P2P Storage Systems

Prof. Chun-Hsin Wu
Dept. Computer Science & Info. Eng.
National University of Kaohsiung

Outline

- Introduction
  - Distributed file systems
  - P2P file-swapping systems
  - P2P storage systems
    - Strengths and design issues
- Case Studies
  - Freenet
  - CFS
  - PAST
  - OceanStore
  - Ivy
- Further Issues

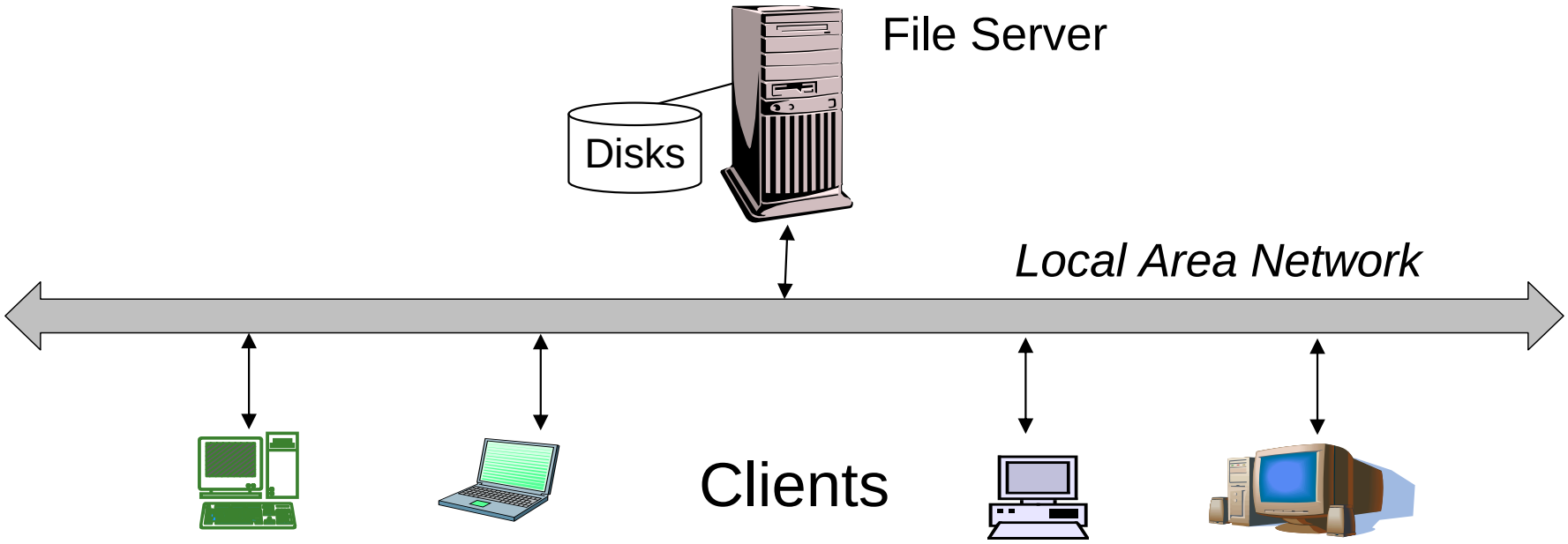
P2P Storage Systems

- A P2P system enables *programs* to store and access remote files exactly *as they do local ones*
  - Build a distributed file system (DFS) in a P2P manner
  - Allow users to share storages or files easily and arbitrarily

1st generation DFS	Client-server model	NFS, CIFS, AFS, …
2nd generation DFS	Cluster of servers	ZebraFS, xFS, …
3rd generation DFS	P2P	CFS, Ivy, …

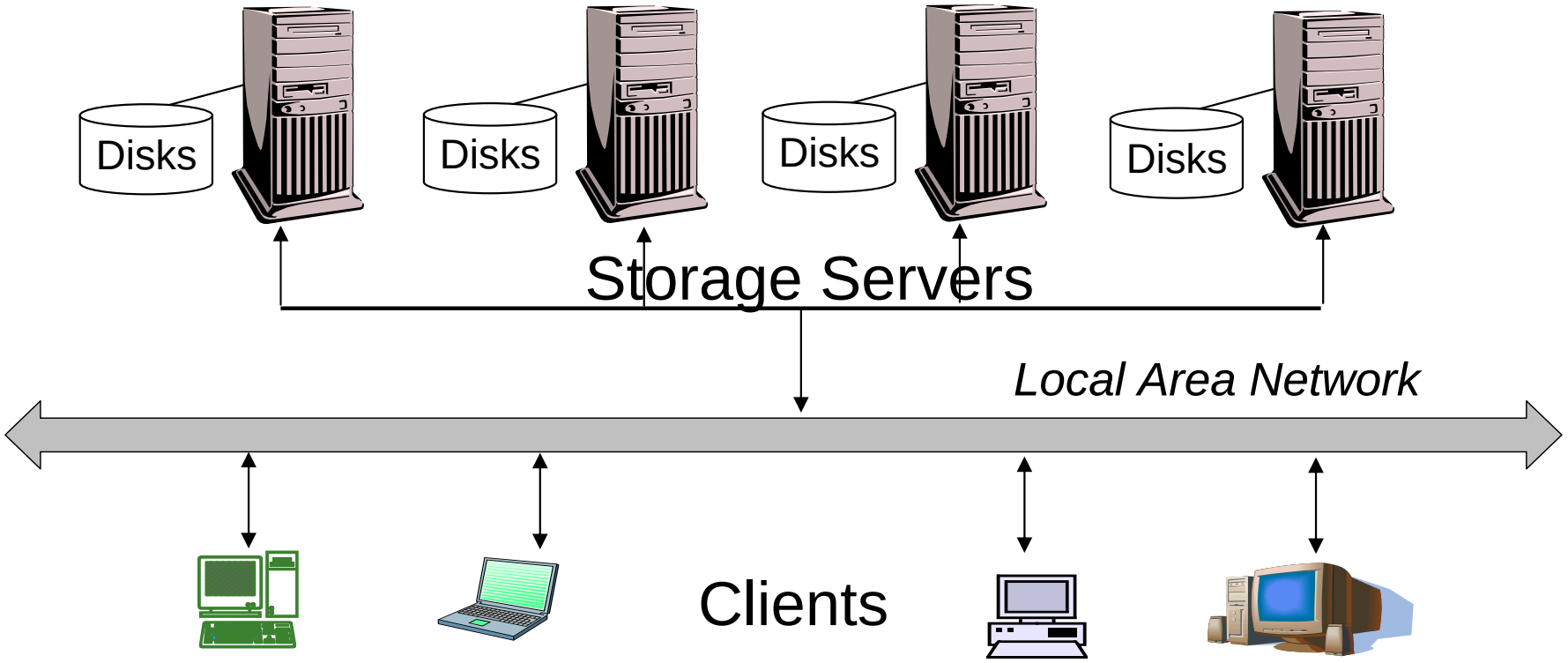
P2P storage systems:
Distributed file systems + P2P file-swapping systems

