Opportunistic Use of Content Addressable Storage for Distributed File Systems

Niraj Tolia[†*], Michael Kozuch[†], M. Satyanarayanan[†*], Brad Karp[†], Thomas Bressoud[†‡], and Adrian Perrig[*]

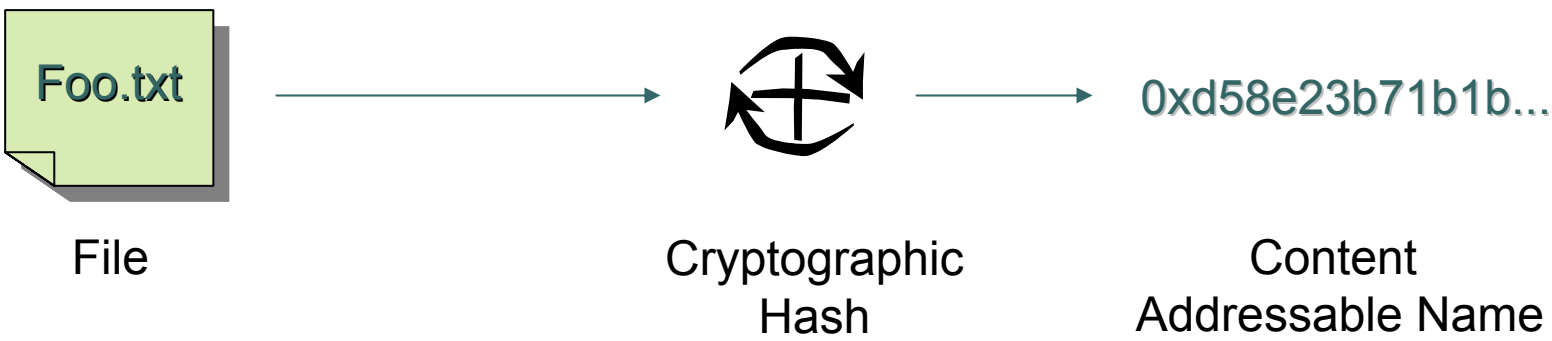
[*]Carnegie Mellon University, [†]Intel Research Pittsburgh and [‡]Denison University

Introduction

- Using a Distributed File System on a Wide Area Network is ***slow!***
- However, there seems to be a growth in the number of providers of Content Addressable Storage (CAS)
- Therefore can we make ***opportunistic*** use of these CAS providers to benefit client-server file systems like NFS, AFS, and Coda?

Content Addressable Storage

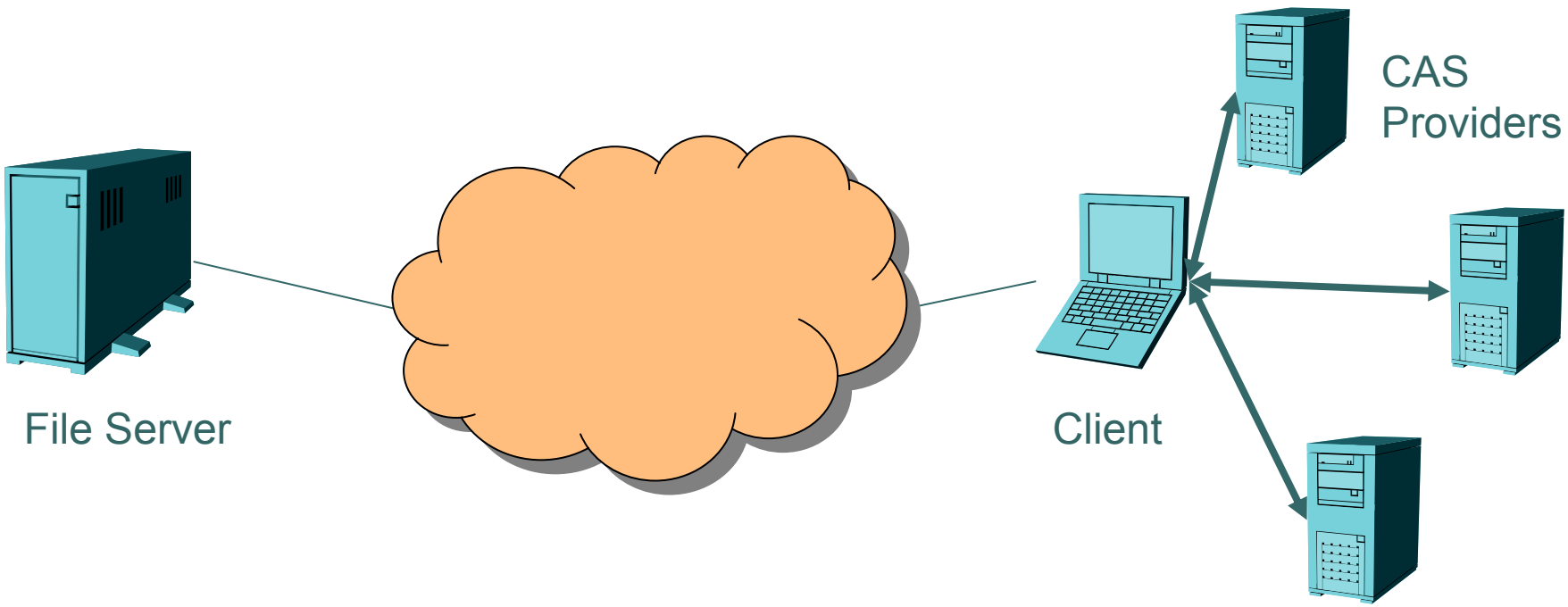
- Content Addressable Storage is data that is identified by its contents instead of a name



- An example of a CAS provider is a Distributed Hash Table (DHT)

Motivation

- Use CAS as a performance enhancement when the file server is remote
- Convert data transfers from WAN to LAN



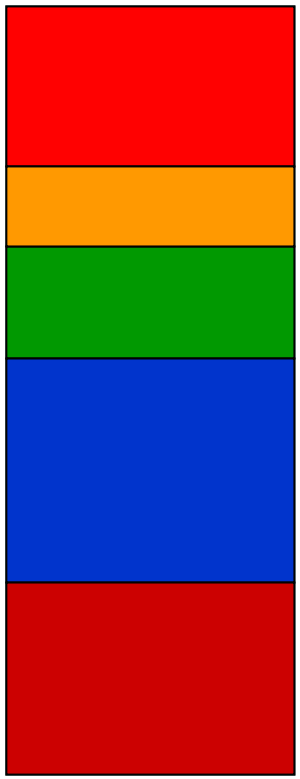
Talk Outline

- Introduction
- The CASPER File System
  - Building Blocks
    - Recipes
    - Jukeboxes
    - Recipe Servers
  - Architecture
- Benchmarks and Performance
- Fuzzy Matching
- Conclusions

