

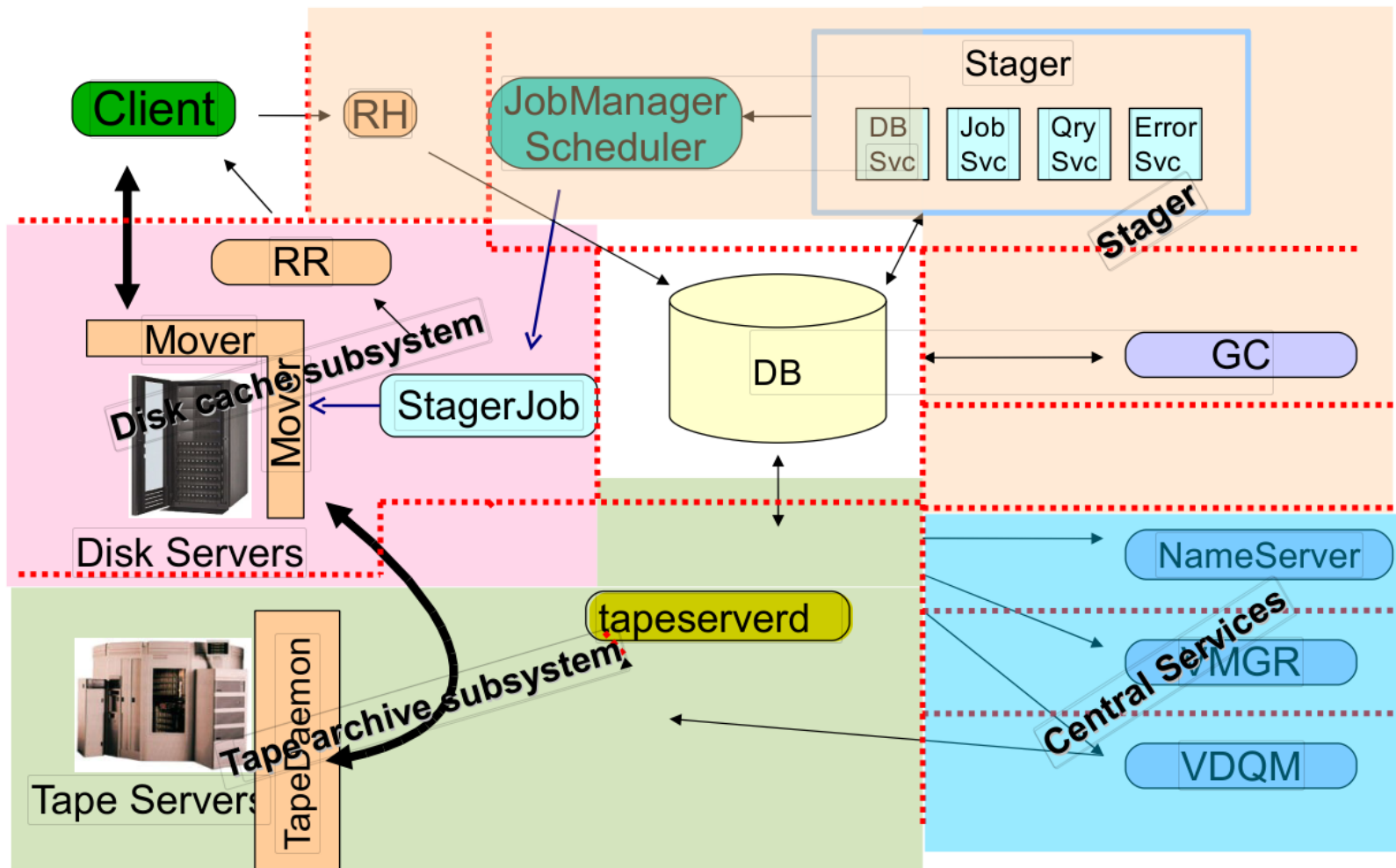
CTA: Object Store Status and Alternatives