1st Generation DFS: Client-Server



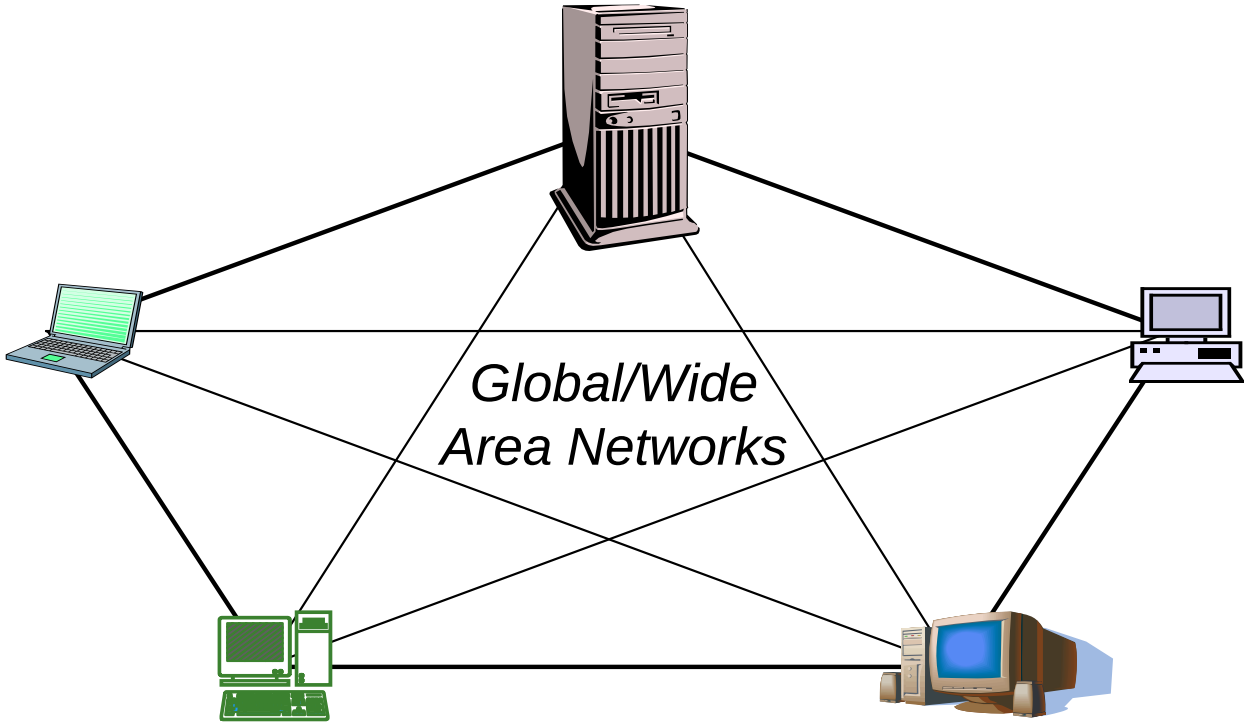
- Provide clients transparent access to a file system stored at a remote server
  - Usually the server and clients are on the same network
  - The server becomes the bottleneck and single point of failure

2nd Generation DFS: Server Clusters



- A file is stripped to a cluster of storage servers like RAID
  - Each storage server stores parts of files on local node
  - All machines can be a server in their symmetric and scalable design: trusted and stable

3rd Generation DFS: P2P Storage System



- No assumptions about trust, churn, and bandwidth
  - Autonomous, self-organizing
  - Heterogeneous

Beyond P2P File-Swapping

- File-swapping vs. storage-sharing
  - File-swapping: enable *users* to swap (publish / retrieve) files using a particular software program
  - Storage-sharing: enable *programs* to manipulate shared files transparently
- P2P storage systems
  - Support file-sharing as P2P file-swapping systems do
  - Provide more file services than P2P file-swapping systems

Strengths of P2P Storage Systems

- Aggregate storage spaces and resources
  - Heterogeneous environment
  - Reliable services with individual redundant resources
- Reduce storage and maintenance costs
  - Autonomous
  - Self-organizing
- Provide transparent, global, persistent storage
  - Globe scale
  - Continuous access
  - Persistent information

Design Issues of P2P Storage Systems

- Peer issues
  - Decentralization
  - Churn
  - Trust
  - Accounting
- File issues
  - Consistency
  - Persistence
  - Security
  - Transparency

Comparison of P2P Storage Systems

System	Scheme	Lookup Time	Unit	Persist-ence	Read/ Write
Freenet	Depth-First Flooding	Variable	File	No	Read Only
CFS	Chord	O(log N)	Block	Yes	Read Only
PAST	Pastry	O(log N)	File	Yes	Read Only
Ocean Store	Bloom Filters + Tapestry	O(log N)	Fragment	Yes	Read/ Write
Ivy	Chord	O(log N)	Log-based	Yes	Read/ Write

Freenet: Introduction

- A distributed anonymous information storage and retrieval system
  - Files stored and replicated across a distributed network environment, with a P2P query and data access system
  - Ian Clarke, et. al. 2000
- Motivation
  - Anonymity for both 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

