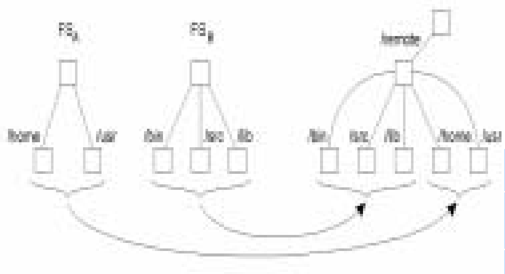


General organization of Plan 9

The Plan 9 file server.



Plan 9: Resources Unified to Files (cont.)

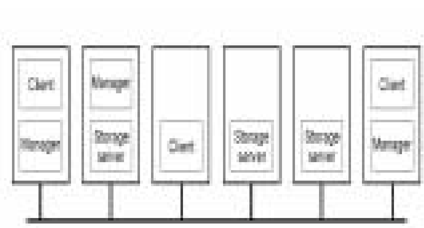


A union directory in Plan 9.

Files associated with a single TCP connection in Plan 9. ⟹

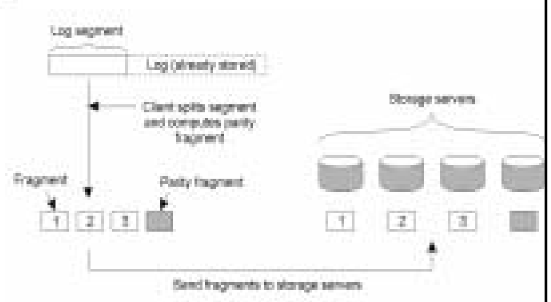
File	Description
ctl	Used to write protocol-specific control commands
data	Used to read and write data
listen	Used to accept incoming connection setup requests
local	Provides information on the caller's side of the connection
remote	Provides information on the other side of the connection
status	Provides diagnostic information on the current status of the connection

xFS: Serverless File Systems

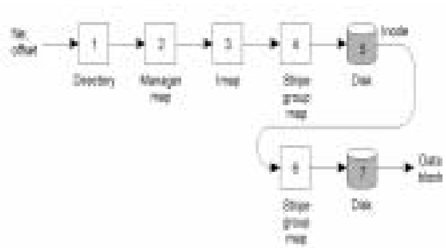


A typical distribution of xFS processes across multiple machines.

The principle of log-based striping in xFS.



xFS: Serverless File Systems (cont.)



Reading a block of data in xFS

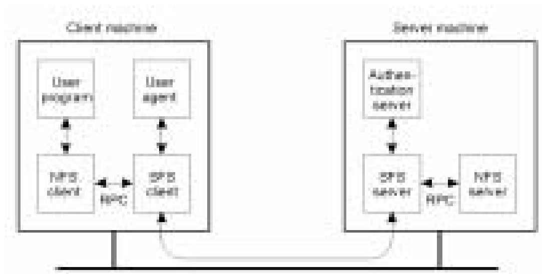
Data structure	Description
Manager map	Maps file ID to manager
Imap	Maps file ID to log address of file's inode
Inode	Maps block number (i.e., offset) to log address of block
File identifier	Reference used to index into manager map
File directory	Maps a file name to a file identifier
Log addresses	Triplet of stripe group, ID, segment ID, and segment offset
Stripe group map	Maps stripe group ID to list of storage servers

Main data structures used in xFS

SFS: Scalable Security

/sfs	LOC	HID	Pathname
/sfs/sfs.vu.sc.nl:ag62hty4wior450hdh63u623i4f0kqere/home/steen/mbox			

A self-certifying pathname in SFS.



- The organization of SFS.