Germán Cancio, Eric Cano, Michael Davis, Anastasia Karachaliou, Steven Murray

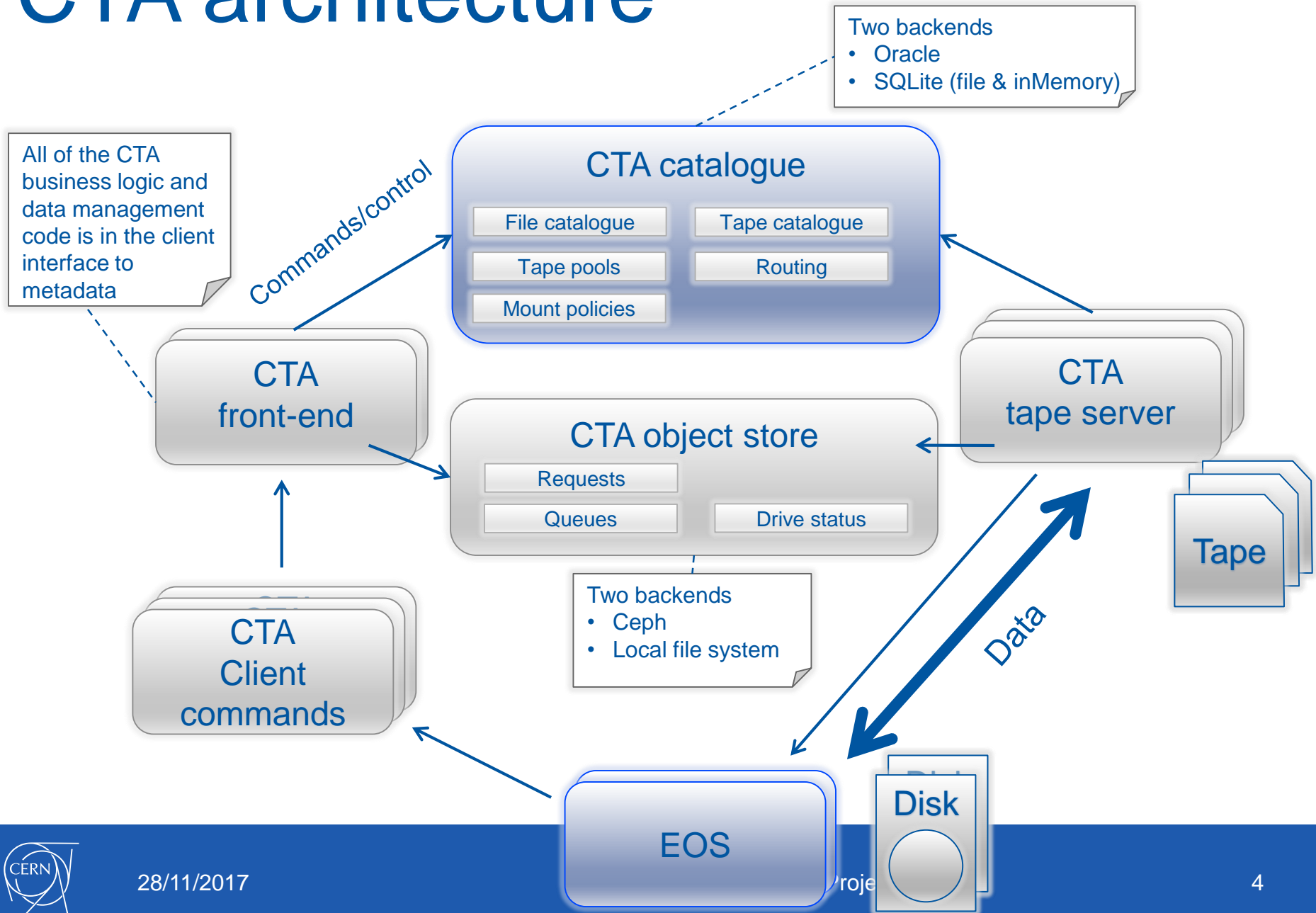
Overview

- What is CTA's object store?
 - Structure
 - Contention management
 - Reliability
 - Performance
- CTA queueing, differences with CASTOR
- Object store vs DBs