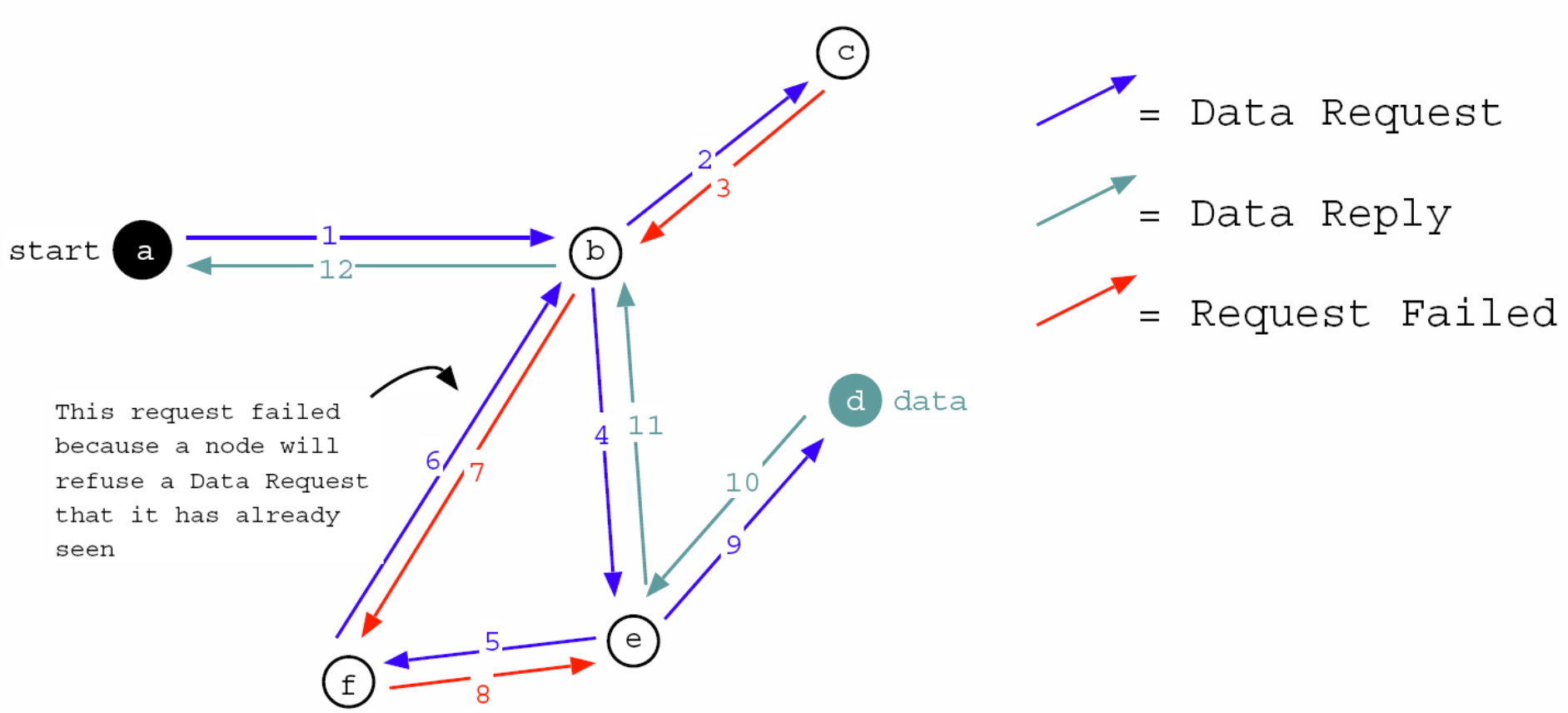
Freenet: Overview

- Unstructured organization
  - A node joins the network by discovering the address of one or more existing nodes through out-of-band means
  - New nodes must somehow announce their presence
- Lexicographically closest match of file hash key
  - When a node receives a request, it looks up the nearest key in its routing table and forwards the request to the corresponding node if not found the requested file locally
- Storing data
  - To insert a new file, a query for the key is invoked to establish an initial query path (bounded by hops-to-live)
  - The user then sends the file to insert, propagated along the above path

Freenet: File Identification and Matching

- Files are identified by a hash key (160-bit SHA-1). There are three types of file key
  - keyword-signed key (KSK): derived from a short descriptive text string chosen by the user when storing a file
    - i.e. john-wang/mp3-music/eric/the-world
  - signed-subspace key (SSK): A public namespace key and KSK are hashed independently, XOR'ed together, and then hashed again to yield the file key
    - A user creates a namespace by randomly generating a public / private key pair to identify her namespace
  - content-hash key (CHK): derived by directly hashing the contents of the corresponding file

Freenet: A Typical Request Sequence



Freenet: Summary

- Personal keys and file hash keys are used to protect privacy and security
- *Depth-first* exhaustive search is performed to reduce message-flooding overhead
- New files are replicated explicitly as they are inserted
- *Store-and-forward* file transfer to support anonymity

