CASTOR: OPERATIONAL ISSUES AND NEW DEVELOPMENTS

O. Bärring, B. Couturier, J.-D. Durand, S. Ponce, CERN, Geneva, Switzerland

Abstract

The Cern Advanced STORage (CASTOR[1]) system is a scalable high throughput hierarchical storage system developed at CERN. CASTOR was first deployed for full production use in 2001 and has expanded to now manage around three PetaBytes and 26 million files. CASTOR is a modular system, providing a distributed disk cache, a stager, and a back end tape archive, accessible via a global logical name-space.

This paper focuses on the operational issues of the system currently in production, and first experiences with the new CASTOR stager which has undergone a significant redesign in order to cope with the data handling challenges posed by the LHC, which will be commissioned in 2007.

The design target for the new stager was to scale to another order of magnitude above the current CASTOR, namely to be able to sustain peak rates of the order of 1000 file open requests per second for a PetaByte disk pool. The new developments have been inspired by the problems which arose managing massive installations of commodity storage hardware. The farming of disk servers poses new challenges to the disk cache management:

- request scheduling; resource sharing and partitioning,
- automated configuration and monitoring,
- fault tolerance of unreliable hardware

Management of a distributed component like CASTOR system across a large farm provides an ideal example of the driving forces for the development of automated management suites. Quattor[2] and Lemon[3] frameworks naturally address CASTOR's operational requirements.

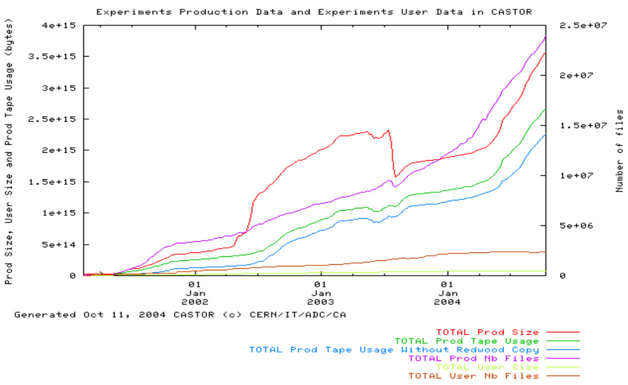

Figure 1

CASTOR CURRENT STATE

The production version has reached significant storage capacity; around three petabytes of data spreaded over 25 million files, as showned in the figure 1.

There are two main subsystems in this version:

- The central services, like a name server, daemons managing tape information and request queuing, and the tape servers
- Diskservers, hosting one or more filesystems, managed by a *stager*.

All data moving and control is done using the Remote File Input/Output (RFIO) layer.

There no direct access to tape: the data is always cached on disk both for read and write. That is, a user wanting to read a file will have to wait for its availability on one of the filesystems, and a user creating a file is doing so on a filesystem as well. The *stager* is responsible of the consistency over all the filesystems, launching *recallers* and *migrators* as needed, freeing space with *garbage collectors*.

In contrary to central services that are using an RDBMS, the stager daemon uses a home-made database and suffers from its architecture, made of layers added to layers every year – leading to some hardly removable features like having an internal copy of its catalogue in memory.

This configuration worked fine until the number of diskserver at CERN started to grow causing the stager catalogue size to exceed the physical memory available.

PROBLEMS WITH CURRENT CASTOR SYSTEM

The management flexibility provided by the current version has become a problem and, in particular its lack of fault tolerance. For instance, if something goes down, or worse becomes very slow, the whole system suffers. This has to do with the stager daemon design, which is not doing true request scheduling or throttling. Instead, it is guessing what is a good matching resource candidate using simple algorithms based on the its internal knowledge of previous assignments rather than the actual system load. A side effect is that the diskservers are not easy to manage individually. For instance, no facility is provided for draining a disk server in preparation for an upcoming intrusive maintenance intervention. As a consequence a large number of administrative scripts have been developed around the stager daemon but the management is still not robust and fault tolerant.

Independently, the increase of the number of disk servers and/or filesystems at CERN pushed the system to its limits:

- scalability problems
- performance hickups
- needs of a proper resource sharing
- internal catalog of the stager daemon limitations
- sub-optimal use of resources
- etc...

Right now, the number of different instances of the CASTOR stager daemon has considerably increased (~50 these days), but the more instances of it, the more idle resources, as soon as a dedicated stager daemon is not used , for example during one month.

It has been acknowledged that the current system do fit well the LEP experiments way of working – but LHC experiments will have very different requirements. For instance, proper security mechanism (for grid access in particular), access policies, etc...

Concrete requirements of LHC era are for example:

- CASTOR should scale up to **500/1000** requests per second (a request is a *file opening*)
- 4 Petabytes of disk cache
- 10 Petabytes to tape per year
- **10000** disks
- increased number of small files
- etc...