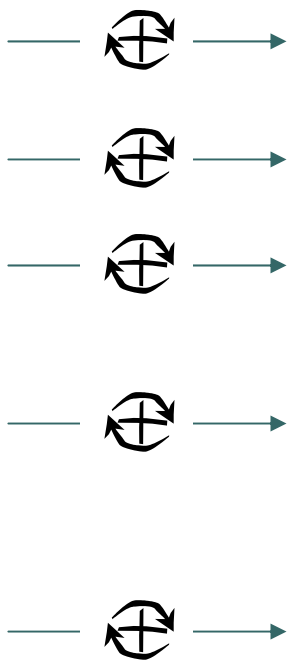
The CASPER File System

- It can make use of any available CAS provider to improve read performance
- However it does not depend on the CAS providers
  - In the absence of a useful CAS provider, you are no worse off than you originally were
- Writes are sent directly to the File Server and not to the CAS provider

Recipes



File Data



Cryptographic Hash

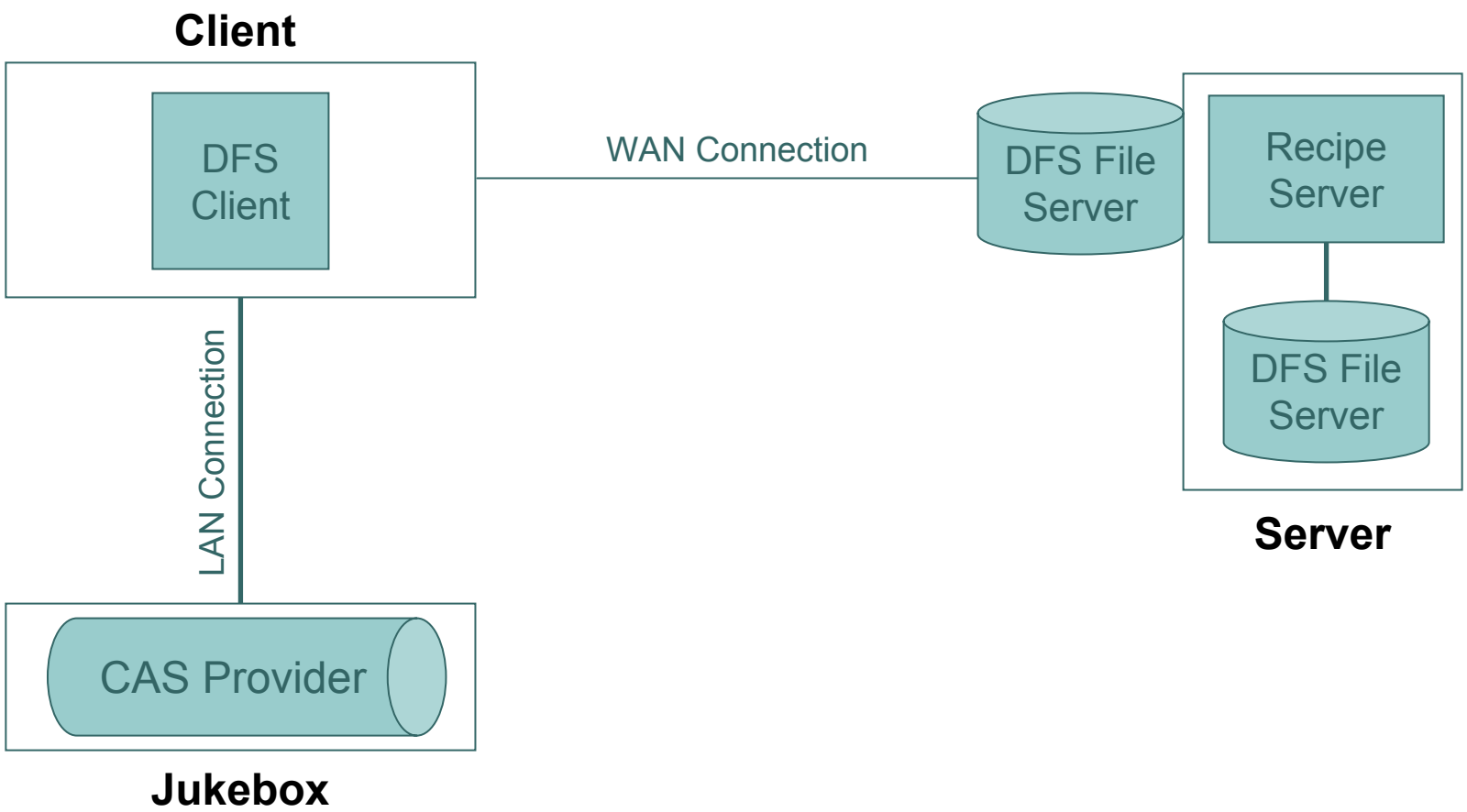
Building Blocks: Recipes

- Description of objects in a content addressable way
- First class entity in the file system
  - Can be cached
- Uses XML for data representation
  - Compression used over the network
- Can be maintained lazily as they contain version information

Recipe Example (XML)

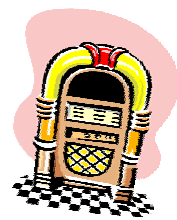
```
<recipe type="file">
  <metadata>
    <version>00 00 01 04 01</version>
    …
  </metadata>

  <recipe_choice>
    <hash_list hash_type="SHA-1" block_type="variable"
                                                number="5">
      <hash size="4189">330c7eb274a4...</hash>
      …
    </hash_list>
  </recipe_choice>
</recipe>
```

CASPER Architecture



Building Blocks: Jukeboxes

- Jukeboxes are abstractions of a Content Addressable Storage provider
  - Provide access to data based on their hash value
- Provide no guarantee to consumers about persistence or reliability
- Support a *Query()* and *Fetch()* interface
  - *MultiQuery()* and *MultiFetch()* also available
- Examples include your desktop, a departmental jukebox, P2P systems, etc.