Cooperative File System: Introduction

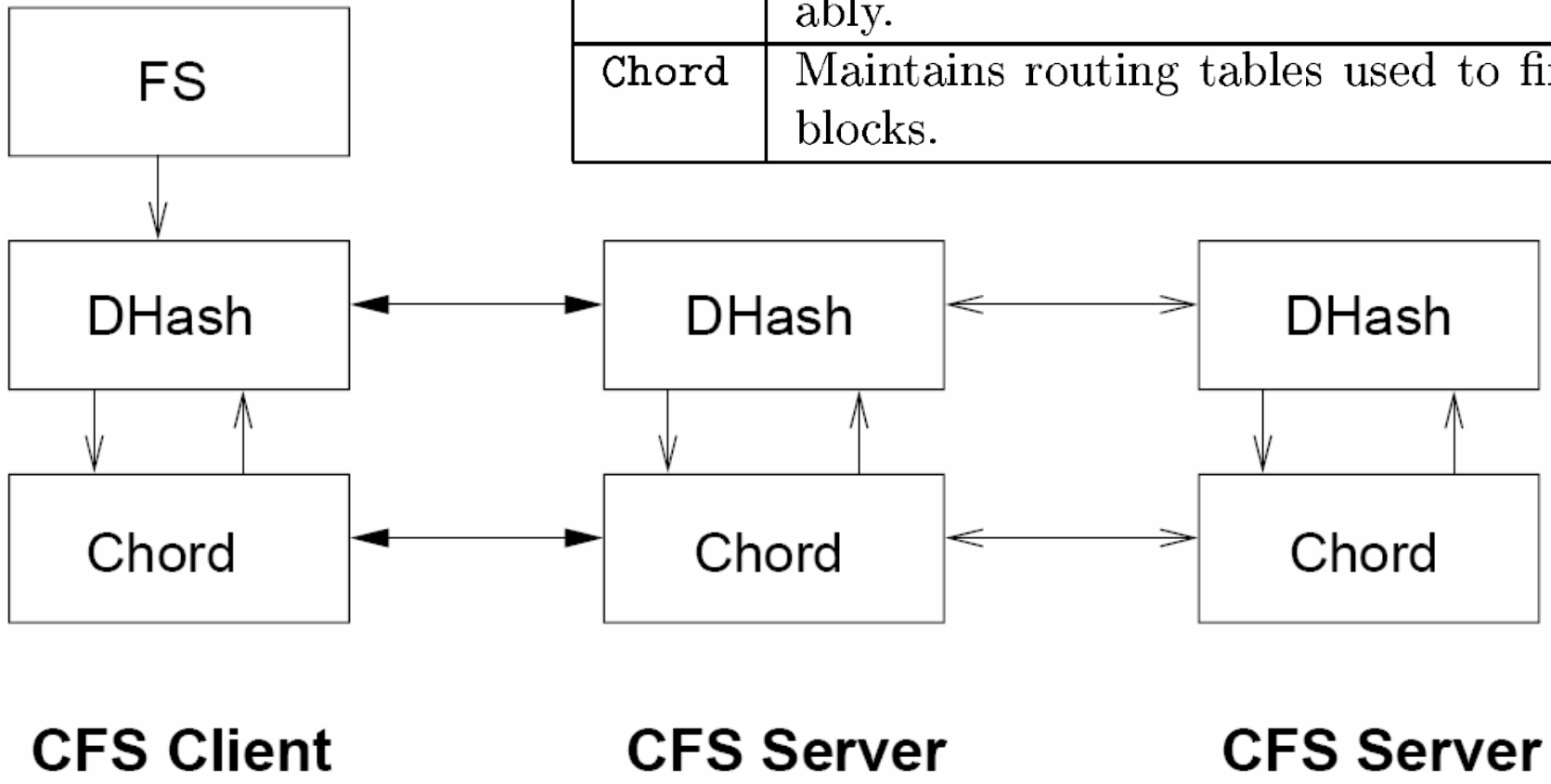
- CFS: wide-area cooperative storage
  - Peer-to-peer read-only storage system
  - The file system is designed as a set of blocks distributed over the CFS servers
- Provide provable guarantees for efficiency, robustness, and load-balance of file storage and retrieval
  - Distribute and cache blocks at a fine granularity to achieve load balance
  - Use replication for robustness
  - Decrease latency with server selection

CFS: Overview

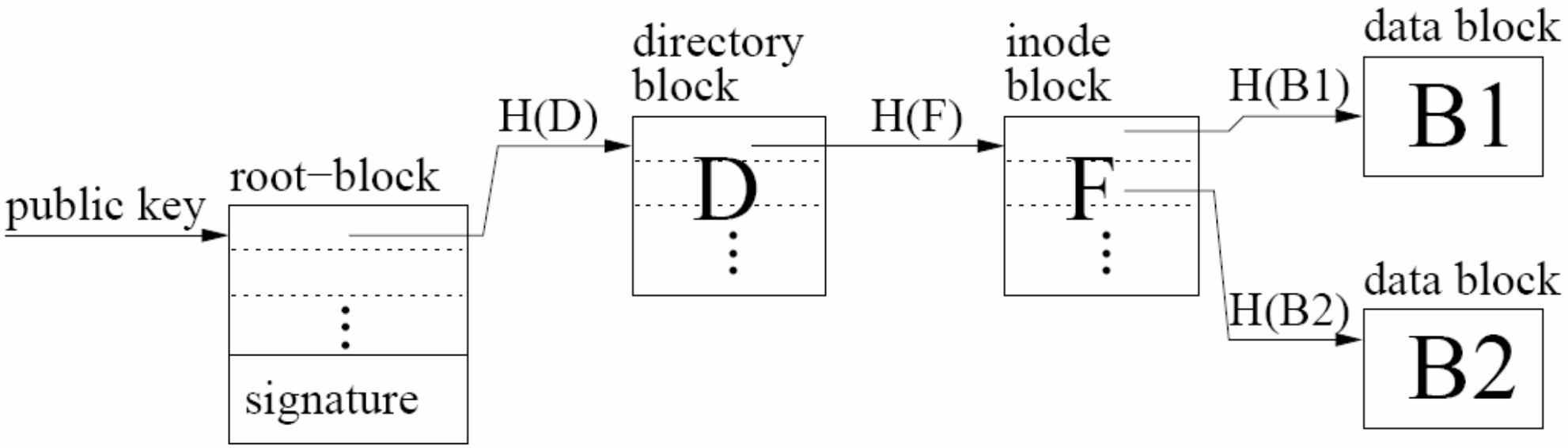
- Two core layers of CFS
  - DHash (distributed hash table)
    - Perform block fetches for the client
    - Distribute the blocks among the servers
    - Maintain cached and replicated copies
  - Chord
    - Support distributed lookup system to locate the servers responsible for a block
- Basic operations
  - DHash maps from block identifiers to servers
  - Chord takes a block id and yield the block successor, the server whose ID most closely follows the block ID on the identifier circle

CFS: Software Architecture

Layer	Responsibility
FS	Interprets blocks as files; presents a file system interface to applications.
DHash	Stores unstructured data blocks reliably.
Chord	Maintains routing tables used to find blocks.



CFS: A File System Structure Example



The root-block is identified by a public key and signed by the corresponding private key. The other blocks are identified by cryptographic hashes of their contents.

