

A Low-bandwidth Network File System

Athicha Muthitacharoen, Benjie Chen

MIT Lab for Computer Science

David Mazières

NYU Department of Computer Science

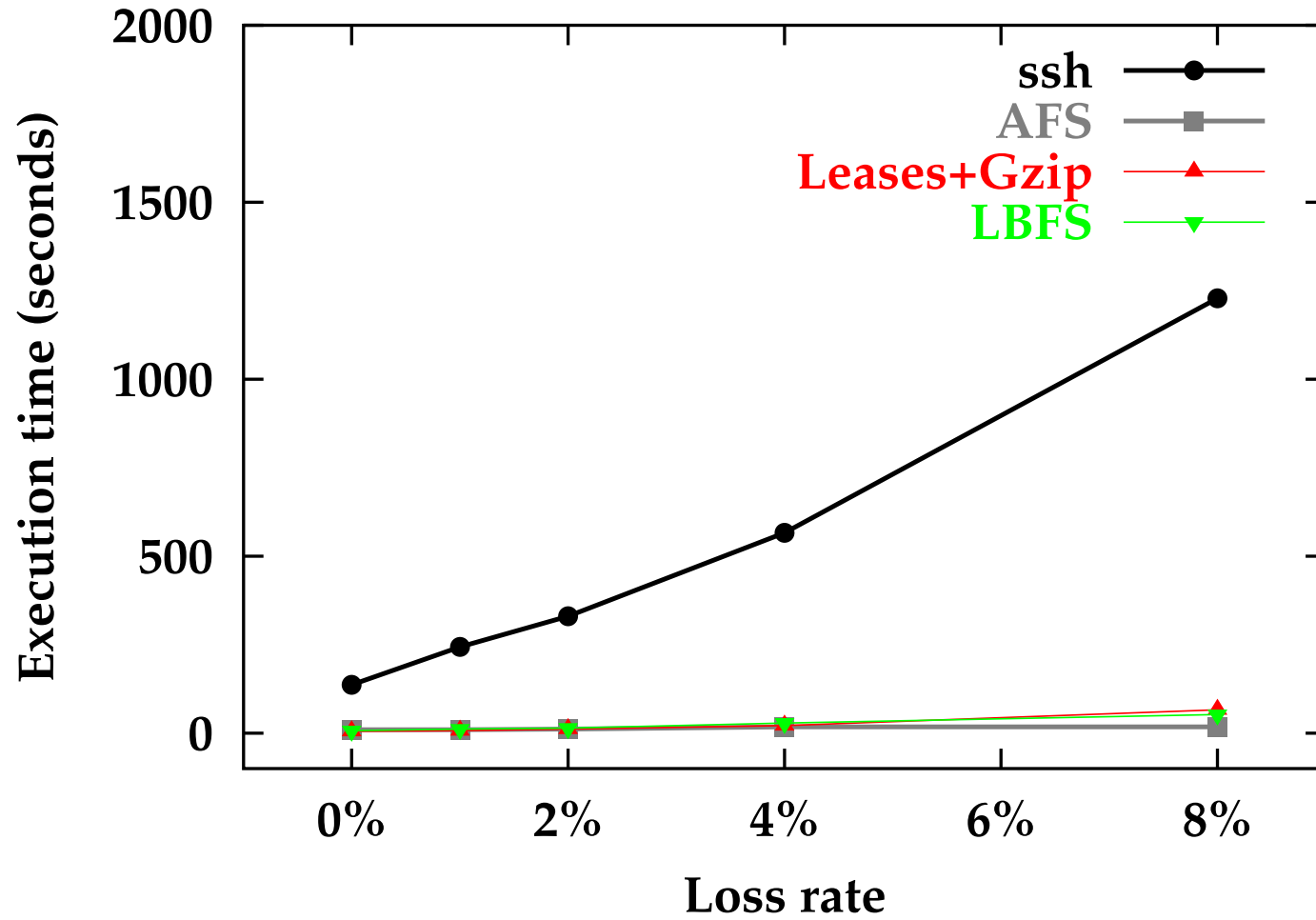
Motivation

- **Network file systems are a useful abstraction...**
- **But few people use them over wide-area networks**
 - E.g., people who travel use network file systems
 - But don't use them over 802.11b while traveling
 - FSes used over WAN provide non-traditional semantics
- **Network file systems require too much bandwidth**
 - Saturate bottleneck links
 - Interfere with other users
 - Block processes for seconds while waiting for network

