

Block-Level Security for Network-Attached Disks

Authors: Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick,
and Erwin Oertli (*Hewlett-Packard Labs*);

Dave Andersen (*Massachusetts Institute of Technology*);

Mike Burrows (*Microsoft Research*)

Timothy Mann (*VMware*)

and Chandramohan A. Thekkath (*Microsoft Research*)

Published in 2003

Presented by: Adam O'Sullivan

Introduction

Network Attached Disks (NADs):

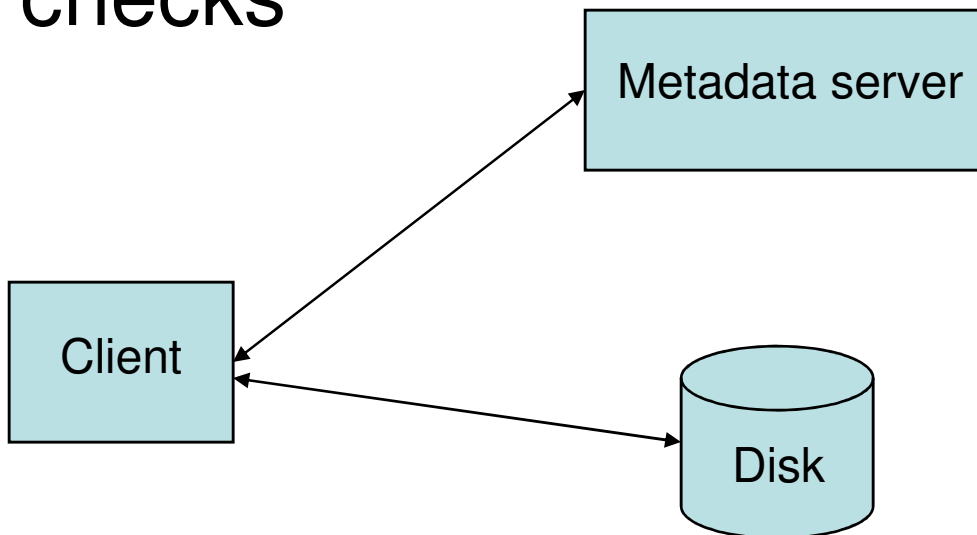
- Storage devices that accept block read / write requests over network
- Provide no support for security (honor any request)
- Often used as part of a Storage Area Network (SAN)

Goals

- Add simple block-level security to NADs
- No changes to data layout on disk
- Minimal changes to existing NADs
- Small changes to standard protocol

Overview

- Use a central “Metadata server” to issue and manage capabilities
- NADs only have to perform simple access checks



Access Control Matrix:

Objects

	File 1	File 2	File 3	...	File n
User 1	R,W	R,W	R		R,W
User 2		R,W			R
User 3			R		
...					
User m		R			R

Subjects

Access Control List:

Store **columns**
of the matrix with
associated
Objects (Files)

Access Control List (ACL)

	File 1	File 2	File 3	...	File n
User 1	R,W	R,W	R		R,W
User 2		R,W			R
User 3			R		
...					
User m		R			R

Capabilities:

Store **rows** of the matrix with associated **Subjects (Users)**

Capability {

	File 1	File 2	File 3	...	File n
User 1	R,W	R,W	R		R,W
User 2		R,W			R
User 3			R		
...					
User m		R			R

Assumptions

- Server and disks can be trusted (these set and enforce security policies)
- Network is vulnerable to eavesdropping and spoofing

Capabilities

- Naïve approach: One capability per block
- Instead: Each capability specifies access to **ranges** of blocks
- **Accessible blocks** and **access mode** stored as part of each **capability**

Capabilities

Composed of two parts:

- Self describing certificate
- Associated secret

Self descriptive part specifies **type of access** to certain **parts of a disk**

Group ID	Capability ID	Disk ID	Extents	Mode
22	123	1	0-200, 400-410	Read

Capabilities

Secret:

Used to prevent:

- Forgeries of illegal capabilities
- Illegal requests using legal capabilities

Capabilities

Secret:

Generated using keyed-hash message authentication code (MAC)

$$\mathbf{mac} = h(\text{data}, \text{key})$$

Unforgeability property: without knowing the key, infeasible to find any new pair of (mac, data) such that $\text{mac} = h(\text{data}, \text{key})$