CFS: Summary

- A highly scalable, available and secure read-only file system
  - Use Chord lookup protocol to map blocks to servers dynamic and implicit
    - Servers store uninterpreted blocks of data with unique identifiers
    - Clients retrieve blocks from the servers and interpret them as file systems
- Using replication and cache to achieve availability and load balance
  - Replicate a block among consecutive servers
  - Cache a block along the lookup path

PAST: Introduction

- A large-scale, persistent storage utility to provide scalability, high availability, persistence and security
  - Built on top of the Pastry lookup system
    - Storage nodes and files are each assigned uniformly distributed identifiers
  - Multiple replicas of files: stored at nodes whose identifier matches most closely the file's identifier
  - Cache additional copies of popular files
    - Any node can retain an additional copy of a file
    - Use 'unused' portion of their advertised disk space to cache files that can be evicted and discarded at any time

PAST: Overview

- A file is inserted on the k nodes whose nodeIds are numerically closest to the FileId
  - Each node is assigned a 128-bit nodeId
  - Each file has a 160-bit FileId that is a SHA-1 hash of the file name and the public key of the client
  - Insertion process is like the lookup process of Pastry
- To retrieve a file, a client uses its FileId, and in some cases, the decryption key
  - Route messages to the numerically closest node for a given FileId

PAST: State Example of a Pastry Node

NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

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

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

State of a hypothetical Pastry node with nodeId 10233102, *b* = 2, and *l* = 8. All numbers are in base 4. The top row of the routing table represents level zero. The shaded cell at each level of the routing table shows the corresponding digit of the present node's nodeId. The nodeIds in each entry have been split to show the *common prefix with 10233102* - *next digit* - *rest of nodeId*.

PAST: Storage Management

- Replica Diversion
  - To balance the remaining free storage space among the nodes in a leaf set
  - If a node A cannot accommodate a copy locally, it chooses a node B in its leaf set to store a copy on its behalf
    - B is not among the k closest and does not already hold a diverted replica of the file
    - A enters an entry for the file in its table with a pointer to B
- File Diversion
  - To balance the remaining free storage space among different portions of the nodeId space in PAST
  - The client node generates a new FileId using a different salt value when a file insert operation fails
    - A file insert operation fails if the k numerically-closest nodes could not accommodate the file nor divert the replicas locally within their leaf set
    - Retries the insert operation for up to three times

PAST: Summary

- An Internet-based global P2P storage utility
  - Pastry ensures that client requests are reliably routed to the appropriate nodes
  - Replicas of a file are stored at the k nodes whose nodeIds are numerically closest to the file's fileId
- Balance the remaining free storage space
  - Replica diversion: among the nodes in a leaf set
  - file diversion: among different portions of the nodeId space

OceanStore: Introduction

- A utility infrastructure designed to
  - Span the globe
  - Provide continuous access to persistent information

  Developed by University of California, Berkeley
- Two unique goals
  - Untrusted infrastructure: servers may crash without warning or leak information to third parties
    - Data is protected through redundancy and cryptographic
  - Nomadic data: data allowed to flow freely
    - Promiscuous caching: data can be cached anywhere, anytime
    - Introspective monitoring: discover tacit relationships between objects

OceanStore: Overview

- Persistent object: fundamental unit
  - Objects can be directories, files, fragments
  - Named by a globally unique identifier (GUID)
    - Secure hash of the owner's key and some human-readable name
    - A pseudo-random, fixed-length bit string
  - Replicated and stored on multiple nodes
- Deep archival storage: objects exist in both active and archival forms
  - active: the latest version of its data together with a handle for update
  - Archival: a permanent, read-only version of the object
    - Encoded with an erasure code and spread over hundreds of thousands of servers

