LBFS: Low Bandwidth Network File System &
SHARK: Scaling File Servers via Cooperative Caching

Presented by - RAKESH .K

OVERVIEW

LBFS

- MOTIVATION
- INTRODUCTION
  - CHALLENGES
  - ADVANTAGES OF LBFS
  - HOW LBFS WORKS?
  - RELATED WORK
- DESIGN
- SECURITY ISSUES
- IMPLEMENTATION
  - SERVER IMPLEMENTATION
  - CLIENT IMPLEMENTATION
- EVALUATION
- SHARK

MOTIVATION

- Users rarely consider running NFS over slow or wide area networks.
- If bandwidth is low, performance is unacceptable.
- Data transfers saturate bottleneck links and cause unacceptable delays.
- Interactive applications are slow in responding to user input.
- Remote login is frustrating
- Solution ?
  - Run Interactive programs locally and manipulate remote files through the file system.
  - Network File system should consume less bandwidth.
  - LBFS.

INTRODUCTION

- LBFS is used for slow or Wide area Networks.
- Exploits similarities between Files or versions of the same file.
- It uses Conventional comparison and Caching.
- In LBF, interactive programs and accessing remote data through file system run locally.
- LBFS requires 90% less bandwidth than Traditional Network File System.

- **Challenges ?**
- **Advantages**
  - **Provides Traditional FS Schematics**
  - **Local Cache**
  - **Exploits Cross File similarities**
  - **Variable Size Chunks**
  - **Indexes chunks by hash values**
- **How LBFS Work?**
  - **Provides Close to Open Consistency**

RELATED WORK

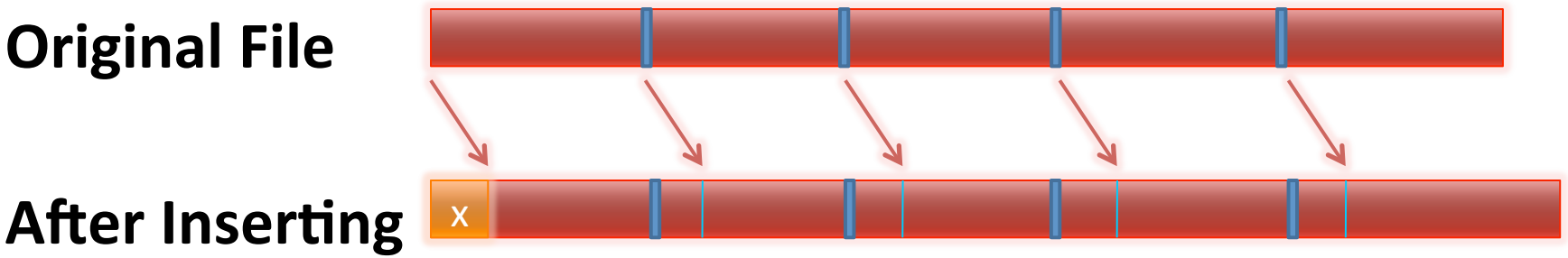
- **AFS** uses server callbacks to reduce network traffic.
- **Leases** are callbacks with expiration date.
- **CODA** supports slow networks and even disconnected operations through optimistic replication.
  - CODA saves bandwidth as it avoids transferring files to the server.
- **Bayou** and **Ocean Store** investigate conflict resolution for optimistic updates.
- **Spring** and **Wetherall** :
  - Use large client and server caches
- **Rsync** exploits similarities between directory trees.

DESIGN

- LBFS uses large persistent file cache at client.
  - It assumes Client has enough cache.
- It Exploits similarities between files and file versions.
  - Divides Files into Chunks.
  - Only transmits data chunks containing new data**.**
- To save chunk transfer, LBFS relies on the SHA-1 Hash.
- LBFS Uses "**gzip"** compression.
- Central challenge in Design is:
  - Keeping the index a reasonable size
  - Dealing with shifting offsets.

PROBLEMS WITH FIXED SIZED BLOCKS

- Single byte insertion shifts all the block boundaries.



- **Possible solutions:**
  - Index files by the hashes of all overlapping 8 KB blocks at all offsets.
  - Rsync: Consider only two files at a time. Existence of a file is found using file name.

LBFS Solution for Block Size

- LBFS
  - Only looks for non-overlapping chunks in files
  - **Avoids sensitivity to shifting file offsets by Setting chunk boundaries based on file contents.**
- To divide a file into chunks, LBFS
  - Examines every (overlapping) 48-byte region of the file.
  - LBFS *Uses Rabin's fingerprints to select boundary regions* called ***breakpoints.***
  - **Fingerprints** are efficient to compute on a sliding window in a file.

Rabin Finger Prints

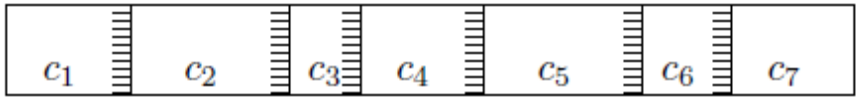
- Polynomial representation of data in 48-byte region modulo an irreducible polynomial.
  - **FingerPrint = f(x) mod p(x)**

- Probability of Collision = $\max(|r|,|s|)/2^{w-1}$

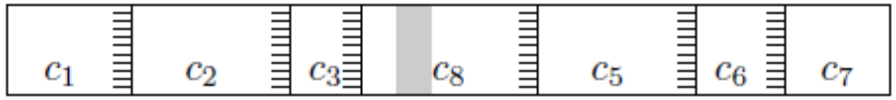
- Boundary regions have the 13 least significant bits of their fingerprint equal to an arbitrary predefined value.

- Method is reasonably fast.

