CERN Tape Archive: a distributed, reliable and scalable scheduling system

Eric Cano[1,*], Vladimír Bahyl[1,*], Cédric Caffy[1,*], Germán Cancio[1,*], Michael Davis[1,*], Oliver Keeble[1,*], Viktor Kotlyar[2,**], Julien Leduc[1,*], and Steven Murray[1,*]

[1]CERN—European Organization for Nuclear Research, 1211 Geneva 23, Switzerland
[2]Institute for High Energy Physics named by A.A. Logunov of National Research Center "Kurchatov Institute", Nauki Square 1, Protvino, Moscow region, Russia, 142281

Abstract. The CERN Tape Archive (CTA) provides a tape backend to disk systems and, in conjunction with EOS, is managing the data of the LHC experiments at CERN.

Magnetic tape storage offers the lowest cost per unit volume today, followed by hard disks and flash. In addition, current tape drives deliver a solid bandwidth (typically 360 MB/s per device), but at the cost of high latencies, both for mounting a tape in the drive and for positioning when accessing non-adjacent files. As a consequence, the transfer scheduler should queue transfer requests before the volume warranting a tape mount is reached. In spite of these transfer latencies, user-interactive operations should have a low latency.

The scheduling system for CTA was built from the experience gained with CASTOR. Its implementation ensures reliability and predictable performance, while simplifying development and deployment. As CTA is expected to be used for a long time, lock-in to vendors or technologies was minimized.

Finally, quality assurance systems were put in place to validate reliability and performance while allowing fast and safe development turnaround.

1 Introduction

The CERN Tape Archive (CTA), in conjunction with the EOS disk system, stores the Physics data of experiments at CERN. All the experiments at the Large Hadron Collider (LHC) have been migrated from the CERN Advanced STORage (CASTOR) to the CERN Tape Archive (CTA), and CASTOR is being phased out gradually for the other experiments, current and past [1].

CTA is a tape backend for file systems and initially targets EOS. Other filesystems could use it and adaptation for deployment behind dCache is being investigated [2]. CTA incorporates the knowledge accumulated during the years of CASTOR operation [3]. Unlike CASTOR, CTA contains no file directory structure or disk storage system: this is the responsibility of the client disk system. Like CASTOR, CTA keeps track of files stored on tape in a file catalogue, keeps track of the transfers between disk and tape in multiple queues and manages the tape drives.

*e-mail: {eric.cano,vladimir.bahyl,cedric.caffy,german.cancio.melia,michael.davis,julien.leduc,oliver.keeble,steven.murray}@cern.ch
**e-mail: viktor.kotliar@ihep.ru

As the time to mount a tape is significant — $O(1\,\text{min})$ — tape systems need to accumulate transfer requests in multiple queues before it is worth mounting a tape. Writes define the destination as a tape pool, while reads are limited to the tape where the file is located. Both CASTOR and CTA logically queue reads to individual tapes, and writes to tape pools, creating a natural grouping of requests. Multiple properties of the content of the queues and drive statuses are then polled to decide when a new tape mount should be started, based on data volume, request age, priority and user drive allowance. Those requirements of queue content introspection make standard message passing packages inadequate.

In practice this means there are up to $O(1000)$ queues, whose properties should be inspected to reach a mount decision. Such a queueing system should be reliable and provide adequate throughput. The latency is not always critical, but some user visible operations require it to be kept low. Finally, as CTA is going to be used in the long run, particular care should be taken in avoiding vendor or technology lock-ins.

This article will describe the new queueing system introduced with CTA, which builds on the experience gathered in CASTOR as well as modern possibilities in databases and object stores to meet those challenges.