Summary of Distributed File Systems

Issue	NFS	Coda	Plan 9	xFS	SFS
Design goals	Access transparency	High availability	Uniformity	Serverless system	Scalable security
Access model	Remote	Up/Download	Remote	Log-based	Remote
Communication	RPC	RPC	Special	Active msgs	RPC
Client process	Thin/Fat	Fat	Thin	Fat	Medium
Server groups	No	Yes	No	Yes	No
Mount granularity	Directory	File system	File system	File system	Directory
Name space	Per client	Global	Per process	Global	Global
File ID scope	File server	Global	Server	Global	File system
Sharing sem.	Session	Transactional	UNIX	UNIX	N/S
Cache consist.	write-back	write-back	write-through	write-back	write-back
Replication	Minimal	ROWA	None	Striping	None
Fault tolerance	Reliable comm.	Replication and caching	Reliable comm.	Striping	Reliable comm.
Recovery	Client-based	Reintegration	N/S	Checkpoint & write logs	N/S
Secure channels	Existing mechanisms	Needham-Schroeder	Needham-Schroeder	No pathnames	Self-cert.
Access control	Many operations	Directory operations	UNIX based	UNIX based	NFS BASED

- A comparison between NFS, Coda, Plan 9, xFS. N/S indicates that nothing has been specified.

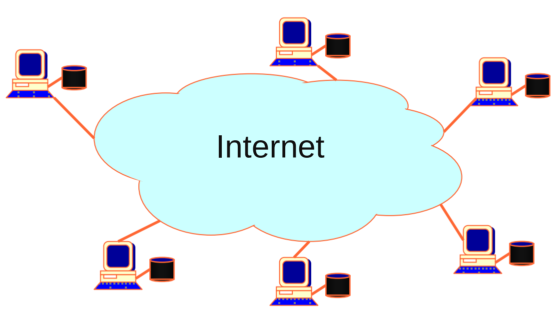
31

Peer-to-Peer Networks

32

How Did it Start?

- A killer application: Naptser
  - Free music over the Internet
- Key idea: share the content, storage *and* bandwidth of individual (home) users

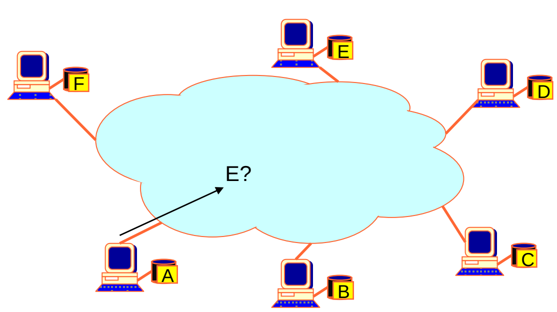


Model

- Each user stores a subset of files
- Each user has access (can download) files from all users in the system

Main Challenge

- Find where a particular file is stored



35

Other Challenges

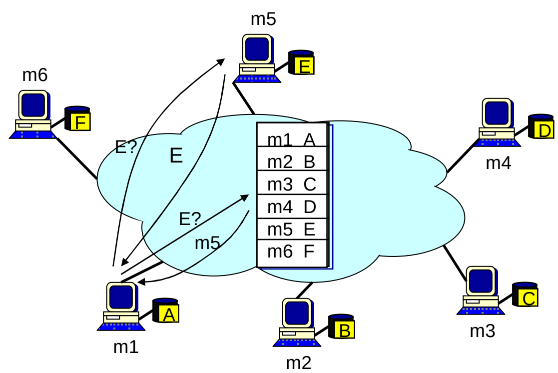
- Scale: up to hundred of thousands or millions of machines
- Dynamicity: machines can come and go any time

36

Napster

- Assume a centralized index system that maps files (songs) to machines that are alive
- How to find a file (song)
  - **Query the index system → return a machine that stores the required file**
    - » **Ideally this is the closest/least-loaded machine**
  - **ftp the file**
- Advantages:
  - **Simplicity, easy to implement sophisticated search engines on top of the index system**
- Disadvantages:
  - **Robustness, scalability (?)**

Napster: Example

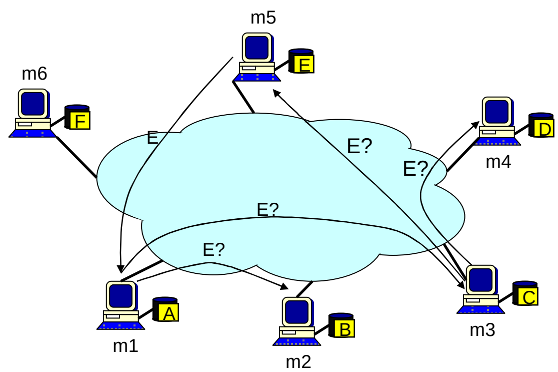


Gnutella

- Distribute file location
- Idea: flood the request
- Hot to find a file:
  - **Send request to all neighbors**
  - **Neighbors recursively multicast the request**
  - **Eventually a machine that has the file receives the request, and it sends back the answer**
- Advantages:
  - **Totally decentralized, highly robust**
- Disadvantages:
  - **Not scalable; the entire network can be swamped with request (to alleviate this problem, each request has a TTL)**