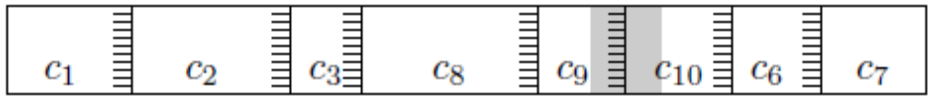
Chunk Boundaries After a Series of Edits?

a. 

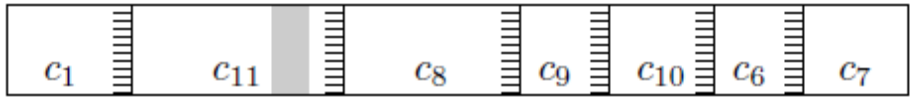
- Figure shows the file divided into variable length chunks with break points determined by hash of each 48 bit region.

b. 

- Effect of inserting some text into the file at chunk **C4.**
- Transfer only C8.

c. 

- Effect of inserting a data in **C5** that contains a break point
- Splitting the chunks into two new chunks. (**C9** and **C10)**
- Transfer only two new chunks **C9** and **C10.**

d. 

- One of the break point is eliminated. **C2+C3 -> C11**
- Transfer only **C11**

PATHOLOGICAL CASES

- Variable size chunks can lead to Pathological behavior.
- If every 48 bytes of a file happened to be a breakpoint.
- Very large chunks would be too difficult to send in a single RPC.
- Arbitrary size RPC messages would be somewhat inconvenient.
- Chunk sizes must be between 2K and 64K
- Artificially insert chunk boundaries if file is full of repeated sequences.

CHUNK DATABASE

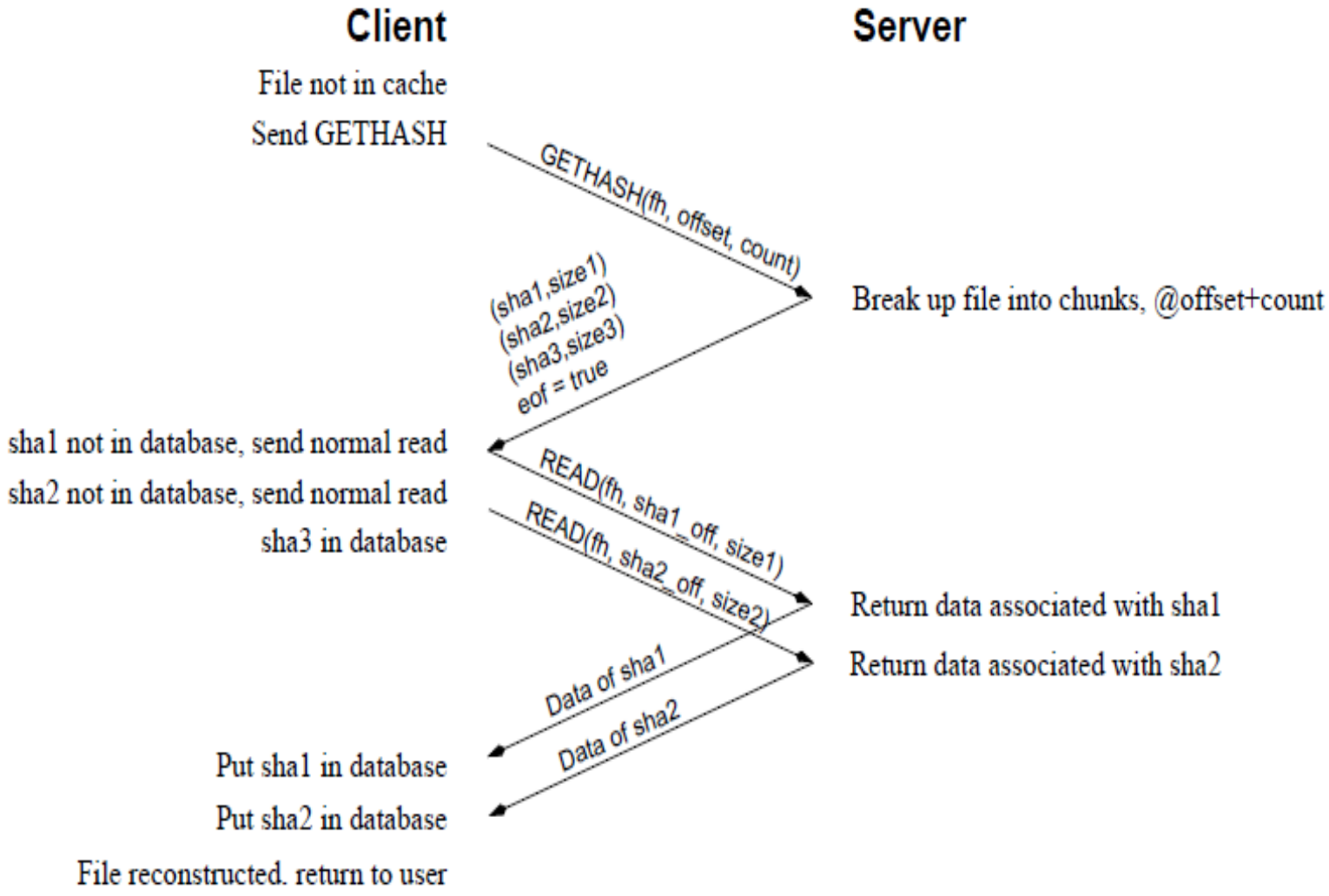
- The chunk database Indexes chunks by first 64 bits of SHA-1 hash.
- The database maps keys to (file, offset, count) triples.
- LBFS never relies on the correctness of the chunk database.
- How to keep this database up to date?
  - Must update it whenever file is updated
  - Can still have problems with local updates at server site
  - Crashes can corrupt database contents.