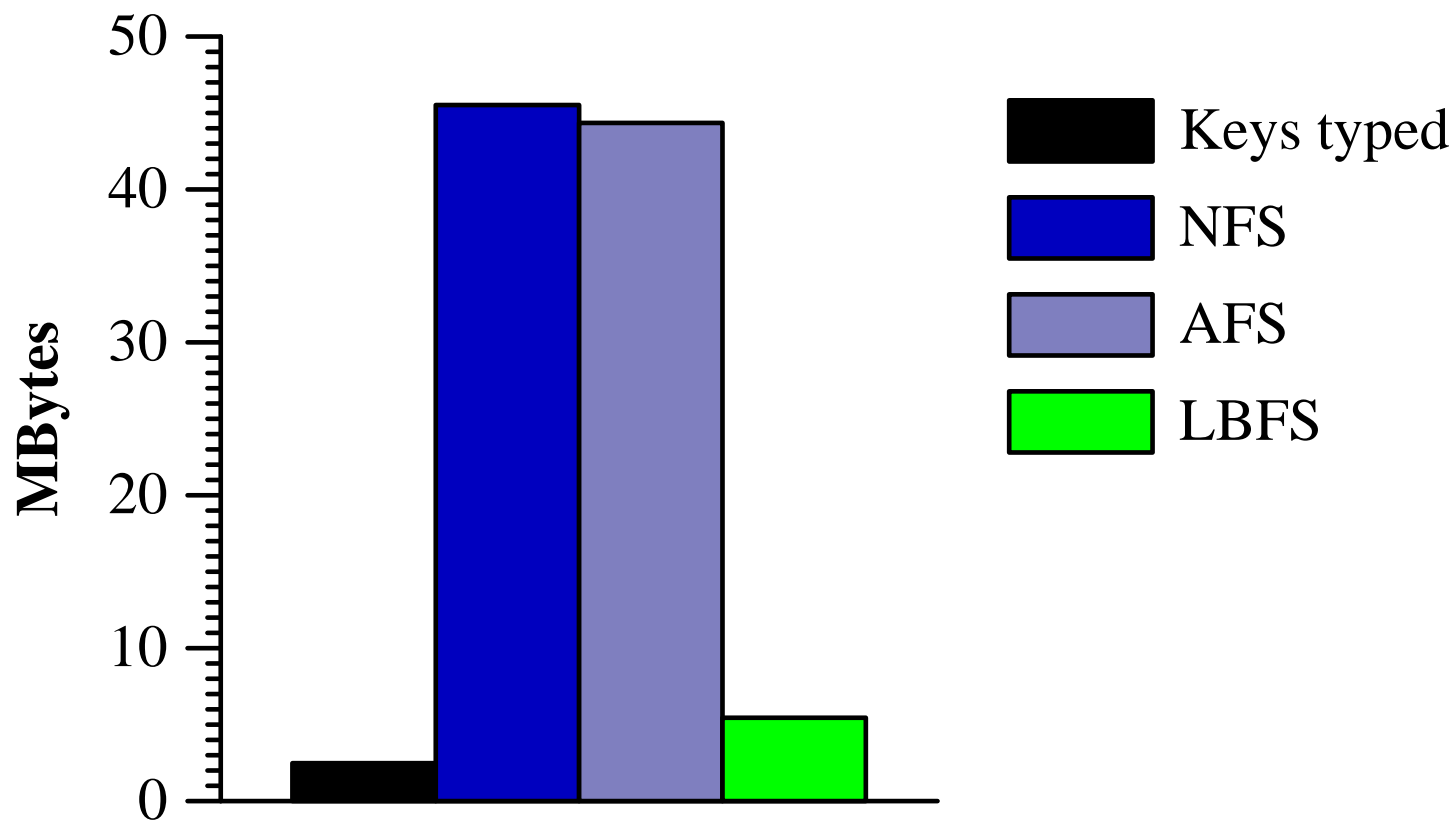
Other ways of accessing remote data

- **Relax consistency semantics (CODA, CVS, ...)**
 - Many applications need strict consistency (email, RCS, ...)
- **Copy files back and forth to work on them**
 - Threatens consistency—where is latest version?
 - Not all files will work if copied (symlinks, CVS/Root, ...)
- **Use remote login to work on files remotely**
 - Graphical applications require too much bandwidth (figure editors, postscript previewers, ...)
 - Interactive programs sensitive to latency and packet loss
 - Delayed character echoes are extremely frustrating!

Remote login frustration!



Client→server bandwidth



Observation: Much inter-file commonality

- **Editing/word processing workloads**
 - Often only modify one part of a large file
 - Generate “autosave” files with mostly redundant content
- **Software development workloads**
 - Modify header & recompile → recreate similar object files
 - Concatenate object files into a library
- **LBFS: Exploit commonality to save bandwidth**
 - Won't always work, but big potential savings