MACs Can be computed efficiently in practice

Quick Recap

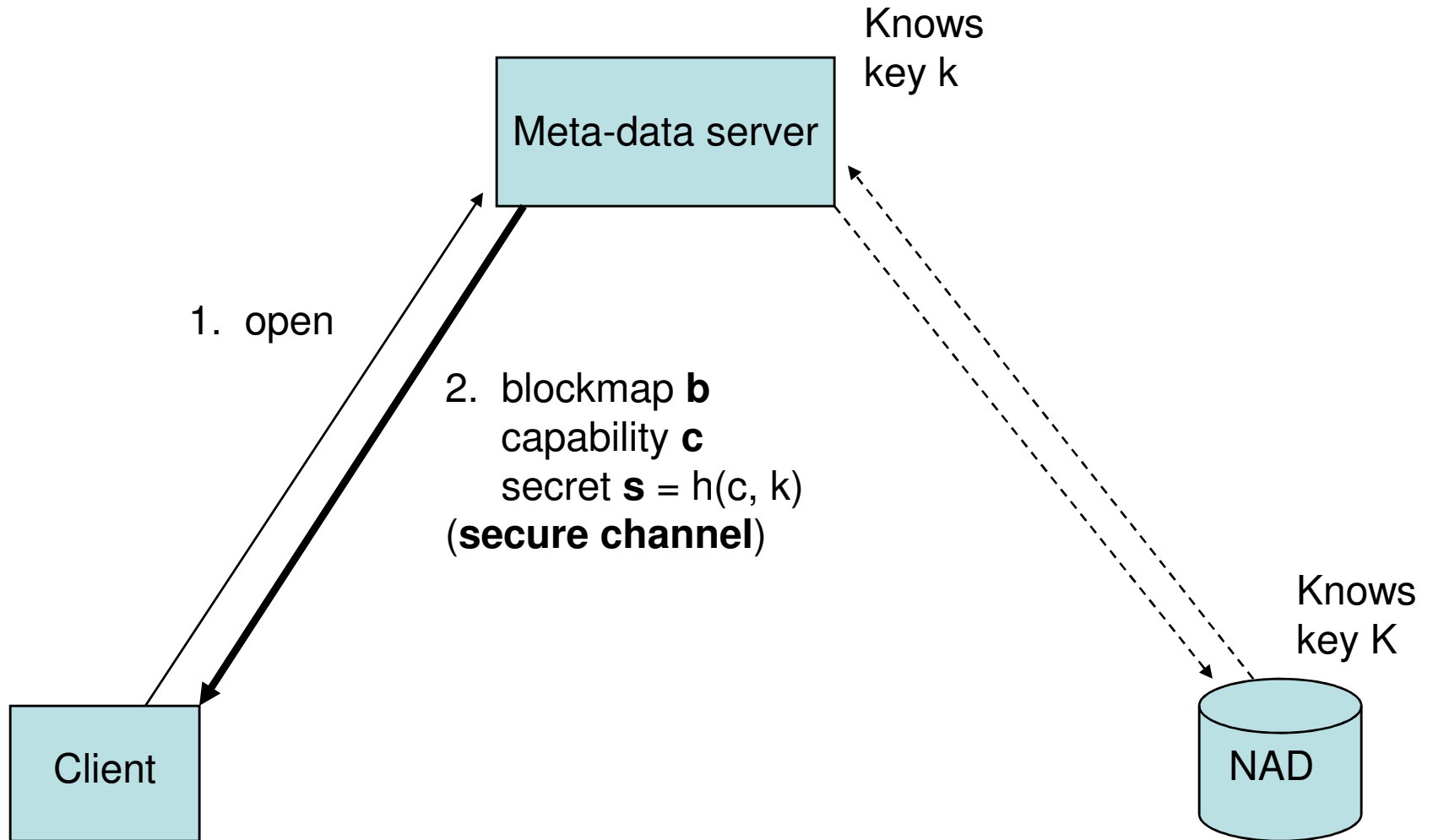
c =

Group ID	Capability ID	Disk ID	Extents	Mode
22	123	1	0-200, 400-410	Read

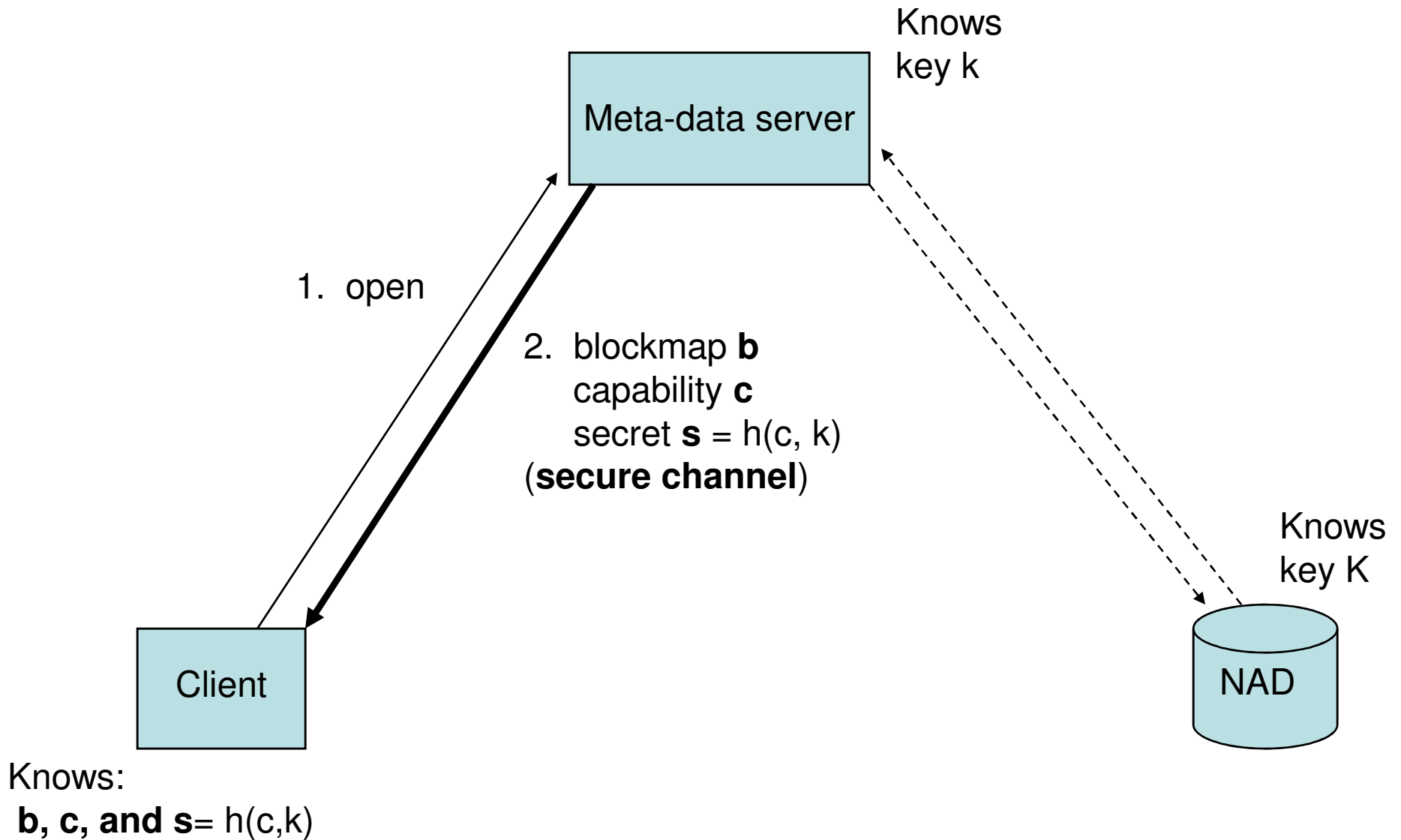
secret $s = h(c, k)$

k = key shared by **meta-data server** and the **disk** (specified by Disk ID in c)

Example



Example



Next

To read or write data on the disk, client sends 3 things:

- Operation op = type of access
range of blocks
data to be written (if writing)
- Capability c (provided by meta-data server)
- MAC $m = h(op, s) = h(op, h(c, k))$

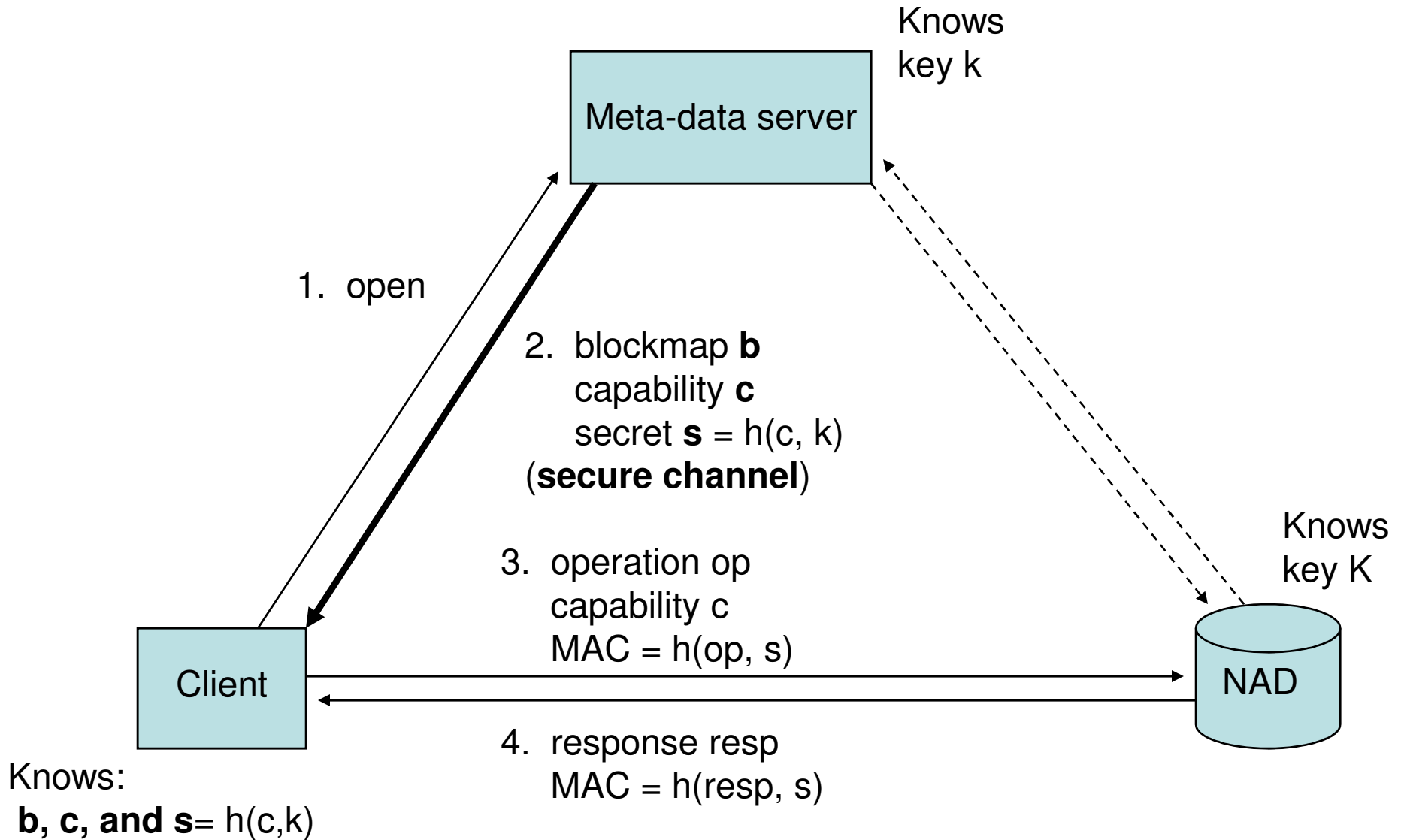
Double MAC

$$\text{MAC } m = h(\text{op}, s) = h(\text{op}, h(c, k))$$

Serves a double purpose:

- 1.) Proof to NAD that client knows s (and has been authorized to use c to issue op)
- 2.) Prevents op from being tampered with (if op is modified, MAC m will verify)

Example



Revoking Capabilities

- Required when client should no longer have the access granted by previous capability
- Examples:
 - Change in file permissions
 - File truncated or deleted

Revoking Capabilities

- Change the key k ?
 - Revokes all capabilities
- Causes “burstiness problem”

Capability Groups

- Place capabilities into groups, and invalidate groups when the system needs to recycle IDs
- Avoids burstiness problem –only a small fraction of capabilities is revoked when the system runs out of IDs.