Castor architecture



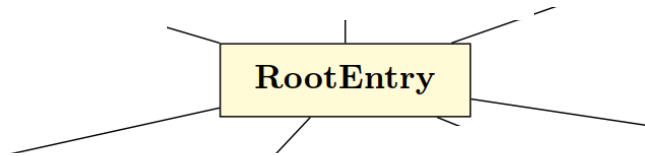
CTA architecture



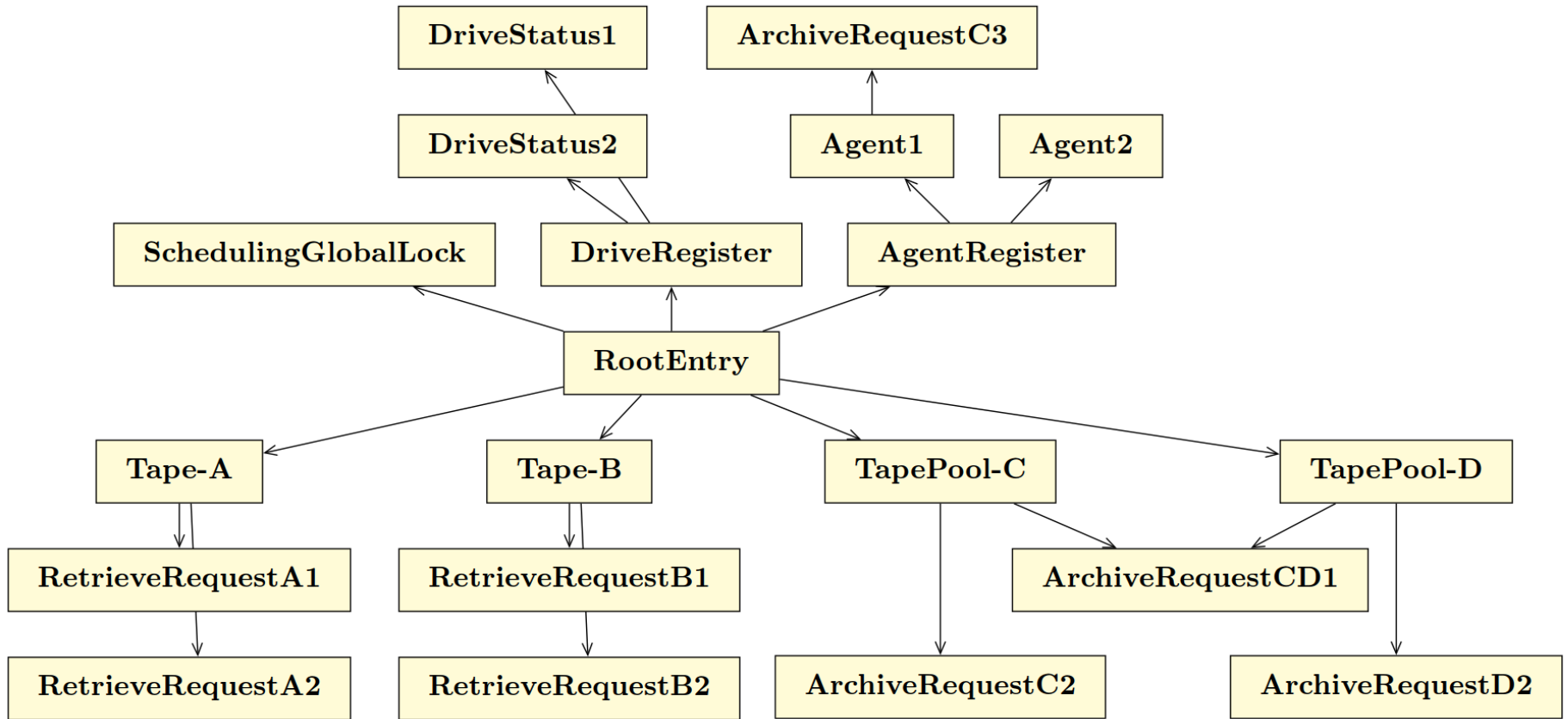
What is CTA's object store?

- A collection of data structures (objects) serialized using protocol buffers
 - Archive/retrieve queues and requests, agents their register (more on them later), drive statuses and their register, scheduling lock, and a root entry
- The objects are stored in a key/value store (backend)
 - Operations: create, read, overwrite, remove, lock, unlock + some async combinations
- Several backends available
 - Rados for distributed running
 - Linux VFS (posix filesystem) for unit tests
 - More can be added
- We rely on the backend for
 - Locking
 - Atomic updates (full object rewrites all or nothing)
 - Asynchronous IO
 - Object level reliability
- Object store is collaboratively updated by all agents of the system
 - No single point of failure
 - Direct access to data: no daemons and internal protocols to maintain
- Only used for transient data (queues and requests)
- Tape file catalog still in a DB

The tree structure (visually)



The tree structure (visually)