2 Issues and opportunities in the CASTOR - EOS scenario

2.1 Vendor lock-in of CASTOR into Oracle due to implementation in PL/SQL

The scheduling of CASTOR [4] is based on a relational database that stores the queued requests both for disk and tape. While the disk operation requests are processed in order and as soon as resources are available, the tape operations require grouping, sorting and summarizing to take decisions and execute requests. Requests are grouped by tape pool or tape volume identifier (VID), while sorting is by location on the tape volume for reads and age for writes. In order to achieve good performance, those operations were implemented as PL/SQL inside the Oracle database. This led to a vendor lock-in and the impossibility to run CASTOR on another database as too much code would have to be ported.

This constant summarizing of queues to get totals and extrema require the database to go through all the elements of the queue repeatedly. Experience in production has shown those queries to yield unreliable performance. As the queues vary in size so do the underlying tables, but the disk blocks for their physical storage are not cleaned up accordingly. This lead to sub-optimal performance, to the point of creating user visible incidents. Mere popping from such a queue led to incidents in the past years.

2.2 High cost of inter-daemon communication

CASTOR having been developed 20 years ago, its architecture reflects constraints that have since disappeared. The most notable one is the maximum number of connections a database allows. Low in the past, this number required servers in front of the database. This is not the case any more for $O(100)$ connections, which is the order of magnitude of the number of drives in production. In turn, creating servers required the development and maintenance of protocols between said daemons and the clients (disk and tape servers), which was a significant part of the development.

This architecture led to the deployment of many utility daemons, which neither interfaced to the user nor directly managed data transfers to the hardware. Deployment of any change of protocol between all those daemons required careful planning between the client and the server.

2.3 Redundancy of disk storage between CASTOR and EOS

With both EOS and CASTOR providing a distributed disk based storage, the duplicated functionality was an opportunity to simplify our software portfolio. CTA removes this duplication by adding a tape back end to EOS, and makes the most of the accumulated experience from CASTOR, while taking advantage of modern opportunities. EOS is the initial target in the development of CTA, but its architecture is open enough to accommodate other file systems.

2.4 CTA Architecture

With databases able to handle one connection per tape drive directly, the intermediate daemons are no longer a necessity. In addition, distributed key value stores like Ceph — which have been deployed at CERN for a long time [5] — naturally support a high number of clients. CTA therefore relies on a shared central storage — database and object store — directly accessible by every process, into which the functions of CASTOR's utility servers are collapsed. The layout of the classes as a software stack is shown in figure 1, and these components are described in the following sections.

2.4.1 Catalogue

The CASTOR namespace is held in a database, with the two biggest tables for file entries and tape copies for said files. CTA does not diverge much from the CASTOR namespace in this area. As the data is stable, and mostly grows with time, the use of a database is appropriate. In addition to the file catalogue, this database also tracks the tapes, tape pools, storage classes – which map files to tape pools – and drive allowance per user. This database is directly accessed by all the processes of CTA: front end to the user, tape server processes and maintenance processes — which will be detailed below. The catalogue C++ object interface is defined as pure virtual interface potentially allowing multiple implementations of the service. In practice, a single umbrella implementation gathers most of the queries, which are common to all supported relational database implementations and inheriting classes implement both the specifics to each database and the access mechanism.

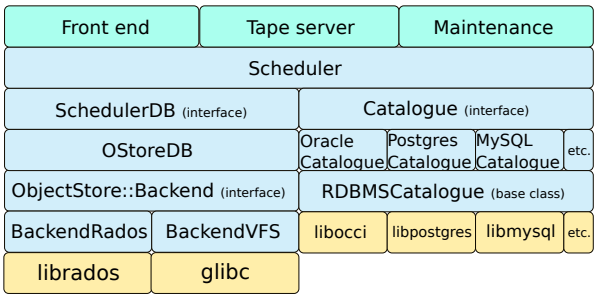

Figure 1. Software stack layout

2.4.2 Scheduler DB, aka Objectstore

On the other hand, CTA's queueing and request tracking is radically different from CASTOR's. The queues for different tapes or tape pools are represented by separate objects in a key value store, and each request is represented by an individual object. In addition, this object store contains the status of tape drives and of each process operating on the objects, in order to recover incomplete operations and ensure object store consistency after a process crash. A completely shared implementation relies on multiple implementations of the access mechanism in backend classes. A pure virtual interface was also foreseen to allow a complete re-implementation of the scheduler DB if needed, but like for the catalogue, no alternative for the scheduler DB was implemented.