FILE CONSISTENCY

- The LBFS client currently performs whole file caching.
- LBFS uses a three-tiered scheme to determine if a file is up to date.
  - OPEN A FILE:
    - IF Lease Not Expired
    - IF Lease Expired
- Client gets a lease first time a file is opened for read.
- Client Renews the expired lease by requesting file attributes.
- It's the job of the Client to check if the cached copy is still current.

FILE READS

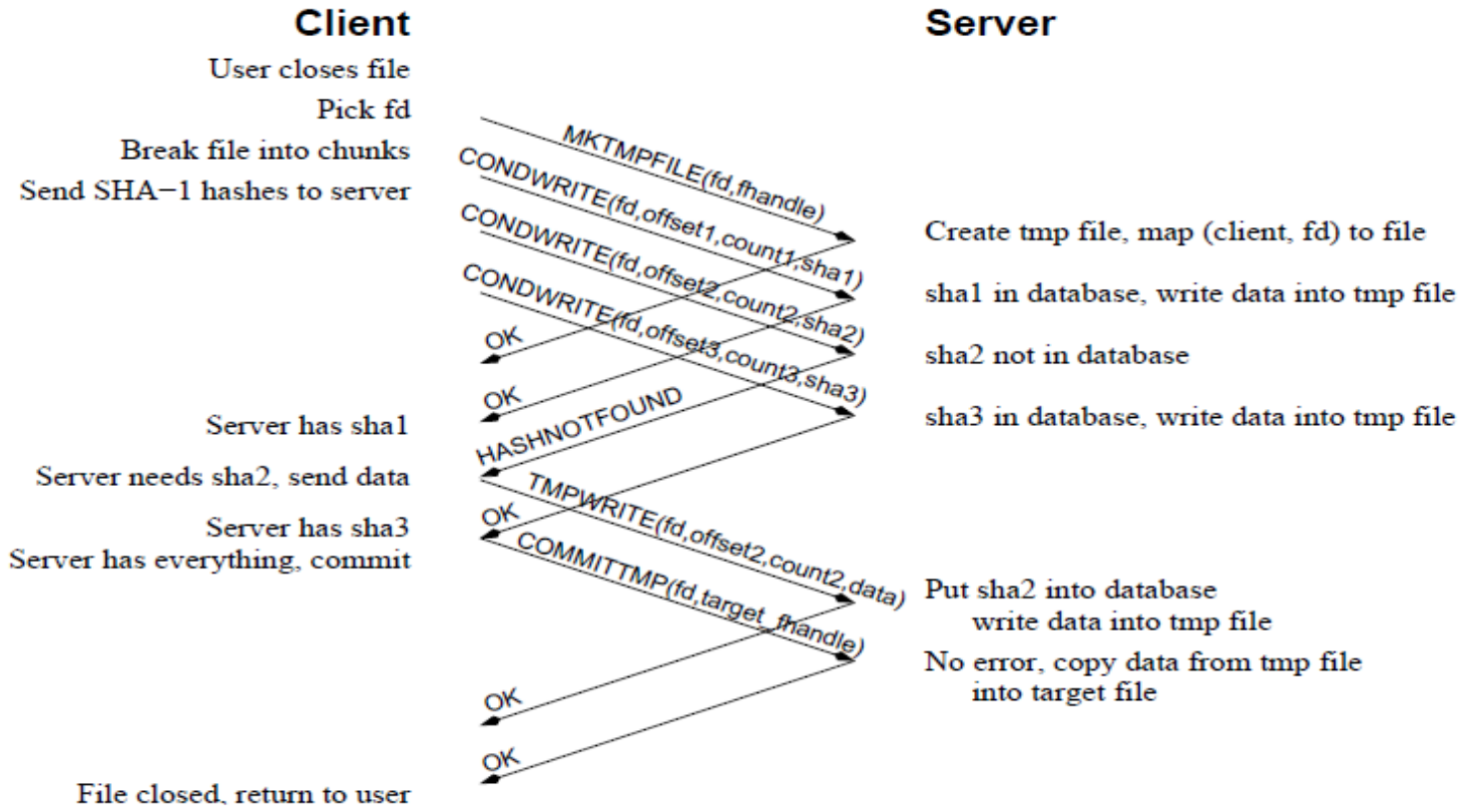
- LBFS Use additional calls not in NFS -> **GETHASH** for reads



FILE WRITES

- LBFS Server updates files **atomically** at close time.
- Uses Temporary Files .
- 4 RPC's are used in update protocol:

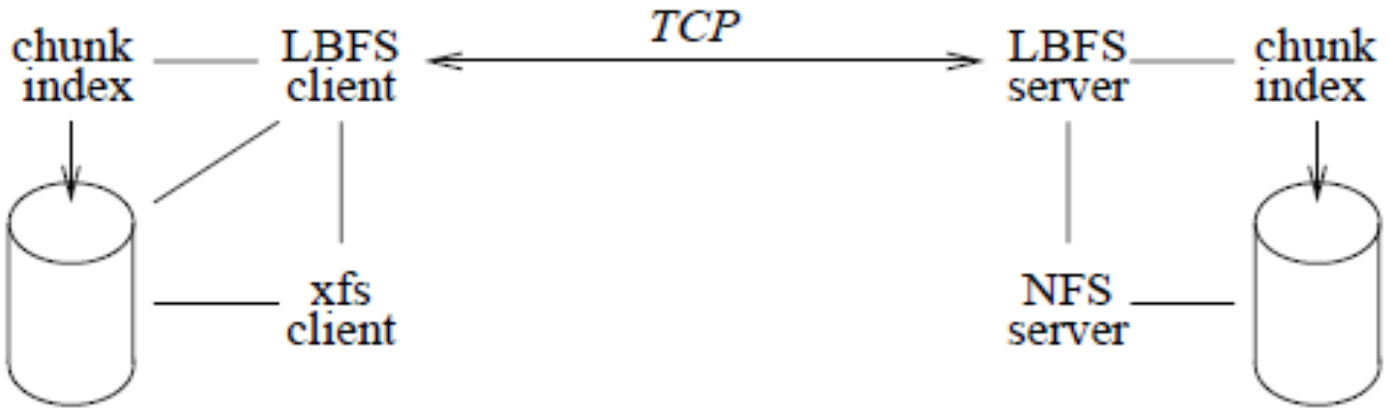
  *1. MKTMPFILE *2.CONDWRITE * 3. TMPWRITE *4. COMMITTMP.