Revocations

Group ID		Revoked capability ID's (bitmap)
Index	Counter	
0	10	100010001000 ...
1	5	001111011010 ...
2	14	111011111000 ...
⋮	⋮	⋮
63	3	000011010111 ...

Figure 3: **Keeping track of revocations.** The table used by the disk controller to keep track of revoked capabilities.

Practical Benefit of Capability Groups

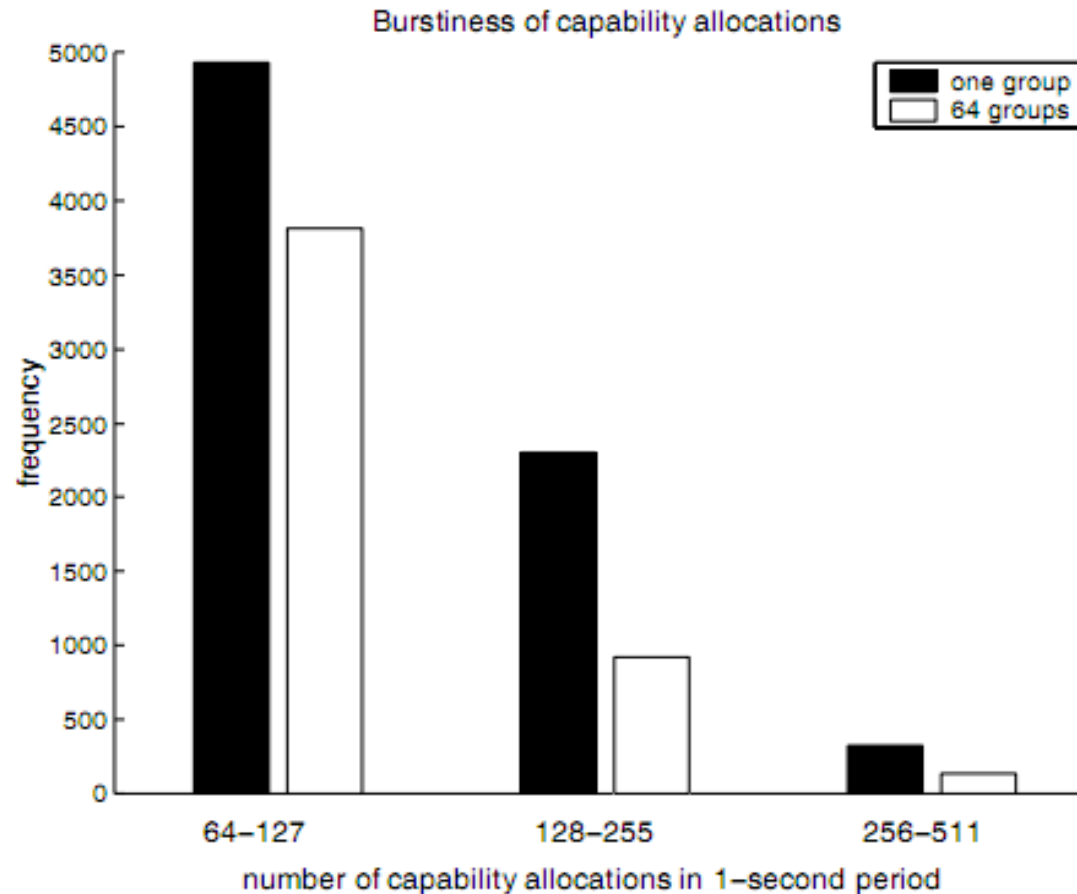


Figure 7: **Using 64 groups instead of 1 substantially reduces the burstiness of capability allocations.**

Preventing Replay Attacks

- While infeasible for adversary to forge new requests, trivial to replay previous requests
- Use a Bloom filter to remember recent requests
 - Low computation and memory cost
- When a request arrives, disk checks to see if the request is already in the filter
 - If “definitely not” disk can proceed
 - Otherwise disk sends replay rejection message
- If client receives replay rejection message, it changes nonce, resends

Preventing Replay Attacks

- Eventually, a filter will begin to fill up
- Determine a filter needs replacement when some proportion of bits are set
- Store several filters, identified by epoch value
- For filter replacement, delete oldest filter, increment epoch, create new empty filter.

Summary of Data Structures

Metadata Server	Client	Disk
Hash table of all currently valid capabilities, indexed by inode number For each disk, a list of valid groups, with the number of valid and revoked capabilities in each	A cache of capabilities issued to this client, that are not known to have been revoked	One counter and bitmap per valid group (64 KB) Bloom filters of recent requests (64 KB)

Additional Disk Functionality

New functionality	Equivalent lines of C
Cryptography	340
Capability groups and revocations	60
Miscellaneous (refresh timer, RPC handlers, logging)	610

Implementation

- Prototype NAD file system called *Snapdragon*
- Modified Linux's kernel-based implementation of NFS 2
- New file system code (7,500 lines)
- New disk functionality (1,000 lines)
- Security library (14,000 lines)

Implementation

- Snapdragon clients run 2 kernel modules:
 - A standard NFS lock daemon
 - Module with core filesystem functionality
 - Exports through VFS
- Snapdragon metadata server runs:
 - Filesystem kernel module(snapd)
 - Device driver(rddev)
 - Lock daemon(lockd)

Implementation

- Changes to NFS protocol
 - Snapdragon metadata server implements superset of NFS protocol
 - 3 commands added to NFS protocol:
 - GETCAPS
 - OPEN
 - CLOSE

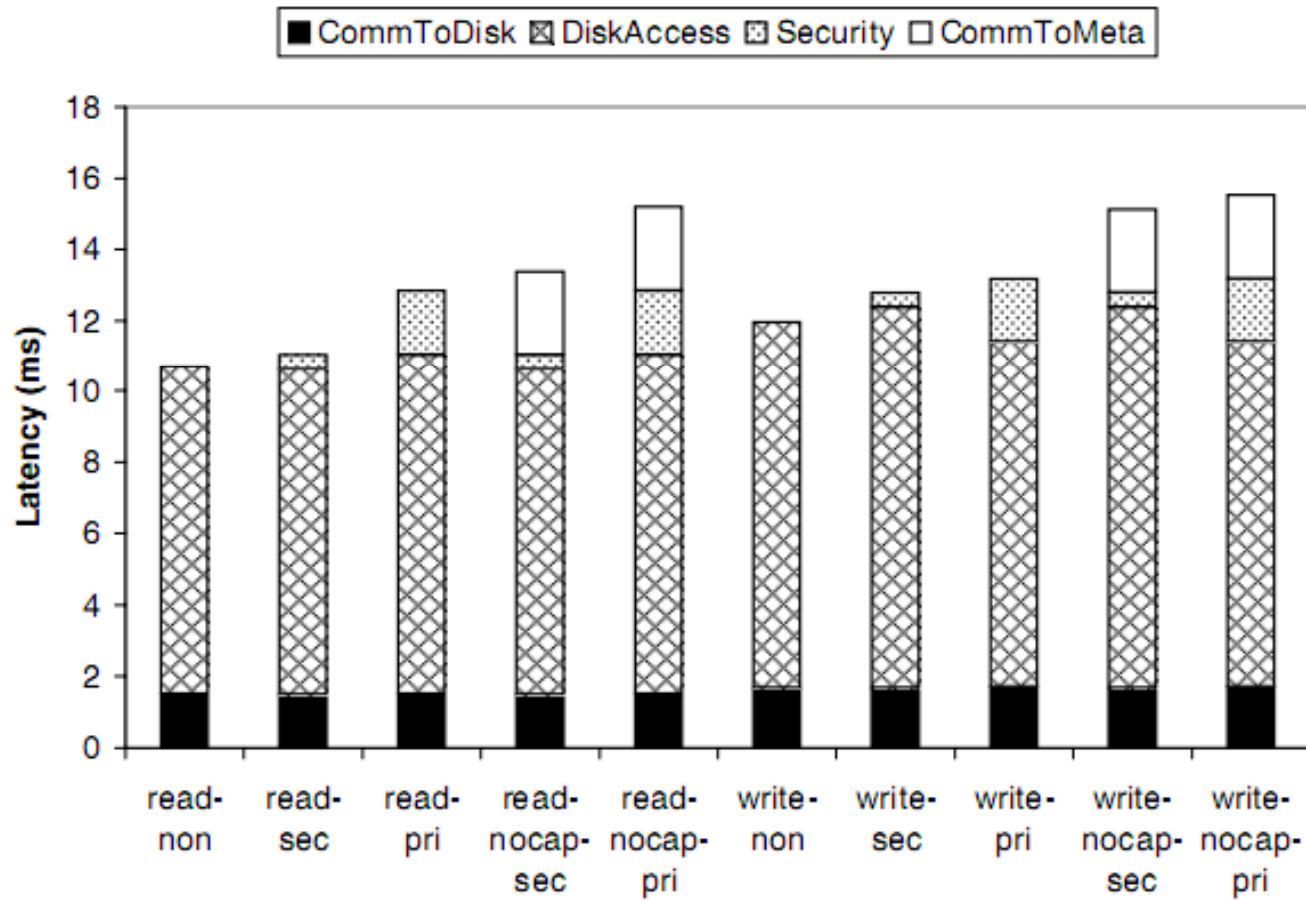
Disk Controller

- Implemented as a PC connected to the network
 - Runs user level program that listens for, checks, and executes block requests