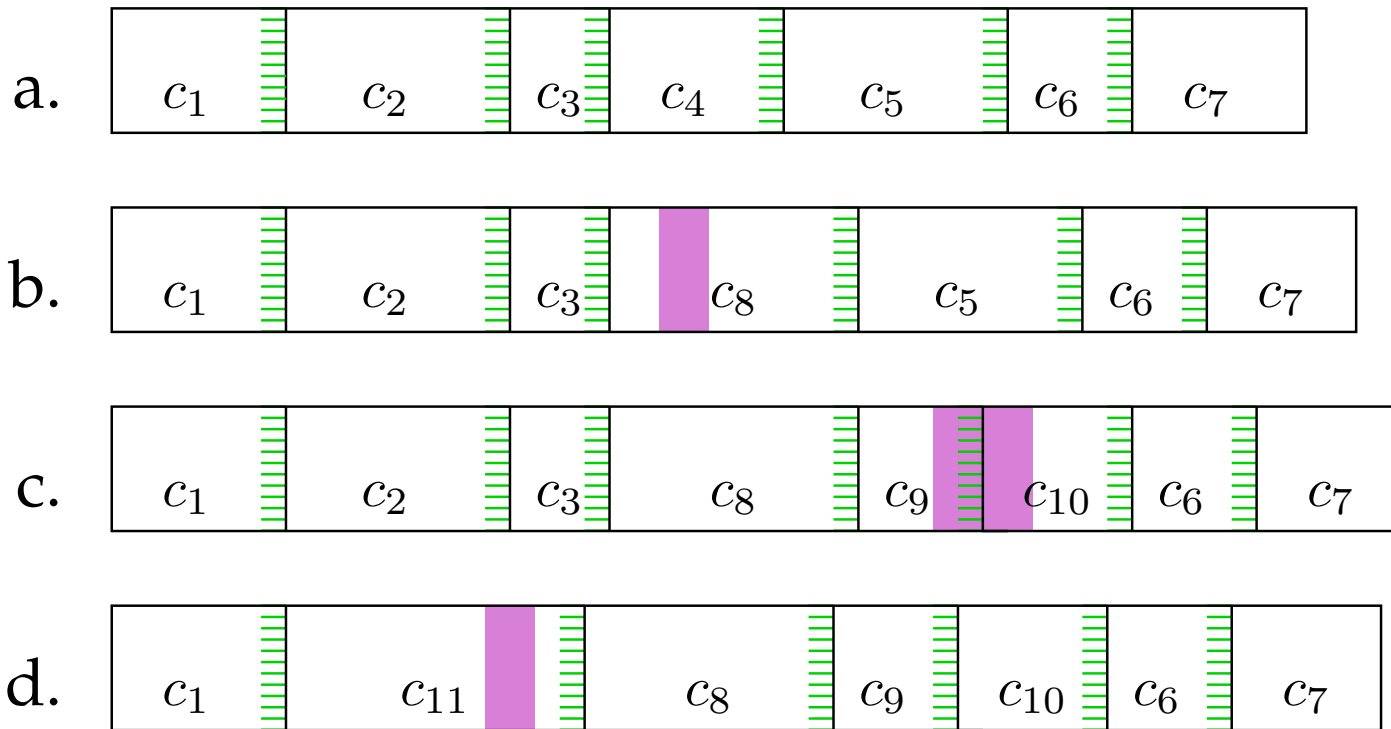
Avoiding redundant data transfers

- **Identify blocks by collision-resistant hash**
- **To transfer a file between client and server**
 - Break file into $\sim 8K$ data chunks
 - Send hashes of the file's chunks
 - Only send chunks actually needed by recipient
- **Index file system and client cache to find chunks**
 - Keep database mapping hash \rightarrow $\langle \text{file, offset, len} \rangle$
 - Use chunks from any file in reconstructing any other

Dividing files into chunks

- **Straw man: Split file into aligned 8K chunks**
 - Inserting one byte at start of file changes all chunks
- **Base chunks on file contents, not position**
 - Allow variable-length chunks
 - Compute running hash of every overlapping 48-byte region
 - If hash mod 8K is special value, create chunk boundary
- **Chunk boundaries insensitive to shifting offsets**
 - Inserting/deleting data only effects surrounding chunk(s)

Example: Breaking a file into chunks



Pathological cases

- **Tiny chunks**

- E.g., caused by unlucky 48-byte region repeated
- Sending hashes consume more bandwidth than data

- **Enormous chunks**

- E.g., long run of all zeros
- Hard to handle (can't hold chunks in memory)

- **Solution: Impose min/max chunk sizes (2K/64K)**

- Could conceivably derail alignment
- Just an optimization, can afford low-probability failures
- “Problem-cases” often very compressible!

LBFS overview

- **Provides traditional file system semantics**
 - Close-to-open consistency
 - Data safely stored on server before close returns
- **Large client cache holds user's working set**
 - Eliminates all communication not required for consistency
 - When user modifies file, must write through to server
 - When different client modifies file, download new version
- **Elides transfers of redundant data**
- **Conventionally compresses remaining traffic**

LBFS protocol

- **Derived from the NFS protocol**
- **Adds more aggressive caching**
 - Persistent, on-disk cache holds user's entire working set
 - Callbacks & Leases save an RPC for many open/stat calls
- **Client and server index data chunks with a B-tree**
- **Five new RPCs exploit inter-file commonality**
 - GETHASH – like read, but returns hashes not data
 - CONDWRITE – a write that takes a hash instead of data
 - 3 RPCs for atomic file updates