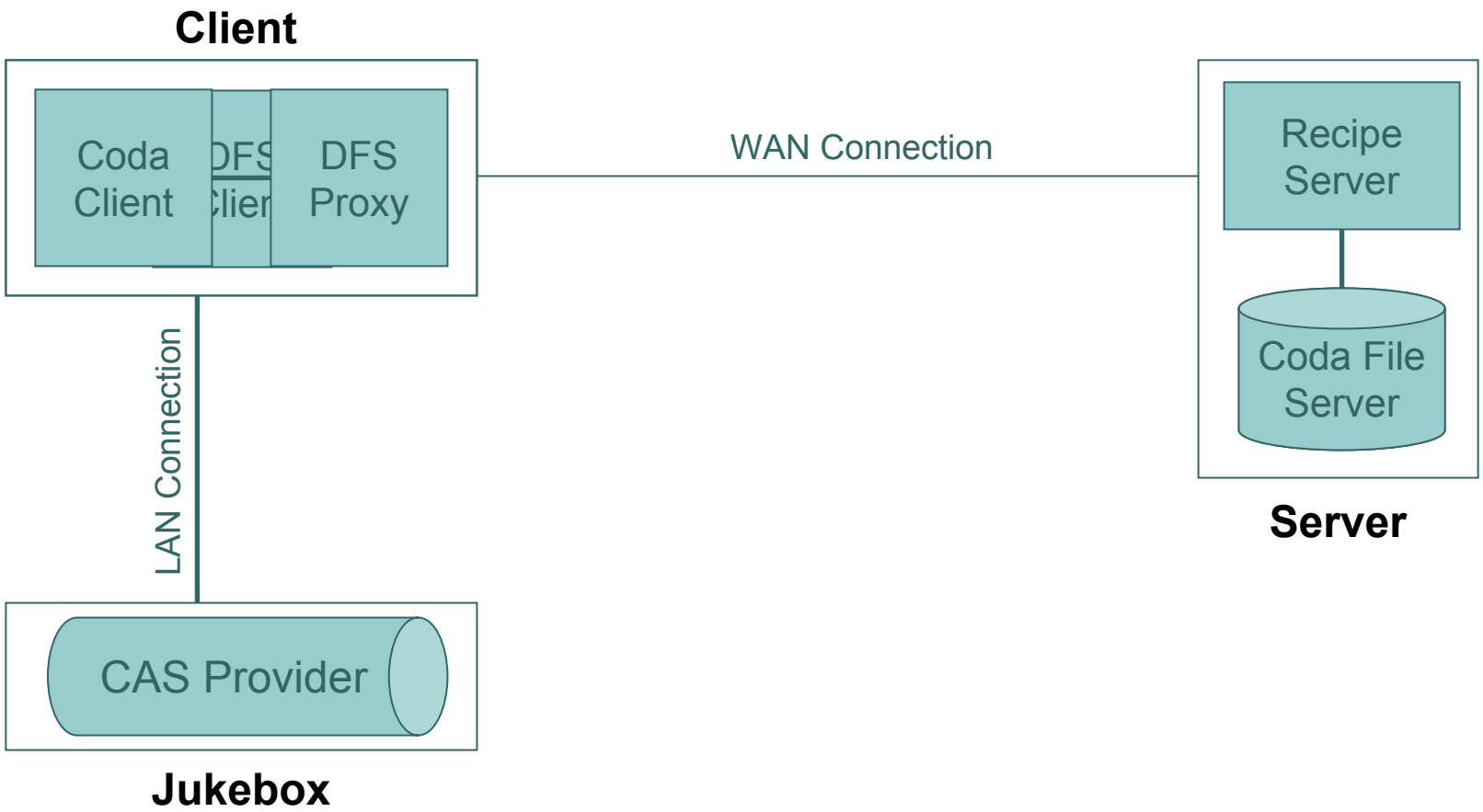
Building Blocks: Recipe Server

- This module generates recipe representation of files present in the underlying file system
- Can be placed either on the Distributed File System server or on any other machine well connected to it
- Helps in maintaining consistency by informing the client of changes in files that it has reconstructed

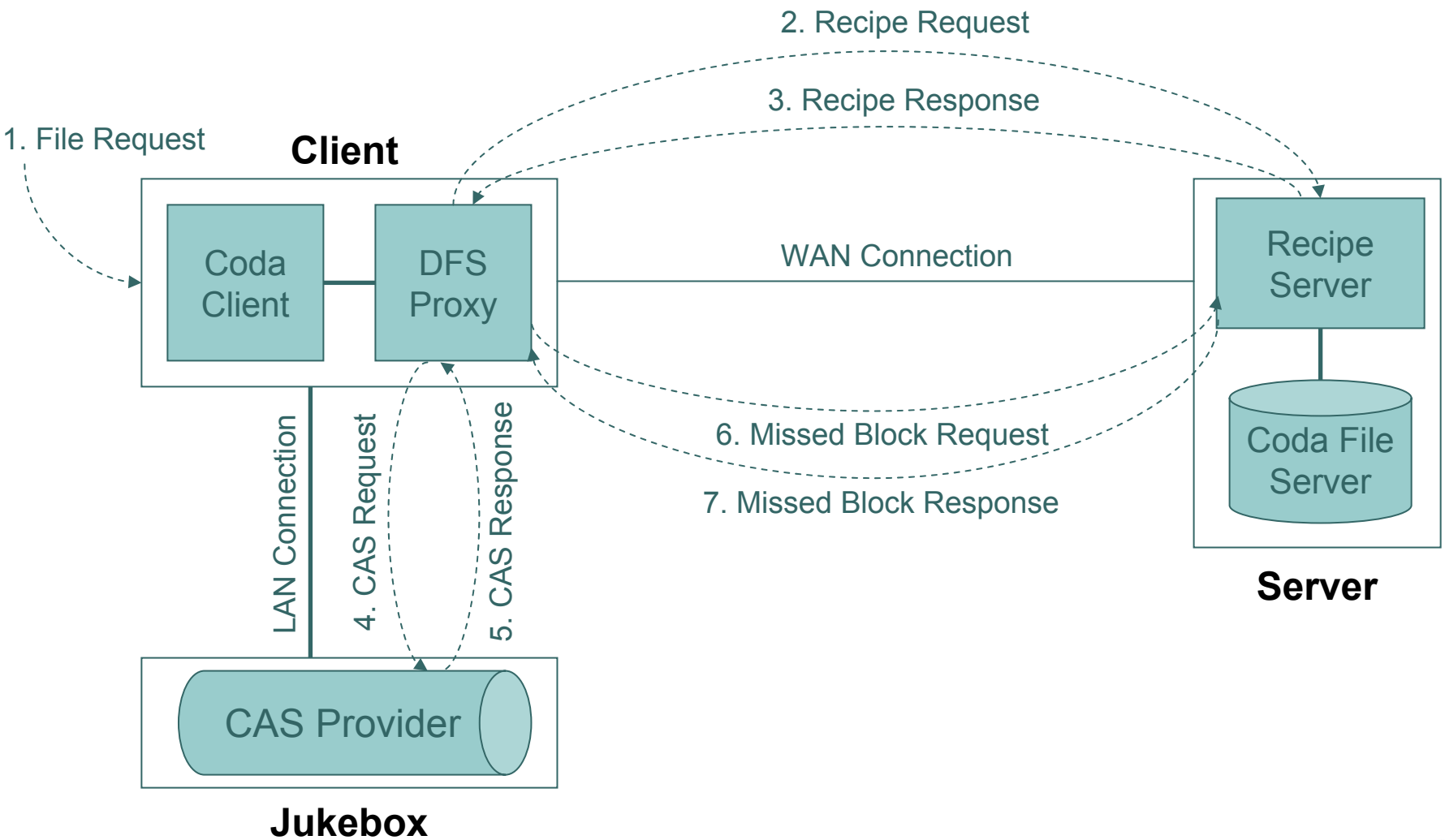
CASPER details…

- The file system is based on Coda
  - Whole file caching, open-close consistency
- Proxy based layering approach used
- Coda takes care of consistency, conflict detection, resolution, etc.
  - The file server is the final authoritative source
- CASPER allows us to service cache misses faster that might be usually possible