SECURITY

- LBFS uses the security infrastructure from SFS.
- All Servers have public keys.
- Access Control.

IMPLEMENTATION



- **mkdb** utility
- If File Size < 8KB
- Trash Directory

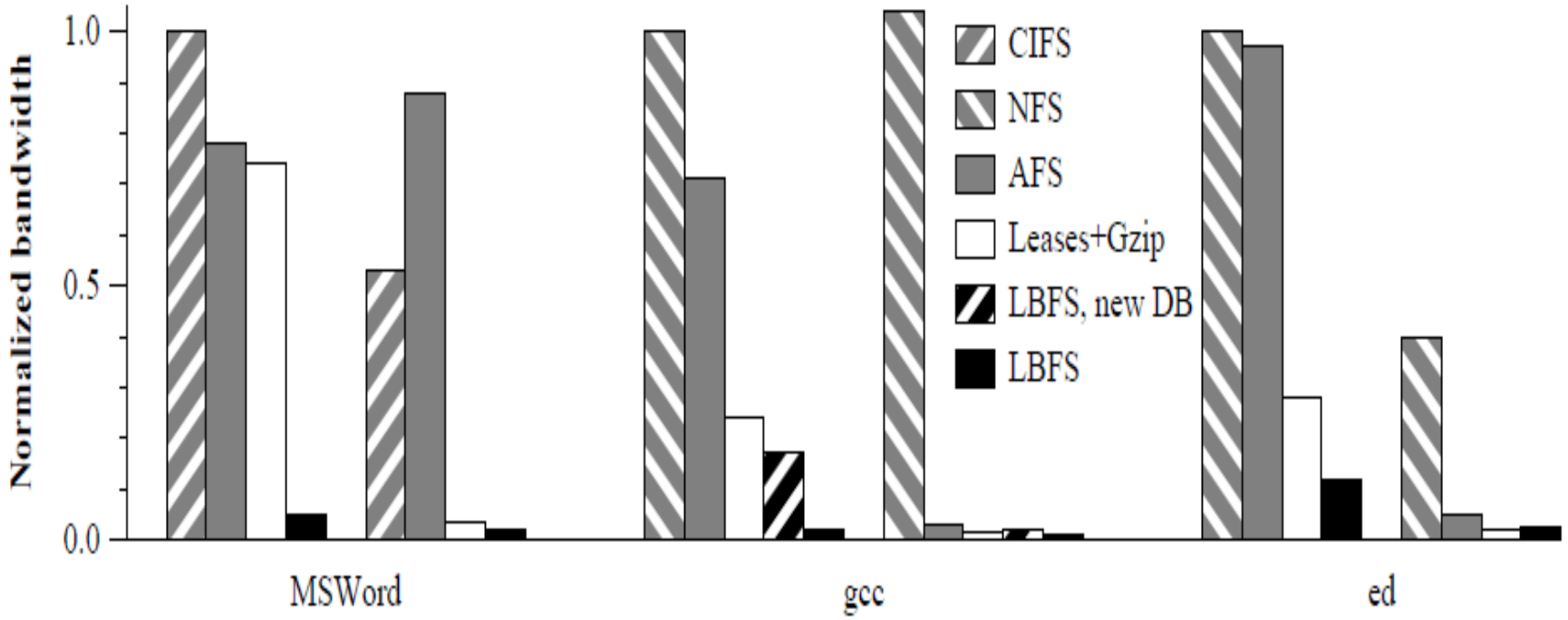
EVALUATION – REPEATED DATA IN FILES

- Bandwidth consumption and network utilization are measured under several common workloads.
- LBFS is compared with :
  - CIFS,
  - NFS version 3 and
  - AFS.

Data	Given	Data size	New data	Overlap
emacs 20.7 source	emacs 20.6	52.1 MB	12.6 MB	76%
Build tree of emacs 20.7	—	20.2 MB	12.5 MB	38%
emacs 20.7 + printf executable	emacs 20.7	6.4 MB	2.9 MB	55%
emacs 20.7 executable	emacs 20.6	6.4 MB	5.1 MB	21%
Installation of emacs 20.7	emacs 20.6	43.8 MB	16.9 MB	61%
Elisp doc. + new page	original postscript	4.1 MB	0.4 MB	90%
MSWord doc. + edits	original MSWord	1.4 MB	0.4 MB	68%

