Andy Hanushevsky Michal Simon / Wei Yang

XRootD object store

Introduction

- XrdEc a high performance scalable EC-based file store motivated by HL-LHC requirements with ALICE as the first tangible well-defined use case.

- Originally developed for EOS and recently extended to work with any type of backend storage

Highlights

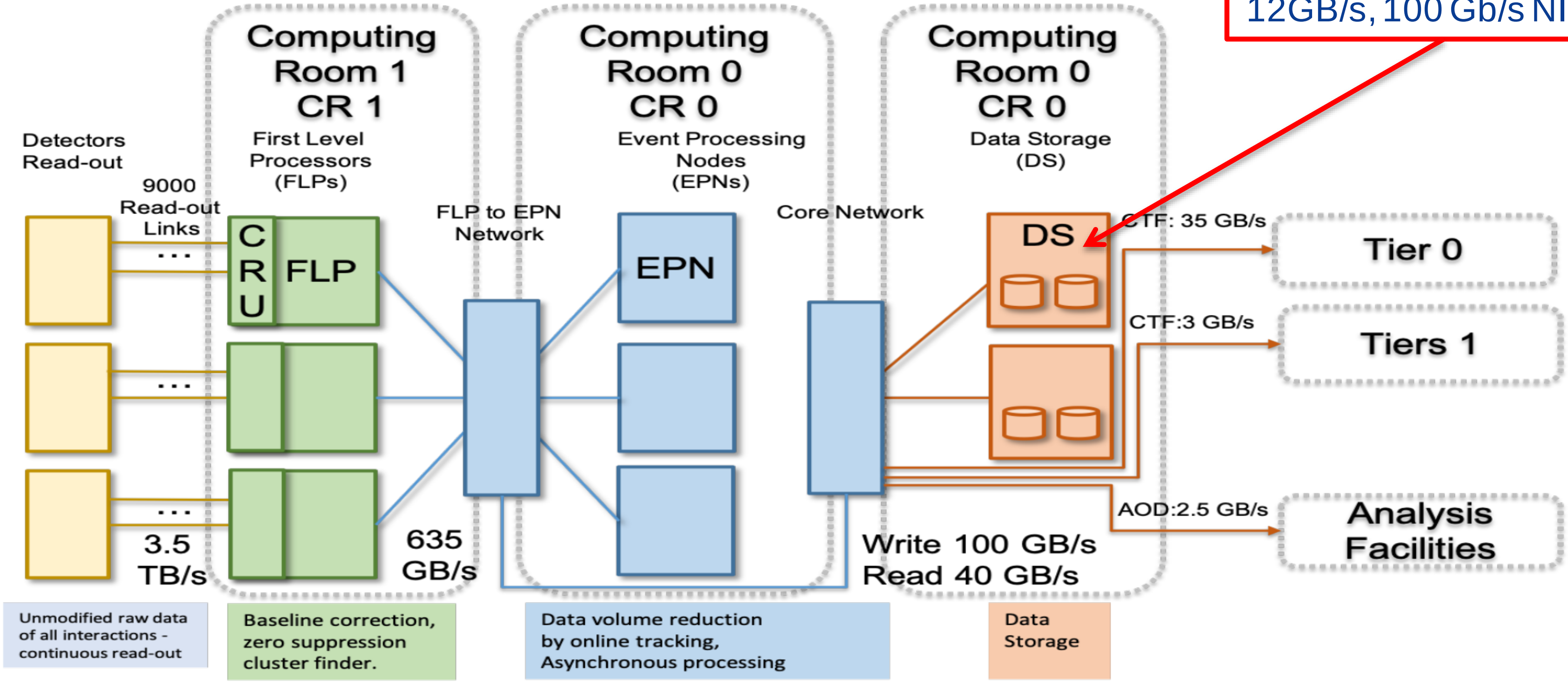
- We use state of the art **Intel ISAL Reed-Solomon** implementation
- **Placement group** for the data chunks can be obtained from EOS namespace or vanilla XRootD redirector
- ZIP is used for bundling data chunks together into stripes
  - Each **chunk is a separate file within a ZIP archive**
  - The file header contains information like the **crc32, size**, etc.
- Implementation details