CASPER Architecture



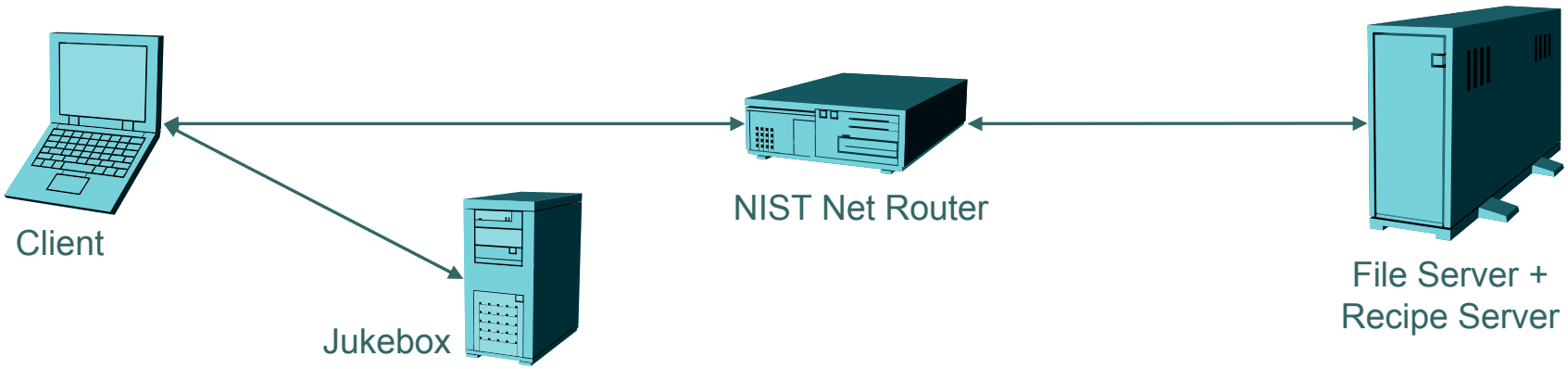
CASPER Implementation



Talk Outline

- Introduction
- The CASPER File System
  - Building Blocks
    - Recipes
    - Jukeboxes
    - Recipe Servers
  - Architecture
- Benchmarks and Performance
- Fuzzy Matching
- Conclusions

Experimental Setup



- WAN bandwidth limitations between the server and client were controlled using NIST Net
  - 10 Mb/s, 1 Mb/s + 10ms, and 100 Kb/s + 100ms
- The hit-ratio on the jukebox was set to 100%, 66%, 33%, and 0%
- Clients began with a cold cache (no files or recipes)

Benchmarks

- Binary Install Benchmark
- Virtual Machine Migration
- Modified Andrew Benchmark

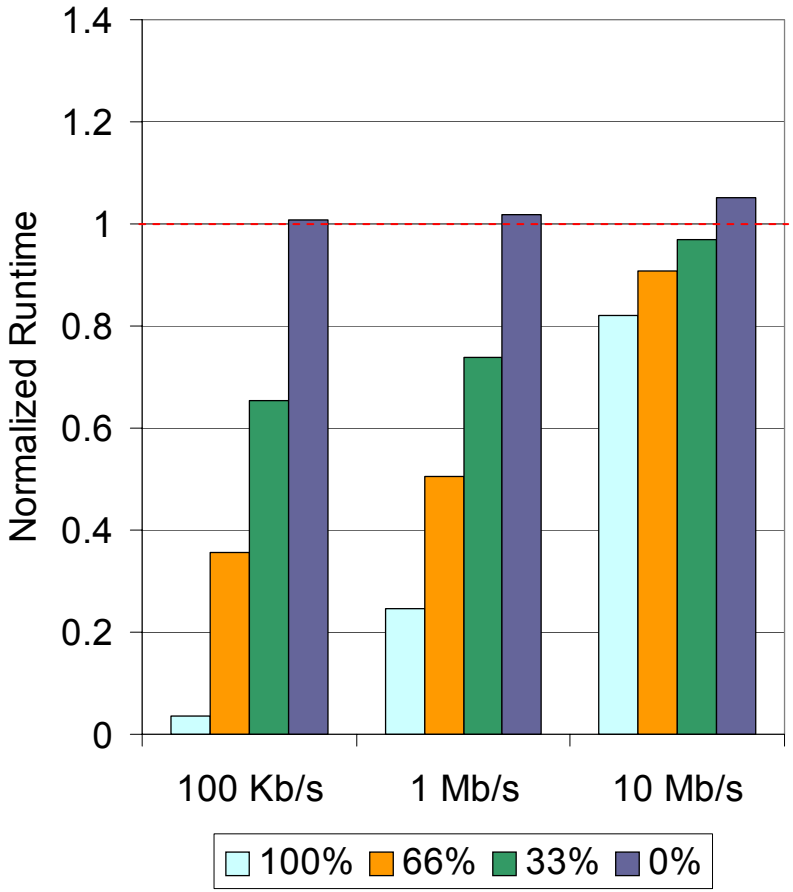
Benchmark Description

- Binary Install (RPM based)
  - Installed RPMs for the Mozilla 1.1 browser
    - 6 RPMs, Total size of 13.5 MB
- Virtual Machine migration
  - Time taken to resume a migrated Virtual Machine and execute a MS Office-based benchmark
  - Trace accesses ~1000 files @ 256 KB each
    - No think time modeled

Benchmark Description (II)

- Modified Andrew Benchmark
  - Phases include: Create Directory, Copy, Scan Directory, Read All, and Make
    - However, only the Copy phase will exhibit an improvement
  - Uses Apache 1.3.27
    - 11.36 MB source tree, 977 files
    - 53% of files are less than 4 KB and 71% of them are less than 8 KB in size