The tree structure

- Object keys are used as pointers
- Objects point to each other
- Starting from root entry (with fixed key name)
 - Pointing to queues and then requests, registries and then agents and drive statuses and the global lock
- All objects referenced only once with “backpointer” trick:
 - Several references to the object can coexist
 - Only one valid at any point in time
 - The object points back to active owner
 - Others are dangling pointers
 - This allows atomic move of objects within the tree
 - Pointer created before inserting object
 - Dangling pointers have to be handled

Contention management

- Tree separated in individual elements as needed
- Cut into sub structures to have finer grained contention points
- Most importantly, queues are separated objects with no cross-contention
- Writes are locked, and hence contended
- Reads can be lock free
 - At the cost of tolerating inconsistencies (object deleted in the meantime)
- Read and writes can be asynchronous (and hence parallel)
 - Example: bulk request updates
- Archetypical example: drive statuses
 - Each drive process locks and updates its status continuously
 - Statuses fetched lockfree for “cta drive ls” and scheduling
 - Drive registry locked only for insertion/removal of status reference (very rare)
 - Initial implementation (drive statuses inside the registry) was not scaling (contention on updates)

Example drive status querying

- “cta drive ls” sequence (fully lock free):
 - fetch root entry (get drive registry address)
 - fetch drive registry (get drive statuses list and addresses)
 - fetch all drive statuses in parallel (async).

```
[cano@tpsrv600 ~]$ time cta drive ls
library   drive      host      desired  request  status  since  vid  tapepool  files  MBytes  MB/s  session  age
IBM1JB    I1JB0402  tpsrv033  Down     -        Down    80148  -    -         -     -       -       -       4
[...]
IBM1JB    I1JB1005  tpsrv036  Down     -        Down    80142  -    -         -     -       -       -       1
tpsrv601  L601D1    tpsrv601  Up       Archive  Transfer 853    V01011 test_eric 1628   0.01    0.00    13778   11
tpsrv602  L602D1    tpsrv602  Up       Archive  Transfer 979    V02012 test_eric 1561   0.01    0.00    13774   2
tpsrv603  L603D1    tpsrv603  Up       Archive  Transfer 797    V03011 test_eric 1921   0.01    0.00    13779   0
tpsrv604  L604D1    tpsrv604  Up       Archive  Transfer 410    V04012 test_eric 711    0.00    0.00    13781   3
tpsrv605  L605D1    tpsrv605  Up       Archive  Transfer 862    V05012 test_eric 1681   0.01    0.00    13777   5
tpsrv606  L606D1    tpsrv606  Up       Archive  Transfer 155    V06012 test_eric 294    0.00    0.00    13782   5
tpsrv607  L607D1    tpsrv607  Up       Archive  Transfer 721    V07011 test_eric 1253   0.01    0.00    13780   15
tpsrv608  L608D1    tpsrv608  Up       Archive  Transfer 115    V08011 test_eric 172    0.00    0.00    13783   10
tpsrv609  L609D1    tpsrv609  Up       Archive  Transfer 918    V09012 test_eric 1817   0.01    0.00    13776   1
tpsrv610  L610D1    tpsrv610  Up       Archive  Transfer 1011   V10011 test_eric 1856   0.01    0.00    13772   3
T10KD5    T10D5214  tpsrv230  Down     -        Down    80137  -    -         -     -       -       -       3

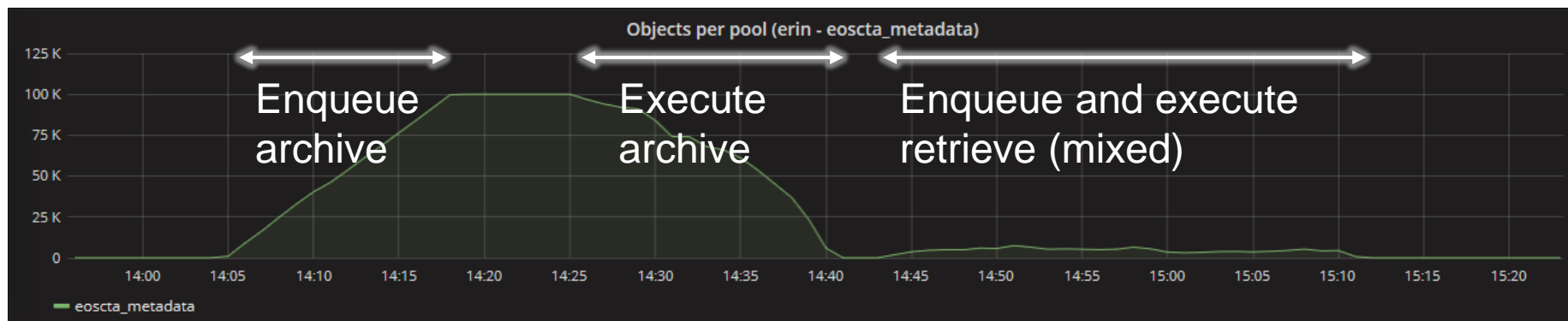
real    0m0.070s
user    0m0.007s
sys     0m0.005s
```

