Striping without Sacrifices: Maintaining POSIX Semantics in a Parallel File System

Jan Stender
Björn Kolbeck

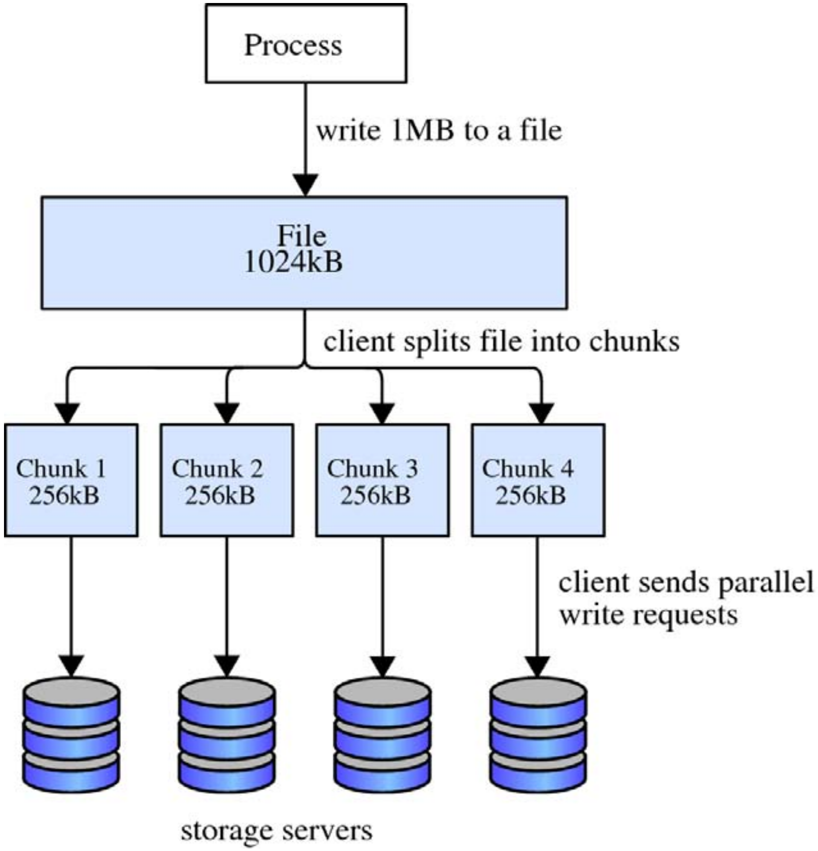
Zuse Institute Berlin

Outline

- Introduction
- Problem Description
- Striping Protocol
- Experimental Results
- Summary

Introduction

- Striping increases the performance of file systems
  - a single file is split up in chunks scattered across multiple storage resources
  - chunks can be accessed in parallel
  - a single file can be accessed with the accumulated performance of multiple storage resources
- Parallel file systems have distributed storage resources
  - chunks reside on different storage servers