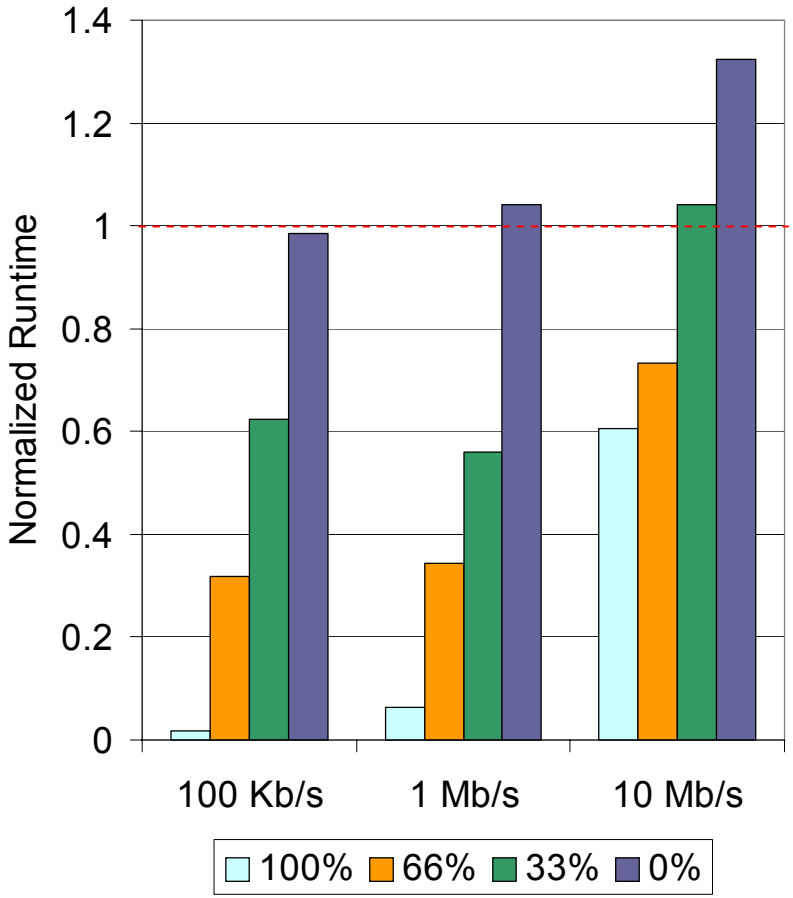
Mozilla (RPM) Install



Baseline	1238 sec	150 sec	44 sec

- Time normalized against vanilla Coda
- Gain most pronounced at lower bandwidth
- Very low overhead seen for these experiments (between 1-5 %)

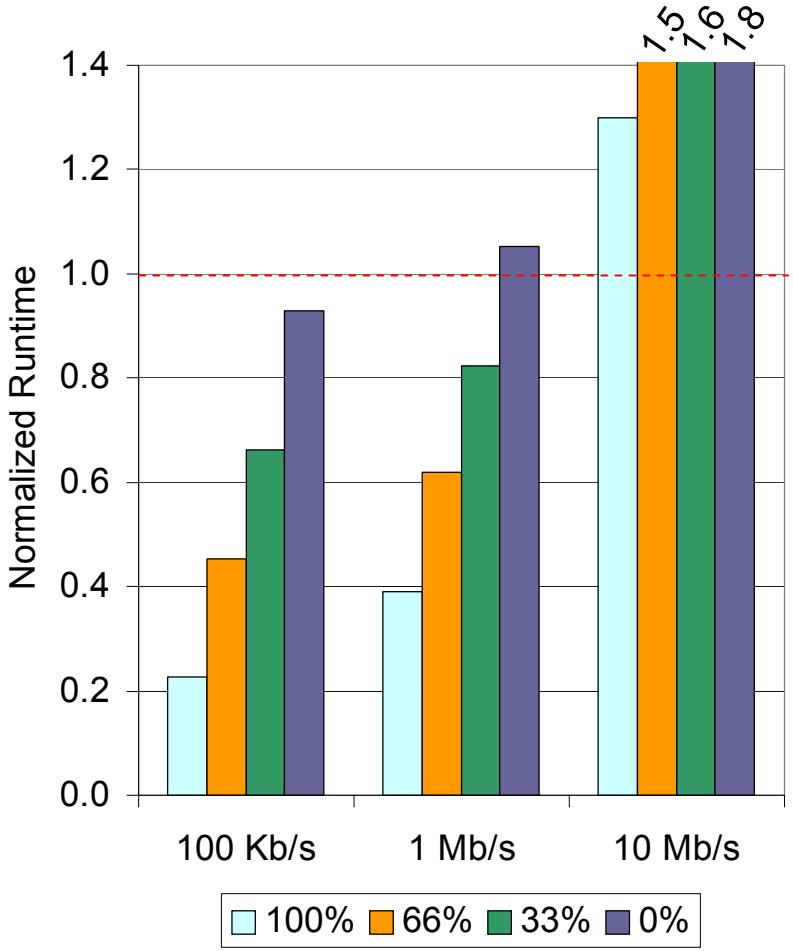
Virtual Machine Migration



Baseline	21523 sec	2046 sec	203 sec

- Time normalized against vanilla Coda
- Large amounts of data show benefit even at higher bandwidths
- High overhead seen at 10 Mb/s is an artifact of data buffering