2.4.3 Scheduler

A scheduler C++ object is a client to both the catalogue and the scheduler DB objects, and in turn provides the functions needed by all of the processes of the system. The scheduler makes the connection between the two persistent storage systems and takes the decisions for action (mounting a tape, finding files to transfer for a tape session or keeping track of successes and failures).

2.4.4 Impact of the architecture

The different daemons having been collapsed into C++ classes, and run with one instance per process, what used to be calls over the network are now simple function calls. Changes to their APIs do not impose constraints on deployment anymore as the update comes in a block at process restart.

Changes in the data schema on the shared storage are handled in a variety of ways. Protocol Buffers provide a lot of flexibility to handle changes. The objects in the object store foresee schema versioning, allowing an opportunistic migration of the objects as they are visited during normal operations, but this mechanism was never implemented as simpler ways were always found.

The consolidation of the scheduling and namespace information in 2 storage systems instead of 4 in CASTOR and their unification by the scheduler object allows a greater flexibility in implementing the scheduling algorithms, with a complete view of the situation. This allows better decisions, like avoiding deciding to schedule for mounting a tape located in a library with no drives available.

2.5 Development techniques

Thanks to the availability of trivial backends — local file system for the object store and in-memory SQLite for the catalogue — unit tests running in a single process can validate not only the base functions but also significant chunks of the software stack, including the main scheduling scenarios, simulating the calls from a tape server and front end. The fact that the unit test runs in a single process also allows validation of memory allocation and threading with Valgrind and Helgrind [6]. All those validations are part of the continuous integration system (CI) and are run automatically. In practice, this made race conditions and memory leaks a non-existent problem in production.

A Kubernetes environment was put in place to validate the deployment in a virtual environment. This allows running a full stack CTA, including EOS, virtual tape drives and Kerberos in a CI environment. The same Kubernetes environment allows running the full stack in a disconnected environment, like a laptop. Running against any other backend is also possible.

2.6 Advantages

The catalogue part, stored in a relational database, is mostly unchanged with respect to CASTOR and carries the usual advantages and drawbacks of a database: queries are simple to write, and algorithms are relatively well chosen automatically by the optimizer, as long as the data profile is stable over time.

The object store allows a direct representation of the data, which can be split into more objects to reduce contention by increasing the granularity of locking, where needed. On the other hand, arbitrarily complex structures can be serialized in objects using Protocol Buffers,

allowing a reduction of code complexity where multiple tables are needed in an RDBMS environment. Each object of a given type can also have its schema updated independently from other objects, allowing many options in case of schema update. In summary, we have complete objects with independent schemas as opposed to schema-synchronized rows in multiple tables, which require joining when dealing with complex cases like structures containing arrays.

The algorithms used on the object store are well controlled and can be tuned to needs, avoiding the RDBMS optimizer issues mentioned in section 2.1.

2.7 Costs

The object store algorithm flexibility comes at the cost of development effort, including the handling of error cases arising from the nature of running on distributed, unreliable processes. This aspect will be developed in section 4.2.1.

3 The Catalogue

3.1 Multiple back-end implementations, included one contributed.

Due to the systematic structure of the catalogue data most of the operations can be implemented in generic SQL statements. A base class gathers all the commonalities using SQL statements applying to all DBs. Derived classes implement the concrete access to each DB with its specificities and implement the few operations that benefit from a dedicated, per backend implementation. The order of magnitude is 10 % of the approximately 130 member functions of the catalogue class interface. Four backends exist: PostgreSQL, Oracle and MySQL/MariaDB (contributed by IHEP, Beijing, China) for production use cases and in-memory SQLite for validation/unit tests.