Gnutella: Example

- Assume: m1's neighbors are m2 and m3; m3's neighbors are m4 and m5;…

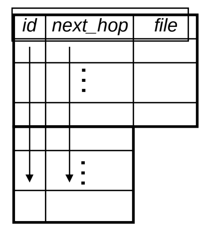


Freenet

- Addition goals to file location:
  - **Provide publisher anonymity, security**
  - **Resistant to attacks – a third party shouldn't be able to deny the access to a particular file (data item, object), even if it compromises a large fraction of machines**
- Architecture:
  - **Each file is identified by a unique identifier**
  - **Each machine stores a set of files, and maintains a "routing table" to route the individual requests**

Data Structure

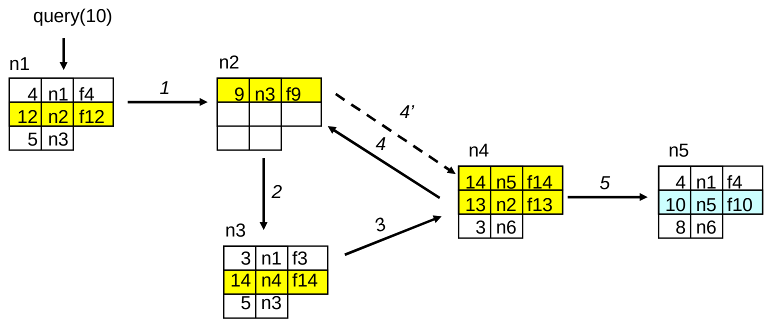
- Each node maintains a common stack
  - *id* – file identifier
  - *next_hop* – another node that store the file id
  - *file* – file identified by *id* being stored on the local node
- Forwarding:
  - Each message contains the file *id* it is referring to
  - If file *id* stored locally, then stop;
  - If not, search for the "closest" *id* in the stack, and forward the message to the corresponding *next_hop*



Query

- API: *file* = query(*id*);
- Upon receiving a query for document *id*
  - Check whether the queried file is stored locally
    - » If yes, return it
    - » If not, forward the query message
- Notes:
  - Each query is associated a TTL that is decremented each time the query message is forwarded; to obscure distance to originator:
    - » TTL can be initiated to a random value within some bounds
    - » When TTL=1, the query is forwarded with a finite probability
  - Each node maintains the state for all outstanding queries that have traversed it → help to avoid cycles
  - When file is returned, the file is cached along the reverse path

Query Example



- Note: doesn't show file caching on the reverse path

Insert

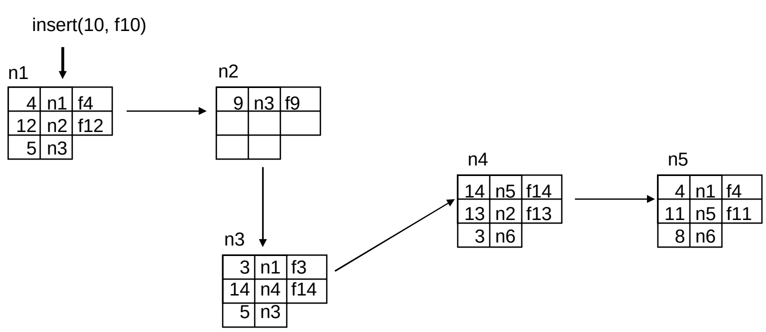
- API: insert(*id*, *file*);
- Two steps
  - Search for the file to be inserted
  - If not found, insert the file

Insert

- Searching: like query, but nodes maintain state after a collision is detected and the reply is sent back to the originator
- Insertion
  - Follow the forward path; insert the file at all nodes along the path
  - A node probabilistically replace the originator with itself; obscure the true originator