The conclusion is that the production version currently running at CERN must change, or the CASTOR product will not meet the LHC requirements – in contrary it will become a bottleneck.

VISION FOR THE NEXT CASTOR GENERATION

With clusters of hundreds of disks and disk servers, the automated management faces more and more the same problems as *Computing* (not storage) resource sharing for CPU clusters:

- Resource management
- Sharing
- Scheduling
- Configuration
- Monitoring

So the vision for the next CASTOR is ***Storage Resource Sharing Facility***.

ARCHITECTURE OF THE NEXT CASTOR GENERATION

The following components are new (see figure 2):

- Request Handler
- Resource Monitoring Agent
- Resource Monitoring Manager
- Scheduler
- Policy Engine
- Security layer
- Distributed Logging Facility
- Expert System

The following has been disassociated from the production version, acting now as stand-alone processes:

- Migrator
- Recaller
- Garbage Collector

These components has been substantially upgraded:

- Tape Mover

The code development methodology has changed. All algorithms, database schemas, flow charts are done in the UML[4] style, and in particular the Database Interfaces are generated using the UML schemas – we add by hand private methods whenever needed (in particular in case of methods accessing several tables).

The main design concepts are:

- Database Centric Architecture
    - Requests States are stored in a relational database (RDBMS)
    - This will allow very big catalogs
    - Locking, transactions will be handled by the database
    - The CASTOR services on top of the database are stateless, which facilitates the administration

  - We benefit from standard (hot) backup procedures that comes along with a well administered RDBMS
  - The system will scale at the same rate as the database and if necessary Database clustering could be used for optimal scalability
- Externalized Scheduling
  - We believe that a complete scheduling system is not trivial to write from scratch. Indeed, schedulers are often offering features like:
    - fair share
    - backfilling
    - reservation
    - statistics and accounting facilities

  Therefore we decided to take advantage of existing extendible scheduling systems and the first candidates were LSF[5] (in production at CERN), and MAUI[6]. Other schedulers has not yet been integrated.
- Improved Security
- Policies and Rules
  - We will use a true policy engine, CLIPS [7], to express all policies. CLIPS integrates well in C programs, but we neverthless install an explicit *expert service*, where all CLIPS rules are grouped. The goal is to let administrators, and experiment responsibles, modify the policies.

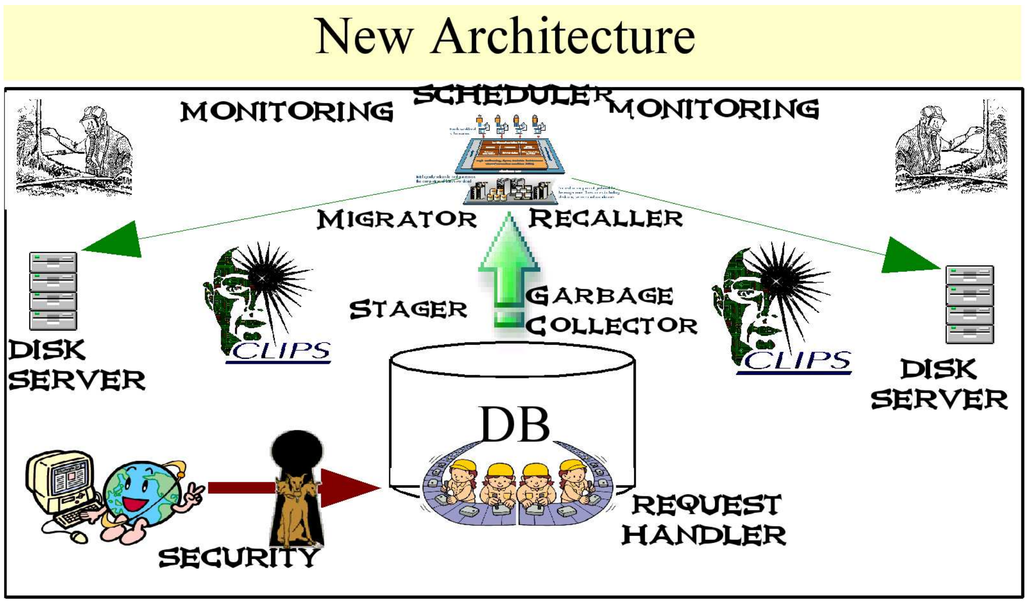

Figure 2

Request Handler

This is how we handle the throttling into CASTOR. All incoming requests are not processed as they arrive, but are stored in a *incoming* table of a database. This component is designed to handle hundreds of requests per second. Another component, which we still call the *stager,* will asynchronously retrieve the requests and process them.

Resource Monitoring Agent

This is lightweight process doing the same as most of the usual monitoring agents: cpu, disk space, processors, i/o, etc... information is sent with UDP packets to a central Resource Monitoring Manager.