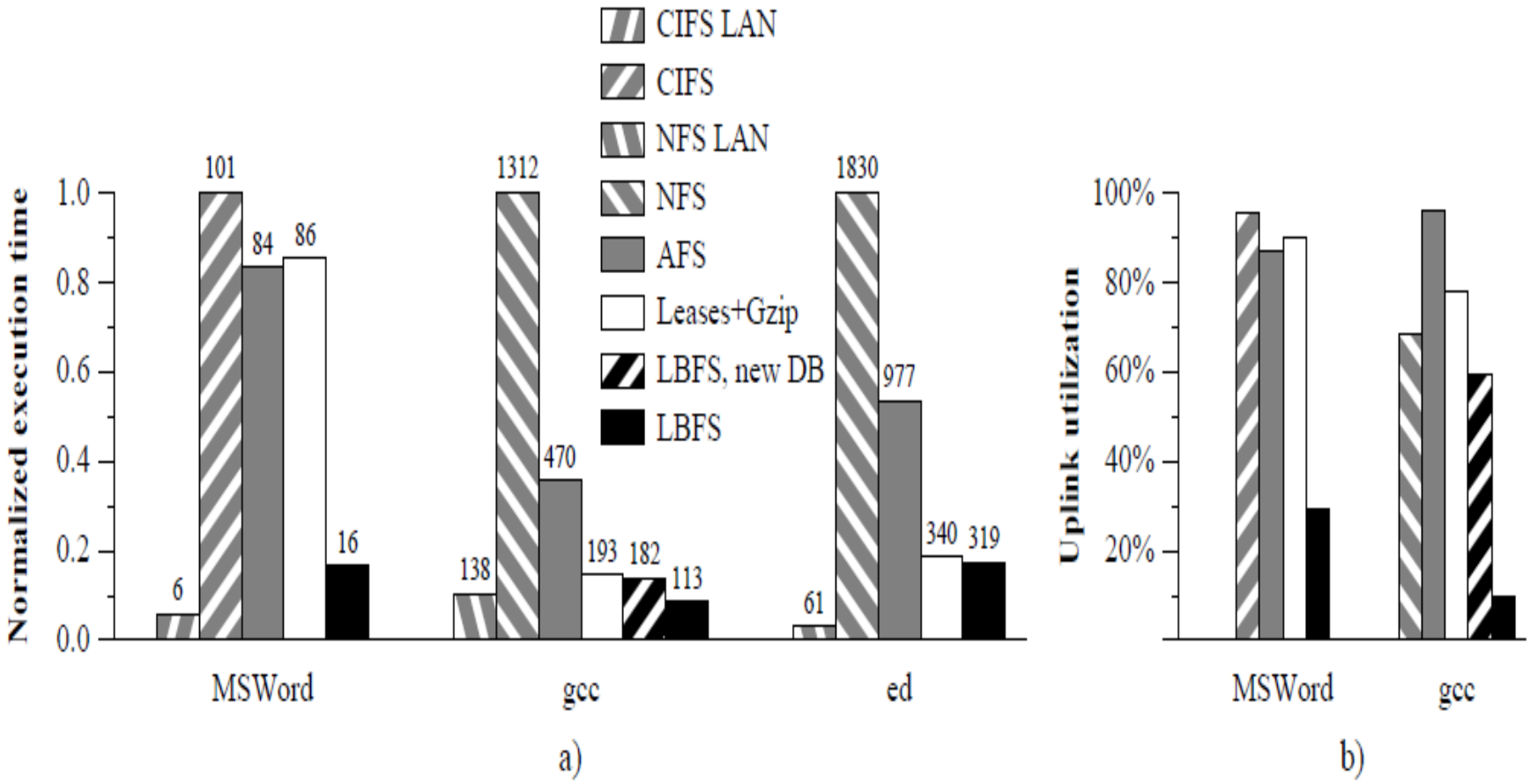
EVALUATION (Cont) – BANDWIDTH UTILIZATION

- Used 3 WORK LOADS .
- (MS Word 1.4MB file, gcc -> Compiled emacs 20.7, ed-> perl)



Normalized bandwidth consumed by three workloads. The first four bars of each workload show upstream bandwidth, the second four downstream bandwidth. The results are normalized against the upstream bandwidth of CIFS or NFS.

EVALUATION (3) – APPLICATION PERFORMANCE



a) Normalized application performance on top of several file systems over a cable modem link with 384 Kbit/sec uplink and 1.5 Mbit/sec downlink. Execution times are normalized against CIFS or NFS results. Execution times in seconds appear on top of the bars. b) Uplink bandwidth utilization of the MSWord and gcc benchmarks.