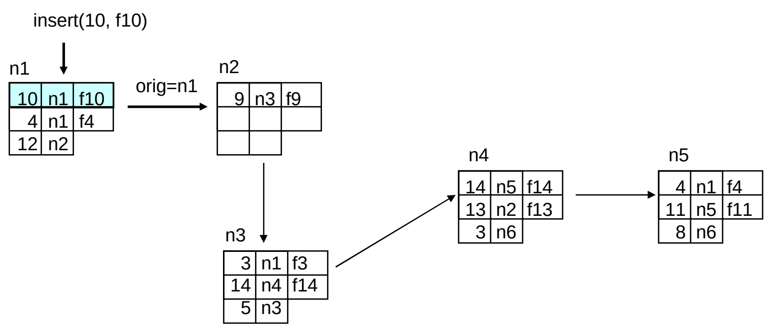
Insert Example

- Assume query returned failure along “gray” path; insert f10



Insert Example

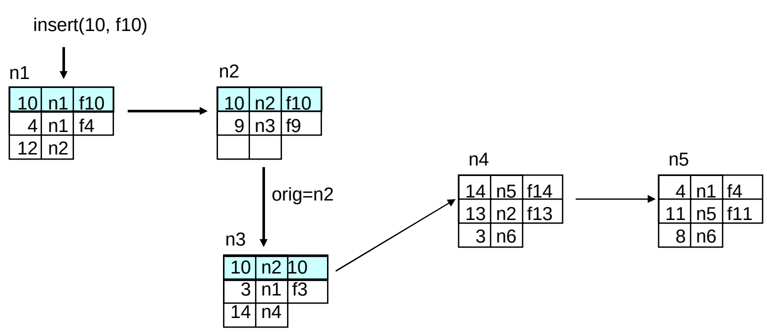
47



Insert Example

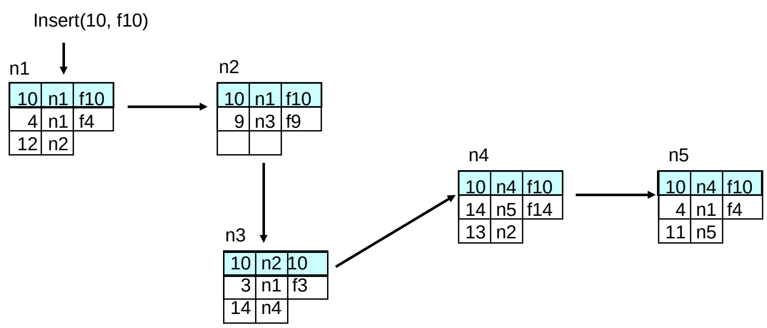
48

- n2 replaces the originator (n1) with itself



Insert Example

- n2 replaces the originator (n1) with itself



Freenet Properties

- Newly queried/inserted files are stored on nodes storing similar ids
- New nodes can announce themselves by inserting files
- Attempts to supplant or discover existing files will just spread the files

51

Freenet Summary

- Advantages
  - Provides publisher anonymity
  - Totally decentralize architecture → robust and scalable
  - Resistant against malicious file deletion
- Disadvantages
  - Does not always guarantee that a file is found, even if the file is in the network

52

Other Solutions to the Location Problem

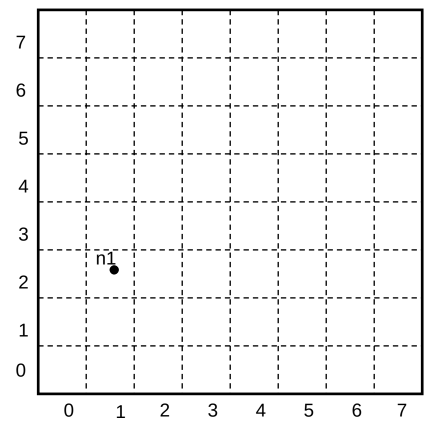
- Goal: make sure that an item (file) identified is always found
- Abstraction: a distributed hash-table data structure
  - insert(id, item);
  - item = query(id);
  - Note: item can be anything: a data object, document, file, pointer to a file…
- Proposals
  - CAN, Chord, Kademlia, Pastry, Viceroy, Tapestry, etc

Content Addressable Network (CAN)

- Associate to each node and item a unique *id* in an *d*-dimensional Cartesian space
- Goals
  - **Scales to hundreds of thousands of nodes**
  - **Handles rapid arrival and failure of nodes**
- Properties
  - **Routing table size O(*d*)**
  - **Guarantees that a file is found in at most $d*n^{1/d}$ steps, where *n* is the total number of nodes**