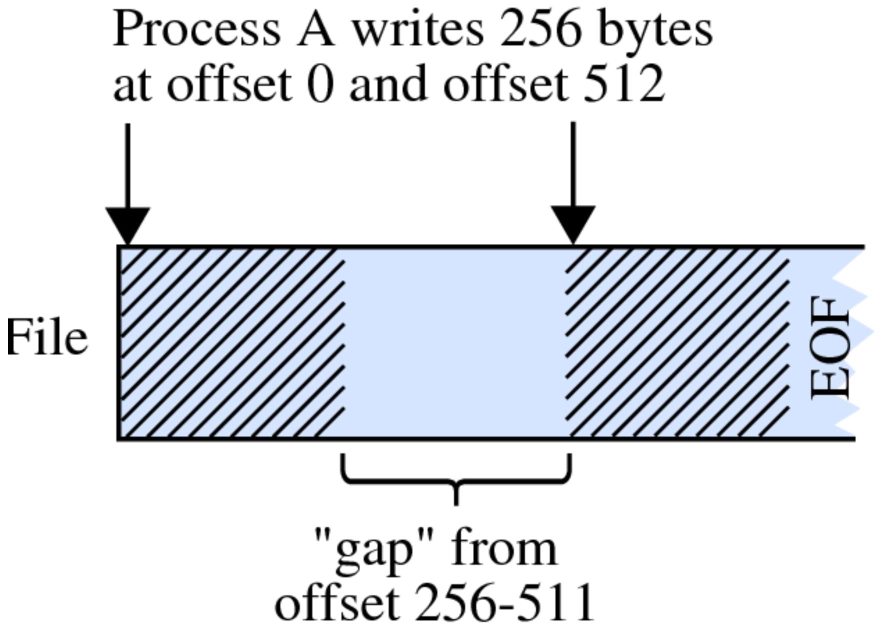


Introduction

- General-purpose file systems are expected to be POSIX-compliant
  - well-defined interfaces and behavior
  - no specific API, applications run w/o being modified or re-linked
  - POSIX-compliant file systems can be used by any application
- POSIX defines how read and write operations behave in certain corner cases:
  - ''gaps''
  - reading beyond EOF

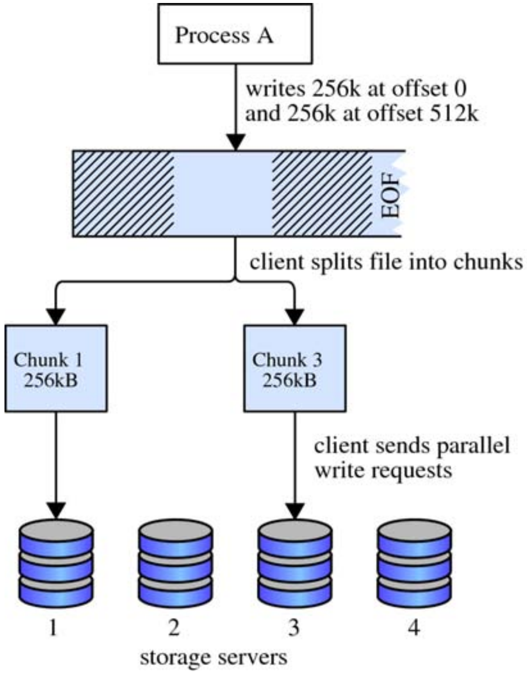
Introduction

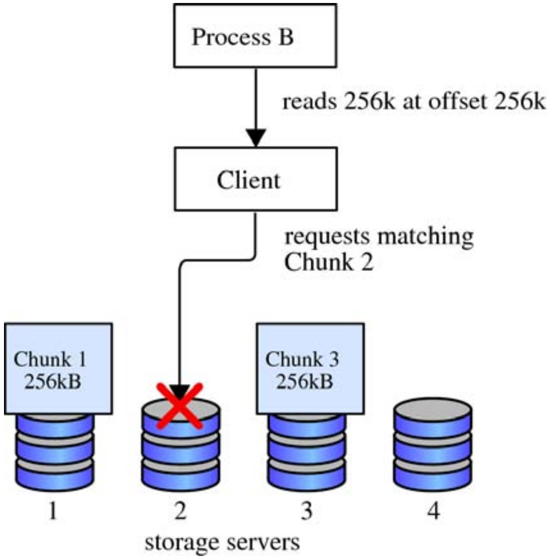
- Gaps
  - writes at an offset beyond EOF implicitly creates a gap, i.e. a region of missing data
  - reading bytes in a gap must return binary zeros
- EOF
  - reading a range of bytes to an offset beyond EOF must prune the resulting buffer (less bytes than requested)



Problem Description

- Problem: How to distinguish between a gap and the EOF in a parallel file system?
  - process *A* creates new file by writing chunk *1* and *3*
  - chunk *2* is not explicitly filled with data