Read caching

- **Leases let client validate cached attributes**
 - Most file operations grant client a lease on attributes
 - Server must notify client if attributes change while leased
- **Attributes let client validate cached file contents**
 - Check modification/change times
- **When client must download a file**
 - Retrieve file's chunk hashes with GETHASH
 - Request chunks not already in cache using normal READs
 - Update the local chunk index to reflect new cache data

Read protocol

Client

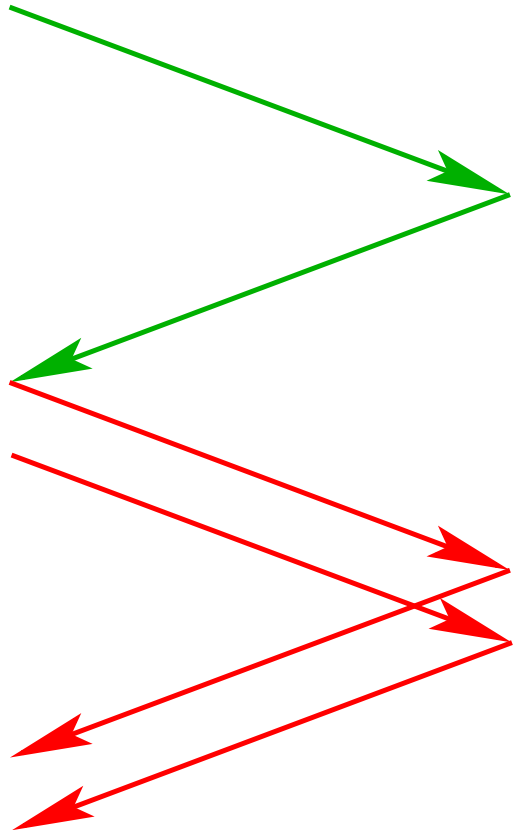
Server

GETHASH

(hash1, size1)
(hash2, size2)
(hash3, size3)
EOF

READ
READ

data2
data3



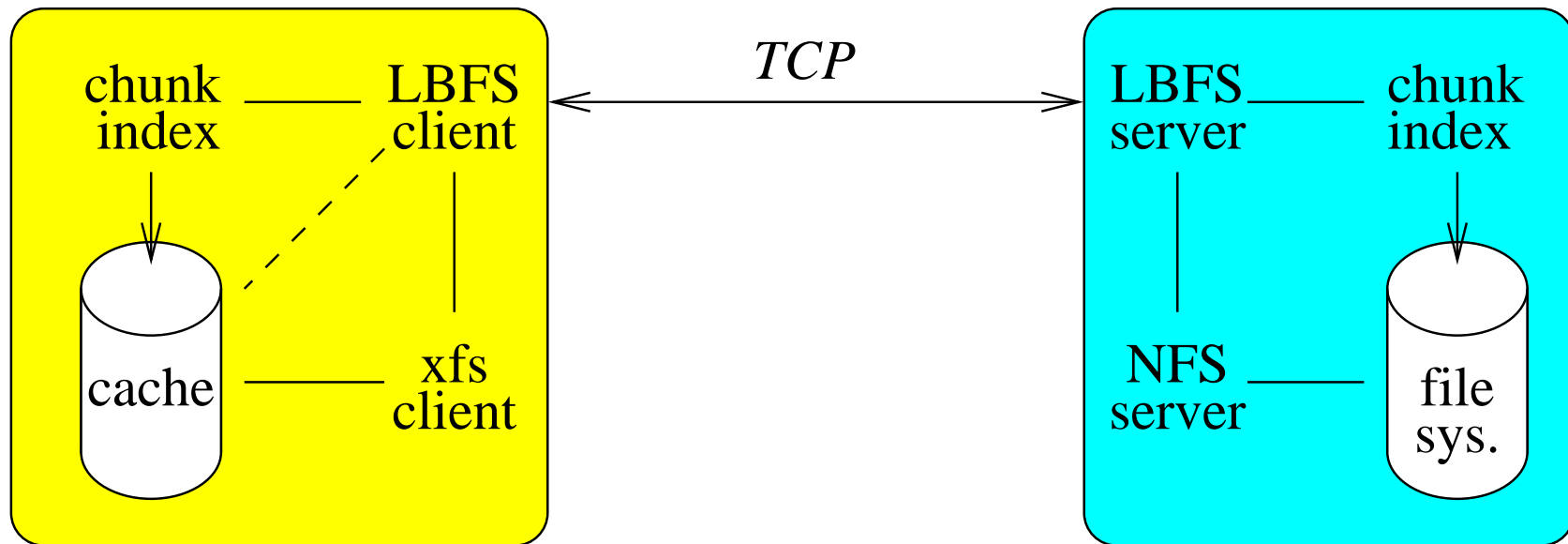
Writing back a modified file

- **Idea: First send hashes, then missing data**
- **Complications:**
 - New file likely contains many chunks it is overwriting
 - Unaligned writes can be expensive (cause disk read)
 - Reordering writes creates confusing intermediary states
 - What if client crashes in the middle of sending file?
- **Solution: Atomic updates**
 - Write data to new temporary file
 - Commit contents of temporary file to file being written