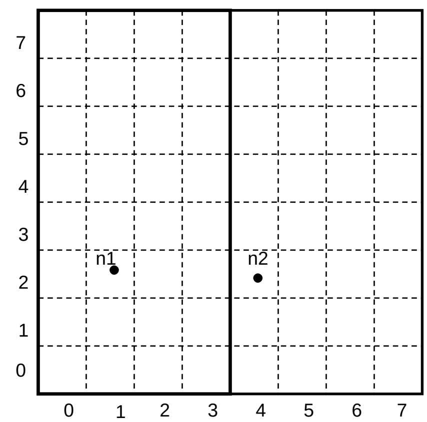
CAN Example: Two Dimensional Space

- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Node n1:(1, 2) first node that joins → cover the entire space



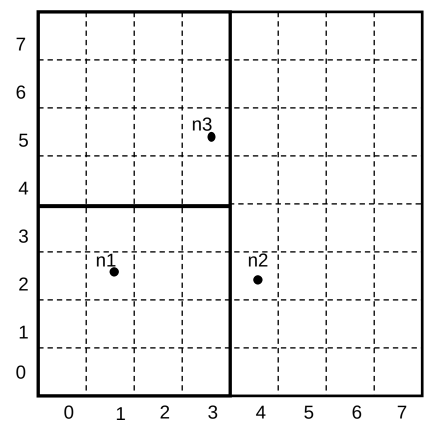
CAN Example: Two Dimensional Space

- Node n2:(4, 2) joins → space is divided between n1 and n2



CAN Example: Two Dimensional Space

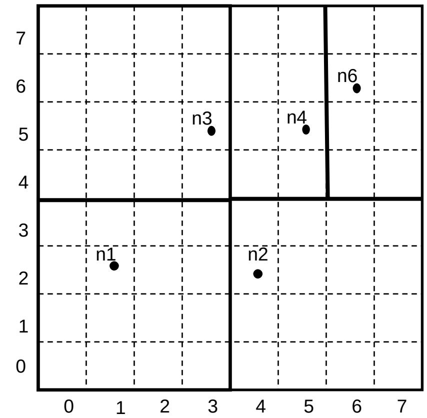
- Node n3:(3, 5) joins → space is divided between n1 and n2



57

CAN Example: Two Dimensional Space

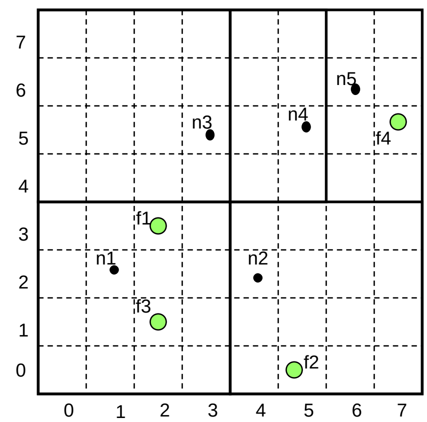
- Nodes n4:(5, 5) and n5:(6,6) join



58

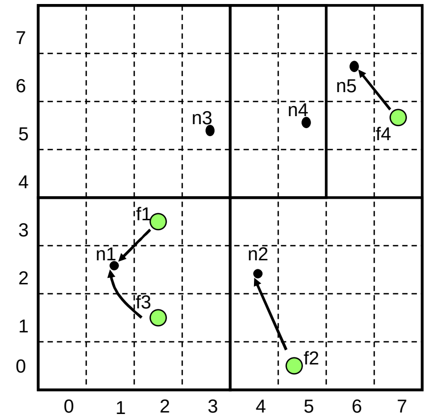
CAN Example: Two Dimensional Space

- Nodes: n1:(1, 2); n2:(4,2); n3:(3, 5); n4:(5,5);n5:(6,6)
- Items: f1:(2,3); f2:(5,1); f3:(2,1); f4:(7,5);