Andrew Benchmark



Baseline	1150 sec	103 sec	13 sec

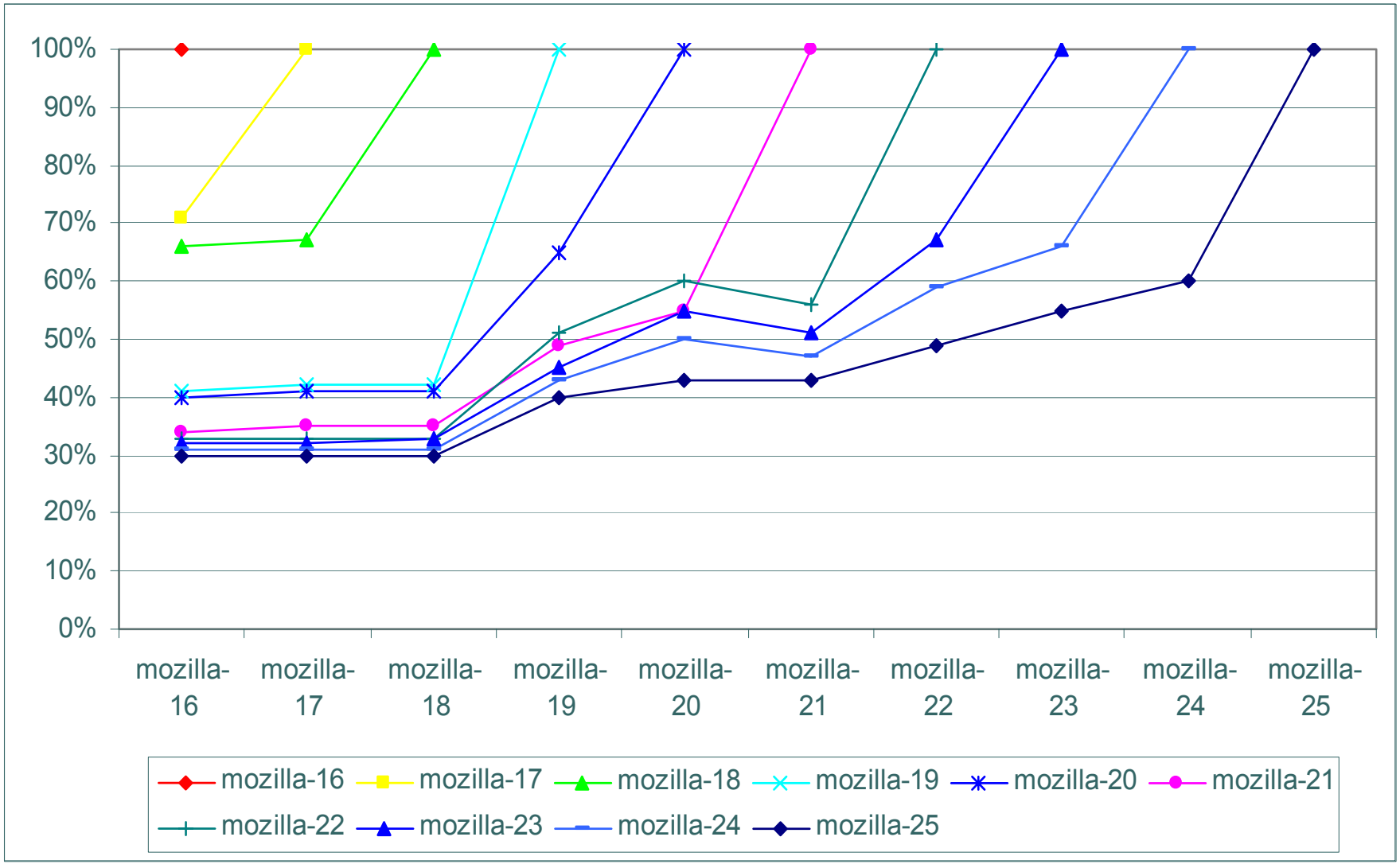
- Only the Copy phase shows benefits
- More than half of the files are fetched over the WAN without talking to the jukebox because of their small size

Commonality

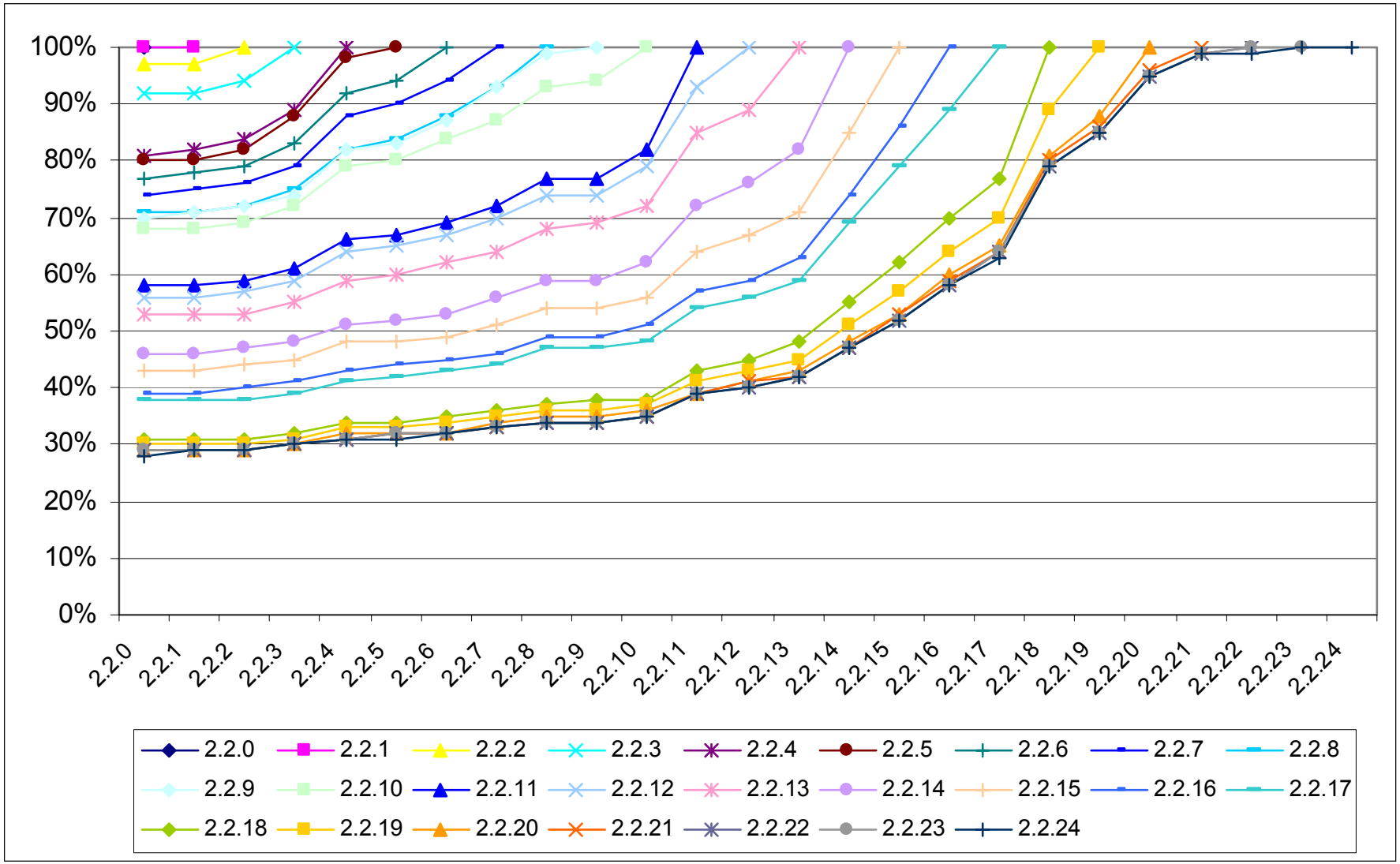
- The question of where and how much commonality can be found is still open
- However, there are a number of applications that will benefit from this approach

- Some of the applications that would benefit include:
  - Virtual Machine Migration
  - Binary Installs and Upgrades
  - Software Development

Mozilla binary commonality



Linux kernel commonality – 2.2



Related Work

- Delta Encoding
  - rsync, HTTP, etc.
- Distributed File Systems
  - NFS, AFS, Coda, etc.
- P2P Content Addressable Networks
  - Chord, Pastry, Freenet, CAN, etc.
- Hash based storage and file systems
  - Venti, LBFS, Ivy, EMC's Centera, Farsite, etc.