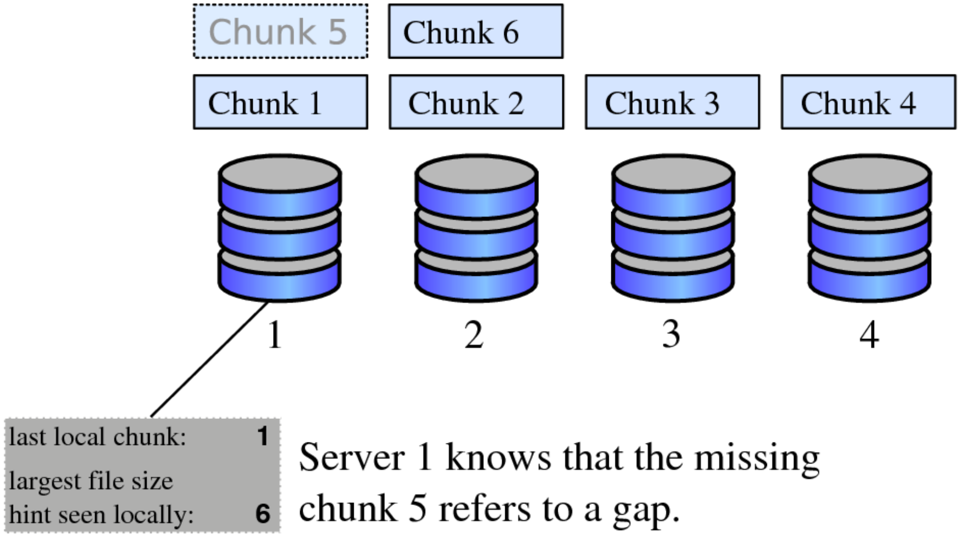
  - process *B* requests missing chunk *2*
  - storage server *2* must decide whether to respond with an empty buffer (EOF) or a zero-padded buffer (gap)

Problem Description

- Basic idea: provide for a consistent view on the file size among all storage servers
- However, ...
  - synchronizing each append-write operation across all storage servers is too expensive
  - a central server that stores the file size would be a bottleneck

Striping Protocol

