Low-Overhead Byzantine Fault-Tolerant Storage

James Hendricks, Gregory R. Ganger
Carnegie Mellon University

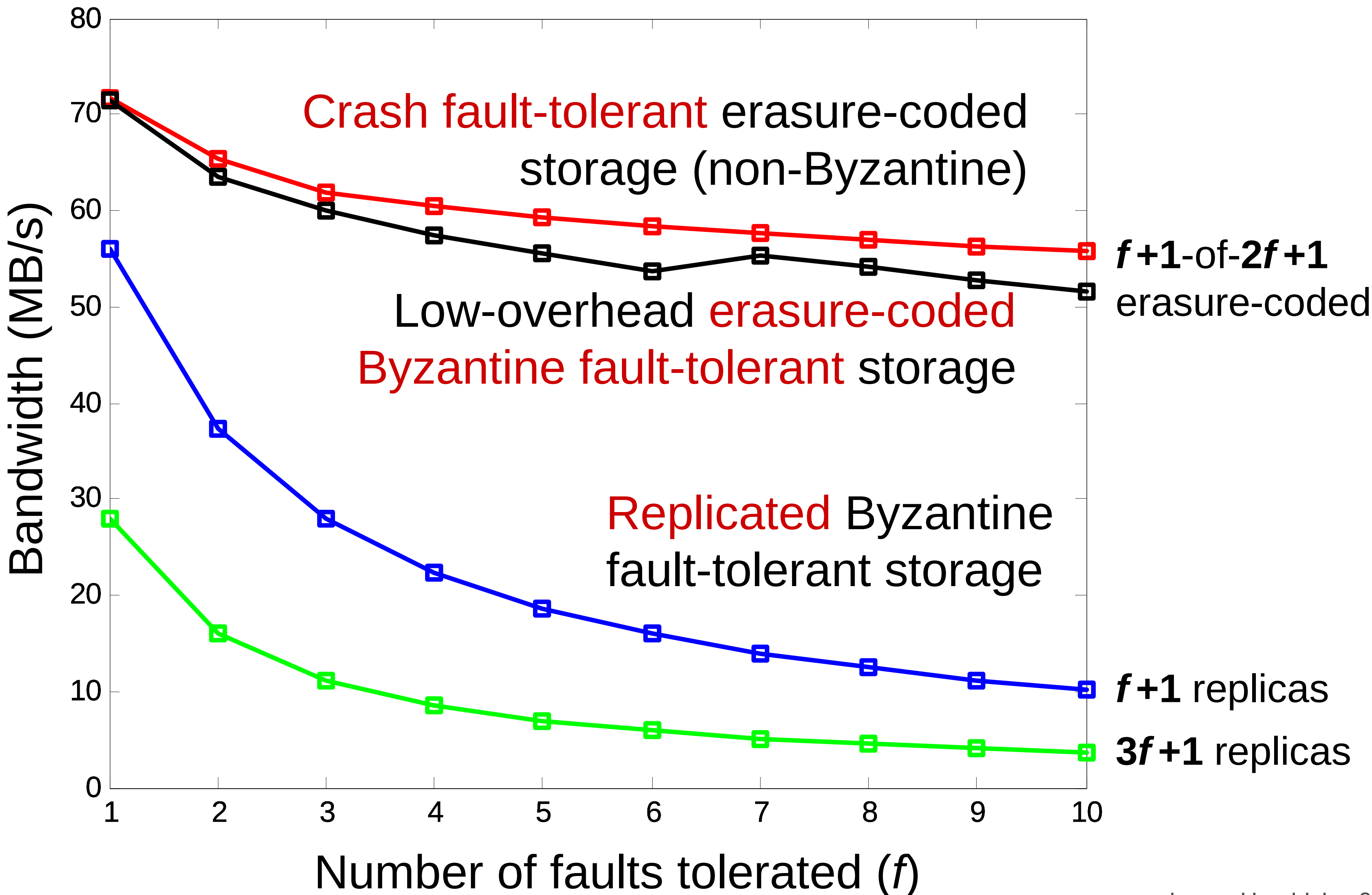
Michael K. Reiter
University of North Carolina at Chapel Hill

Motivation

- As systems grow in size and complexity…
  - Must tolerate more faults, more *types* of faults
  - Modern storage systems take ad-hoc approach
- Not clear which faults to tolerate

- Instead: tolerate arbitrary *(Byzantine*) faults
- But, Byzantine fault-tolerance = expensive?
  - Fast reads, slow large writes

Write bandwidth



Summary of Results

We present a *low overhead* Byzantine fault-tolerant erasure-coded block storage protocol

- Write overhead: 2-round + crypto. checksum
- Read overhead: cryptographic checksum

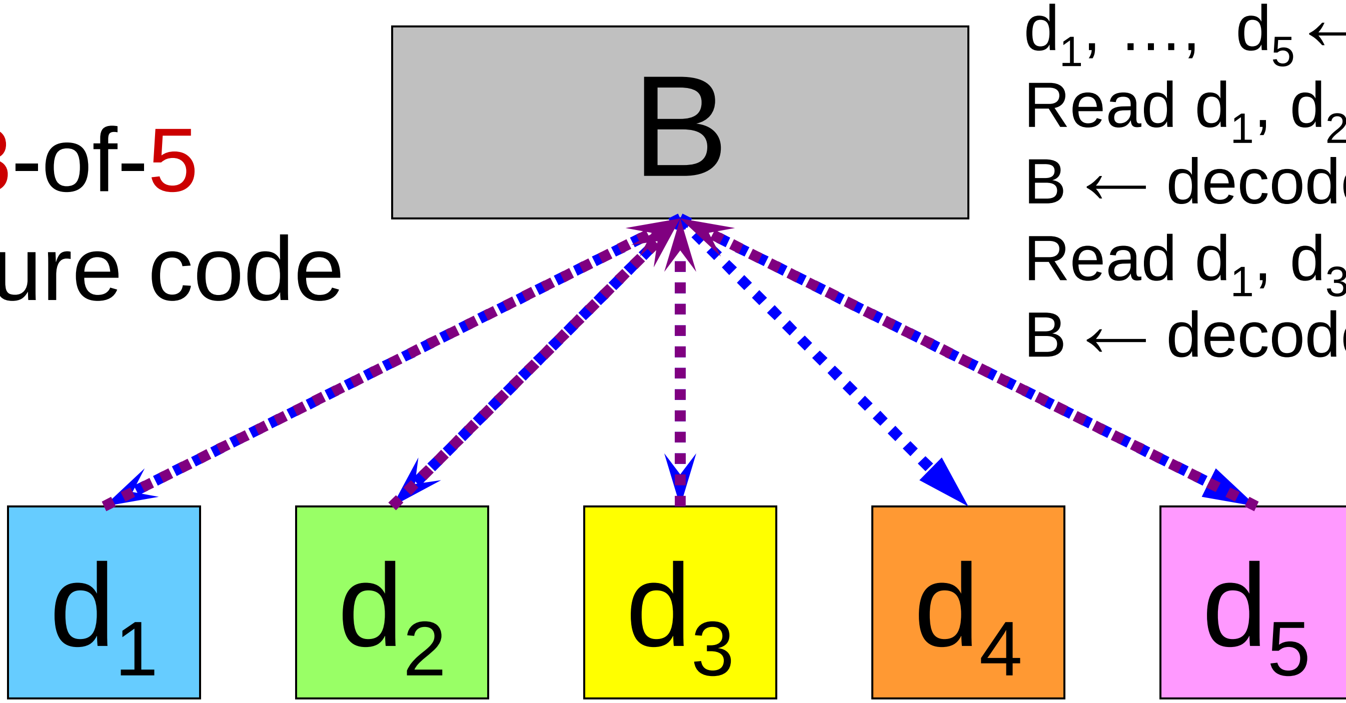
Performance of our *Byzantine*-tolerant protocol nearly matches that of protocol that tolerates *only crashes*