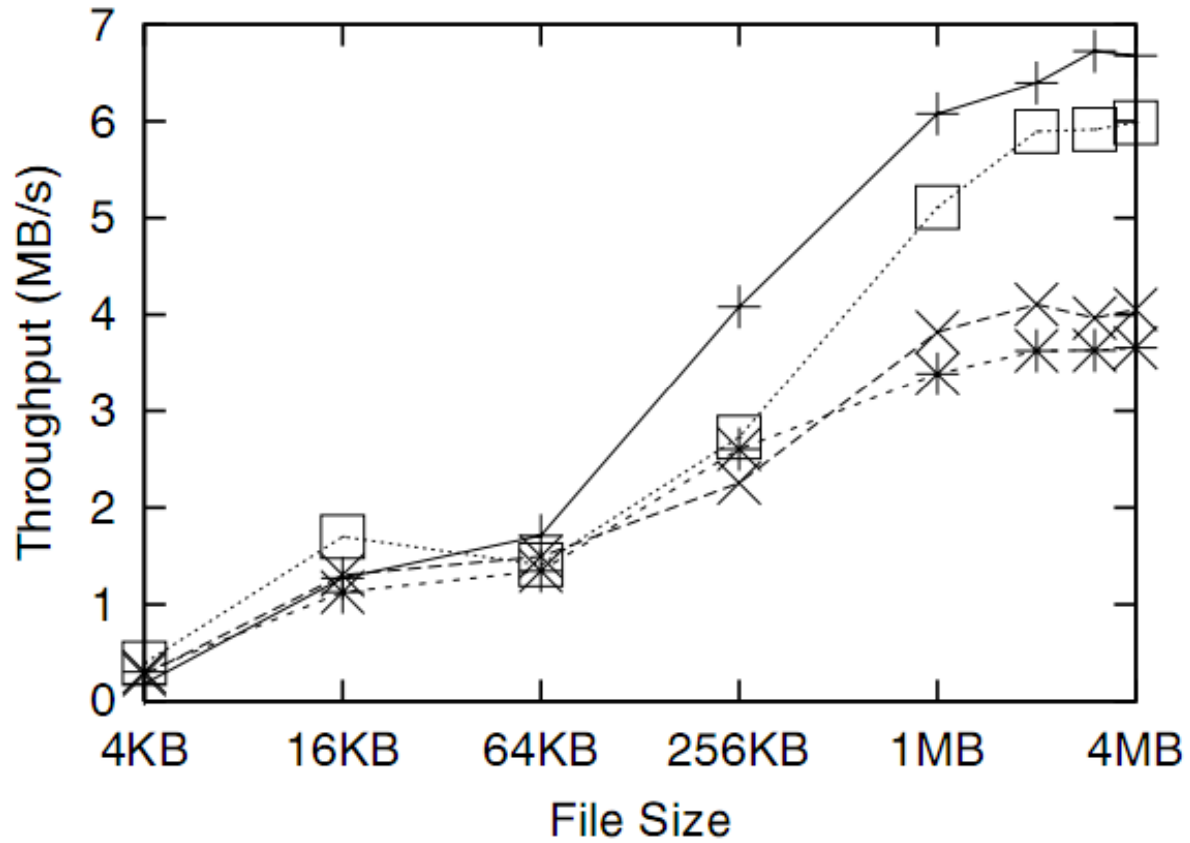
Performance

- Evaluate overhead of security
 - MAC computation, capability revocation, and encryption
- Evaluate system throughput and scalability
 - Under bandwidth intensive workload