OVERVIEW

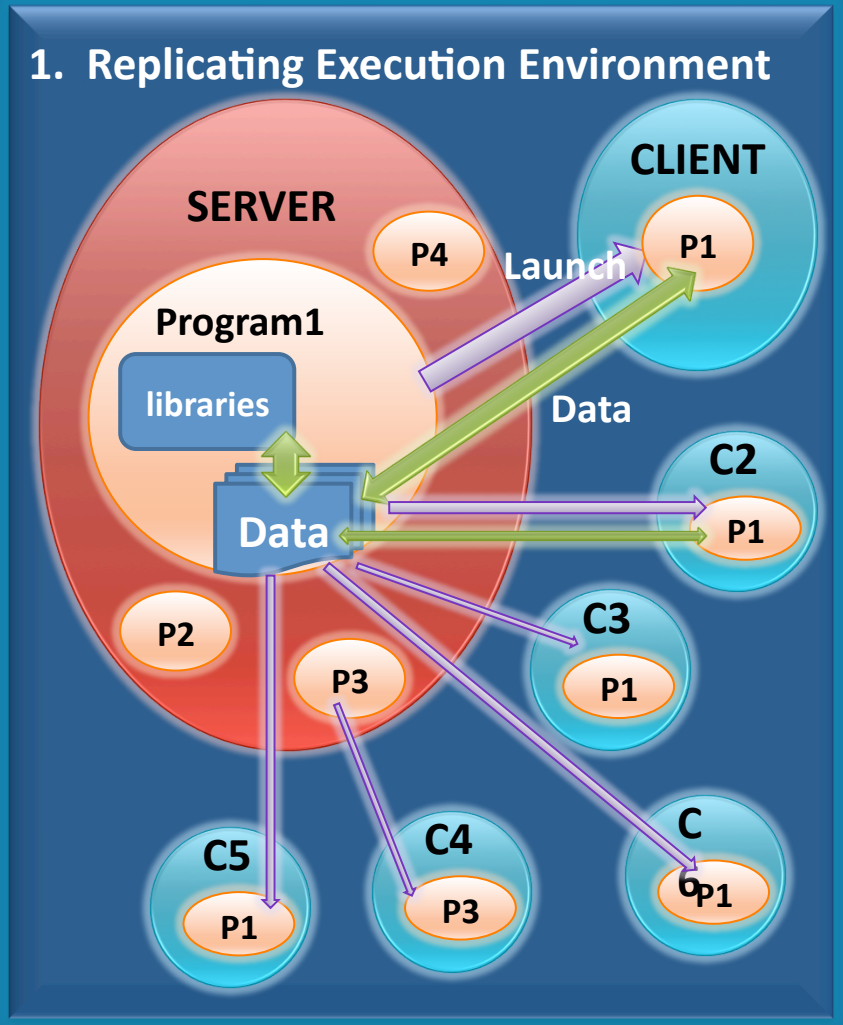
SHARK

- MOTIVATION
- INTRODUCTION
  - CHALLENGES
  - ADVANTAGES OF SHARK
  - HOW SHARK WORKS?
- DESIGN
- IMPLEMENTATION
- EVALUATION
- CONCLUSTION
- DISCUSSION (QUESTIONARIES)

MOTIVATION

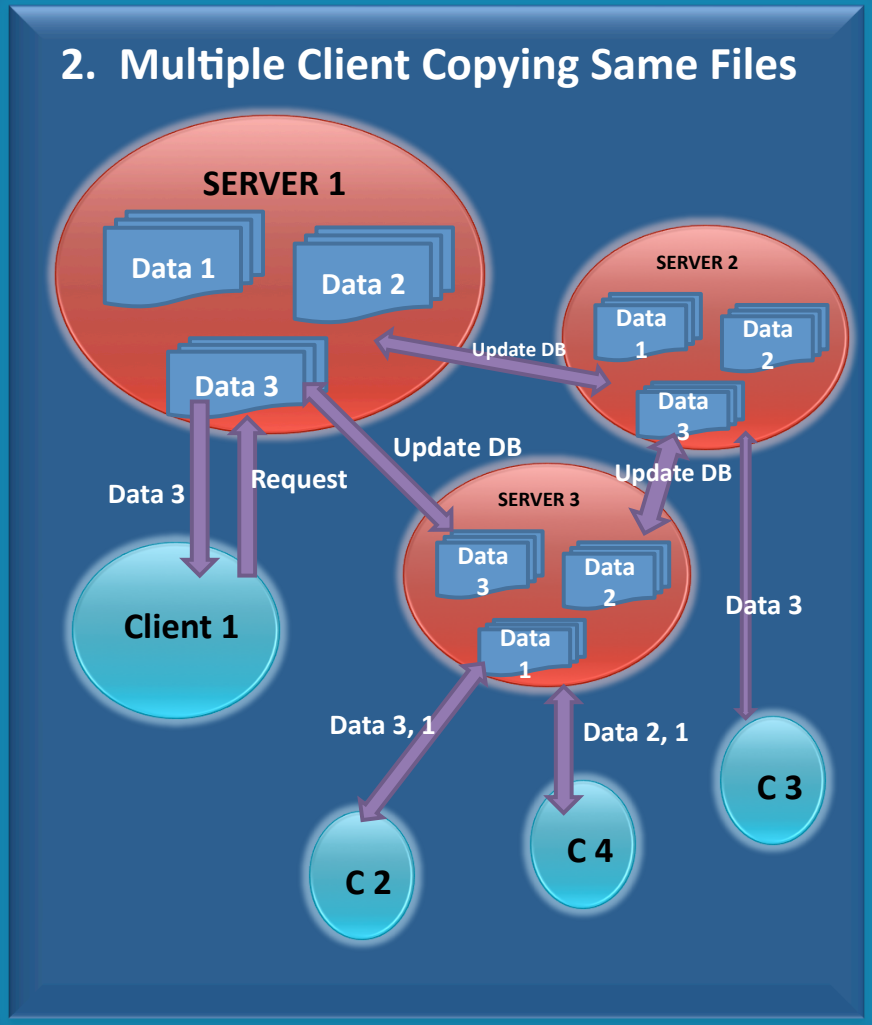
Current Systems

1. Replicating Execution Environment



Replicate their execution environment on each machine before launching distributed application -> WASTE RESOURCES + DEBUG

2. Multiple Client Copying Same Files



Replicating data and serving same copy of data to multiple clients -> INCONSISTENCIES

INTRODUCTION

- Shark is a distributed file system
- Designed for large-scale, wide-area deployment.
- Scalable.
- Provides a novel cooperative-caching mechanism
  - Reduces the load on an origin file server.

CHALLENGES

- **Scalability:** What if a large program (say in MB's) is being executed from a file server by many clients?
  - Because of bandwidth, server might deliver unacceptable performance.

- As the model is similar to P2P file systems, administration, accountability, and consistency need to be Considered.

HOW SHARK WORKS?

- Shark clients find nearby copies of data by using distributed index.
- Client avoids transferring the file/chunks of the file from the server, if the same data can be fetched from nearby, client.
- Shark is compatible with existing backup and restore procedures.
- By shared read, Shark greatly reduces server load and improves client latency.

DESIGN – PROTOCOL (1/2)

1. Shark server divides the file into chunks by using ***Rabin finger print algorithm***.

2. Shark interacts with the local host using an existing NFSv3 and runs in user space.

3. When First Client reads a particular file
   - *Gets file and Registers as* ***replica proxy*** for the chunks of the file in the distributed index.

4. Now when 2nd client wants to access the same file:
   - it discovers the proxies of the file chunks by querying the distributed index.
   - establishes a secure channel to (multiple such) proxy(s) , and download the file chunks in parallel.

DESIGN PROTOCOL (2/2)

5. After fetching, the client then registers itself as a replica proxy for these chunks.

6. Server exposes two Api's.
   - Put: Client executes put to declare that it has some thing.
   - Get: client executes get to get the list of clients who have some thing.