Use Case: Alice O2

- 500 EPNs (Event Processing Node), each hosting 4 GPUs, each GPU generating a Time Frame every 40 seconds
  - **2000 data sources** in total
  - Aggregate throughput of **100GB/s**
- A Time Frame (TF) corresponds to a single 2GB file in EOS
  - **TF has to be copied to EOS in less than 40 seconds**
- Data sources transfer data directly to EOS (CERN CC) in (kind of) round robin fashion at 20 ms intervals
  - **every 20 ms a new file will be created and 2GB of data transferred**

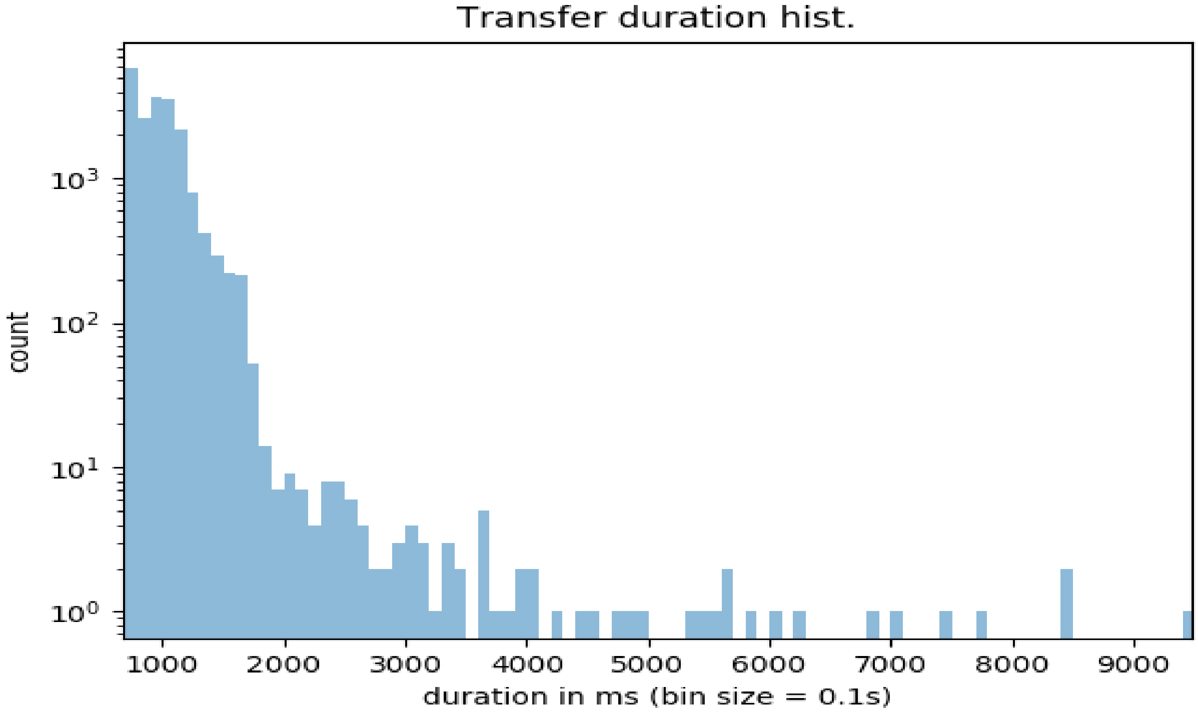
Alice O2

6 data servers: 96 HDDs / 280MB/s, 4 HBA: 12GB/s, 100 Gb/s NIC



Use Case: Alice O2

~10% of the target production load, ~10% of the cluster capacity

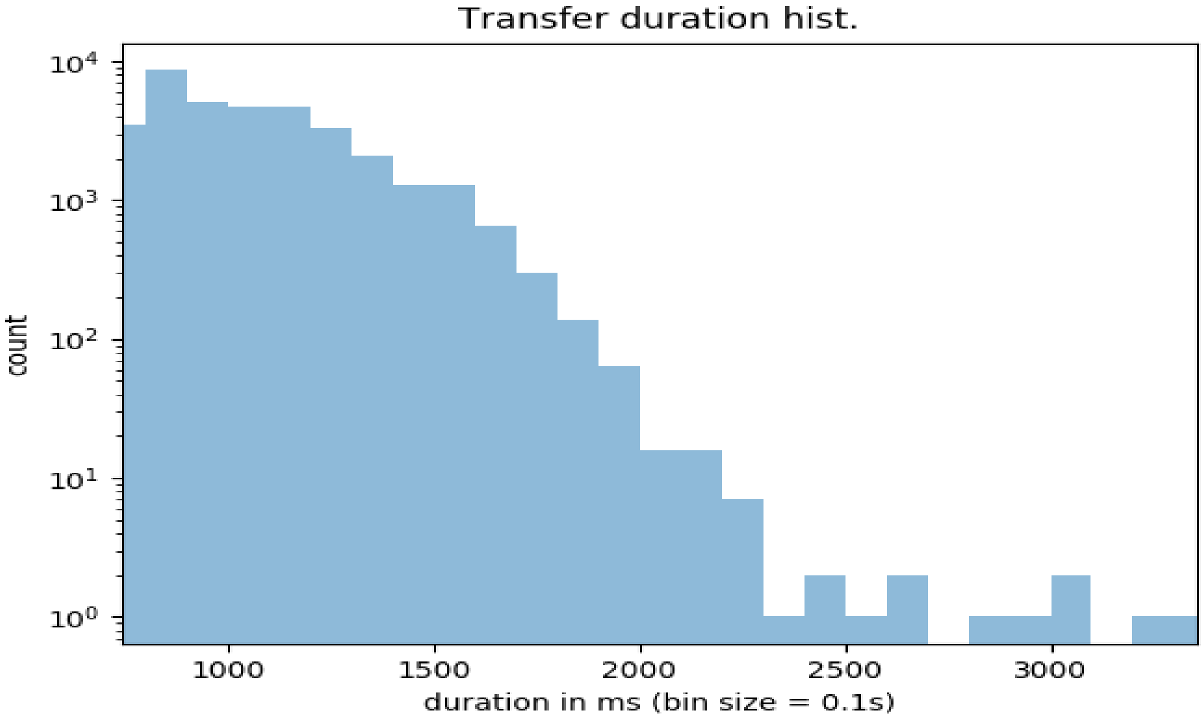


10+2 layout,
10GB/s of aggregate throughput (200 streams),
1 hour run, 6 data servers

Avg duration: 974 msec
Avg transfer rate: 2.15GB/s
Transfer rate stdev: 0.418
Transfer duration stdev: 290