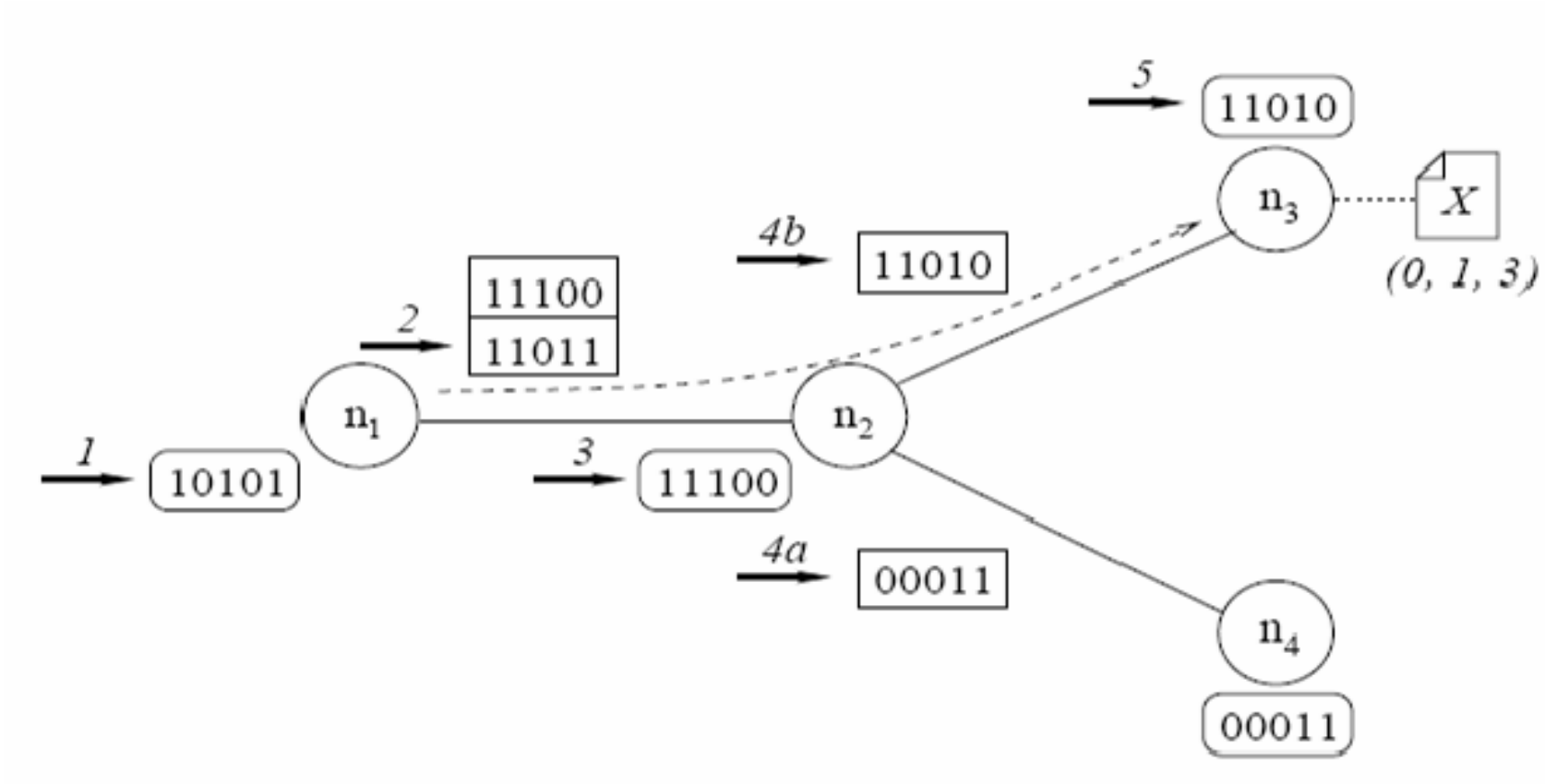
OceanStore: Two-levels of Routing

1. Perform a fast, probabilistic algorithm first
   - probabilistic search for "routing cache"
   - Built from *attenuated* bloom filters
   - Approximation to gradient search
2. Backup with a slower, global deterministic algorithm
   - Redundant *Plaxton Mesh* used for underlying routing infrastructure - Tapestry
   - Randomized data structure with locality properties

OceanStore: Probabilistic Routing

- The probabilistic algorithm is fully distributed and uses a constant amount of storage per server
  - If a query cannot be satisfied by a server, local information is used to route the query to a likely neighbor
  - An attenuated Bloom filter is used to implement this function
- An attenuated Bloom filter of depth ***D*** can be viewed as an array of ***D*** normal Bloom filters
  - The first Bloom filter is a record of the objects contained locally on the current node
  - The *i*th Bloom filter is the union of all Bloom filters for all of the nodes a distance *i* through any path from the current node