4 The Scheduler DB, aka Objectstore

The scheduler DB provides the tracking of the transient data transfer requests — to and from tape — as well as the internal states of the drives and of all the processes interacting with the scheduler DB. The underlying storage is a persistent key-value store. The interface to the backends is defined by an abstract class, and two concrete implementations exist: one using Rados, the distributed key-value store of Ceph, and one relying on a Linux Virtual File System (VFS). The latter is only used in tests, and while a Network File System (NFS) based deployment would work in theory this was never tested because it presents a single point of failure unless expensive hardware is used.

The scheduler database has a heterogeneous tree structure. Objects are stored as values in the key-value store, and referenced by their key. A special object with the key "root" is at the root of the structure. The objects contain arbitrary structures serialized with Protocol Buffers [7]. Protocol Buffers is an open source serialization and deserialization package developed by Google. It allows storage of any composition of structures, arrays, scalars and strings in a size-efficient and portable binary format.

Objects reference each other by their key. Some objects representing processes and tape drives have a static location and are referenced by fixed registries. The objects representing processes are called "agents". Queues also have a static location.

Other objects have a more dynamic life cycle: the requests. Requests to transfer data to and from tape, respectively named ArchiveRequests and RetrieveRequests, are referenced

by various structures during their life cycle. They are attached to queues when pending processing and are attached to the agent object of the process working on them. This per-status queueing simplifies the writing of the processing stages. The tree structure is illustrated in figure 2.

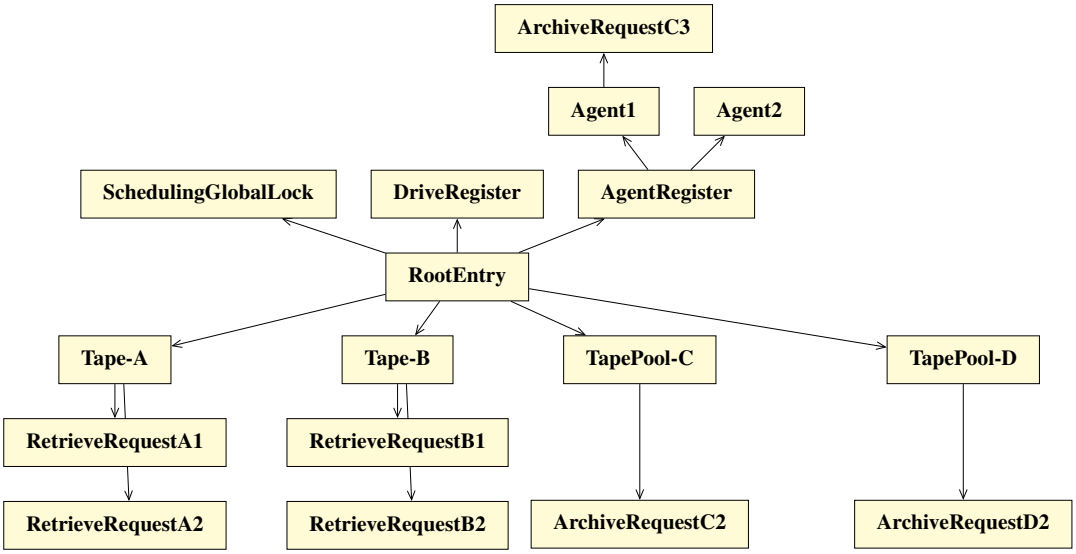

Figure 2. Object store's instance diagram showing retrieve queues for Tape-A and B, archive queues for TapePool-C and D, and 2 Agents, one of which is processing an archive request

4.1 Challenges

As the data structures are shared, permanent access from multiple processes should ensure consistency at all times, including after a process crash. The problem is hence similar to classic multi-threaded programming with extra considerations for process crash and latency, the distributed key-value stores having much higher latency than local memory. On the other hand, the key-value stores allow asynchronous IO, which eases algorithmic complexity to $O(1)$ in many cases.