- Within 10% for large enough requests

Erasure codes

An *m-of-n erasure code* encodes block B into *n fragments*, each size |B|/*m*, such that any *m* fragments can be used to reconstruct block B

3-of-5
erasure code



$d_1, \ldots, d_5 \leftarrow$ encode(B)
Read d_1, d_2, d_3
B $\leftarrow$ decode(d_1, d_2, d_3)
Read d_1, d_3, d_5
B $\leftarrow$ decode(d_1, d_3, d_5)

Design of Our Protocol

Parameters and interface

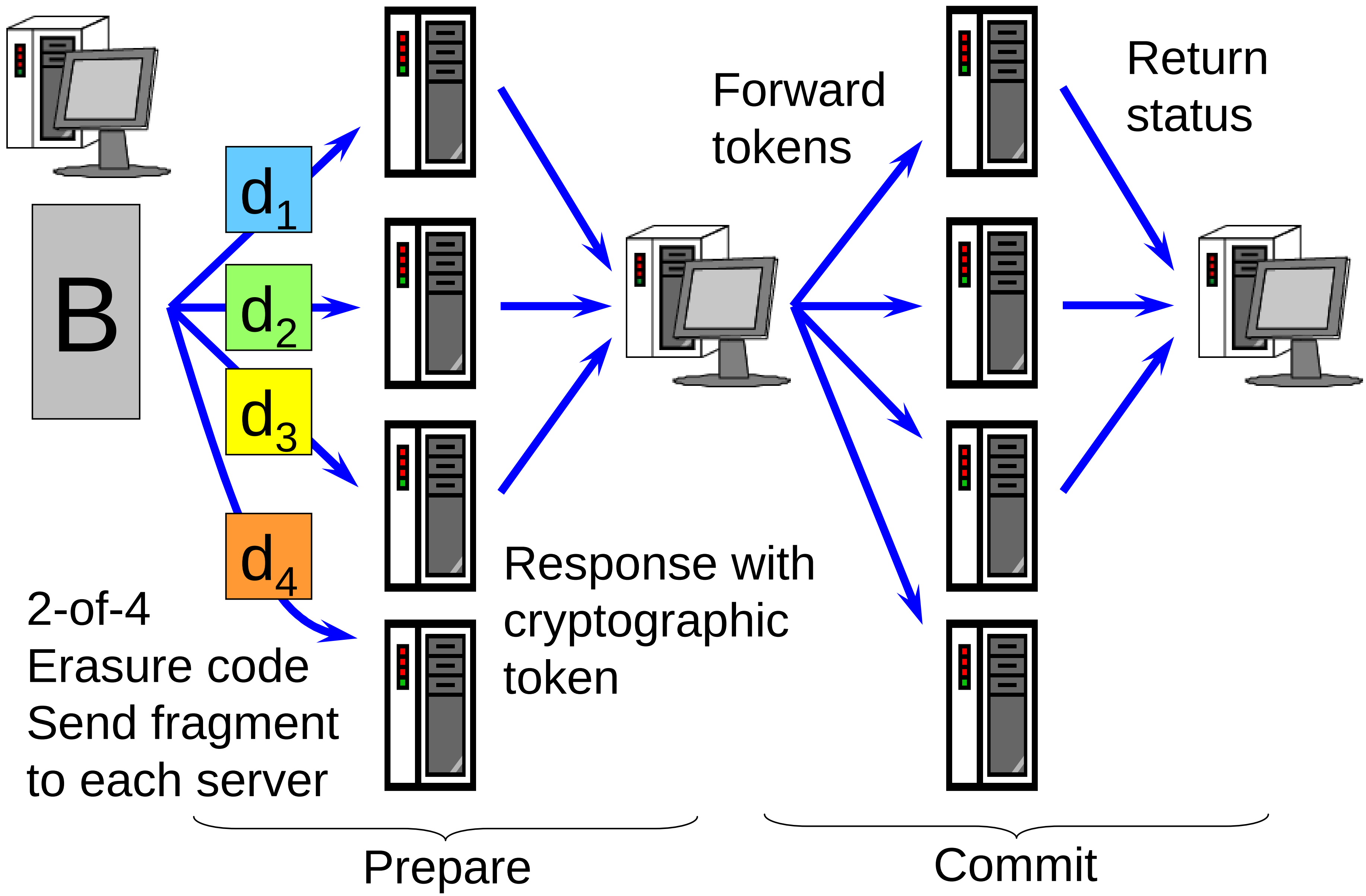
Parameters

- f: Number of faulty servers tolerated
- $m \geq f + 1$: Fragments needed to decode block
- $n = m + 2f \geq 3f + 1$: Number of servers

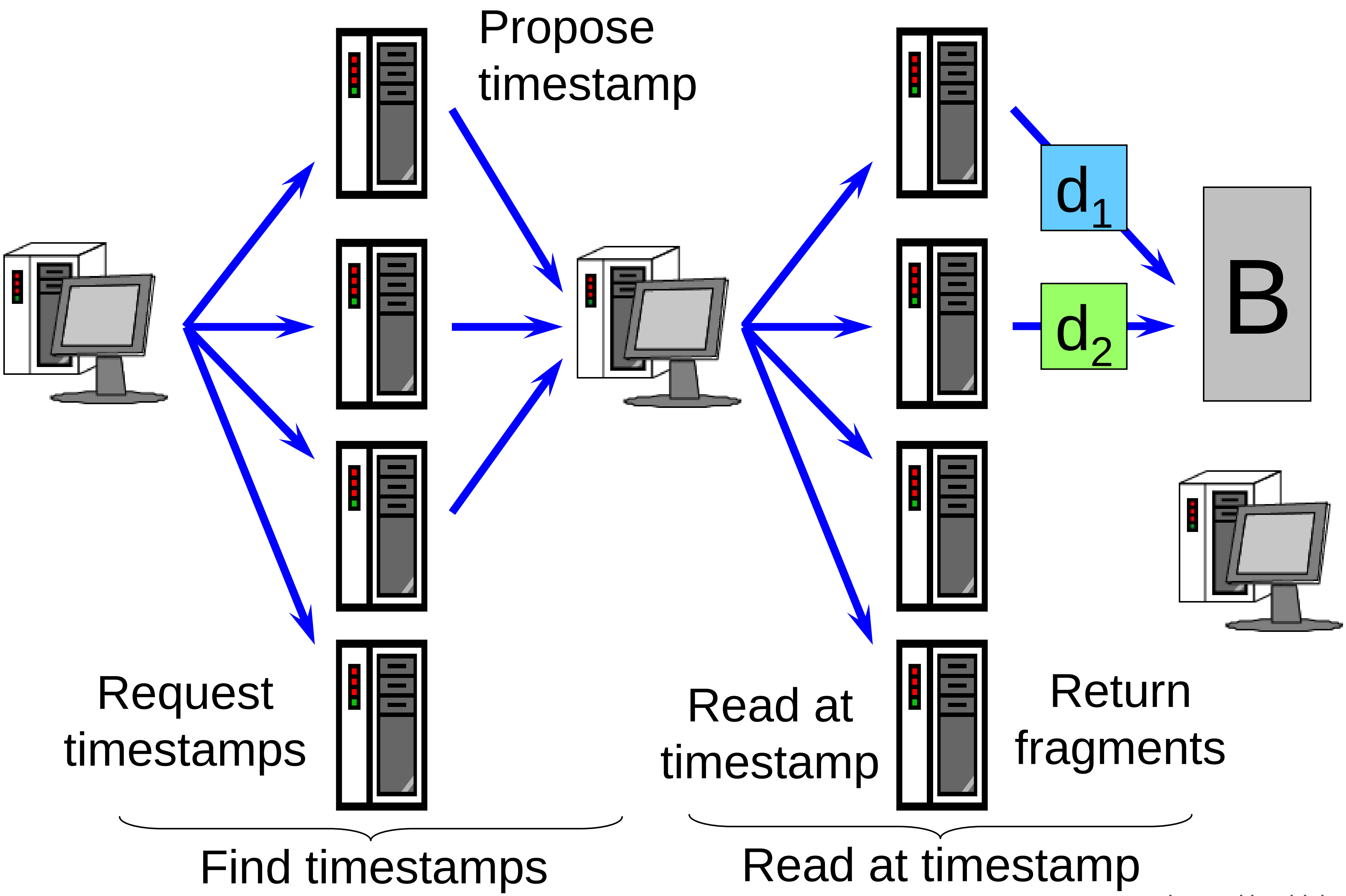
Interface: Read and write fixed-size blocks