Latency for Read and Write



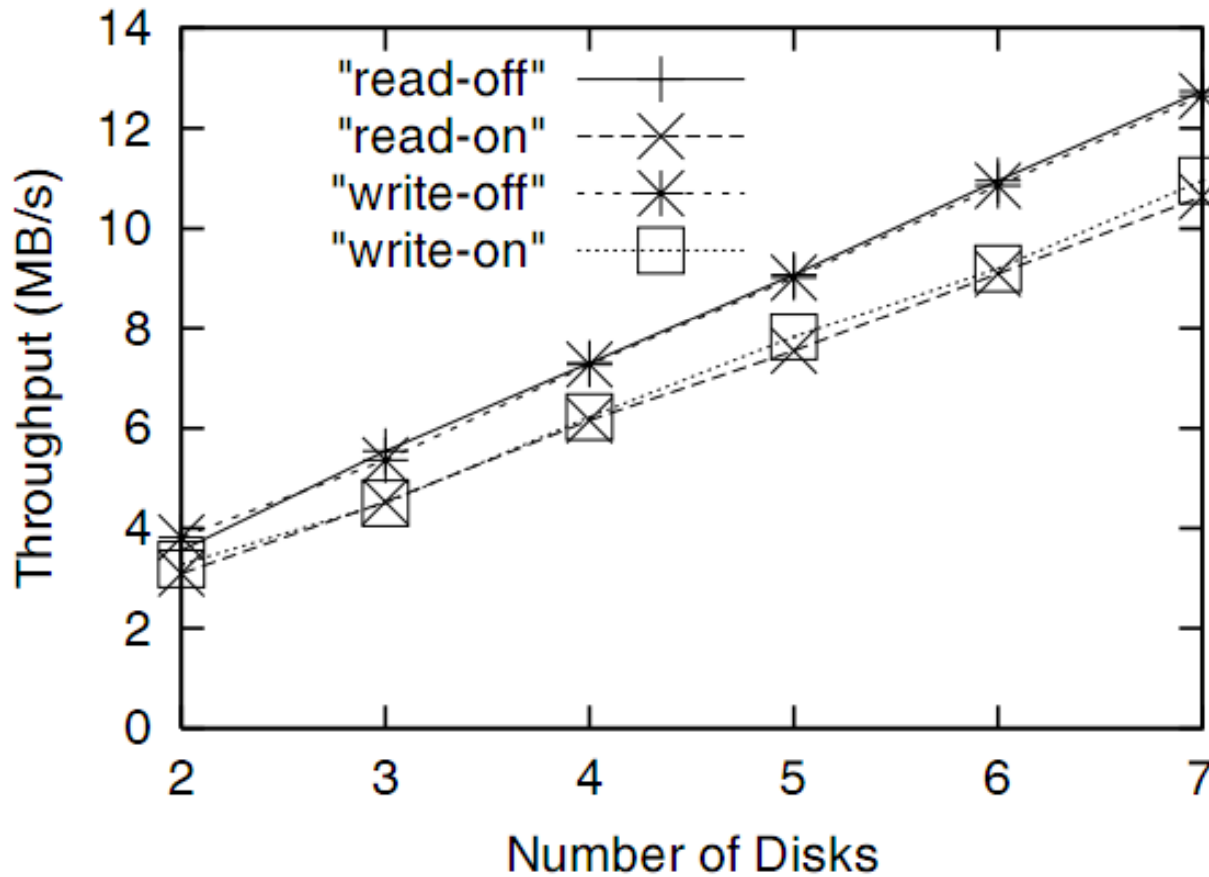
Throughput



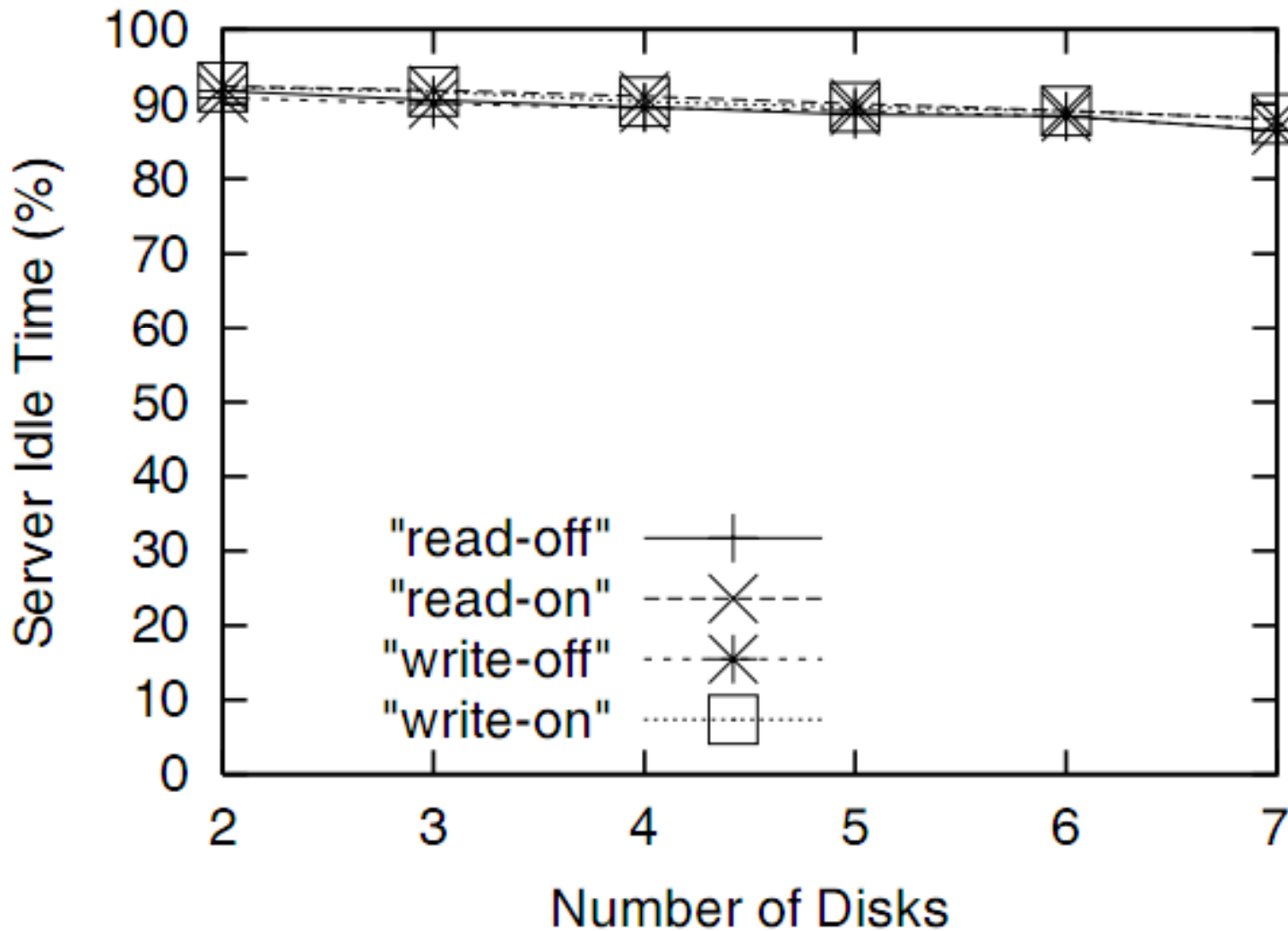
"write-nfs" —+—
"write-off" - -x- -

"write-on" - -*- - -
"write-un" . .□. . .

Aggregate Read/Write Bandwidth with multiple disks



Average Percent Idle CPU Time on Metadata Server



Andrew Benchmark

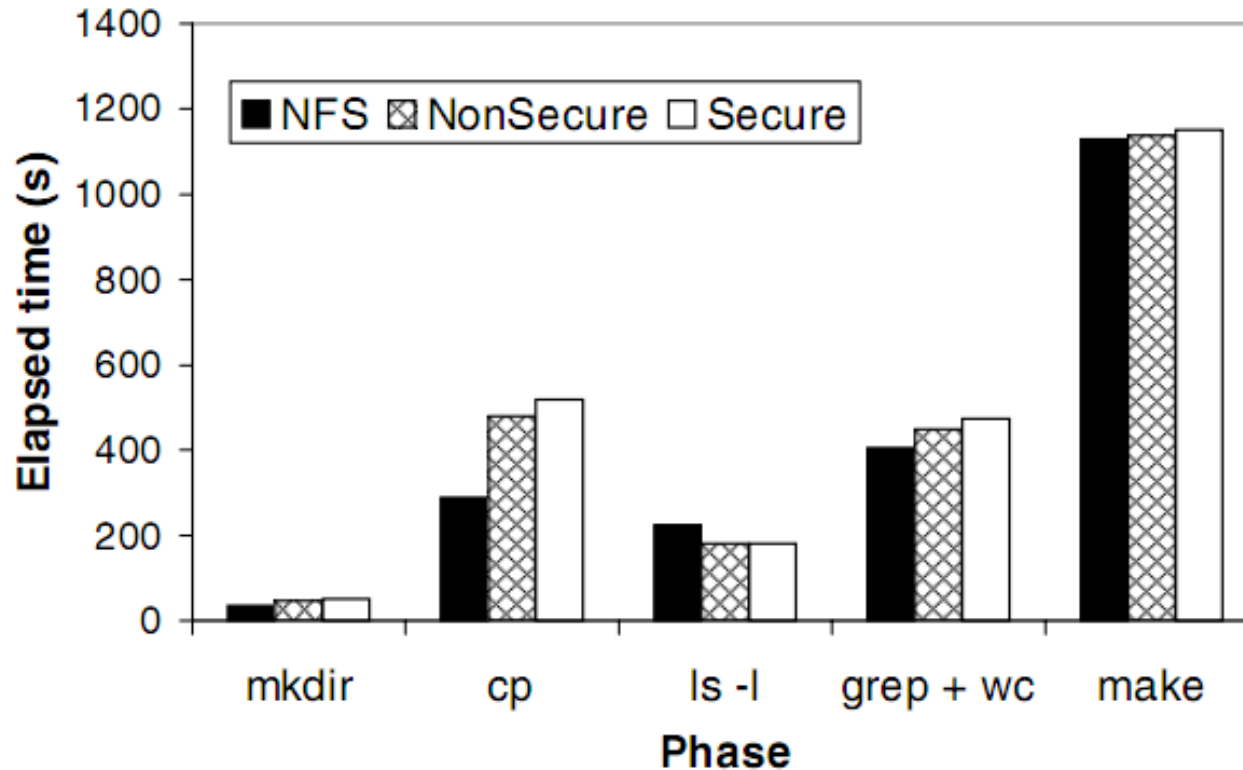


Figure 14: **Andrew benchmark with Linux kernel source.**

Conclusion

- Pros
 - Interesting use of bloom filters and capability groups to offer functionality in a limited memory environment (on NADs)
- Cons
 - Used separate computers with fast processors as disk controllers in experiments
 - Didn't compare against other similar solutions (NAS or another NAD solution), only NFS

Venti: A New Approach to Archival Storage

Sean Quinlan and Sean Dorward
Bell Labs, Lucent Technologies

Presented by: Adam O'Sullivan

Problems:

- Archival storage is a second class citizen
- Data typically stored on tapes
- Usually offline and not accessible for users
- Restoring from tape requires an administrator
- Tapes are slow (sequential access)
- Restoring data can be tedious/error prone

Introduction

- Online block-level network storage system
- Intended for archival data
- Enforces a write-once policy (no deletion or overwrites)
- Building block for a variety of storage applications (backups, snapshot, etc.)
[It does not provide these applications itself]

Write-once policy?

- Hard drives currently inexpensive and keep getting cheaper
- When current storage fills up, simply add / replace hard drives
- Feasible today

Design - Fingerprints

- Identifies blocks by hash of their contents
- Uses collision resistant hash function with large output (authors assume that each block hashes to a unique value)
- Refer to unique hash as *fingerprint* of a block

Design – Fingerprints (cont.)