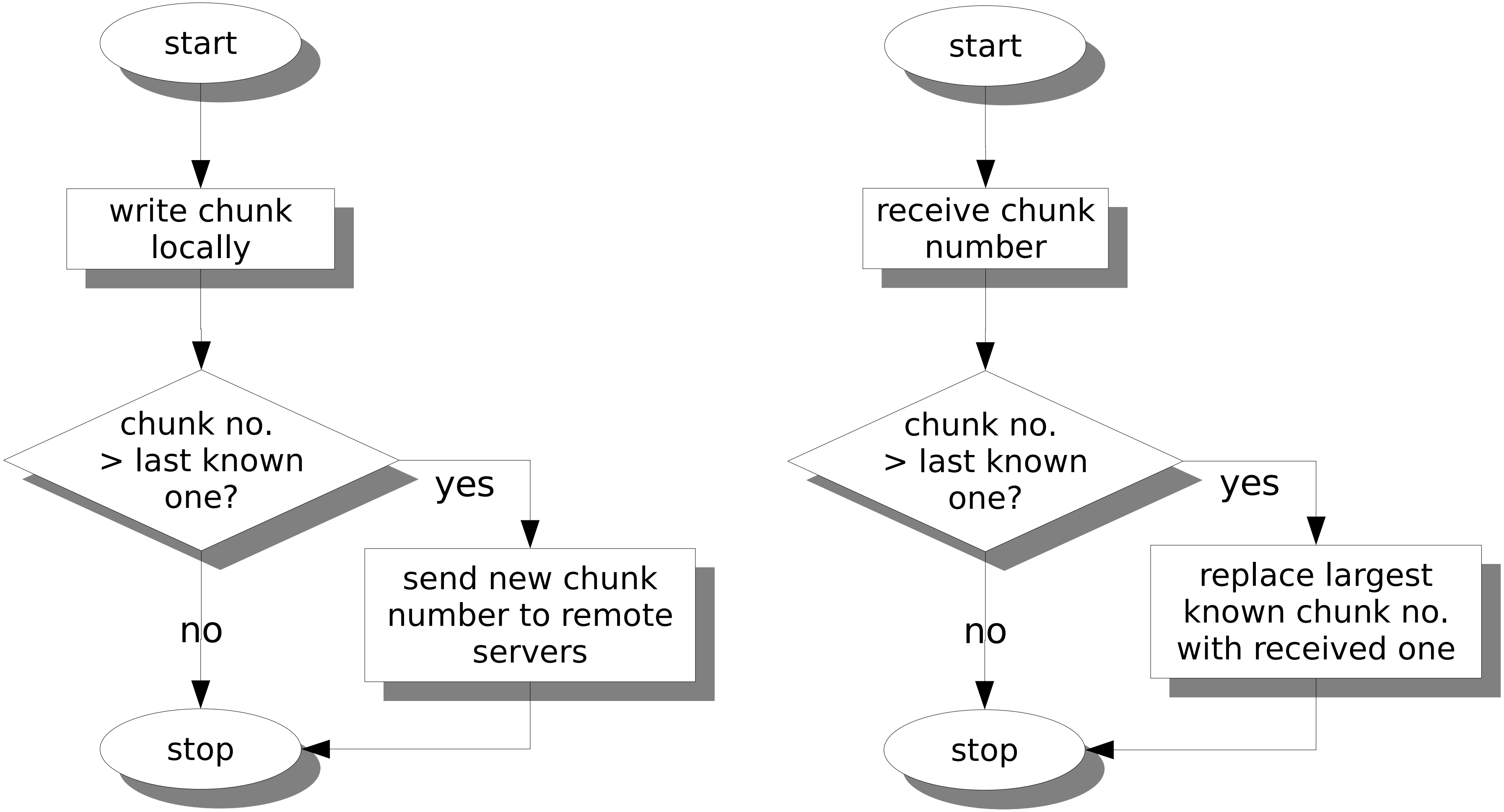
- Solution: decentralized, loosely-synchronized approach
  - storage servers disseminate and keep track of hints about the current file size (i.e. the globally last chunk number)
  - if a requested chunk is missing, these hints are used to decide between a gap and an EOF
  - if no decision is possible, the file size is explicitly synchronized by fetching the last chunk number from all storage servers
  - implicit assumption: files grow monotonously