- Not a filesystem. No metadata. No locking. No access control. No support for variable-sized reads/writes.
- A *building block* for a filesystem

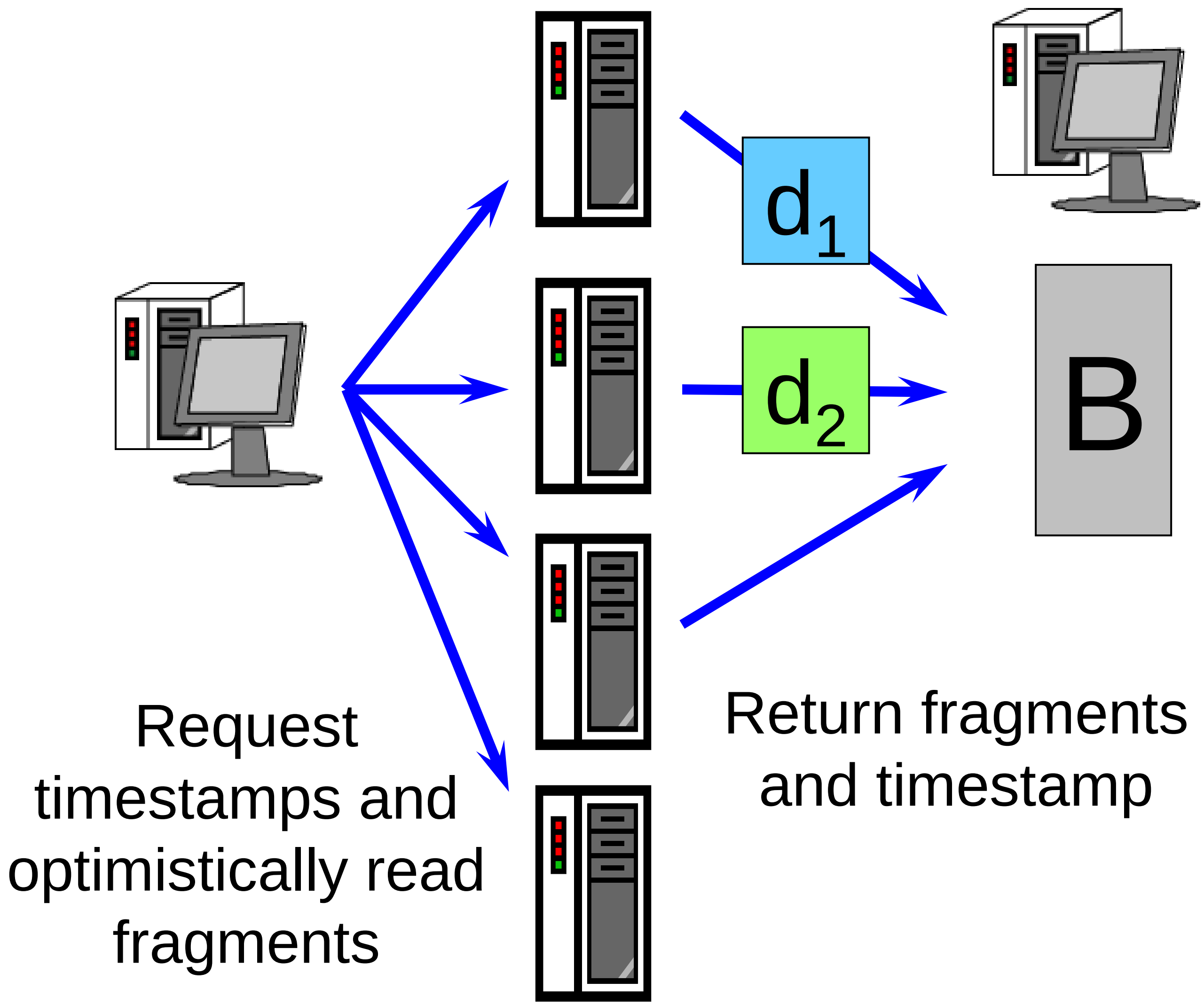
Write protocol: prepare & commit



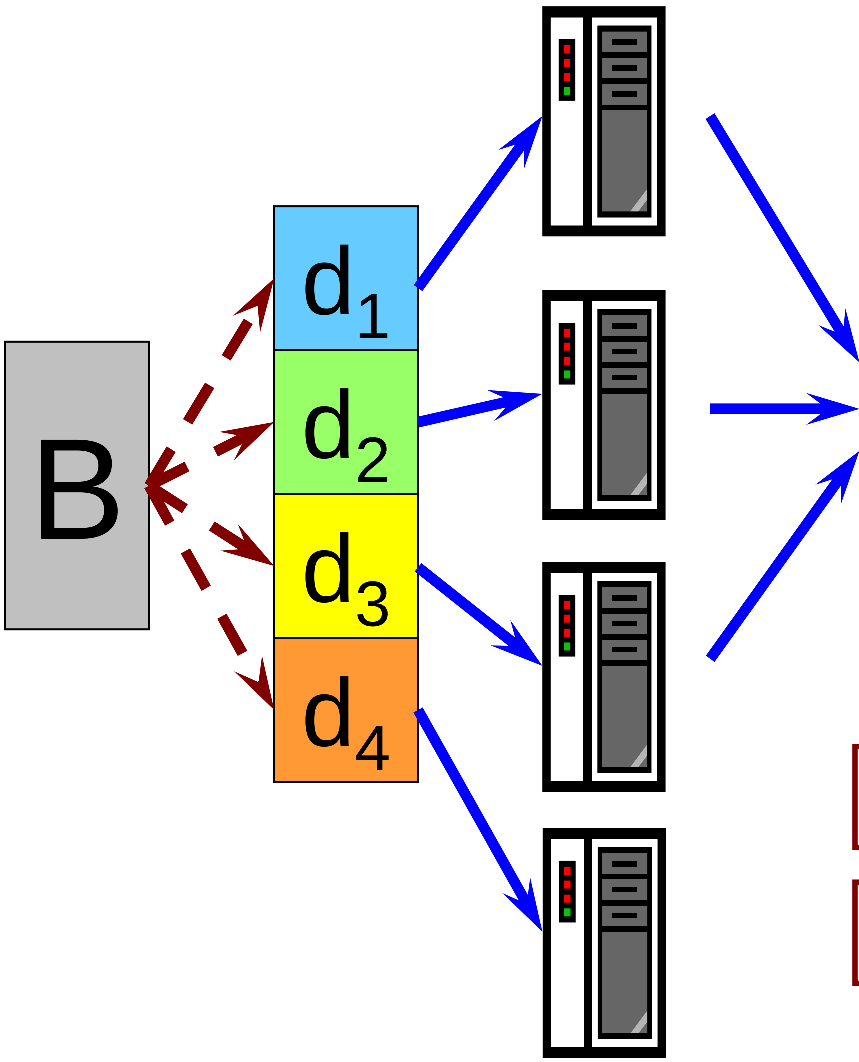
Read protocol: find & read



Read protocol common case



Issue 1 of 3: Wasteful encoding

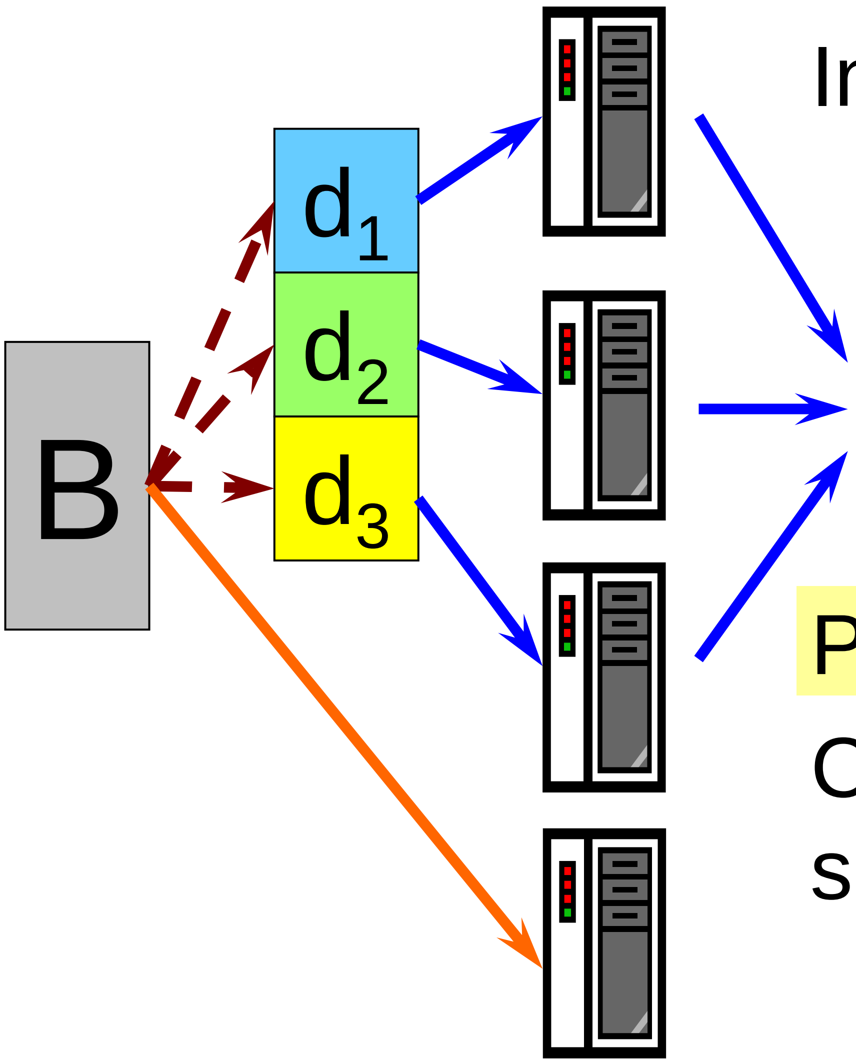


- Erasure code $m + 2f$
- Hash $m + 2f$
- Send to each server
- But only wait for $m + f$ responses
  - → This is wasteful!

m: Fragments needed to decode

f: Number of faults tolerated

Solution 1: Partial encoding



Instead:

- Erasure code *m+f*
- Hash *m+f*
- Hear from *m+f*

Pro: Compute *f* fewer frags

Con: Client may need to send entire block on failure

- Should happen rarely

Issue 2: Block must be unique

Fragments must comprise a unique block

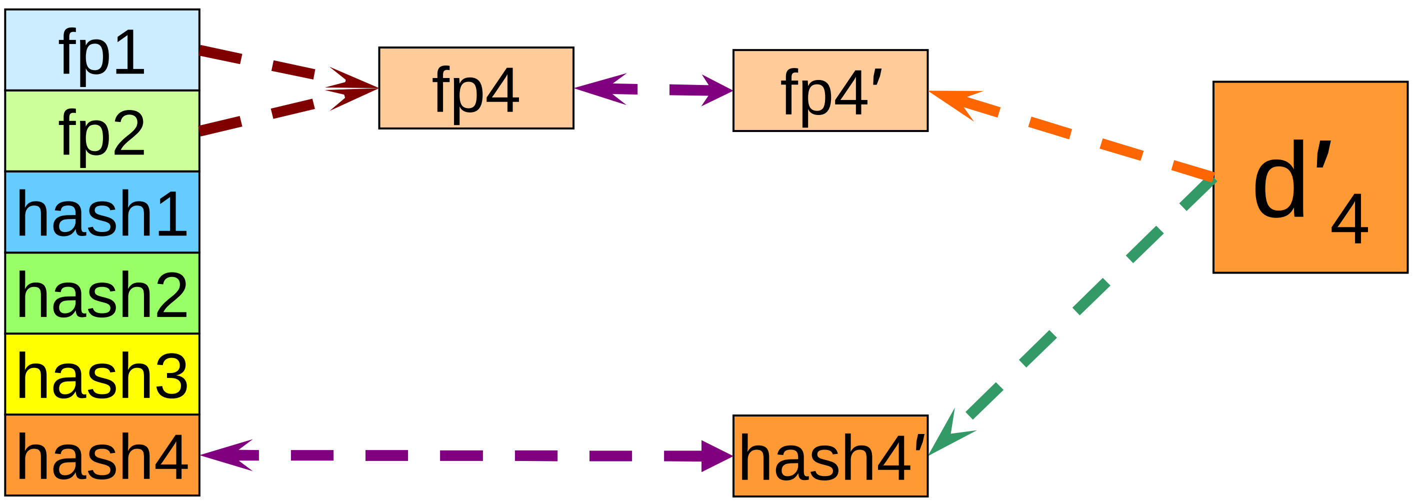
- If not, different readers read different blocks

Challenge: servers don't see entire block

- Servers can't verify hash of block
- Servers can't verify encoding of block given hashes of fragments

Sol’n 2: Homomorphic fingerprinting