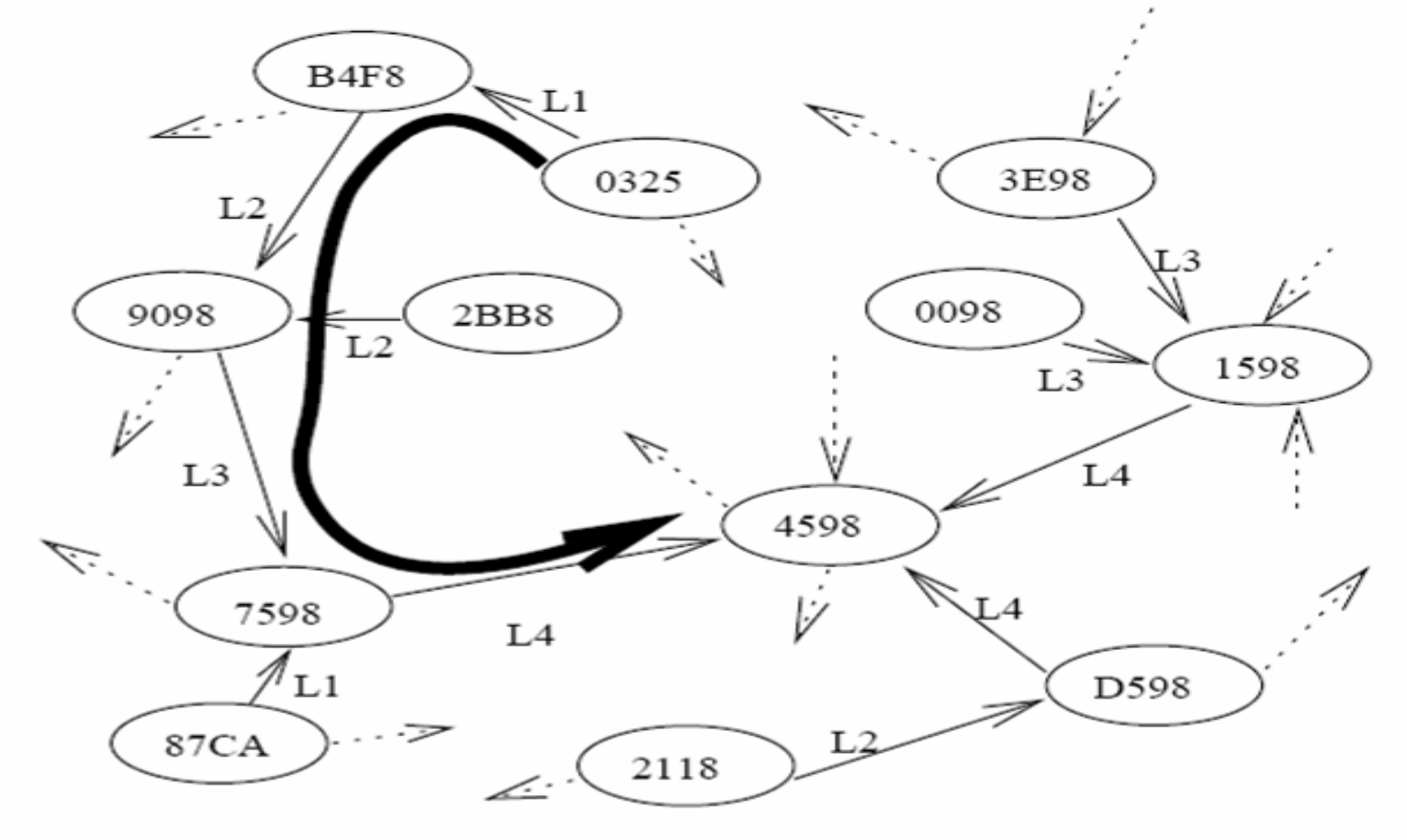
OceanStore: Probabilistic Query Example



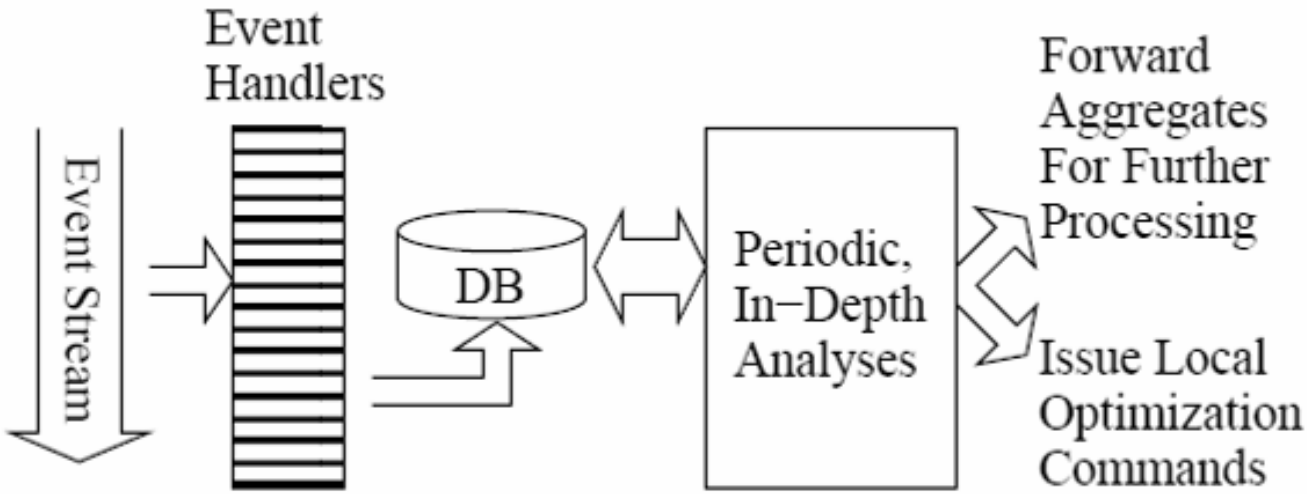
OceanStore: Tapestry Routing

- A variation on Plaxton randomized hierarchical distributed data structure
  - Scheme to directly map GUIDs to root node IDs
  - Replicas publish toward a document root
  - Search walks toward root until pointer located⇒*locality!*
- OceanStore enhancements for reliability
  - Documents have multiple roots
  - Each node has multiple neighbor links
  - Searches proceed along multiple paths
- Dynamic node insertion and deletion algorithms
  - Continuous repair and incremental optimization of links

OceanStore: Deterministic Query Example



OceanStore: Introspective Architecture



- A level of fast event handlers summarizes local events
  - Process local events
    - Events include any incoming message or noteworthy physical measurement
  - Forward summaries up a distributed hierarchy to form approximate global views of the system

OceanStore: Summary

- The rise of ubiquitous computing has spawned an urgent need for persistent information
- OceanStore is a utility infrastructure designed to span the globe and provide secure, highly available access to persistent objects
- Several properties distinguish OceanStore from other systems: the utility model, the untrusted infrastructure, support for truly nomadic data, and use of introspection to enhance performance and maintainability