- As a result, a block cannot be modified without changing its *fingerprint*
- Intrinsically this behavior is write-once
- Writes are idempotent. Multiple writes of same data do not require more storage space

Design – Fingerprints (cont.)

- Reduces redundant data
- Very useful for backup / snapshots, much data is the same between snapshots
- Inherent integrity checking of data
- Using fingerprint as identifier facilitates replication, caching, and load balancing

Hash Function

- SHA-1
- Collision resistant
- Cryptographic hash function
- 160 bit hash value

Hash Function

- Probability of a collision:

$$p \leq \frac{n(n-1)}{2} \times \frac{1}{2^b}$$

n = number of blocks, b = number of bits generated by hash function

Example:

A system with an exabyte (10^{18} bytes) stored as 8 Kbyte blocks ($\sim 10^{14}$ blocks)

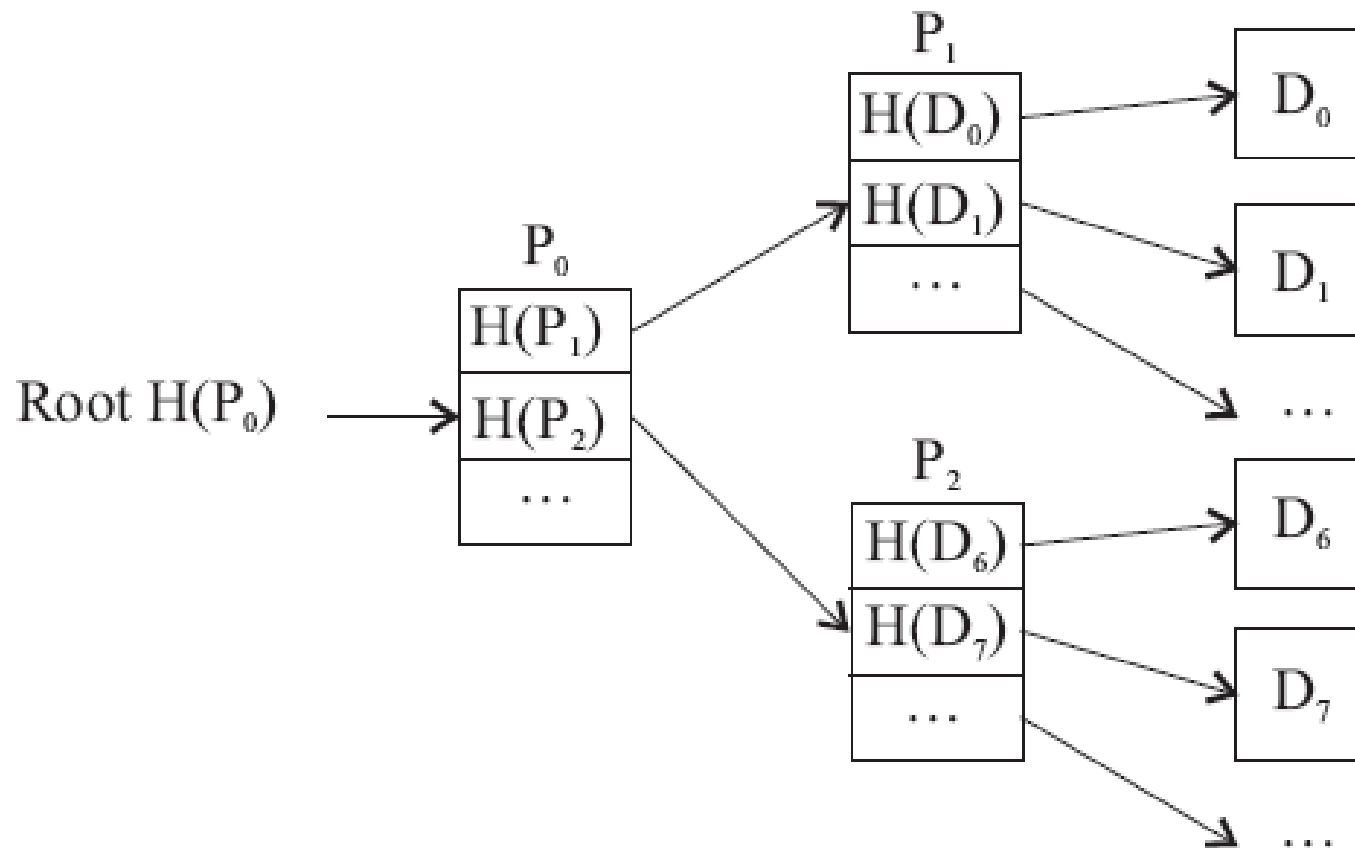
Using SHA-1: The probability of a collision is less than 10^{-20}

Choice of Storage Technology

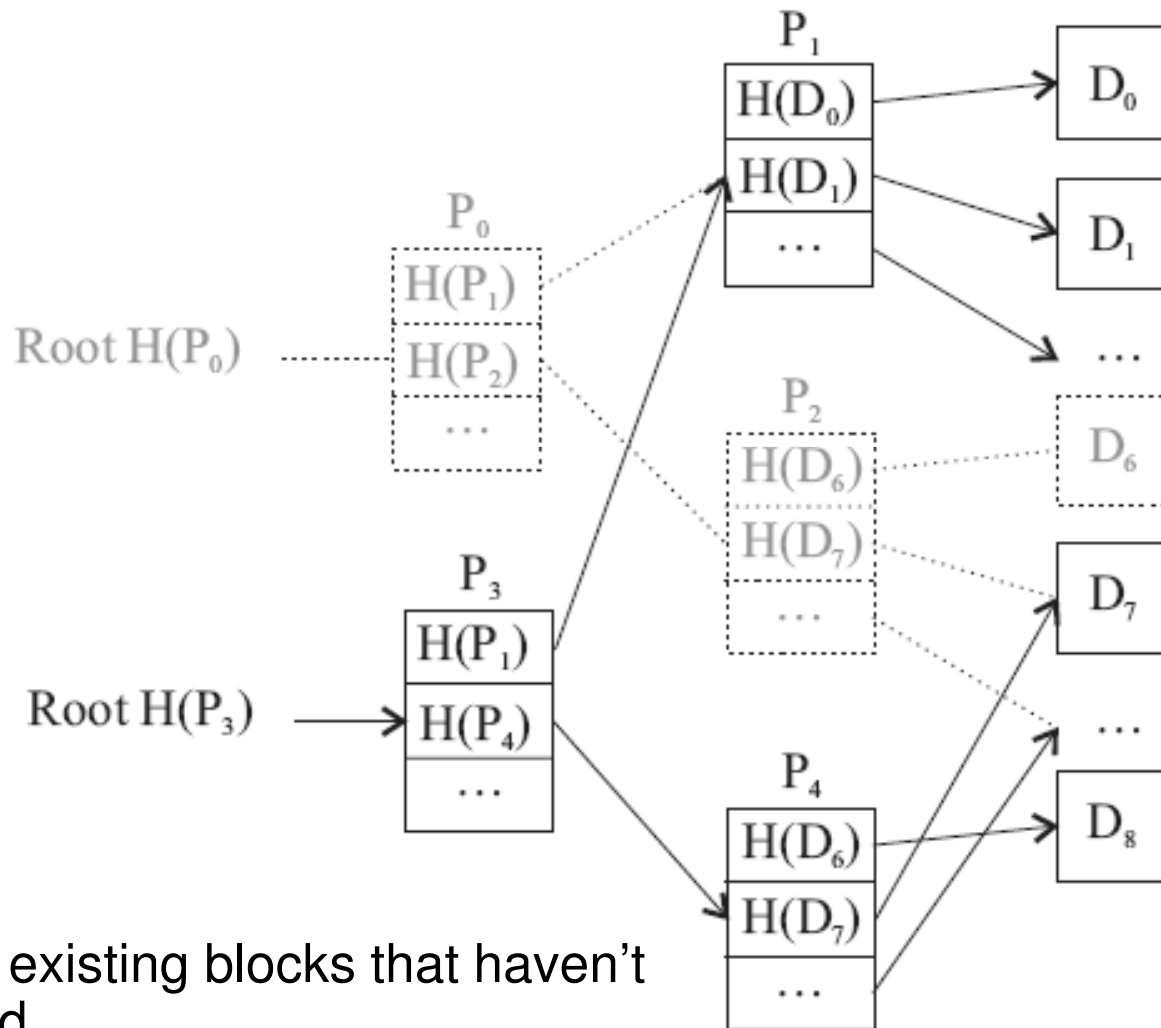
- Use magnetic hard disks
- Cost effective today
- Can use RAID for reliability
- Archival storage speeds now comparable to primary storage
- Archives are easily accessible

- To enable applications to retrieve data, they must record the fingerprints of blocks
- One approach stores fingerprints in additional blocks, called *pointer blocks*
- Repeat recursively until single fingerprint is obtained (root of the tree)

Example Tree Structure



Example: How to update?