Conclusions

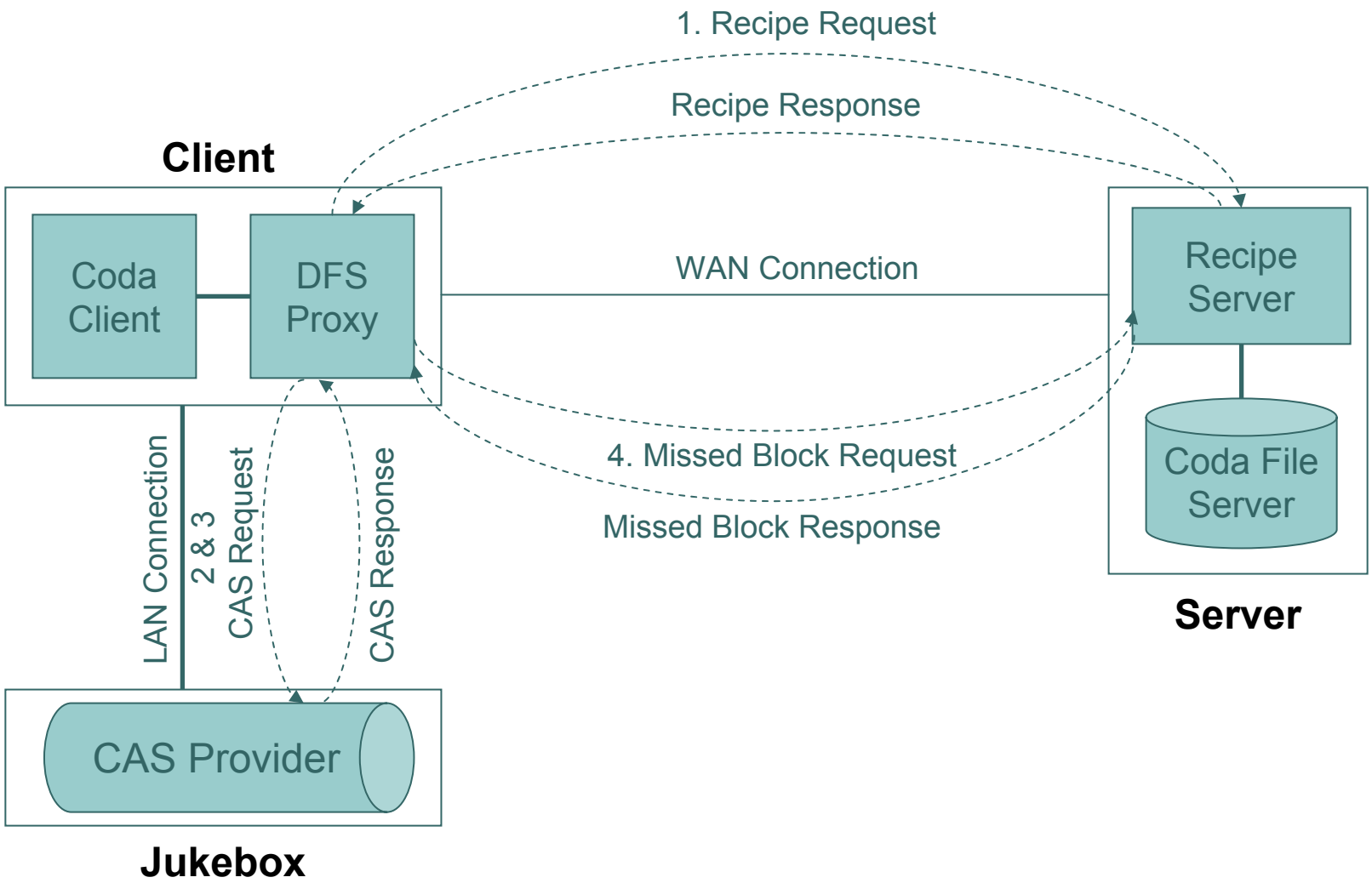
- Introduction of the concept of recipes
- Proven benefits of opportunistic use of content providers by traditional distributed file systems on WANs
- Introduced “Fuzzy Matching”

Backup Slides

Where did the time go?

- For the Andrew benchmark
  - Reconstruction of a large number of small files takes 4 roundtrips
  - There is also the overhead of compression, verification, etc.
  - Some part of the system (CAS requests) can be optimized by performing work in parallel

Number of round trips



Absolute Andrew

Jukebox Hit-Ratio	Network Bandwidth		
	100 Kb/s	1 Mb/s	10 Mb/s
100%	261.3 (0.5)	40.3 (0.9)	17.3 (1.2)
66%	520.7 (0.5)	64.0 (0.8)	20.0 (1.6)
33%	762.7 (0.5)	85.0 (0.8)	21.3 (0.5)
0%	1069.3 (1.7)	108.7 (0.5)	23.7 (0.5)
Baseline	1150.7 (0.5)	103.3 (0.5)	13.3 (0.5)

Andrew Benchmark: Copy Performance (sec)

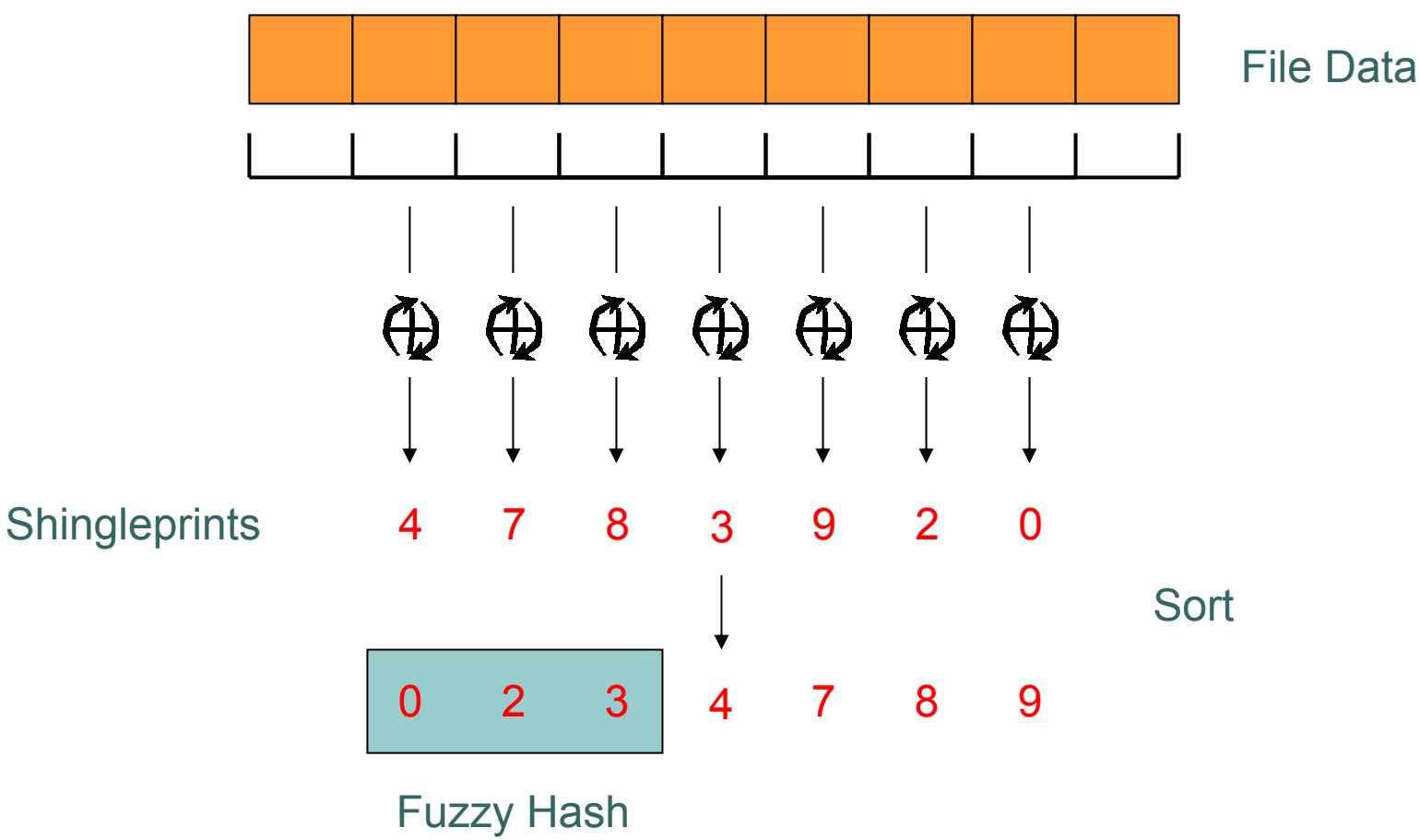
NFS Implementation?