Atomic update RPCs

- **MKTMPFILE RPC creates a temporary file**
 - File named by client-chosen descriptor
- **CONDWRITE sends hashes of chunks**
 - Can be immediately pipelined behind MKTMPFILE
 - Server writes chunk if in DB, else returns NOTFOUND
- **TMPWRITE sends data for NOTFOUND chunks**
- **COMMITTMP copies temporary file to target file**
- **Server updates chunk index**

Implementation

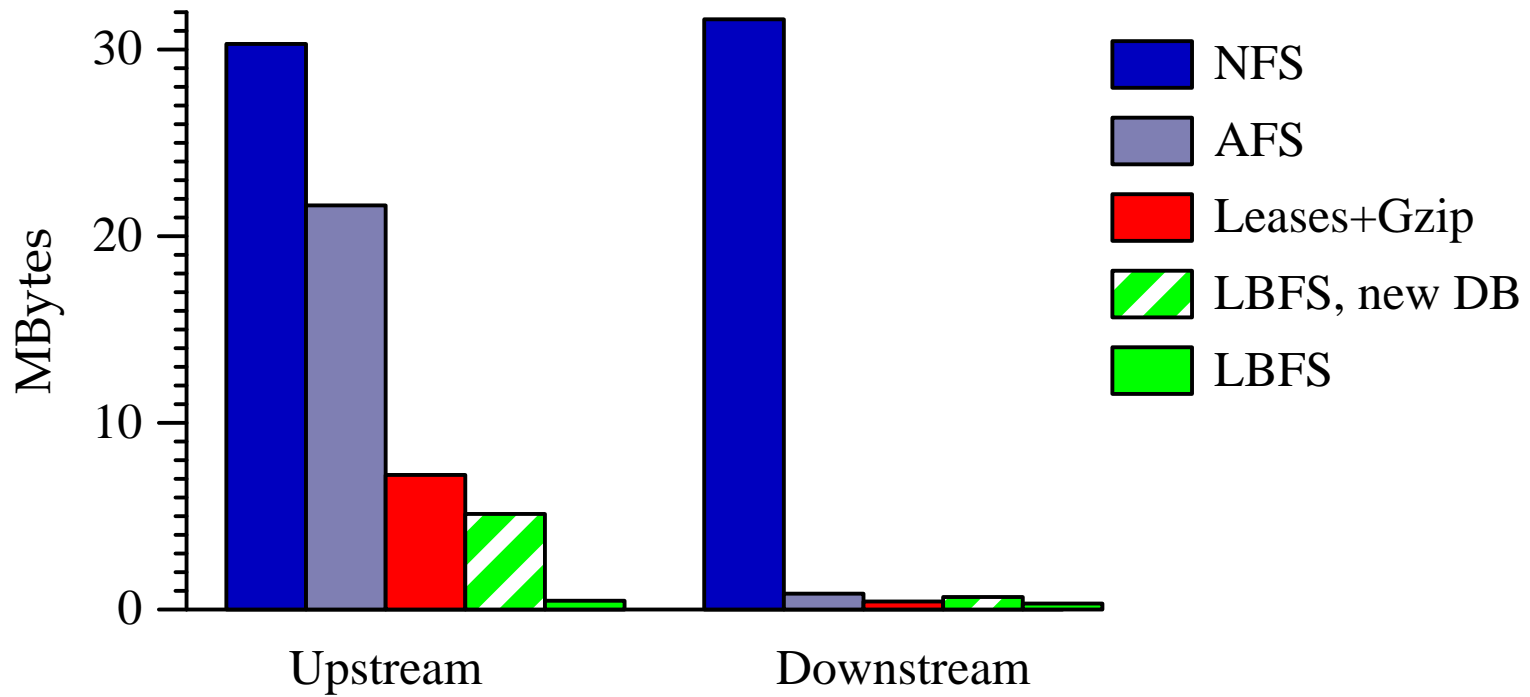


- **Client** – uses *xfst*, device driver of ARLA AFS clone
- **Server** – accesses FS by pretending to be NFS client
- **Index** – uses BerkeleyDB B-tree