Fragment is *consistent* with checksum if hash and homomorphic fingerprint [PODC07] match



Key property: Block decoded from consistent fragments is unique

Issue 3: Write ordering

Reads must return most recently written block

- Required for *linearizability* (atomic)
- Faulty server may propose uncommitted write
- Must be prevented. Prior approaches: $4f+1$ servers, signatures, or 3+ round writes

Our approach:

- $3f+1$ servers, MACs, 2 round writes

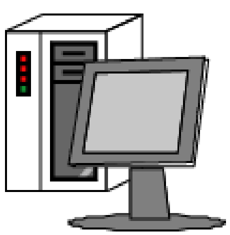
Solution 3: Hashes of nonces

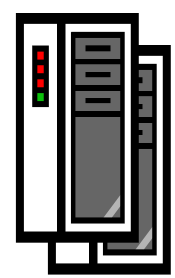
Write

Client		Servers
Prepare	→	Prepare: Store hash(nonce)
Collect nonces	← nonce	Return nonce
	→	Commit: store nonces
	←	

Read

Client		Servers
Find timestamps	→	Return timestamp, nonces
Read at timestamp	← nonces	
	→	Return nonce_hash with fragment
Compare hash(nonce) with nonce_hash	←	

Client 

 Servers

Bringing it all together: Write

- Erasure code m+f fragments
- Hash & fingerprint fragments
- Send to first m+f servers

d_i

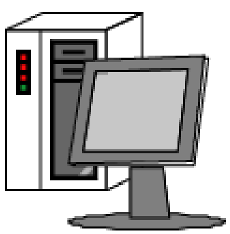
- Verify hash, fingerprint
- Choose nonce
- Generate MAC
- Store fragment