Server 1 knows that the missing chunk 5 refers to a gap.

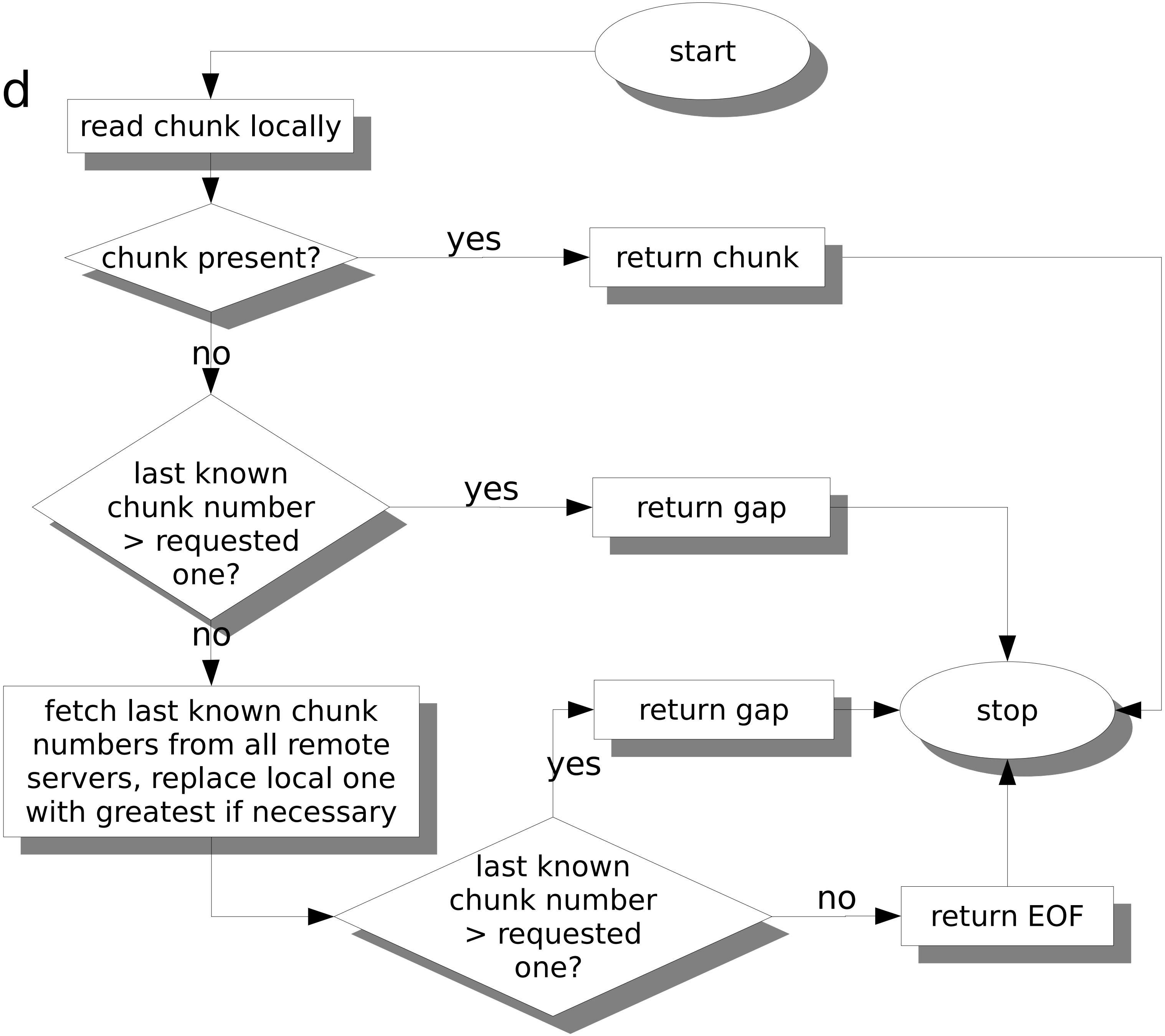
Striping Protocol

- Write


start
write chunk locally
chunk no. > last known one?
yes
send new chunk number to remote servers
no
stop
start
receive chunk number
chunk no. > last known one?
yes
replace largest known chunk no. with received one
no
stop

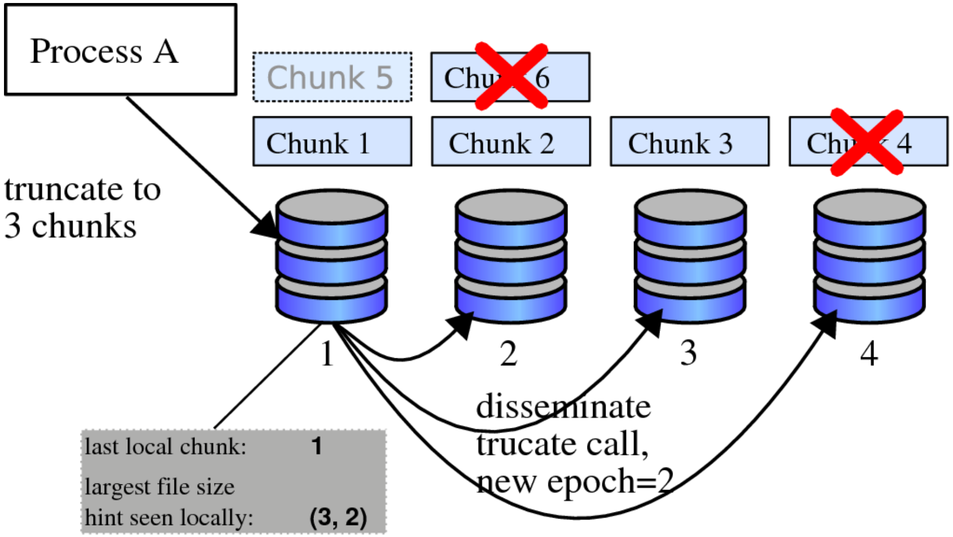
Striping Protocol

- Read


start
read chunk locally
chunk present?
yes
return chunk
no
last known chunk number > requested one?
yes
return gap
no
fetch last known chunk numbers from all remote servers, replace local one with greatest if necessary
last known chunk number > requested one?
yes
return gap
no
return EOF
stop