In order for the data to remain coherent in the event of a process crash the data structures must have a valid state at each object update. Key-value stores do not provide multi-object transactions, so this limitation was worked around.

As the data are accessed from multiple processes locking is needed and we used classic locking techniques to synchronize them.

Blocking between processes can limit performance and locking strategies should ensure contention between processes accessing the object store does not impact performance.

We had to ensure that multiple back ends can be used or added if necessary (as with the catalogue).

4.2 Implementation and algorithms

4.2.1 Reliability with unreliable processes

The interface to the backend is defined in an abstract class according to the needs of the relying class, the schedulerDB. This includes atomic creation, fetch, update and deletion of ob-

jects, per-object locking and asynchronous operations. The backend objects implement those calls in the native libraries of the underlying storages. In all cases the key is any arbitrary string and the value an arbitrary bitstream. Other backends can be added by implementing the interface in a new class.

In order to avoid clashes, while providing a human-readable naming convention, each process determines a unique name for itself and adds a serial number to each object it creates. In addition the name of the object has a prefix indicating the type of the object.

In order to prevent objects from being orphaned and allow safe deletion and atomic transfer from one position in the tree to another, references to objects are allowed to be stale, and the software handles the situation. Objects confirm being owned by a given container by having an "owner back reference" to the active owner. Stale references cover both cases of reference to a non-existing object and references to an object not confirming the ownership with the back reference to the owner. This allows atomic transfers of ownership where the new owning container (queue or agent) first references the object. The new reference is then confirmed by updating the object, which is an atomic operation. The old, invalid reference is then cleaned up from the previous container. A process crashing before the owned object update leaves the ownership unchanged, while if crashing afterwards, leaves at most an invalid reference that will be disregarded and cleaned up eventually. The stale references are not normally seen by other processes thanks to locking. This safe ownership transition is illustrated in figure 3.

A typical ownership change happens when the state of a request gets updated. The owner backpointer update and the internal status update in the request object are committed in the same object write, as this both saves a round trip to the storage and ensures both operations happen in the same intra object transaction.

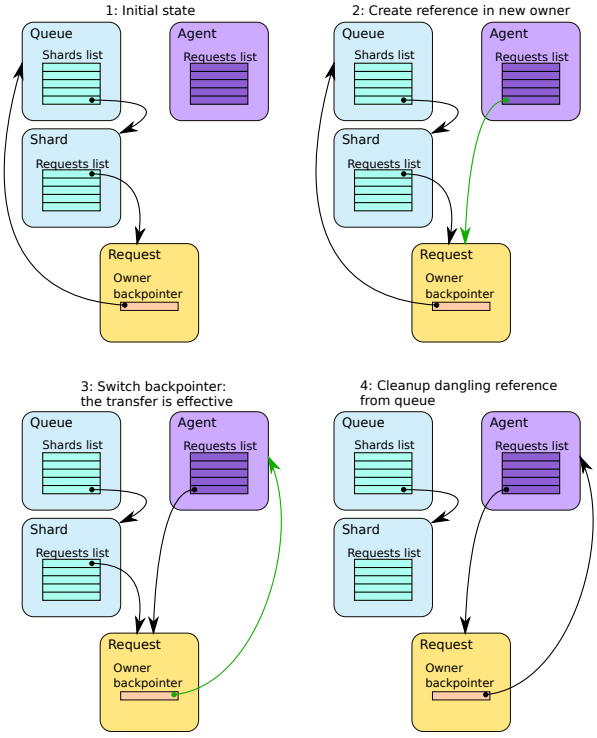

Figure 3. Sequence of the safe ownership change from a queue to an agent (queue pop)

Each ownership change operation involves at least 9 round trips to the object store: locking of the destination container, fetching of the container, adding the new reference, locking, fetching and committing the request, and then locking, fetching and committing the source container. As requests are processed in bulk in the tape system, many new references are added to the destination container in one go. Then the requests are locked, fetched and updated in parallel using asynchronous IO, and finally the source container is updated. This parallelism allows an update bandwidth mostly independent from the key value store la-

tency, at the cost of slightly more complex error handling (source and destination references to requests might need to be corrected, adding an extra round trip).