Use Case: Alice O2

~20% of the target production load, ~10% of the cluster capacity

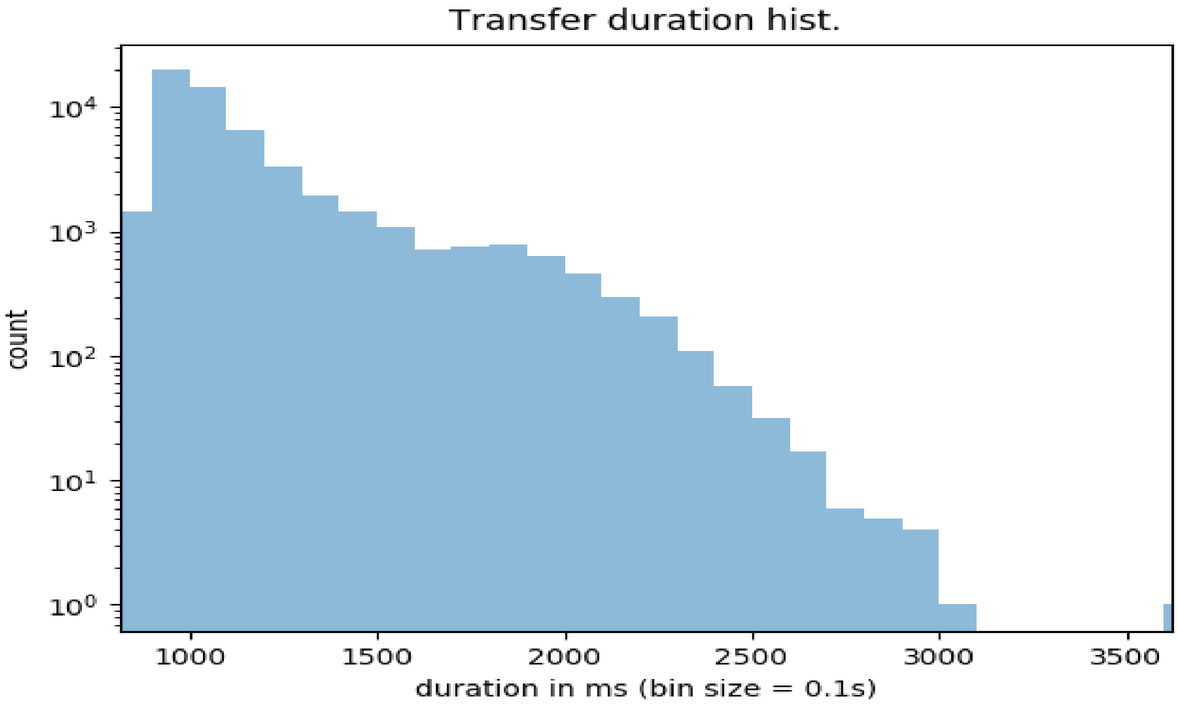


10+2 layout,
20GB/s of aggregate throughput (400 streams),
1 hour run, 6 data servers

Avg duration: 1063 msec
Avg transfer rate: 1.97GB/s
Transfer rate stdev: 0.400
Transfer duration stdev: 244

Use Case: Alice O2

~30% of the target production load, ~10% of the cluster capacity



10+2 layout,
30GB/s of aggregate throughput (600 streams),
1 hour run, 6 data servers

Avg duration: 1127msec
Avg transfer rate: 1.84GB/s
Transfer rate stdev: 0.317
Transfer duration stdev: 272

Paths to integrate XrdCl+EC with the xrootd storage

1. Mode 1. Use xrootd storage directly as an EC store
    - Xroot protocol and xrootd client (with EC support) only

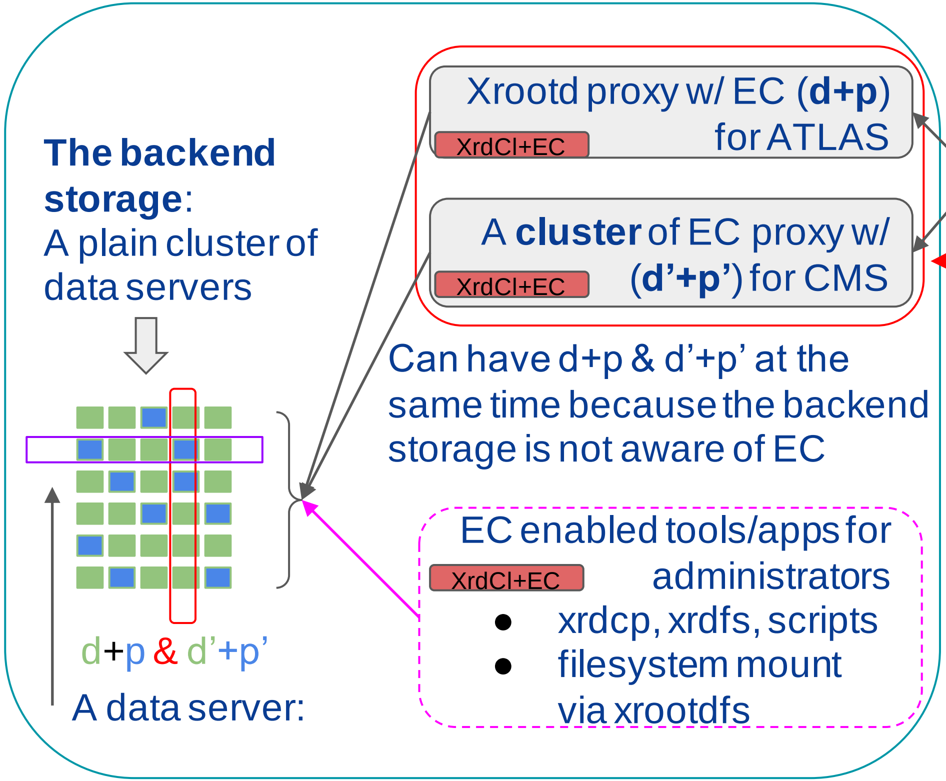
This mode is good for local administration