SECURE DATA SHARING

7. Data is encrypted by the sender and can be decrypted only by the client with appropriate read permissions.

8. Clients cannot download large amounts of data without proper read authorization.

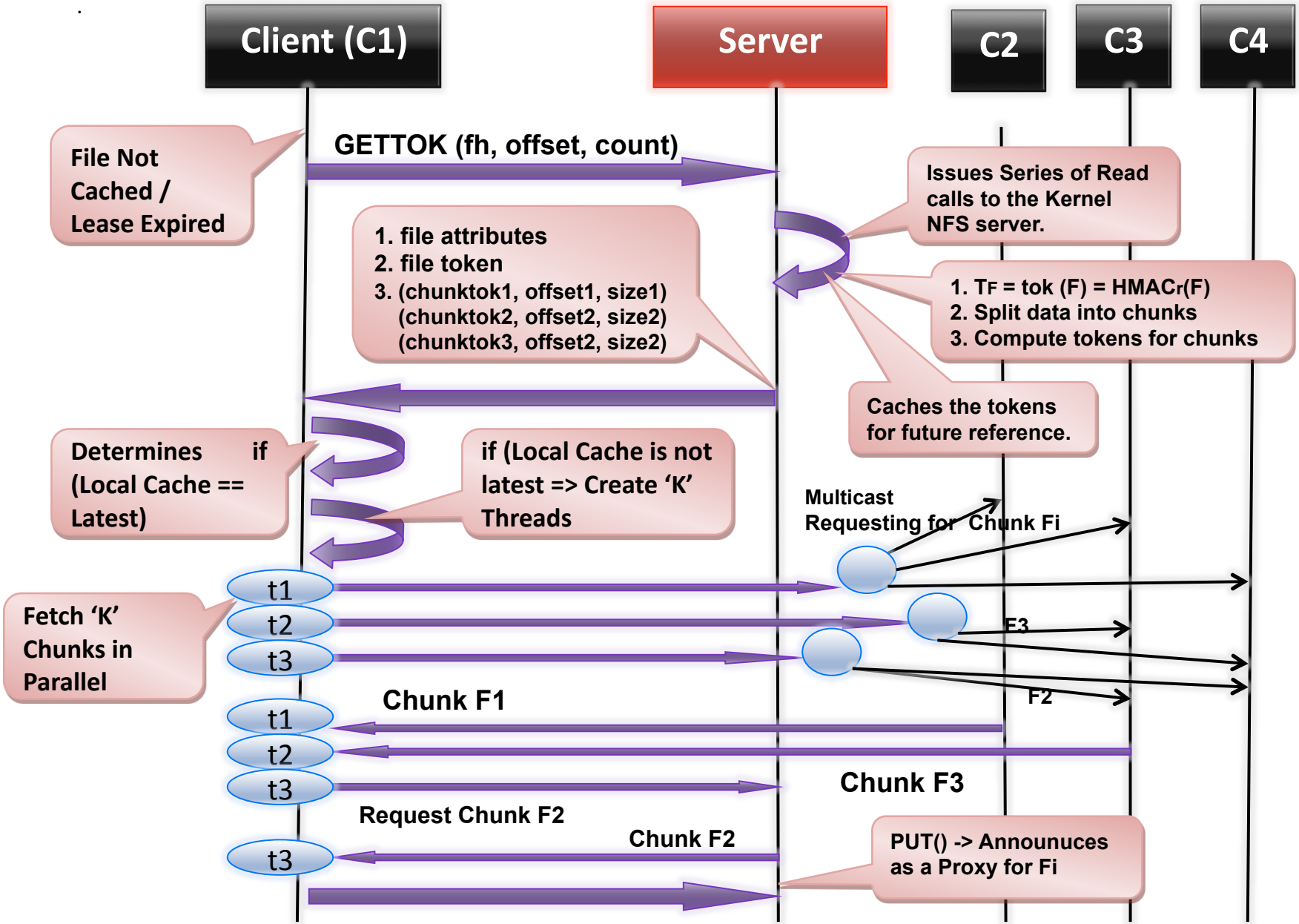
SECURE DATA SHARING

- **How?**
  - ***Cross-file-system*** *sharing*
  - Shark uses token generated by the file server as a shared secret between client and proxy.
  - Client can verify the integrity of the received data.
  - For a sender client (proxy client) to authorize requester client, requester client will provide the knowledge of token
  - Once authorized, receiver client will establish the read permission of the file.

FILE CONSISTENCY

- Shark uses two network file system techniques:
  - ***Leases***
  - **AFS-style whole-file caching**
- When client makes a read RPC to the file server, it gets a read lease on the particular file.
- In Shark default lease duration = 5mins.
- For File Commonalities : It uses LBFS.

COOPERATIVE CACHING



DISTRIBUTED INDEXING

- Shark uses global distributed index for all shark clients.
- System maps opaque keys onto nodes by hashing the value onto a key ID.
- Assigning ID's to nodes allows lookup on the O(log(n))
- Shark stores only small information about which clients stores what data.
- Shark uses Coral as its distributed index.
- Coral Provides Distributed Sloppy Hash Table (DSHT).
- Coral caches key/value pairs at nodes whose IDs are close.