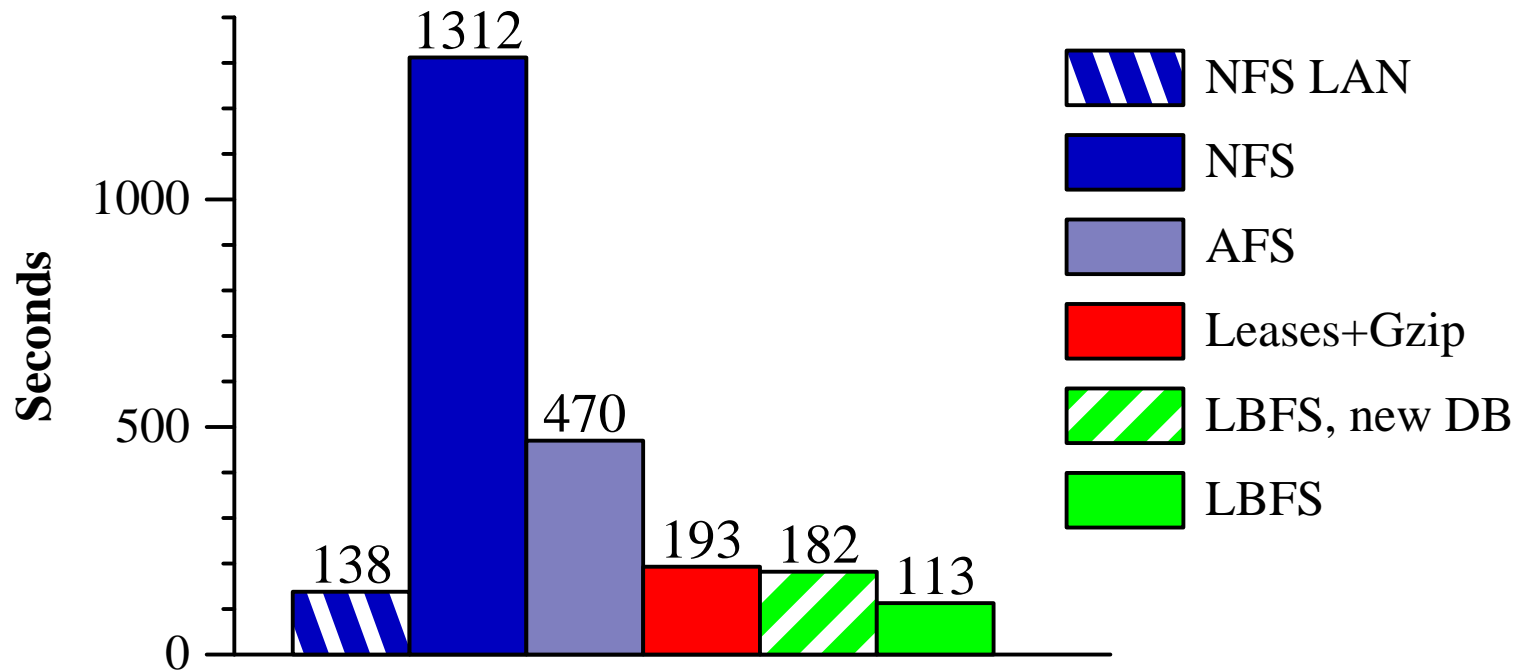
Implementation details

- **Never assume chunk index is correct**
 - Automatically fix errors as encountered
 - No need for expensive crash-recovery precautions
 - Allows server to be updated by non-LBFS clients
- **Keep old temporary files around**
 - Often contain useful chunks for subsequent files
 - Move to trash directory, evict in FIFO order
- **Background thread deletes invalid DB entries**