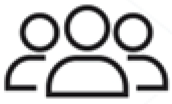
2. Mode 2. Use XRootD Proxy as gateway to backend storage
    - Enable EC in the proxy's xrootd client component.
    - EC is invisible to the users
        - They use existing xrdcp/xrdfs, gfal, curl
    - Support all WLCG security, protocols, TPC, etc.
    - The backend xrootd storage is plain and simple

This mode is better for user access

- The rest of the slides are about this mode

The Object Store: Xrootd with Erasure Coding (XEC)





Add a xrootd proxy (DTN) **cluster**

- A xrootd proxy is both
  - a **User Facing** Xrootd server
  - a xrootd client (talking to "remote" xrootd storage)
    - **XrdCl EC happens at here.**
- Support Object Store functions
  - GET/PUT/DEL/LIST/RENAME
  - HTTP protocol and xroot protocol
- Support all DTN functions used by WLCG
  - VOMS and token authentication
  - TPC (HTTP and xrootd)
  - Checksum query
- Scale out by expanding the cluster

Interface to users

Nothing changed: users will still work with root(s) or http(s) URL:

- https://atlas.cern.ch:1094/atlas/rucio/user/jdoe/my.data or
- root://atlas.cern.ch:1094**//**atlas/rucio/user/jdoe/my.data
- Think of “atlas/rucio/user/jdoe” as bucket, folder, whatever you like.
    - Your access permission may be based on top level buckets/folders.

Three sets of tools for GET/PUT/DEL/LIST/RENAME

- **xrdcp/xrdfs**: work mostly with root(s) URLs
- **gfal2**: works with both root(s) URL and http(s) URLs
- **curl**: works with http(s) URLs

Performance test environment

One goal is to reach the hardware limit

Backend: Xrootd storage:

- 19 nodes of retired Dell R510s, each:
  - 24GB RAM, 1Gpbs NIC, 12x 3TB HDD (some have 11)
  - Each HDD is presented to the OS as its own SCSI device (via LSI RAID controller)
  - CentOS 7, Xrootd 5.3.4 (later auto-updated to 5.4.0), xrootd “sss” security
- 312 pre-placed test files (ATLAS data files) ranging from 30MB to 1.1GB, all with known adler32 checksum

Frontend: Xrootd EC proxy

- 64 core, 128GB, 100Gbps NIC
- CentOS 7, unreleased Xrootd (2021-12-17+patch ← this is newer than 5.4.0)
- EC configuration: **8+2**, chunk size 1MB (So a block has 8+2 MB)

Single stream performance with xrdcp

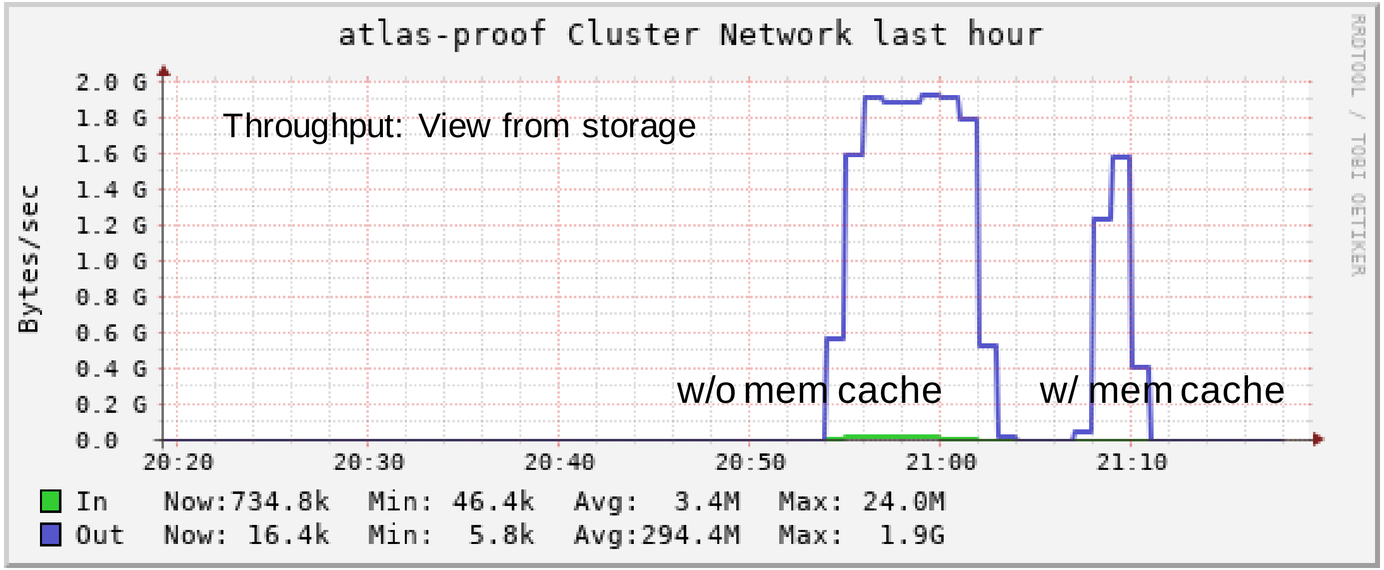
Getting the baseline performance numbers using Mode 1