- Re-use existing blocks that haven't changed

Example Applications

- VAC – user level archive utility
- Physical level backup utility
- New version of Plan 9

Vac

- Stores a collection of files and directories as a single object (similar to tar or zip)
- Contents of each file stored as a separate tree of blocks in Venti
- Root fingerprint for tree is stored in a 45 byte Vac archive file for the user
- Corresponding utility unvac restores files from a Vac archive

Application: Physical Backup

- This approach is block level / physical backup
- File system disk blocks are copied directly to Venti without interpretation
- Coalesce duplicate blocks
- Can represent free blocks using Null value
- User still sees a full backup
- Currently only a concept

Plan 9 File system

- Old version used magnetic disks and write-once “optical jukebox”
- Use Venti as the primary location for data (when combined with a small amount of read/write storage)
- New version of Plan 9 file system

Implementation

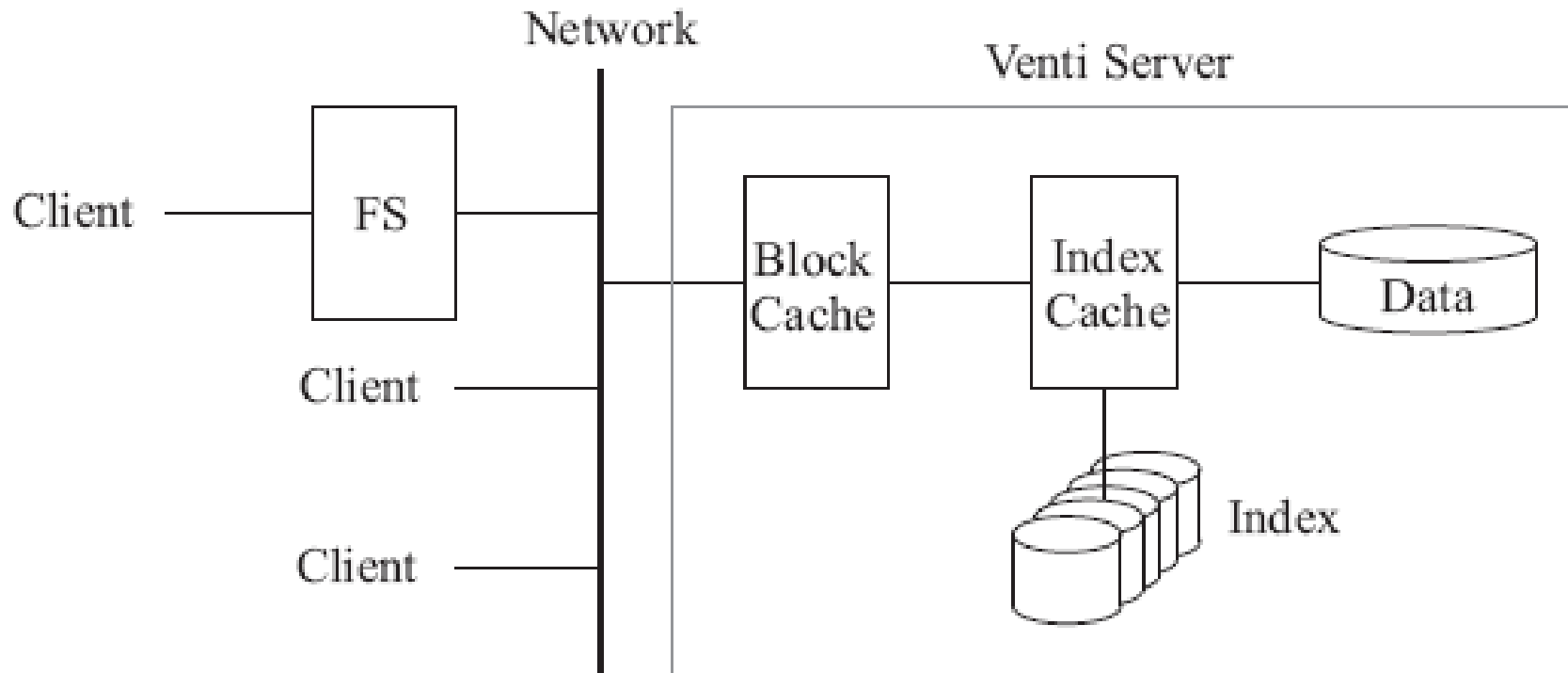


Figure 3. A block diagram of the Venti prototype.

Data Log Format

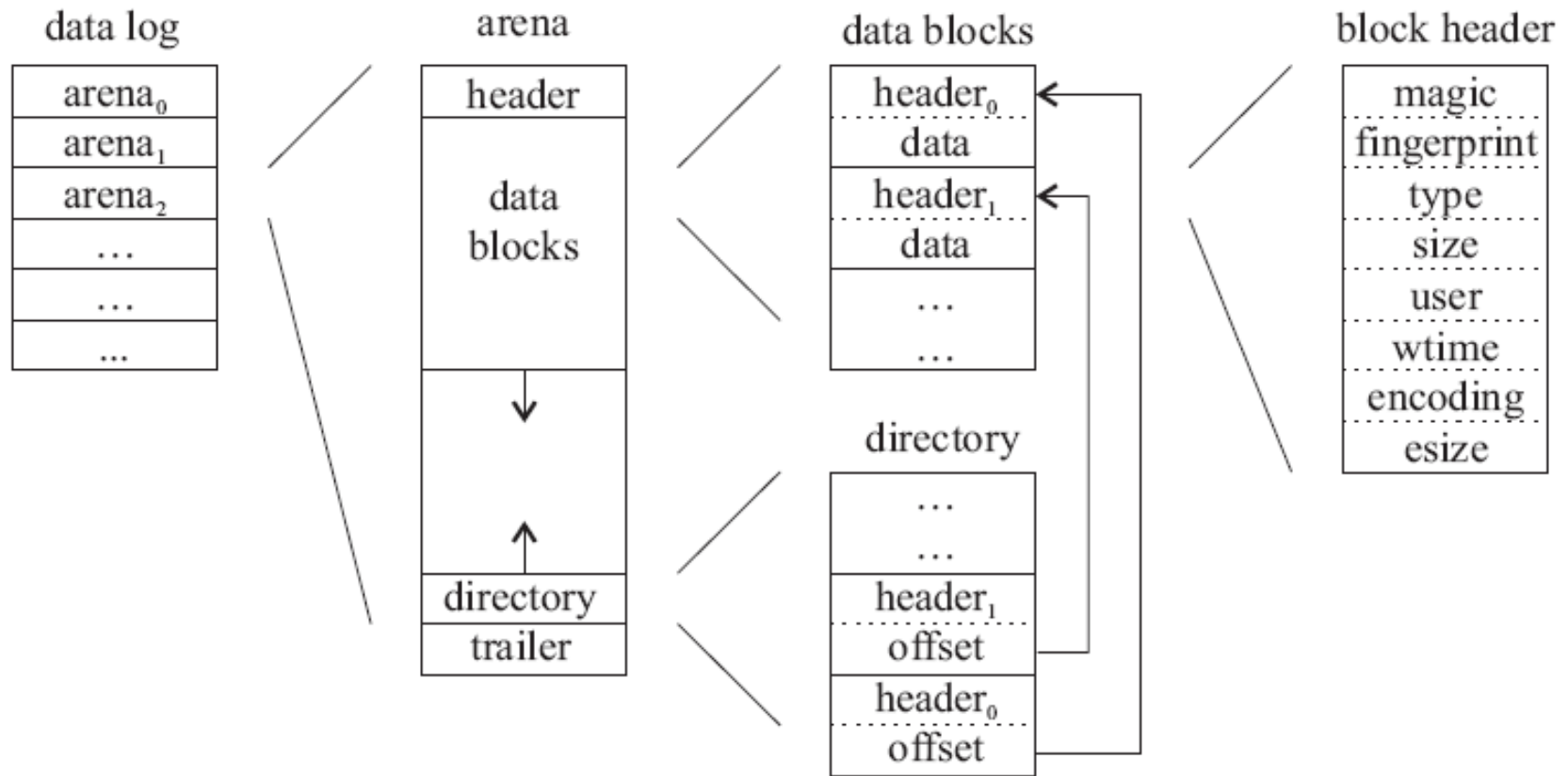


Figure 4. The format of the data log.