Striping Protocol

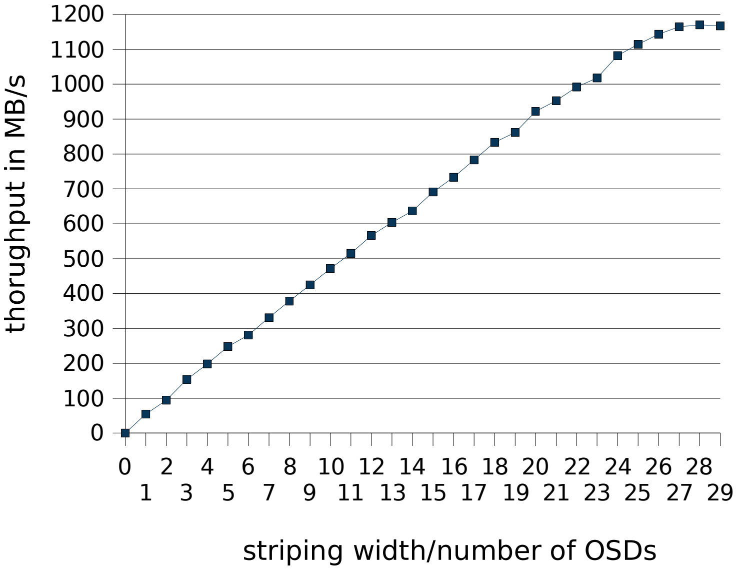
- Truncate
  - problem: violates our monotony assumption on the file size
  - solution: "truncate epochs"
  - file size hints consist of chunk number + epoch number
  - a designated server is responsible for truncate operations
    - it increments the epoch number
    - it synchronously updates the file size + epoch on all remote servers
  - a server receiving a file size hint updates its local chunk and epoch number if
    - the received epoch number is greater than the local one
    - both epoch numbers are equal and the received chunk number is greater than the local chunk number

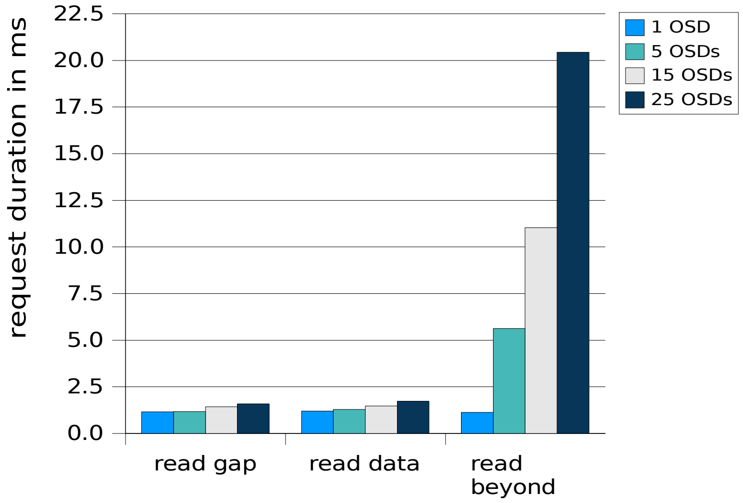


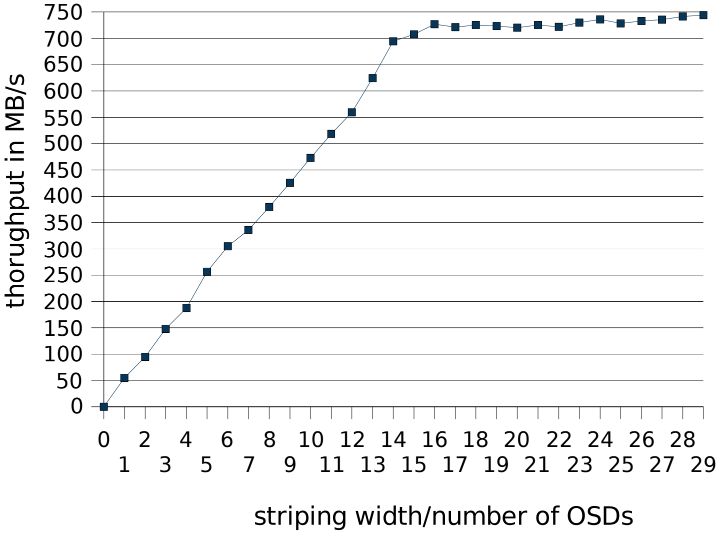
last local chunk:	**1**
largest file size hint seen locally:	**(3, 2)**

Experimental Results

- reads and append writes scale linearly
- low latency for reading gaps and data, as no file size synchronization is necessary
- higher latency for reading beyond the EOF, due to file size synchronization