- Run a EC enabled xrdcp, write and read a single ~8GB file.
- Memory to memory (between RAM disk in client node and page cache in Dell 510s)
- Write: 904MB/s (Actual writing speed: 904 * (8+2)/8 = 1130MB/s ← near the line speed)
- Read: 1017MB/s
  - EC doesn't need to read the parity chunks (unless there is an error)
  - **This is a good indication that EC code isn't the bottleneck in this environment.**

Single stream performance by a client, read from and write to storage via the EC proxy (Mode 2)

- Write: 904MB/s ← near the line speed limit (1250MB/s)
  - **This is a good indication that EC code and EC proxy setup do not present a bottleneck for writing**
- Read:
  - ~155MB/s ← because Xrootd proxy internally break down read request to 2MB chunks
    - It is tunable, to be tested.
  - Add a memory cache in proxy (8MB page size ← to align with EC block size, 1 prefetching): ~505MB/s
    - Memory cache is a feature in Xrootd proxy. Can be turn on if there are sufficient memory

Aggregate read performance by many clients

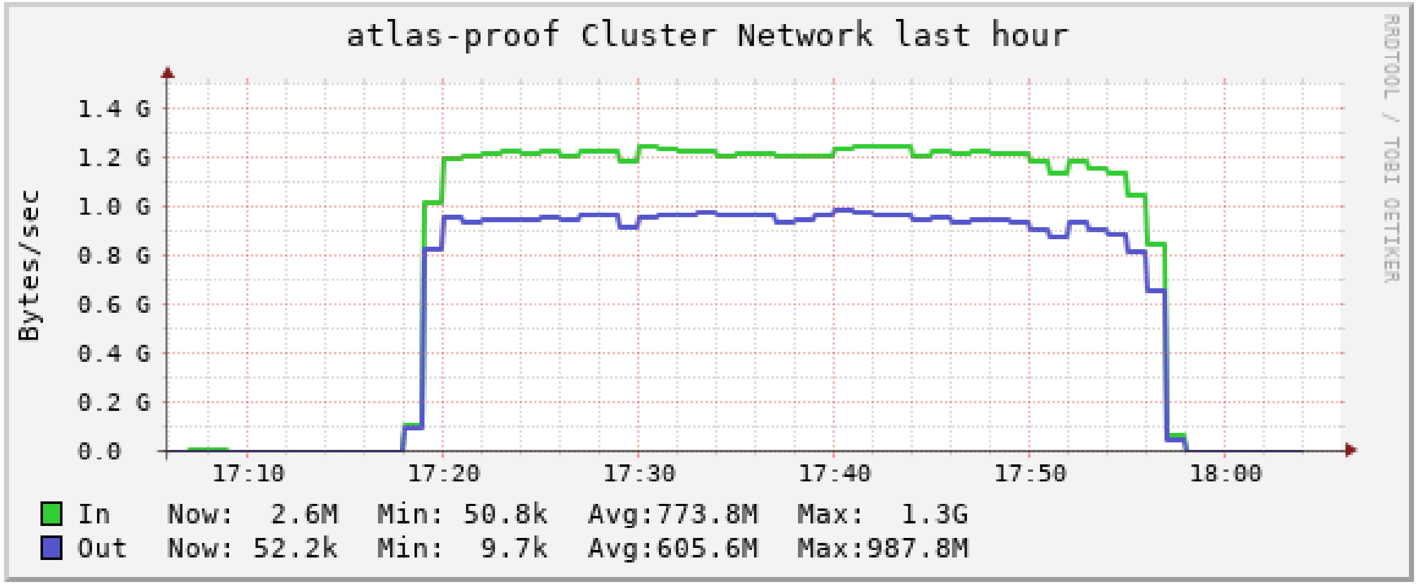


Network upper limit

- 19 Gbit/s or
- 2.375GB/s

- Read the pre-placed 312 data files, repeat 5 times
- Spread the read to 150 concurrent clients
- Memory cache clearly helped, it both
  - cache (reduce read from storage)
  - enable large block read (align with EC blocks)