Resilience against crashes

- Machines do crash
- Tree structure coherent at any point in time with backpointers...
- ... but what about a request half queued, or selected by a drive after the corresponding process crashed?
- Each process is represented in the system (by agent objects) and references owned objects
- Agent object updated with heartbeat counter
- Garbage collector processes detect dead agents and put back owned object in the right queues
 - One GC per tape server
 - All GCs watch all agents (including GCs) lockfree
- Additionally, dangling pointers are discarded when encountered (in all cases, not only GC)

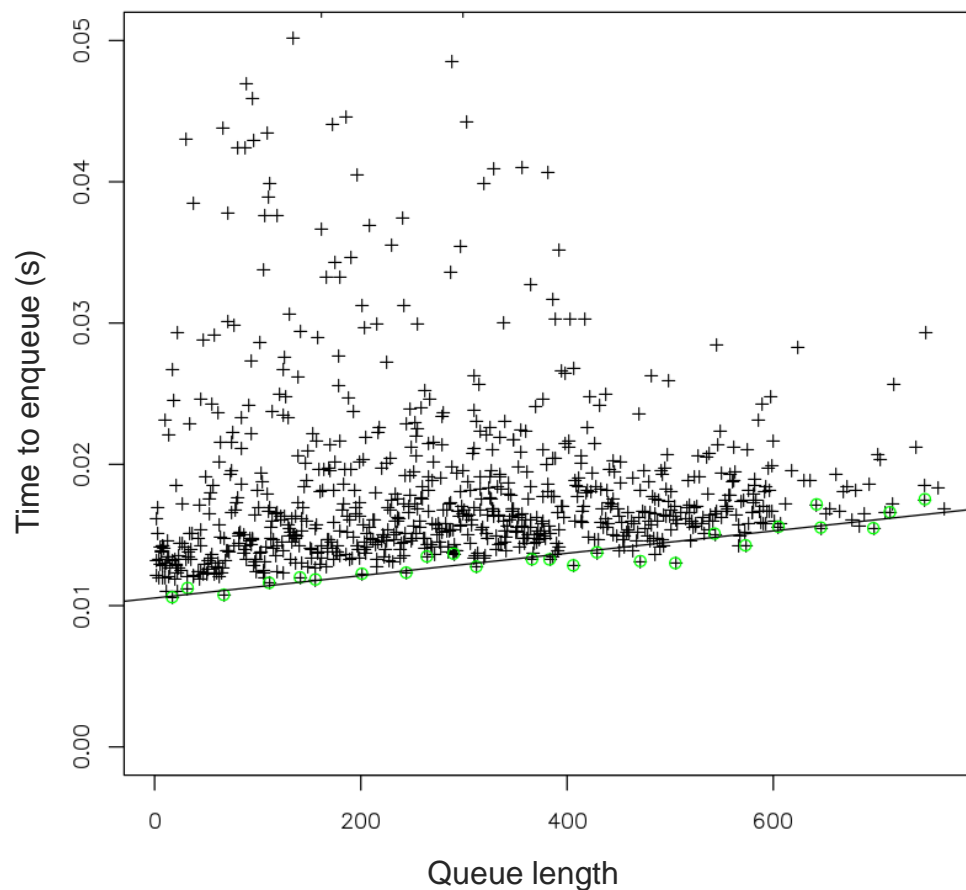
Achieved metadata performance

- 100k small (4B) files test (archive, then retrieve)
- Drives down until all archive requests created
- Enqueuing in 15 min => 111Hz
- Executed in 16 min => 104Hz
- Mixed enqueue+execution in 28min => 60Hz
- Statistically, CASTOR sees few 100k transfers max per day (below 1M)
- We get here ~5M per day, per queue (queues are independent)
- 13 real drives competed for the queue
- Aggressive mount policy