Index

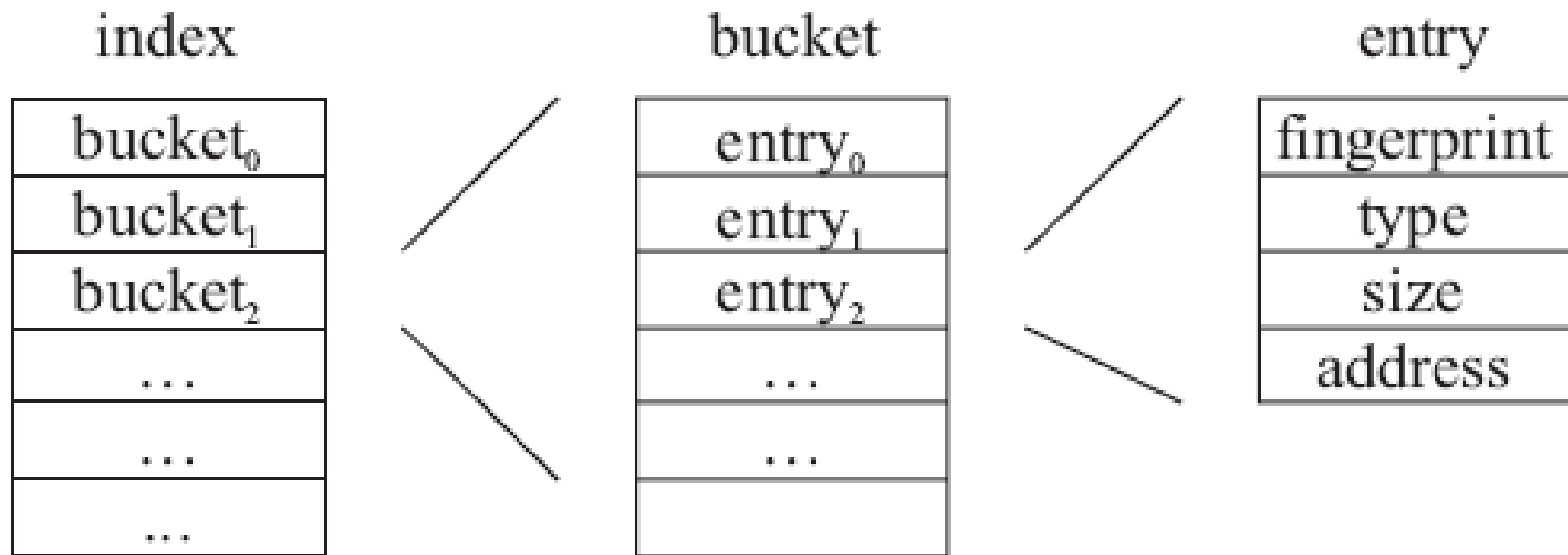


Figure 5. Format of the index.

Performance

Table 1. The performance of read and write operations in Mbytes/s for 8 Kbyte blocks

	Sequential Reads	Random Reads	Virgin Writes	Duplicate Writes
Uncached	0.9	0.4	3.7	5.6
Index Cache	4.2	0.7	-	6.2
Block Cache	6.8	-	-	6.5
Raw Raid	14.8	1.0	12.4	12.4

Venti Server:

Dual 550 Mhz Pentium III

2 Gbyte of memory

100 Mbs Ethernet network

Data log: 500 Gbyte MaxTronic IDE Raid 5

Index: String of 8 Seagate 9 Gbyte SCSI drives

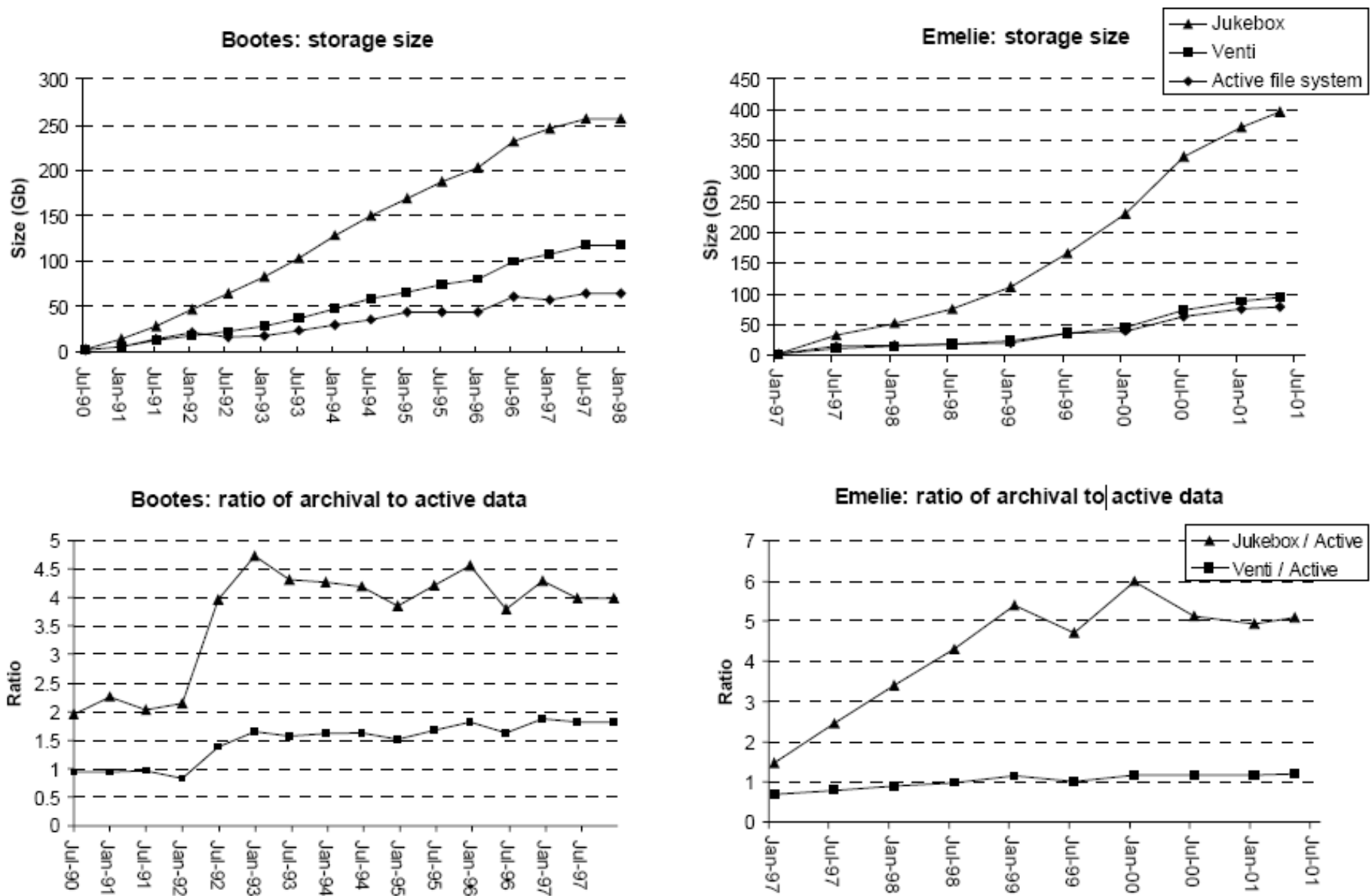


Figure 6. Graphs of the various sizes of two Plan 9 file servers.

Percent Reductions

	bootes	emelie
Elimination of duplicates	27.8%	31.3%
Elimination of fragments	10.2%	25.4%
Data Compression	33.8%	54.1%
Total Reduction	59.7%	76.5%

Table 2. The percentage reduction in the size of data stored on Venti.

Conclusion

- Pros
 - Simple write once approach, removes potential for a lot of errors
 - Venti could be a useful building block
 - Backups are online and easily accessible
 - Convincing feasibility argument
- Cons
 - Performance isn't always great, there is room for optimization
 - Would have been nice to see an implementation of the physical backup application