Aggregated Read/Write performance



Backend storage view

- In: write
- Out: read

Memory cache: off

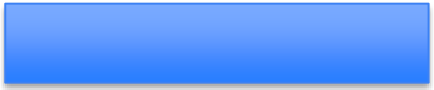
- By 200 concurrent clients
- Randomly pick 20 files from the 312 sample files
- Read and write back at the same time
  - Note: FS prioritizes write over read

Write/Read Implementation

Backup Slides Follow

Writing

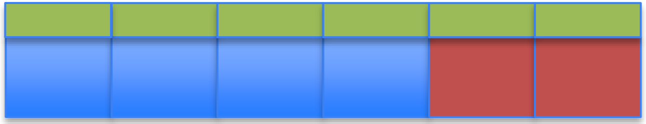
- Client buffers the data until it has a full block



- The block is divided into chunks

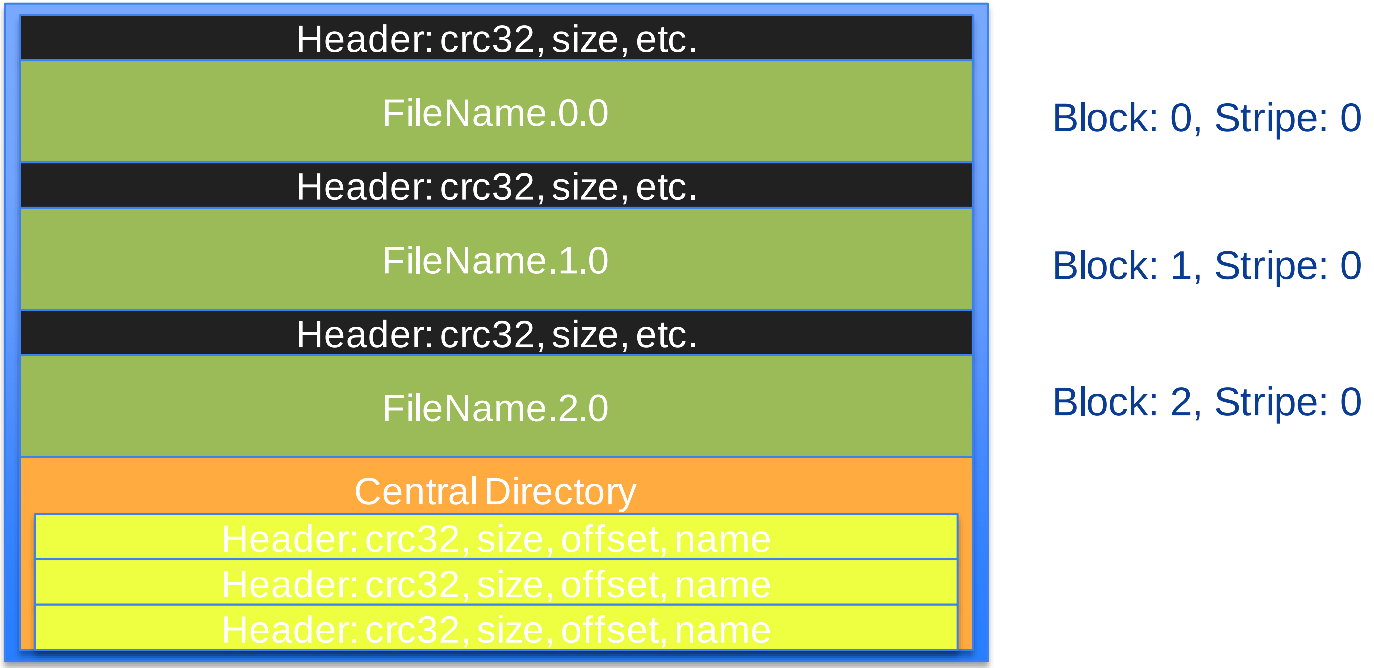
- The chunks are erasure coded

- All chunks (data/parity) are checksumed



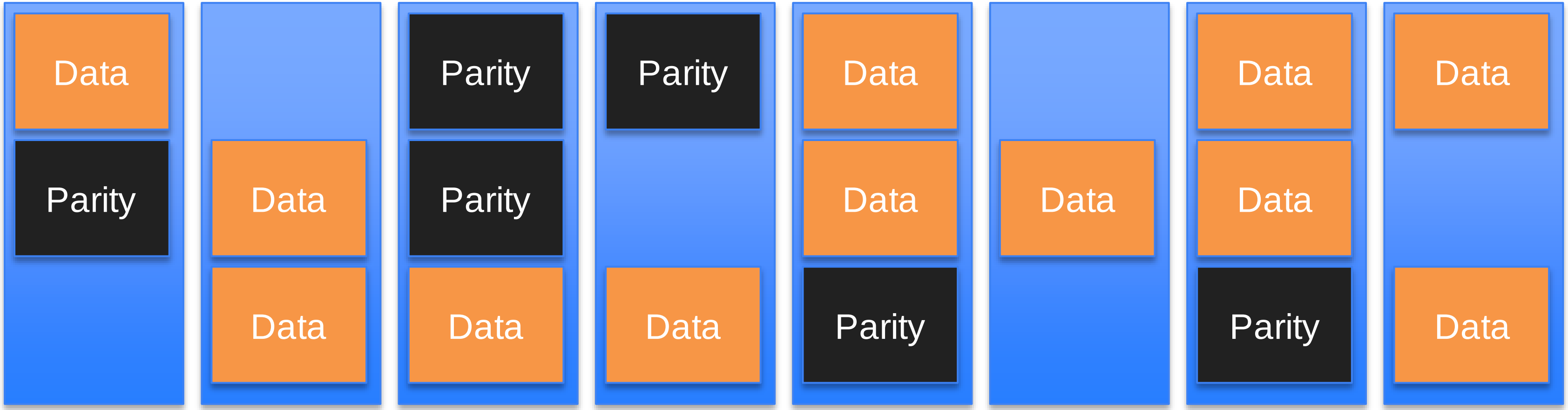
Writing

- Each stripe is stored in a ZIP archive, each chunk is a separate file within the archive



Writing

- If the placement group has more locations than the number of data and parity stripes (> n + m) we choose locations randomly for each block (uniform distribution)
- 4+2 with 8 locations:



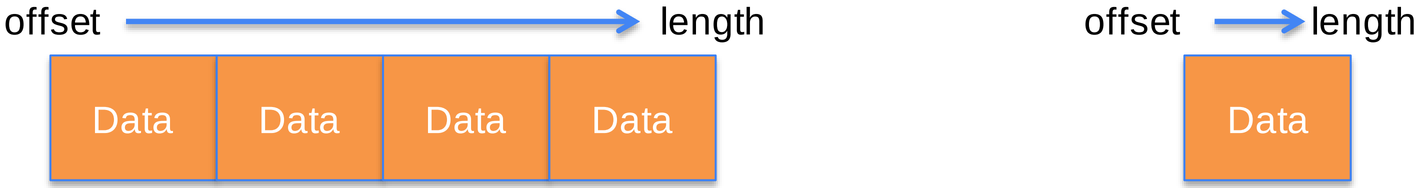
- **Allows to recover errors on write at spare locations**

Reading

- Either the server **on open request tells the client to load the EC plugin**, or access through **proxy server**, again:
  - Static configuration: **number of data and parity chunks, block size**, etc.
  - **Placement group** needs to be discovered dynamically (EOS namespace or through standard **locate** request)
- On ZIP open client **reads/parses the CD** of each stripe
  - Afterwards each chunk locations is known

Reading

- There is **no need to reconstruct a block** for every read
  - Unless the client needs to do error correction
  - While streaming the data user can benefit from full performance boost due to striping
- In order to verify the checksum the client at minimum needs to read a whole chunk
  - **Reads are translated into respective chunks**
  - **Chunks are cached** until user is accessing data within same block



Back to presentation