Bandwidth: emacs recompile



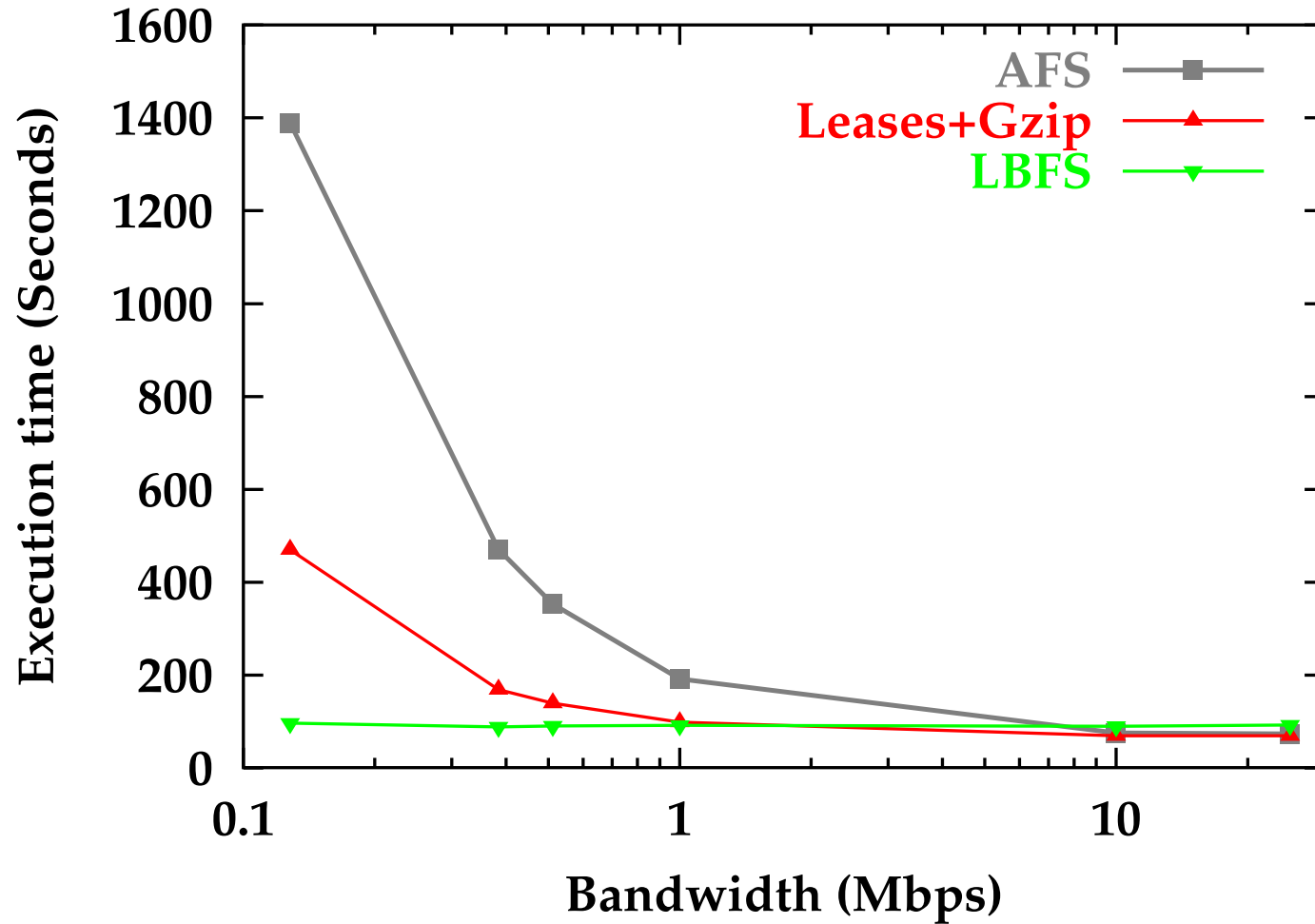
Performance: emacs recompile



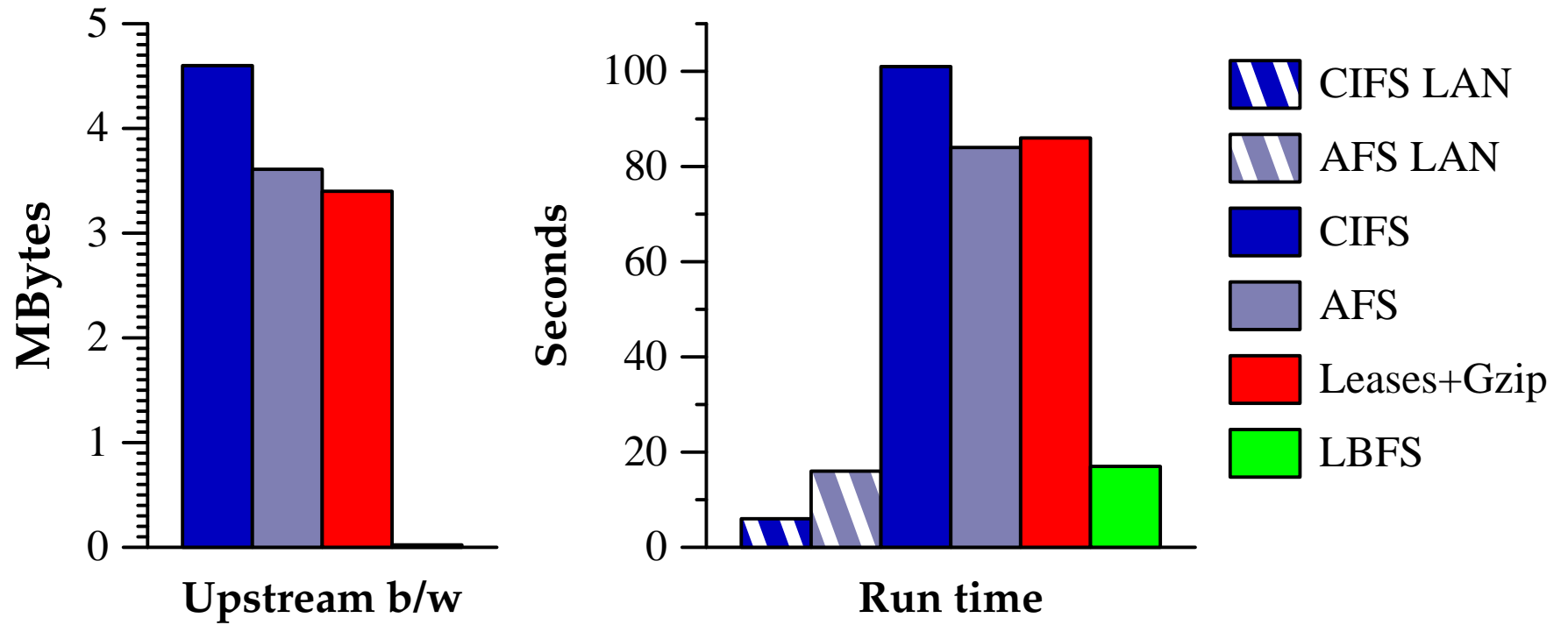
- **Evaluated over simulated ADSL line**