- Forward MACs to servers

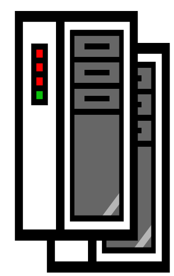
- Verify MAC
- Free older fragments

- Write completed

Client



Overhead: Not in crash-only protocol



Servers

Bringing it all together: Read

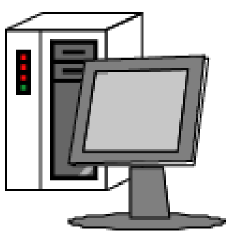
- Request fragments from first *m* servers
- Request latest nonce, timestamp, checksum

- Return fragment (if requested)
- Return latest nonce, timestamp, checksum

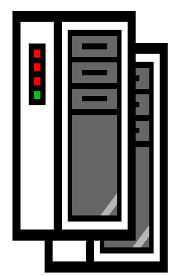
d_i

- Verify provided checksum matches fragment hash&fp
- Verify timestamps match
- Verify nonces
- Read complete

Client



Overhead: Not in crash-only protocol



Servers

Evaluation

Experimental setup

- $m = f + 1$
  - Number of faults tolerated (f)
  - Fragments needed to decode block (m)
- Single client, NVRAM at servers
- Write or read 64 kB blocks
  - Fragment size decreases as f increases
- 3 GHz Pentium D, Intel PRO/1000

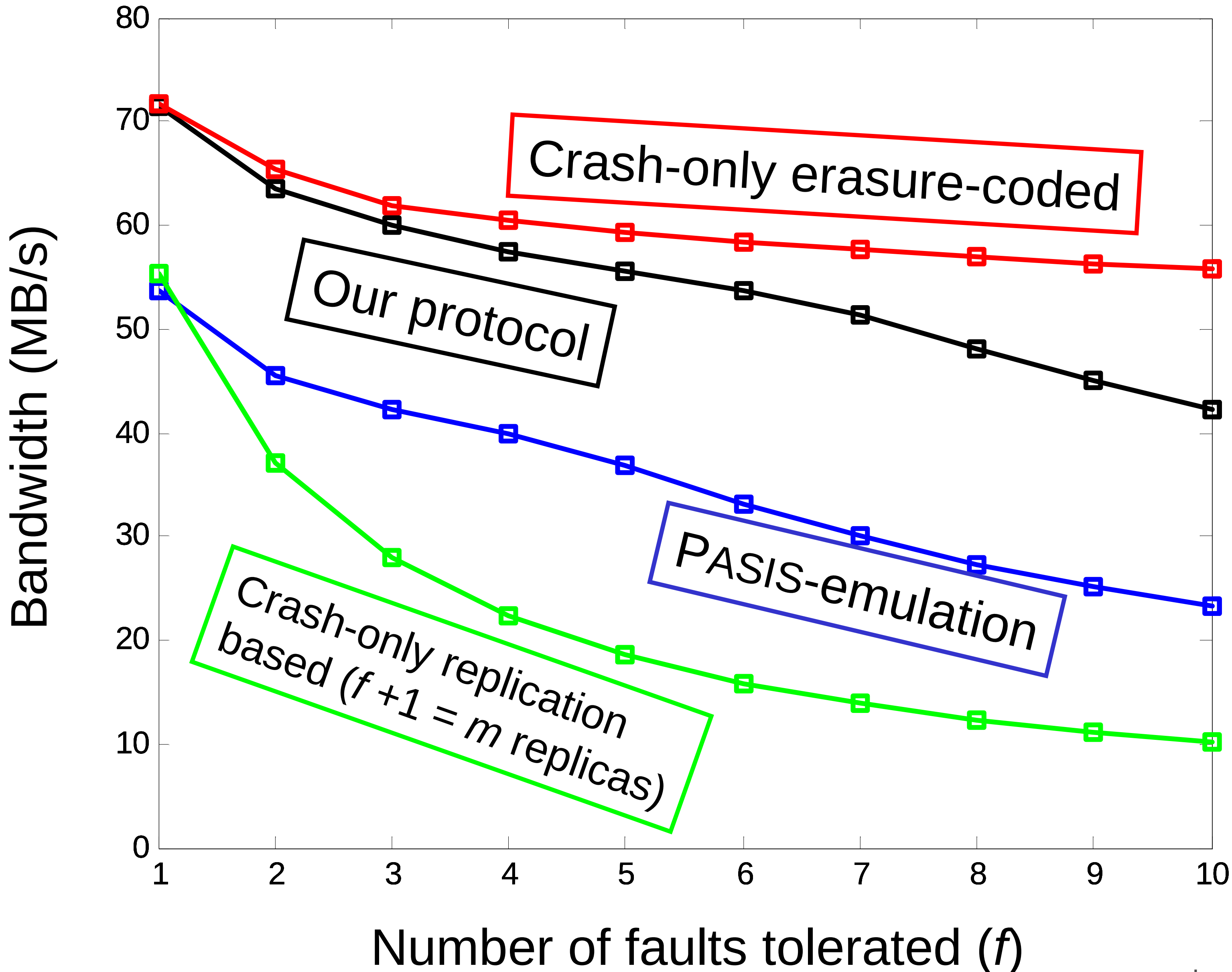
Prototype implementation

Four protocols implemented:	Erasure coded	Byzantine tolerant
Our protocol	➤	➤
Crash-only erasure-coded	➤	
Crash-only replication-based		
PASIS [Goodson04] emulation	➤	➤

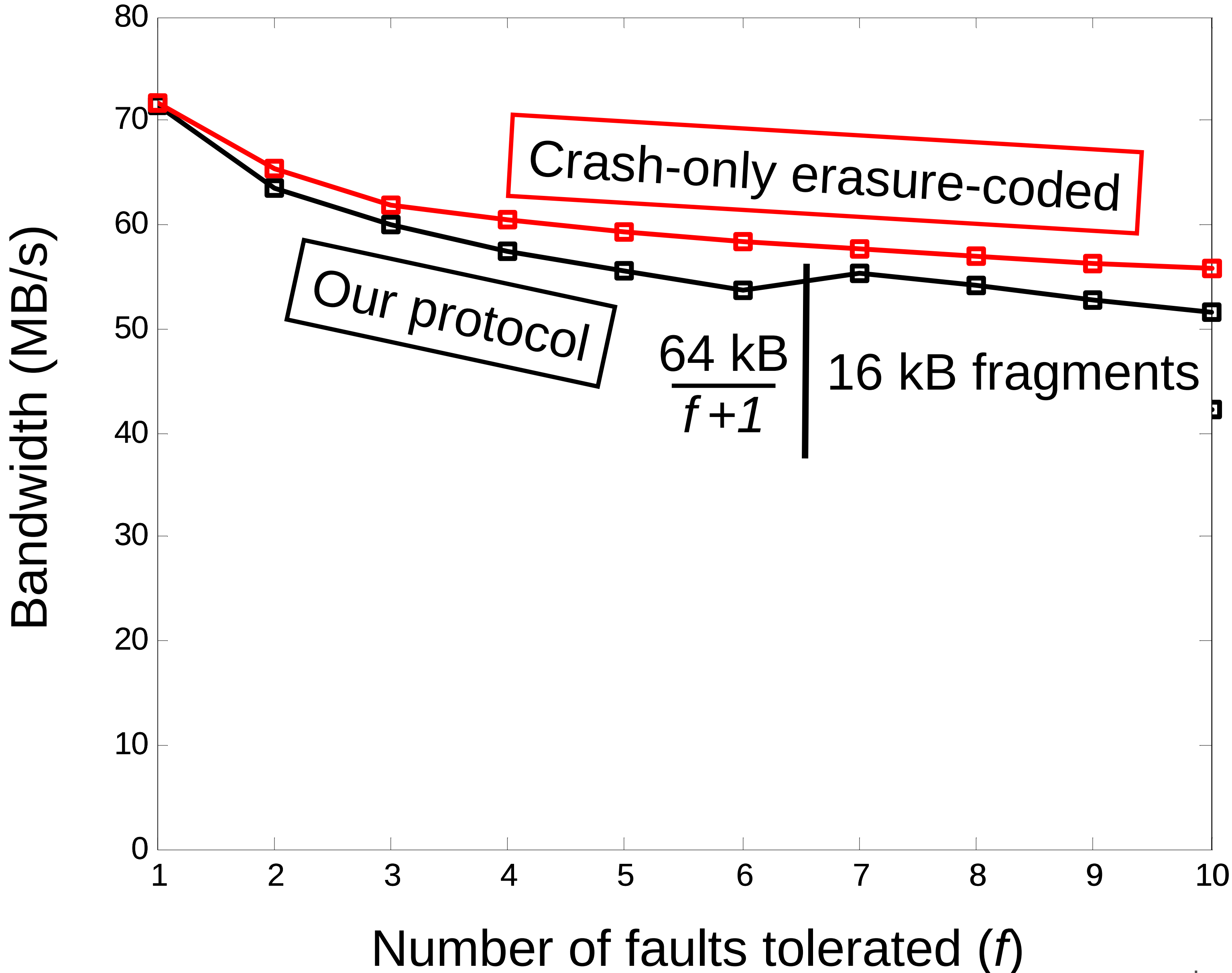
- Read validation: Decode, encode, hash 4f+1 fragments
- 4f+1 servers, versioning, garbage collection