- The benchmark results would not change significantly (with the possible exception of the Virtual Machine migration benchmark).
- It is definitely possible to adopt a similar approach
  - In fact, an NFS proxy (without CASPER) exists.
- However, the semantics of such a system are still unclear…

Fuzzy Matching

- Question: Can we convert an incorrect block into what we need?
- If there is a block that is “near” to what is needed, treat it as a transmission error
- Fix it by applying an error-correcting code
- Fuzzy Matching needs three components
  - Exact hash
  - Fuzzy hash
  - ECC information

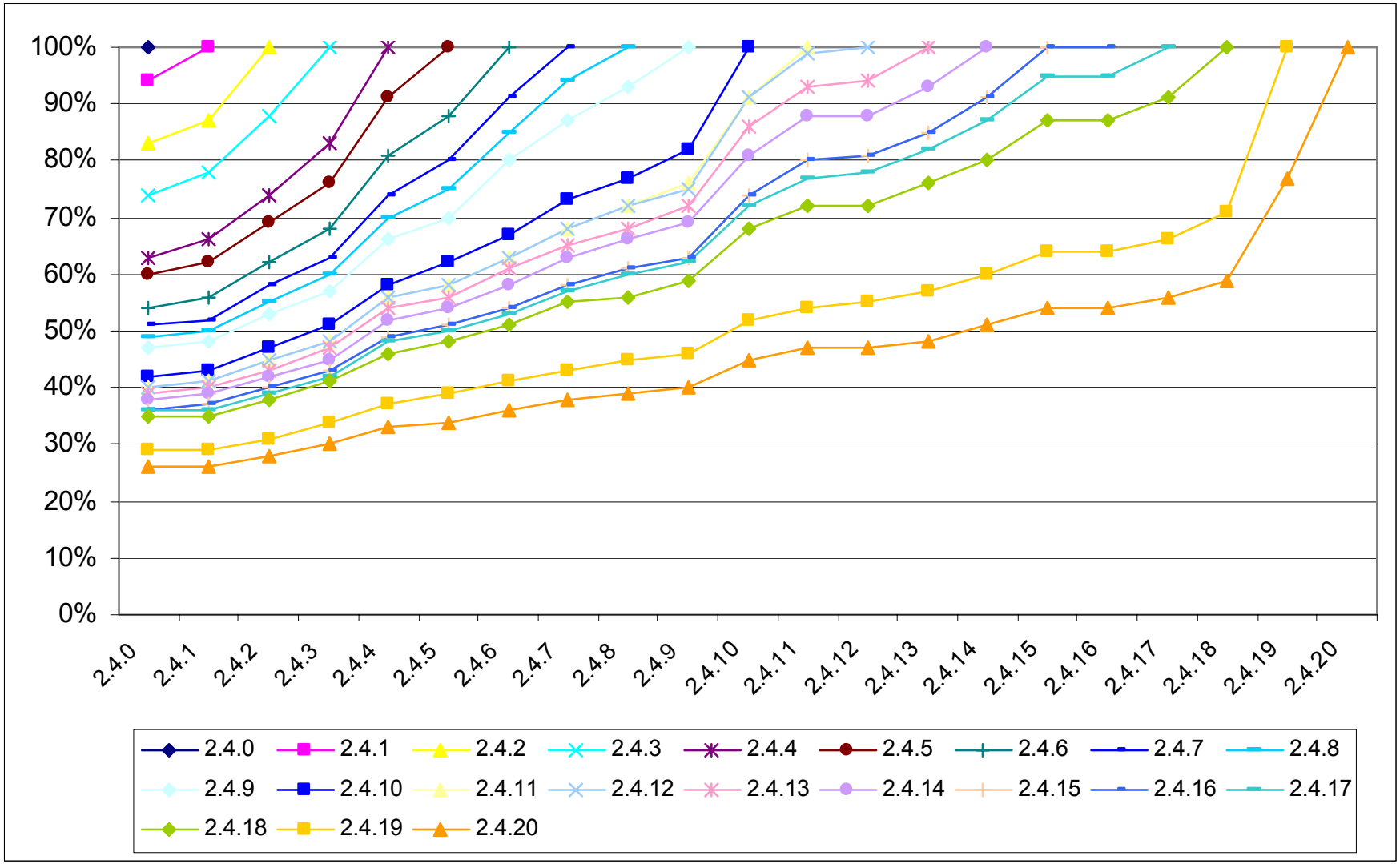
Fuzzy Hashes



Fuzzy Hashing and ECCs

- A fuzzy hash could simply be a *shingle*
  - Hash a number of *features* with a sliding window
  - Take the first *m* hashes after sorting to be representative of the data
  - These *m* shingles are used to find “near” blocks
- After finding a similar block, an ECC that tolerates a number of changes could be applied to recover the original block
- Definite tradeoff between recipe size and Fuzzy Matching but this approach is promising

Linux kernel commonality - 2.4



Linux kernel – 2.2 (B/W)