- 1.5 Mbit/sec downstream, 348 Kbit upstream, 30 ms latency
- LBFS on ADSL beats NFS on 100Mbit/sec LAN

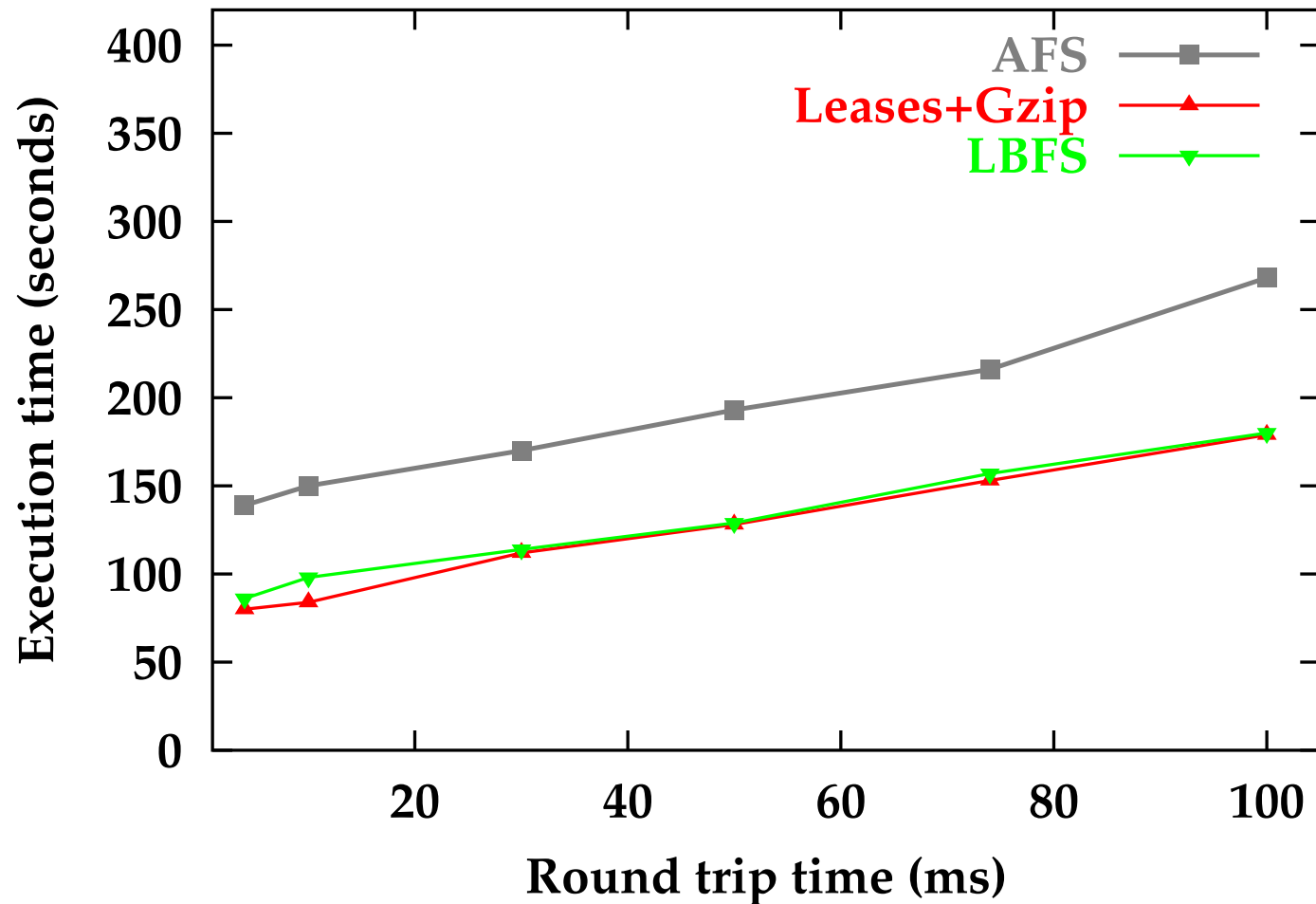
Compile time vs. bandwidth



Saving 1.4 MByte MSWord doc



Effect of network latency on performance



Related work

- **Weaken consistency (CODA)**
- **Send deltas (Diff/patch, CVS, xdelta)**
 - Requires server to keep around old versions of files
- **The rsync algorithm (synchronize two files)**
 - One file often contains chunks of many files (e.g., ar)
 - Not obvious which file to choose at receiving end
(emacs: #foo#→foo, RCS: _1v22825→foo, v, ...)

Conclusions

- **Network file system often best way to access data**
 - Copying files back and forth threatens consistency
 - Remote login frustrating given latency or packet loss
- **Most file systems too bandwidth-hungry for WAN**
- **LBFS exploits file commonality to save bandwidth**
 - Break files into variable-size chunks based on contents
 - Index chunks in file system and client cache
 - Avoid sending chunks already present in other files
- **LBFS works where other file systems impractical**