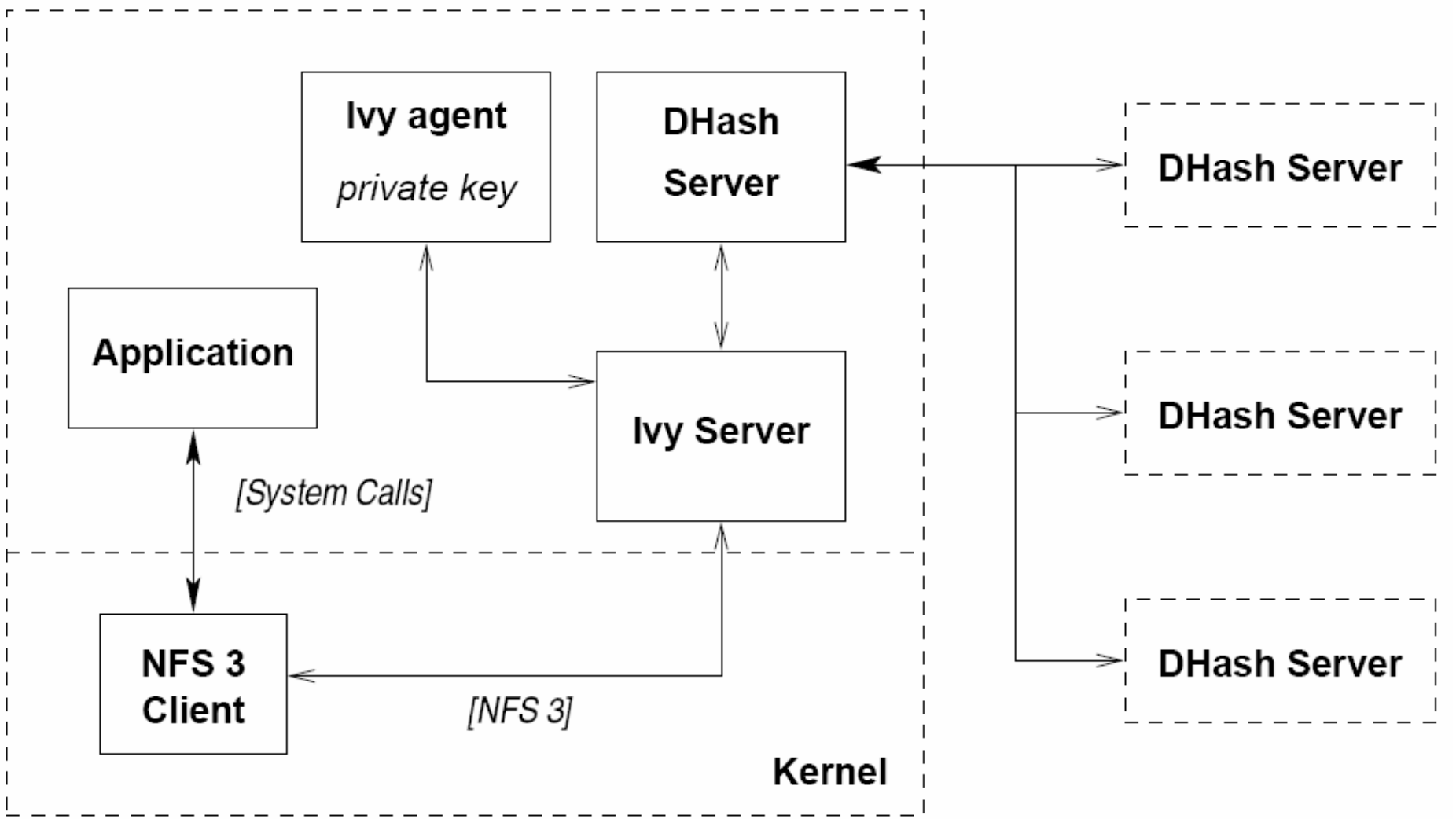
Ivy: Introduction

- A multi-user read/write peer-to-peer file system
  - Provide an NFS-like file system view
  - No centralized or dedicated components
- An Ivy file system consists solely of a set of logs
  - Each participant has its own log
  - Logs are stored in DHash distributed hash table
- Provide convention file system interface
  - Each participant finds data by consulting all logs
  - A participant modifies data by appending only to its own log
  - Snapshot mechanisms prevent scanning of all but the most recent log

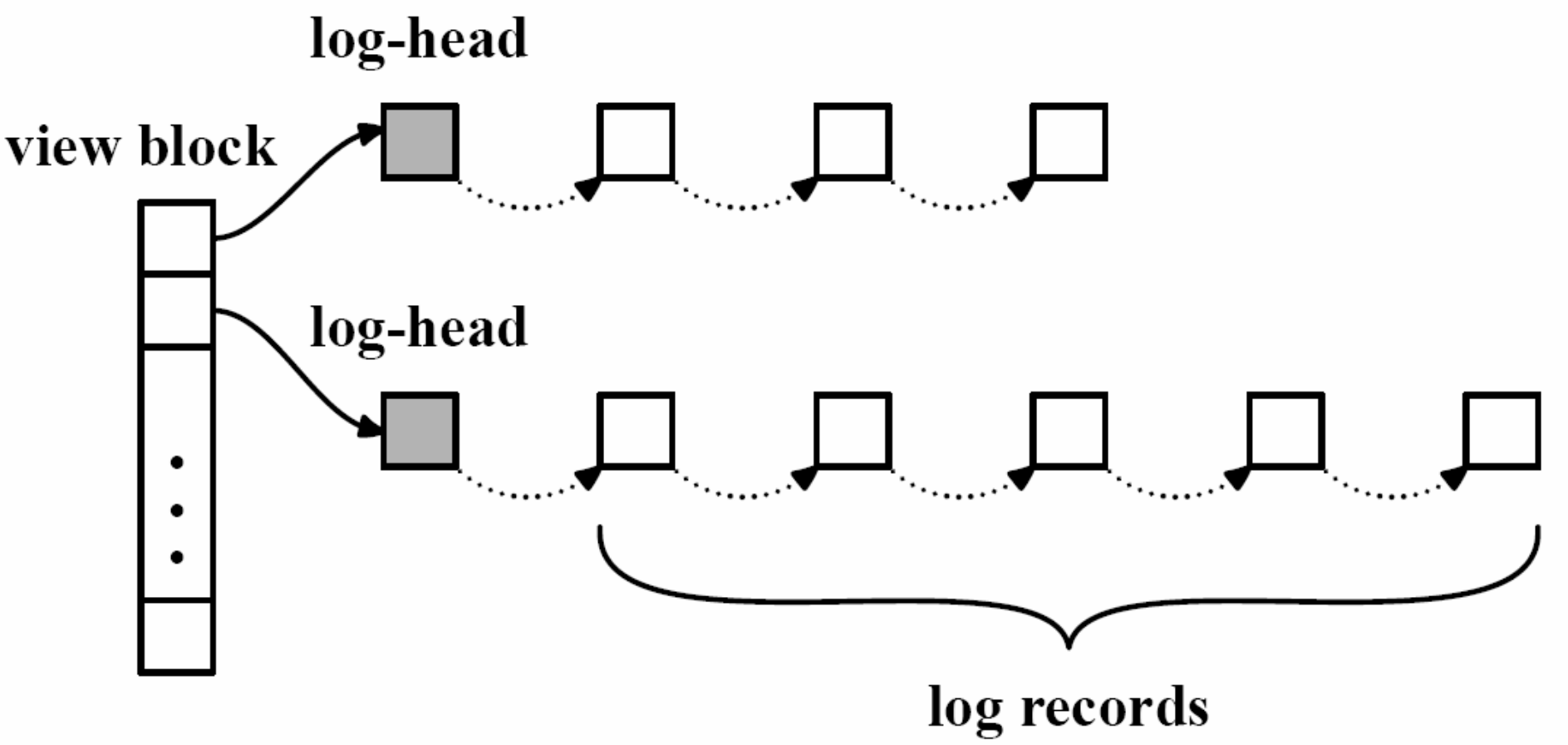
Ivy: Overview

- One log per participant to record the changes made to the system
  - A log contains all of one participant's changes to file system data and meta-data
- Each participant appends only to its own log, but reads from all logs
  - The logs are themselves stored in DHash
  - Provide per-record replication and authentication
- Periodically, each participant takes snapshots
  - A snapshot summarizes the file system state as of a recent point in time
  - Avoid future repeat scanning of the entire log

Ivy: Software Architecture



Ivy: View and Log Example



White boxes are DHash content-hash blocks; gray boxes are public-key blocks

Ivy: Summary

- Operate in a relatively open P2P environment
  - It does not require participants to trust each other
  - Underlying DHT stores and retrieve logs efficiently
- Use of per-participant logs
  - Avoid the need for locking to maintain integrity of Ivy meta-data
  - Allow Ivy users to choose which other participants to trust
- Periodically take snapshots of the file system to minimize time spent reading the logs

Further Issues

- Performance guarantee
- Accounting and quota
- Malicious users
- Robustness
- Deployment