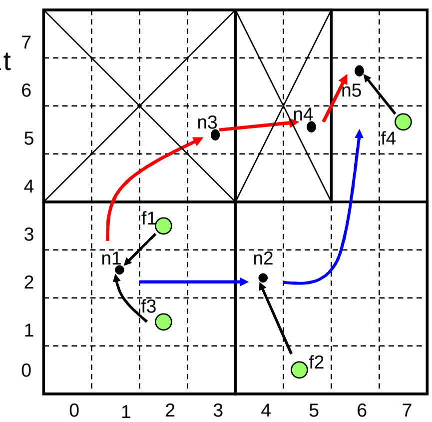
CAN Example: Two Dimensional Space

- Each item is stored by the node who owns its mapping in the space



CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures



Node Failure Recovery

- Simple failures
  - Know your neighbor's neighbors
  - When a node fails, one of its neighbors takes over its zone
- More complex failure modes
  - Simultaneous failure of multiple adjacent nodes
  - Scoped flooding to discover neighbors
  - Hopefully, a rare event

Chord

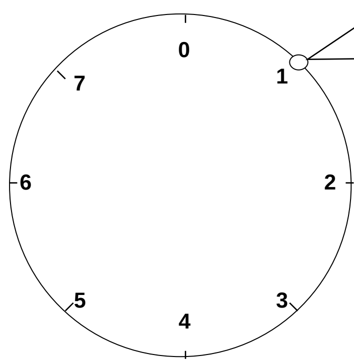
- Associate to each node and item a unique *id* in an *uni-*dimensional space
- Goals
  - Scales to hundreds of thousands of nodes
  - Handles rapid arrival and failure of nodes
- Properties
  - Routing table size O(log(N)) , where N is the total number of nodes
  - Guarantees that a file is found in O(log(N)) steps

Data Structure

- Assume identifier space is $0..2^m$
- Each node maintains
  - Finger table
    - Entry *i* in the finger table of *n* is the first node that succeeds or equals $n + 2^i$
  - Predecessor node
- An item identified by *id* is stored on the succesor node of *id*

Chord Example

- Assume an identifier space 0..8
- Node n1:(1) joins→all entries in its finger table are initialized to itself