All use same hashing and erasure coding libraries

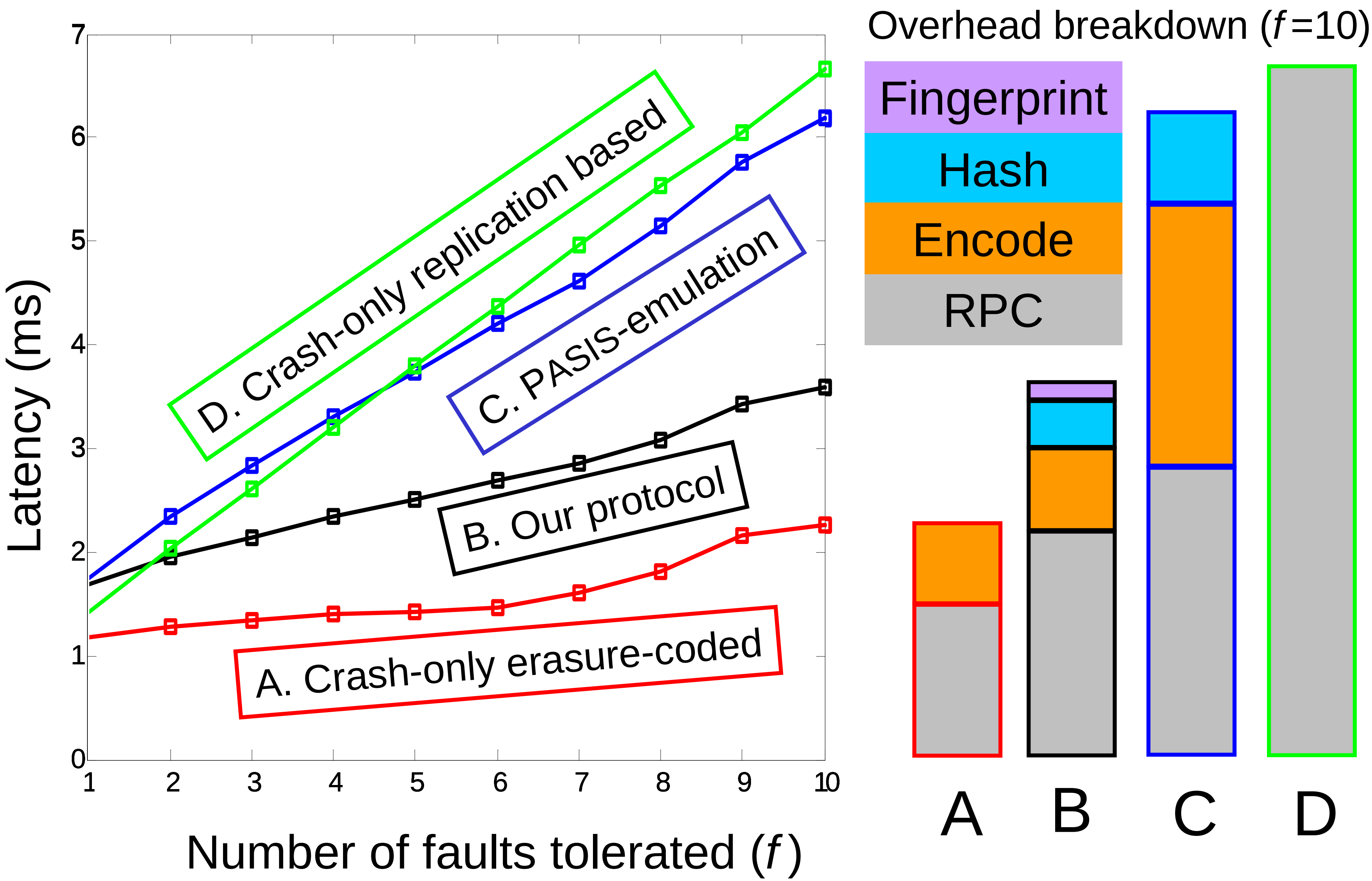
Write throughput



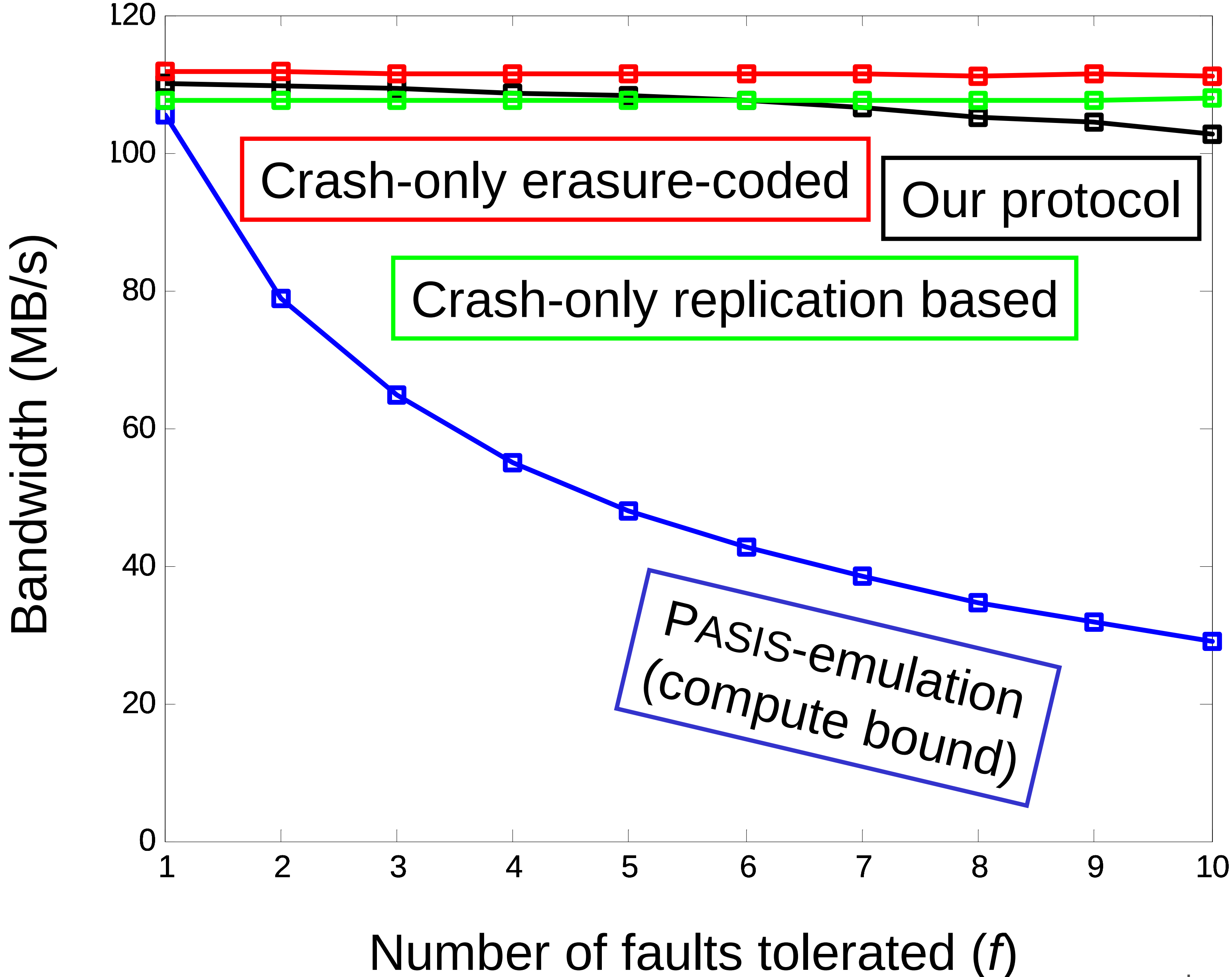
Write throughput



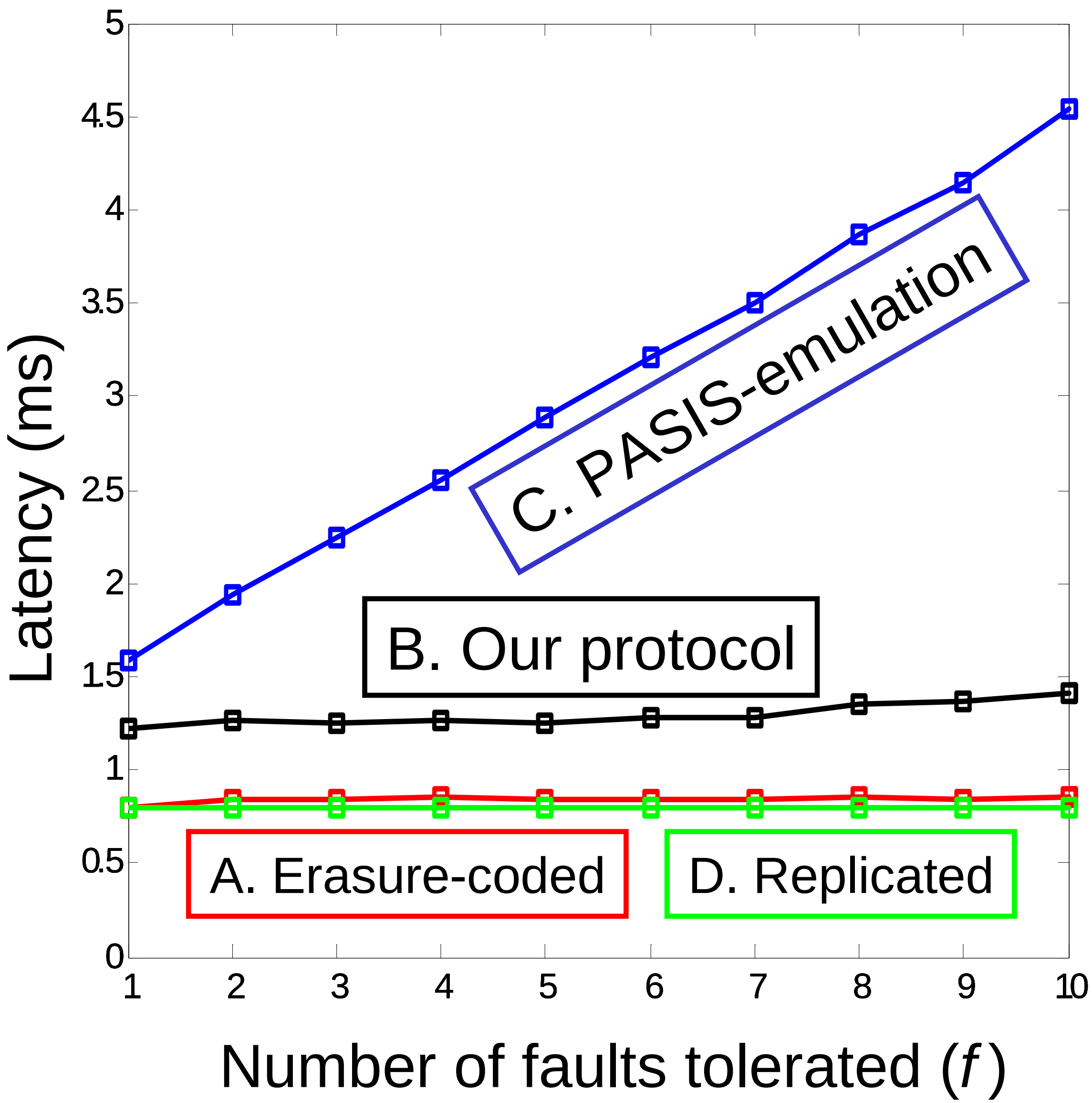
Write response time

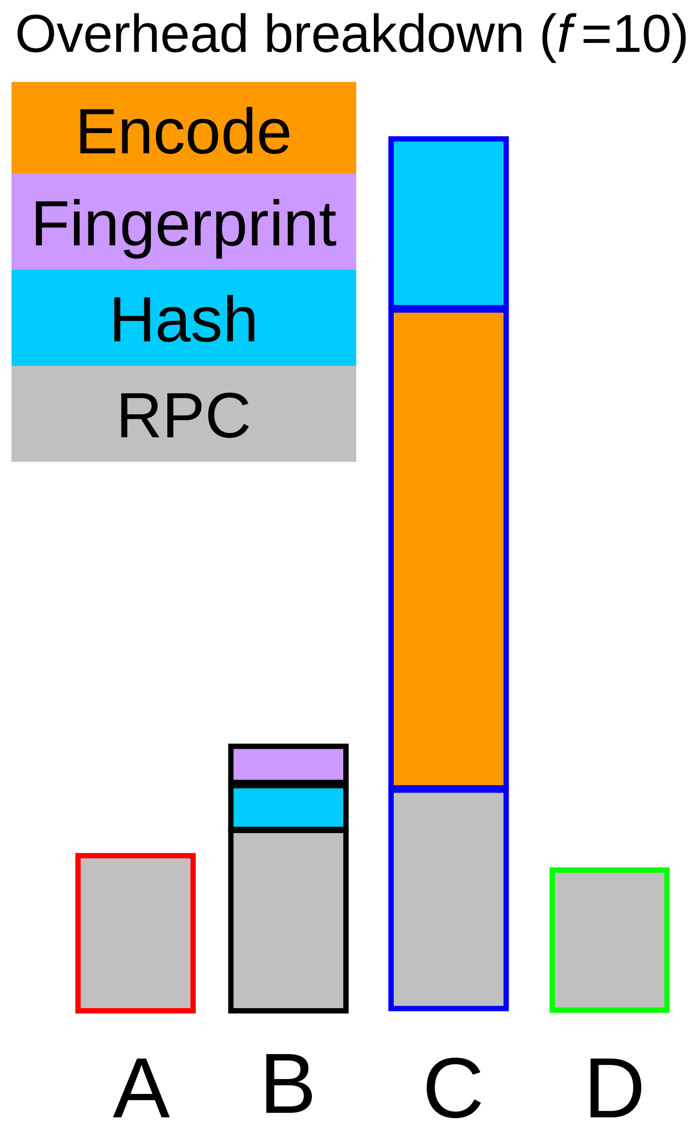


Read throughput



Read response time





Conclusions

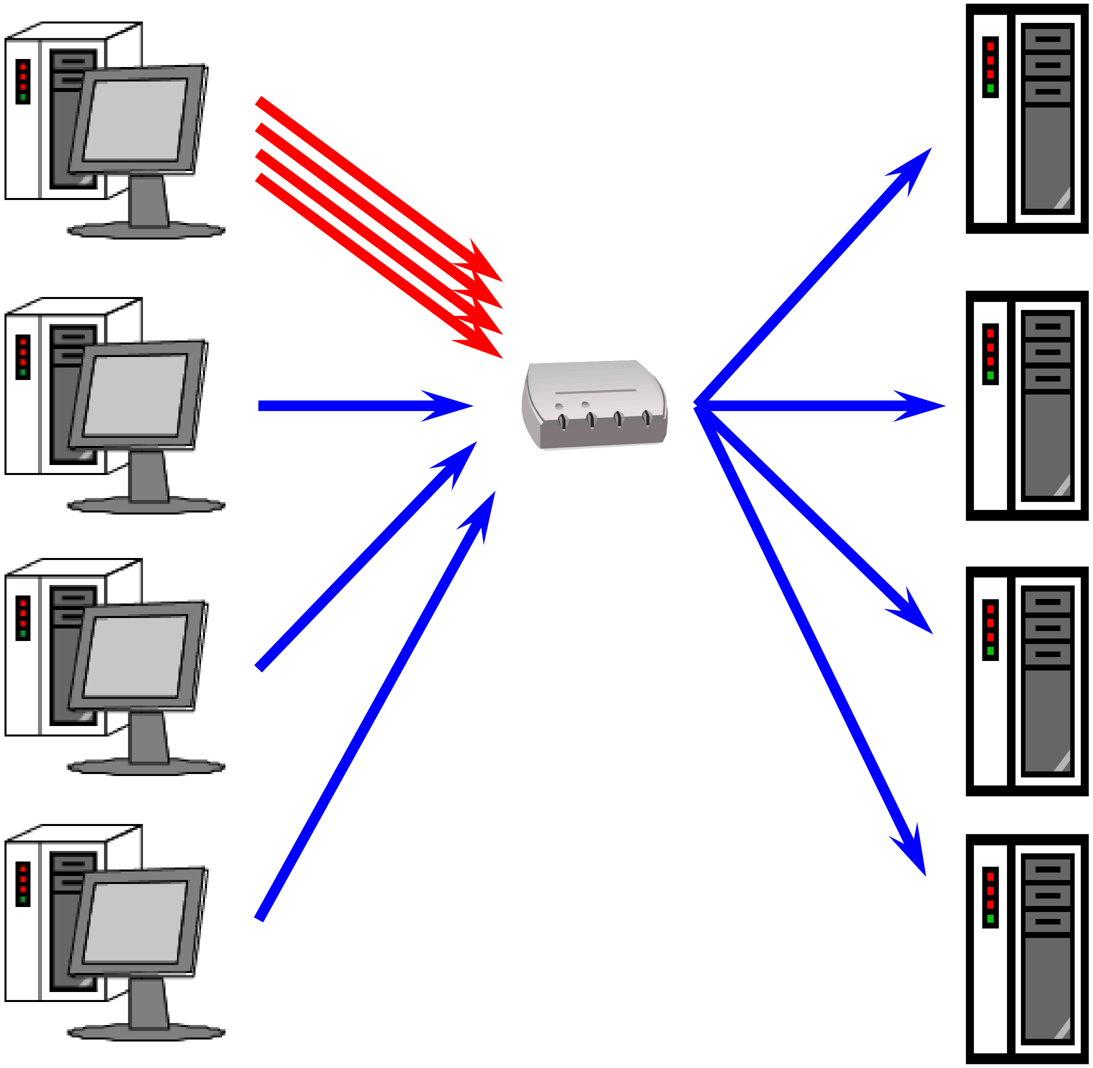
Byzantine fault-tolerant storage can rival crash-only storage performance