4.2.2 Performance: latency hiding, contention domain control

The interface to EOS is driven by file-by-file user interactions. Transfer requests in both directions, to and from tape, are created and queued at this point. In order to hide the queueing latency, which can vary depending on the contention on the queue, the front end first creates a reference to the request in its agent object and then the request itself, before reporting to the client that the request has been received. The request is indeed safe in the system at that point, as it is referenced as owned by the front end process. In case of process crash at that point, the queueing of the requests will be done by the garbage collection.

As the requests are created from multiple threads independently, and to avoid competition between threads of the same process for objects in the object store, both agent object and queue accesses are mutualized by a thread synchronization mechanism. The first thread needing to access a container creates a batch with its own request reference, and waits for any running access to the container to complete. Other threads, noticing the existence of the batch, simply piggy back on the batch and wait. Once the previous access completes, the first thread wakes up, closes the batch and signals it is accessing the container to a potential followup thread, references the requests in the container, and signals all the waiting threads they can proceed with their per-request part of the queueing.

Contention can be controlled by adjusting the size of contention domains. This means spinning sections of a structure off, into distinct objects. For example, initially the drive states were all stored in a single object, leading a non-scaling contention when more drives are added to the system and competed to update their value on the drive registry. The drive registry was then turned into a two-stage structure where each drive has it own state object, and the central registry only references the state objects. The drives stopped competing to update their statuses and only the rare referencing and de-referencing of a drive can lead to contention. On the other extreme of the spectrum, we could imagine a single root object containing all the scheduling data. Any update of it would require taking a global lock and block any other update in the system.

The majority or reads are non-contended as lockless reads are generally adequate. Some operations can also be probed in read-only mode before deciding if a locked run is necessary. This is used for scheduling new tape mounts, where a dry-run scheduling is executed without locks, and only if a mount seems possible is a global scheduling lock taken, followed by a committing scheduling. As the actual scheduling of a new mount is a rare event at the scale of the object store accesses, a single global lock is adequate in this case.

4.2.3 Queue structures for arbitrary size

In order to keep the size of objects capped, the queue objects are sharded. This keeps the update time bound even in the presence of arbitrarily long queues — tests were pushed to several millions — as well as keeping the size of individual objects below the underlying storage limits (100 MB for Rados). The queue is composed of a header, referencing the shards and their properties, and shards referencing the individual requests. To avoid accessing the individual objects, necessary values for them are cached with their references. Most notably for the extrema cases like oldest request age, each extremum for a shard is cached next to the reference to the shard, allowing the efficient re-computation of the global extremum over all shards after updating one. Likewise, any value of interest from a request (age, size, and position on the tape for reads) is cached inside the shard with the request reference, allowing

the recomputation of the shard's totals with only the shard itself loaded in memory from the object store. This relies on the fact that in CTA request properties are immutable for the lifetime of the request.

The queues for reads are sorted, as reading a tape in order is more efficient than a random location order, which would result from FIFO ordering. Algorithms have been devised to assign requests to the proper shards before creating or updating them in turn, with the main queue object. When insertions happen in the middle of the queue, a given shard can be split in two to keep the shards size bound.

With queues sharded, updates to the queue contents are no longer atomic. For this reason, queueing always ensures that references are added to one shard before being removed from another, in the shard split scenario. This ensures that references are present at least once in the queue in the case of a process crash. Double references will lead to slightly off contents summaries, but as the software tolerates dangling references, the queue will eventually return to correctness as processes consume from it.

Finally, when scheduling a tape mount, the tape drives load all the main queue objects asynchronously and without locking, which costs on first order one round trip time, irrespective of the number of queues present. The main queue objects contain all the necessary values for deciding which queue to process in the next mount, if any.