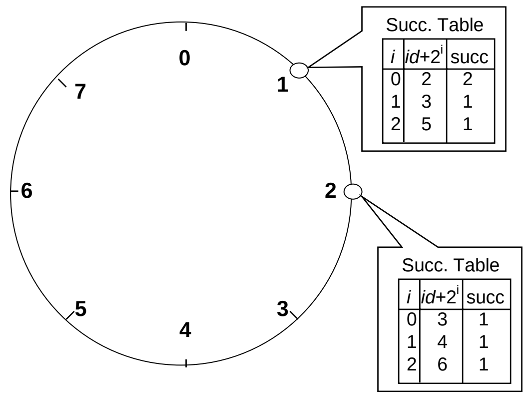


Succ. Table

i	$id+2^i$	succ
0	2	1
1	3	1
2	5	1

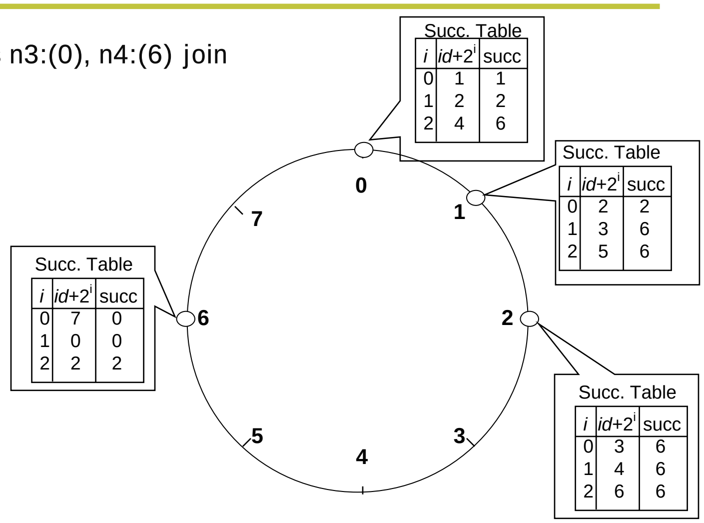
Chord Example

- Node n2:(3) joins



Chord Example

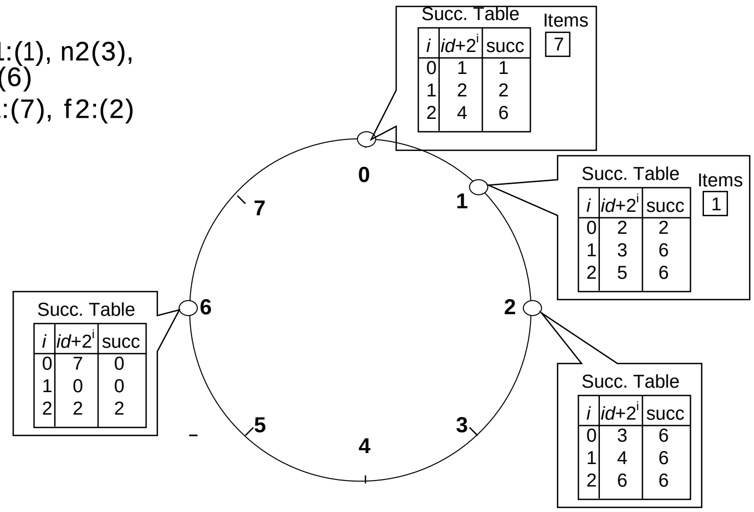
- Nodes n3:(0), n4:(6) join



67

Chord Examples

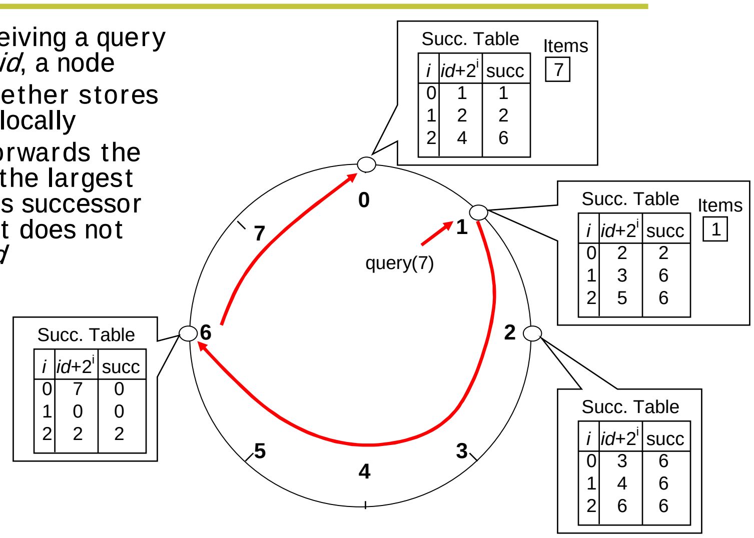
- Nodes: n1:(1), n2(3), n3(0), n4(6)
- Items: f1:(7), f2:(2)



68

Query

- Upon receiving a query for item *id*, a node
- Check whether stores the item locally
- If not, forwards the query to the largest node in its successor table that does not exceed *id*



Node Joining

- Node n joins the system:
  - **n picks a random identifier, id**
  - **n performs n' = lookup(id)**
  - **n->successor = n'**

State Maintenance: Stabilization Protocol

- Periodically node n
  - Asks its successor, n', about its predecessor n"
  - If n" is between n' and n"
    - n->successor = n"
    - notify n" that n its predecessor
- When node n" receives notification message from n
  - If n is between n"->predecessor and n", then
    - n"->predecessor = n
- Improve robustness
  - Each node maintain a successor list (usually of size 2*log N)

CAN/Chord Optimizations

- Weight neighbor nodes by RTT
  - When routing, choose neighbor who is closer to destination with lowest RTT from me
  - Reduces path latency
- Multiple physical nodes per virtual node
  - Reduces path length (fewer virtual nodes)
  - Reduces path latency (can choose physical node from virtual node with lowest RTT)
  - Improved fault tolerance (only one node per zone needs to survive to allow routing through the zone)
- Several others

Conclusions

- Distributed Hash Tables are a key component of scalable and robust overlay networks
- CAN: O(d) state, O(d*n1/d) distance
- Chord: O(log n) state, O(log n) distance
- Both can achieve stretch < 2
- Simplicity is key
- Services built on top of distributed hash tables
  - p2p file storage, i3 (chord)
  - multicast (CAN, Tapestry)
  - persistent storage (OceanStore using Tapestry)