We present a low overhead Byzantine fault-tolerant erasure-coded block storage protocol and prototype

- Write overhead: 2-round, hash and fingerprint
- Read overhead: hash and fingerprint
- Close to performance of systems that tolerate only crashes for reads and large writes

Backup slides

Why not multicast?



May be unstable or unavailable (UDP)

More data means more work at server (network, disk, hash)

Doesn't scale with clients!

Cryptographic hash overhead

Byzantine storage requires cryptographic hashing.
Does this matter?

Systems must tolerate non-crash faults

- E.g., “misdirected write”

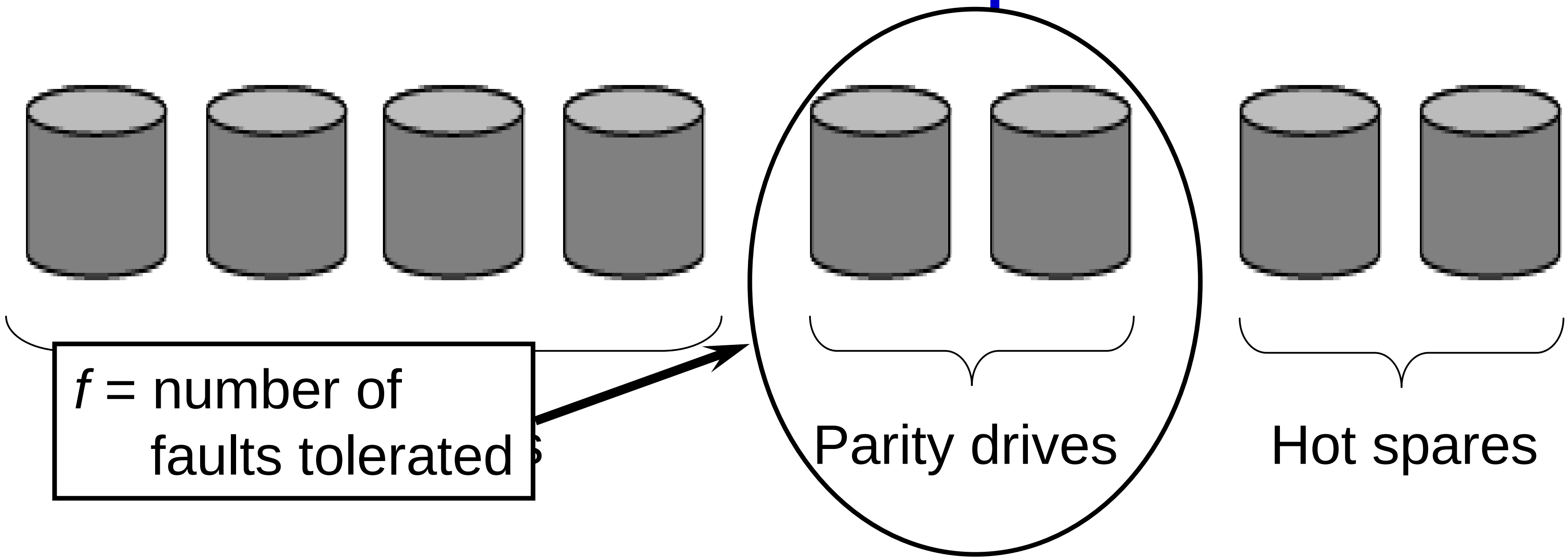
Many modern systems checksum data

- E.g., Google File System
- ZFS supports SHA-256 cryptographic hash function

May hash data for authentication

Conclusion: BFT may not introduce new hashing

Is 3f+1 servers expensive?



Consider a typical storage cluster

- Usually more *primary* drives than *parity* drives
- Usually several *hot spares*

Conclusion: May already use 3*f* +1 servers