IMPLEMENTATION

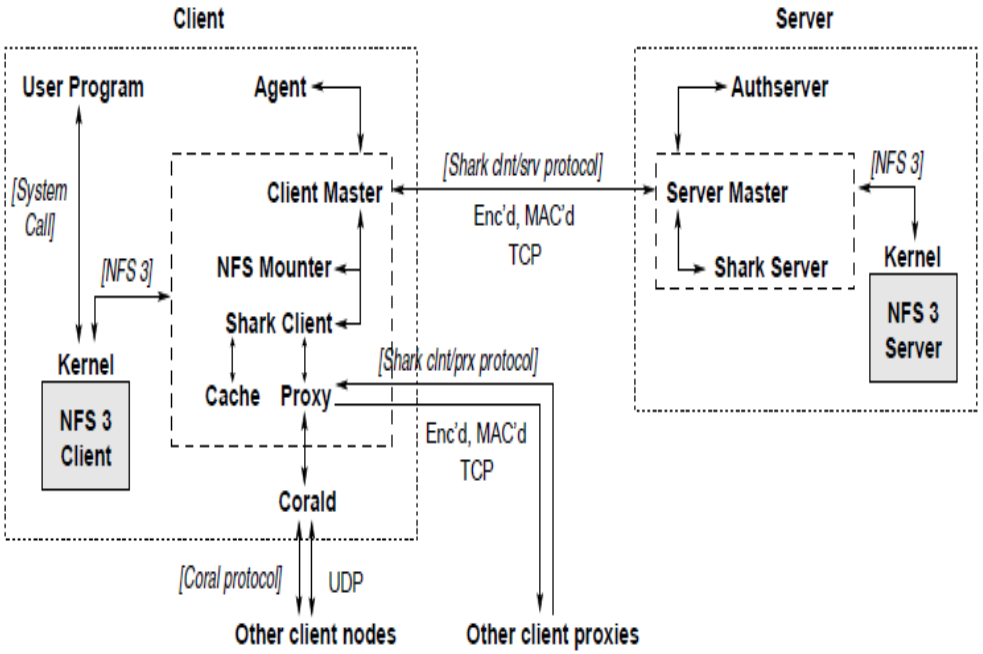


Figure 5: The Shark system components

- **Shark consists of 3 Main Components:**
  - Server side daemon
  - Client side daemon
  - Coral daemon

- Implemented in C++ and are built using SFS toolkit.

- **Client side daemon**
  - Biggest Component of shark.
  - Handles User Requests
  - Transparently incorporates whole file caching.
  - Code is ~12,000 Lines

EVALUATION (1/2)

- Shark is evaluated against NFSv3 and SFS.

- Read tests are performed both
  - within the controlled Emulab LAN environment and
  - In the wide area on the PlanetLab v3.0 test-bed.

- The server required **0.9 seconds** to compute chunks for a **10 MB** random file, and **3.6 seconds** for a **40 MB** random file.

EVALUATION - Micro benchmarks

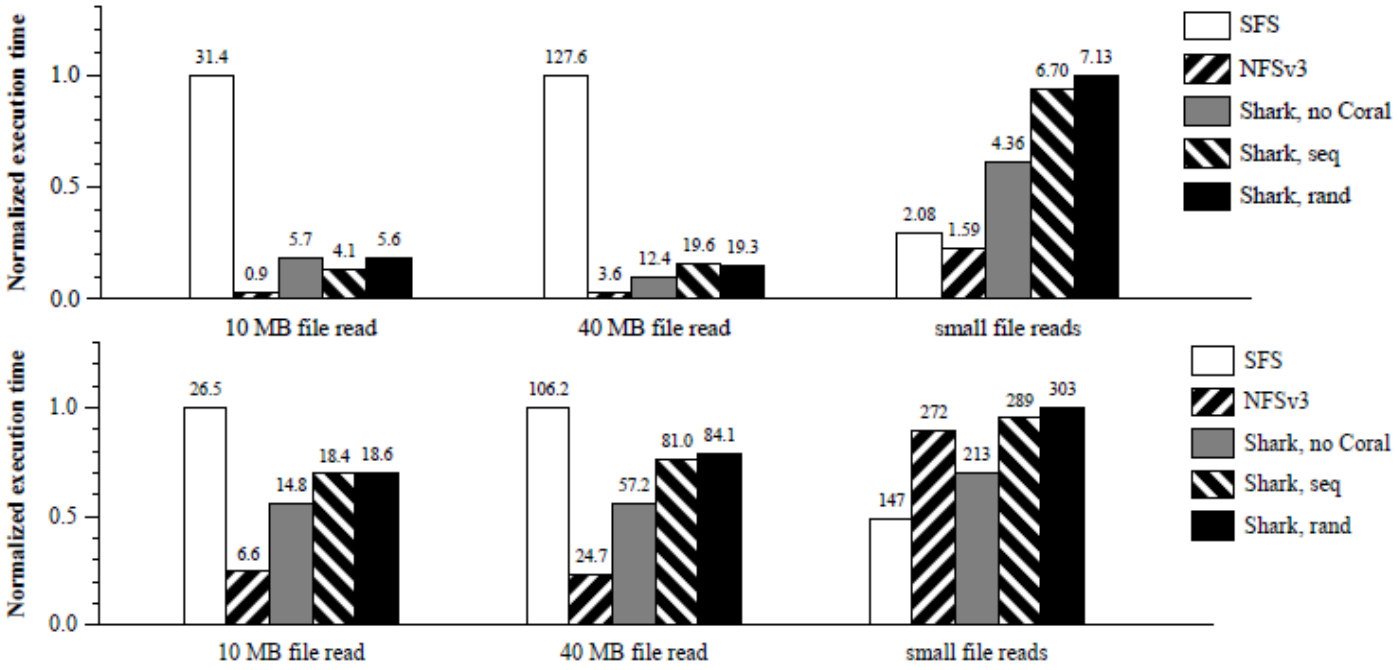


Figure 6: *Local-area* (top) *and wide-area* (bottom) *microbenchmarks.* Normalized application performance for various types of file-system access. Execution times in seconds appear above the bars.

- For the local-area micro-benchmarks, local machines at NYU are used as a Shark client.
- In this micro-benchmark, Shark's chunking mechanism reduces redundant data transfers by exploiting data commonalities.

QUESTIONS

THANK YOU ☺