Summary

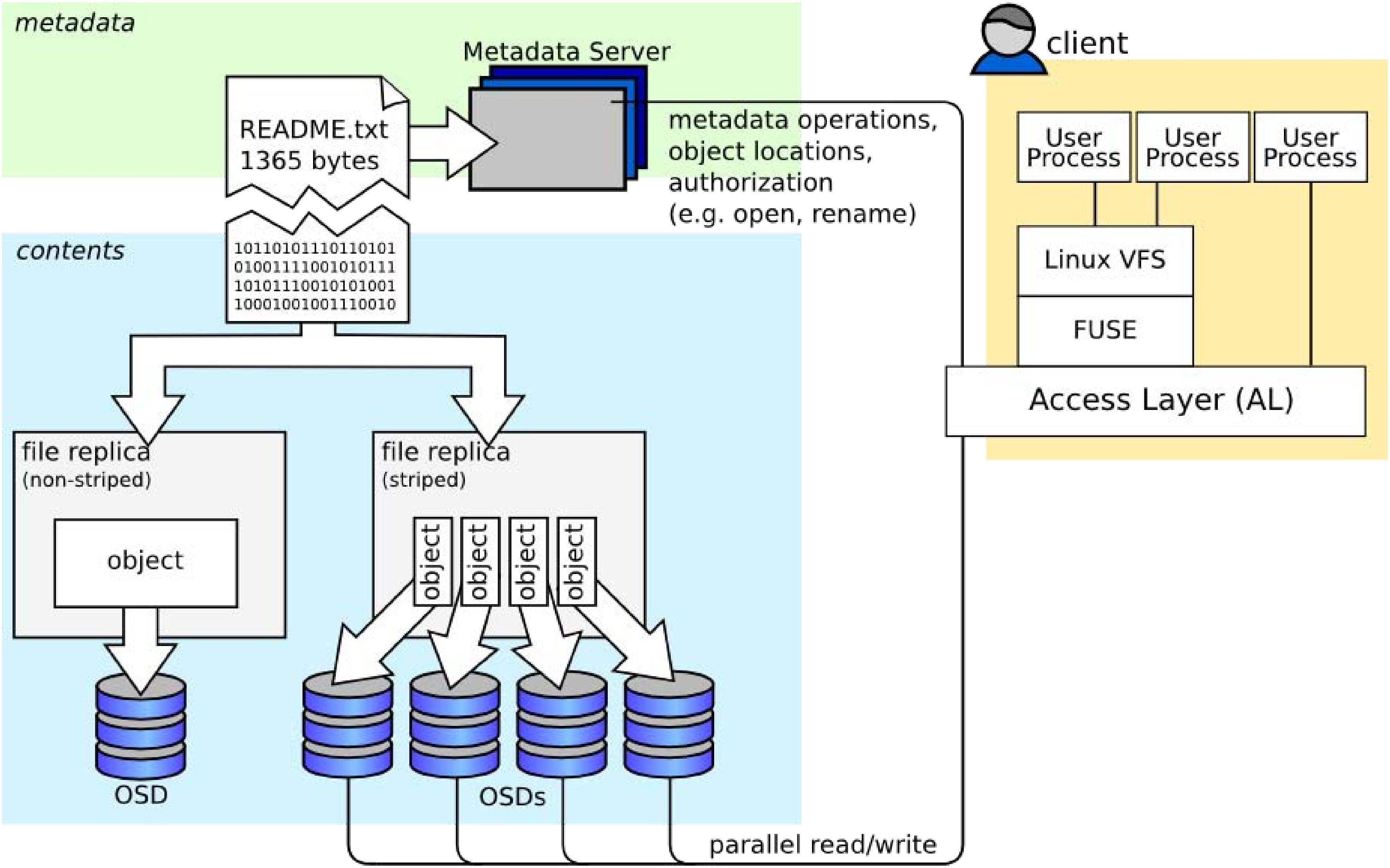
- The suggested protocol exhibits a POSIX-compliant behavior while ensuring scalability
- Frequent operations are fast
  - append and random writes
  - reads in file bounds
- The protocol does not enforce locking
  - parallel access is possible by multiple clients
- The protocol inherently supports sparse files

Thank you!

Questions?

Funded by the European Commision's FP6 programme
under contract #FP6-033576

XtreemFS - Architecture


metadata
Metadata Server
README.txt
1365 bytes
metadata operations,
object locations,
authorization
(e.g. open, rename)
client
User Process
User Process
User Process
Linux VFS
FUSE
Access Layer (AL)
contents
10110101110110101
01001111001010111
10101110010101001
10001001001110010
file replica
(non-striped)
object
file replica
(striped)
object
object
object
object
OSD
OSDs
parallel read/write