These queues are the workhorse of CTA scheduling and required a significant amount of development, but provide guaranteed performance queueing with no vendor lock-in.

4.2.4 Maintenance process, garbage collection

As the tape sessions lock a tape drive while they work, the amount of scheduler housekeeping was kept to a minimum in the tape daemon process, in order to ensure the tape drive is utilized as much as possible.

The rest of the life cycle of requests is executed in a different process. This maintenance process makes sure that successful transfers to tape are reported to the EOS client system, and ultimately to the user. Having this step of the transfer outside the tape session provides the additional benefit of allowing a retry of the reporting step in the case of failure.

The maintenance process also handles repack scheduling, where operators queue tape volume identifiers (VIDs) and the system automatically expands them to file-level detail on demand, so that millions of files are not queried from the catalogue in one go. The maintenance process ensures the expansion is just a few tapes ahead of the current execution. The repack requests are executed in a similar way to the user requests, with a special reporting process.

Finally, the maintenance process handles the garbage collection of dead processes. Processes are represented by agent objects, which are normally owned by a central agent registry. Each maintenance process watches all the agents in the system. The agent objects contain an increasing heartbeat counter which stops being incremented after a crash. When a maintenance process detects such a crash, it takes ownership of the agent object and then requeues all the requests, and more generally returns the owned objects to their original location. For example, if a maintenance process crashes during garbage collection, the owned agent will be returned to the registry, re-detected as a dead process and garbage collected again by another process.

4.3 Limitations

While vendor and technology lock-in were carefully avoided where reasonable, CTA is dependent on Protocol Buffers serialization at its core. Serialized objects and their memory

representations all rely directly on the Protocol Buffers calls. No wrapper was devised for this part as the work required would probably barely break even when introducing a second serialization package. As Protocol Buffers are a standard part of current Linux distributions, the need for such a replacement is unlikely in the foreseeable future.

5 Conclusion and possible generalization

This article described methods used to develop a reliable shared structure that is accessed in a distributed manner. The structure inherits the reliability of its underlying storage, and in the case of Ceph provides the absence of any single points of failure, while running on commodity hardware.

By moving the queueing burden and complexity out of a relational database, this queueing system future proofed CTA by ensuring it is not vendor or technology locked.

While the methodologies used are systematic, the source code is only partially generic, allowing reuse of queues in different contexts. Further generalization might provide in-process container-like interfaces to those distributed structures.

References

[1] Cano, Eric, Bahyl, Vladimír, Caffy, Cédric, Cancio, Germán, Davis, Michael, Kotlyar, Viktor, Leduc, Julien, Lin, Tao, Murray, Steven, EPJ Web Conf. **245**, 04013 (2020)

[2] Tigran Mkrtchyan et al., *dcache: Inter-disciplinary storage system* (Presented at CHEP2021, to be published)

[3] S. Murray, V. Bahyl, G. Cancio, E. Cano, V. Kotlyar, D.F. Kruse, J. Leduc, Journal of Physics: Conference Series **898**, 062013 (2017)

[4] G. Lo Presti, O. Barring, A. Earl, R.M.G. Rioja, S. Ponce, G. Taurelli, D. Waldron, M.C. Dos Santos, *CASTOR: A Distributed Storage Resource Facility for High Performance Data Processing at CERN*, in *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007), 24–27 September 2007, San Diego, California, USA* (IEEE Computer Society, 2007), pp. 275–280, ISBN 0–7695–3025–7, http://doi.ieeecomputersociety.org/10.1109/MSST.2007.7

[5] D.C. van der Ster, M. Lamanna, L. Mascetti, A.J. Peters, H. Rousseau, Journal of Physics: Conference Series **664**, 042054 (2015)

[6] N. Nethercote, J. Seward, SIGPLAN Not. **42**, 89–100 (2007)

[7] *Protocol Buffers*, https://developers.google.com/protocol-buffers//