Queue performance scaling w/size

- Time to insert (one or several) entries
- Variations due to network timings
- Minimum time can be identified
 - Grows with queue size (full write)
- Linear regression of local minimums (green circled points):
 $t \approx 10ms + n \cdot 8\mu s$
- To get $t < 100ms$, $n < 100k$
- => Queues sharding to be implemented (cut queue into sub blocks)



CTA architecture vs CASTOR's

- CASTOR
 - File level requests in per stager queues
 - Queues statistics only inside each stager
 - No global view
 - They are recomputed on each (SQL) request (including a sum of sizes)
 - Each stager decides to generate mount independently
 - Tape from any library, without regard for drive availability
 - Mounts queued in central VDQM (with data reduction, resulting in user visible delays and resource wasting: one library idle, the other saturated)
 - Many daemons as “middle men” (VDQM, VMGR, CUPV...)
- CTA
 - File level requests queued centrally
 - Queue statistics gathered at the queue object level
 - Global view available
 - Statistics updated on insertion/pop (running counters)
 - No computation for querying
 - No central single point of failure: tape drives poll the queues directly and run the scheduling algorithm for themselves
 - Single step decision, taking all resources into account:
 - Tape availability for this drive
 - Global fairness for users

Examples: queues querying

- Heart of the scheduling poll loop for both CASTOR and CTA
 - DB job in CASTOR (mount creation in stager)
 - Polling in each drive in CTA (self scheduling)

```
[root@c2repack-1 ~]# time printmigrationstatus
      TAPEPOOL FILES FILES_MIGRATING TOTAL_SIZE SIZE_MIGRATING MIGRATIONS MIGRATIONS_QUEUED MIGRATIONS_RUNNING MIN_AGE MAX_AGE
-----
stuck migrations      0          0      0B          0B          0          0          0          0s      0s

real    0m10.576s
user    0m0.048s
sys     0m0.028s

[cano@tpsrv600 ~]$ time cta showqueues -h
      type  tapepool  vid  files queued  MBytes queued  oldest age  priority  min age  max drives  cur. mounts  cur. files  cur. MB
Archive  test_eric  -    24316      0.10      558      1      1      13      10      10690      0

real    0m0.273s
user    0m0.006s
sys     0m0.018s
```

DBs pros and cons for queue management

- Pros
 - SQL language is (mostly) standard
 - Easy to find competency
 - Faster development time
 - Easy data consistency (multi-table transactions)
 - Good performance for easy cases
 - Reliability handled by the DB (but not free: gold plated hardware, DBA)
 - All transaction are stored and can be recovered
- Cons
 - Performance tweaks are not standard
 - Normalized data requires describing the structure over and over in each query
 - Execution plans stability requires DBA or ad-hoc tweaking
 - Optimizer driven by (possibly out of date) stats for tables
 - Poor performance when tables grow and shrink (queues)
 - Query rewrite in MySQL
 - Hinting in Oracle
 - Execution plan freeze in Oracle (by DBA)
 - Table storage accumulates history
 - Blocks are never released, leading to costly full table scans (with zero row)
 - Single/dual point of failure
 - Storage is a single point of failure (see recent FS corruption in NetApps)
 - DB server in a single/dual point of failure

Object store pros and cons

- Pros

- Richer objects (using protobuf)
- No need to join at run time
- Distributed object stores
 - Best object level reliability and availability
- Contention can be tuned for a given application
 - Can be made to scale well
- Single language use (C++)
- Single thread of execution in a single place (easy algorithm debugging with gdb)
- Objects can be easily listed, dumped
- Storage size in control (requests/queues removed after use, system goes back to a consistent “ground state”)
- Easy to switch backends as little assumptions are made on them
 - Redis + Redlock
 - Zookeeper(?)
 - NFS or other shared file systems

- Cons

- Dependent on the serialization package (protobuf)
- Locking support required
- More round trips to the storage (3 in the “drive ls” example)
 - Small queue size performance less than DB based systems
- More coding required to handle process failures
 - Dangling pointers (object not owned, missing)
 - Garbage collection
- Locking implementation required some fine tuning to give good performance

Conclusion

- CTA's object store provides reliable and performant queueing
- Improvement still in the pipeline
 - Very deep queue scaling
- Performance on par with CASTOR's already
- Object store approach give us more control and better availability and scalability at the expense of more development effort
- Minimal lock-in (except protobuf)

Castor stats 2017

Total requests over 24h (write), Jan–Nov 2017 for c2public