Resource Monitoring Manager

It receives all monitoring information from all the agents. It means that we have a global view of the whole cluster by querying the resource monitoring manager.

This component is also doing more: some schedulers, like MAUI, can work with an external resource manager, polling it regularly to get all the metrics that have a meaning for them. This is what is happening: the Resource Monitoring Manager has an explicit interface to the MAUI scheduler, answering to "*give me all the nodes you know*" and *"give all the jobs you know*" commands. In return, the MAUI scheduler might decide to schedule, in which case it will use the same interface to say "start this job". In the MAUI terminology, these three protocols are called GETNODES, GETJOBS, STARTJOB respectively.

The MAUI scheduler does not really start the jobs but the Resource Monitoring Manager will do so when it receives the STARTJOB order.

Some other schedulers will work the other way round. For example LSF will start the job itself. In such a case, a

CASTOR job will subscribe itself to the list of running jobs to the Resource Monitoring Manager.

The Resource Monitoring Manager also provides another function needed by the MAUI interface: Jobs Monitoring Manager. This component regularly checks if jobs are alive and updates its knowledge correspondingly.

Scheduler

Two schedulers are curently integrated:

- LSF
  - This scheduler offers a built-in plugin facility. A shared object providing well-defined entry points is provided.
  - Version 5.1 of LSF was succesfully tested, with the price of some hacks to bypass internal limitations of the knowledge that the plugin has. Our plugin showed also few deficiencies in this version of LSF, which has been overcome in the new version 6.x of LSF thanks to an open and very fruitful collaboration with the LSF developers.
  - Version 6.x of LSF is almost ready, waiting for remaining minor modifications of the LSF master scheduler
  - LSF has a built-in Resource Manager collecting information using a special syntax. Our Resource Monitoring Agent supports the LSF syntax, and reports the information to LSF as well as our Resource Monitoring Manager.
  - Jobs are started by LSF itself.
- MAUI
  - The CASTOR plugin for MAUI has to be explicitely linked with the scheduler binary.
  - A fruitful collaboration with MAUI developpers allowed to have a satisfactory version based on a development version of the next MAUI scheduler generation, called MOAB.
  - MAUI tells when to start a job, and we execute the order.

Thus, both LSF and MAUI schedulers provide the functionality we need.

Policy engine

We prefer to give up with rules and policies that 'look like' dynamic, when in reality there are not at all: in the production version of CASTOR, for example, algorithms are hardcoded in the services, and only some parameters can be changed on the fly. The administrator can change the behaviour of a given algorithm, or choose another one, but *cannot* create its own policies.

Having a true support for policies implies a component designed for that.

A market survey concluded that the best compromise between:

- Policy Engine
- Cost to our daemons or machines

is the widely used CLIPS product (see [7]). It integrates easily in a C program, and can be tested on the fly using manual commands, and it is lightweight.

So basic policies and rules has been written for Garbage Collection, Recallers and Migrators.

In particular the Migrator rules are very important: we use them to decide what is the best file to migrate to tertiary storage (tapes, for instance) at any time.

Security Layer

We use *strong authentication*, and do not plan to use encryption once the connection is established. This is because encryption would seriously hamper the data movement performance.

We have developed a plugin system, based on the GSSAPI supporting the following mechanisms:

- GSI
- Kerberos V
- Kerberos IV is added by hand for backward compatibility with current CERN architecture, but is likely to disappear in the near future
- This has an impact on the machines *configuration*: need for service keys, in particular.

Distributed Logging Facility

A clear disadvantage of a distributed system is when we want to synchronize information to do a request survey over the whole system.

Alternatives based on a concept similar to the netlogger [8] product are promising, and we wrote our own component, capable of storing requests in an RDBMS, with predefined members like a Universal Unique Id.

Using this new logging facility, any process has the possibility to continue local logging as usual, and at the *same time* to send the log to a service that will store it to the database.

Expert System

In order to centralize all policies and rules, and because CLIPS is not thread-safe, we wrote a lightweight daemon that is executing CLIPS rules and return the output to the client.

This daemon accepts *facts* that are given to CLIPS. These *facts* will also call for the policy engine to run.

Migrator, Recaller

Migrators and recallers are services living on top of the database, flagging files as candidates to tertiary storage access. Those services act on behalf of the users.

In the prototype version, migrators and recallers are explicitely instanciated when needed:

- a user wanting to read a file not yet available will provocate internally an explicit recall of the file onto the filesystem for which it has been scheduled
- a user creating a file will internally cause the file to be flagged as an eligible candidate for migration. Nevertheless, the CLIPS rules for migration will decide the optimal time when the file is to be physically transferred to the tape mover and written to tape.

Garbage Collector

The garbage collector works quite similar to the migrator and the recaller: this is a component that is living on top of the database, getting a list of files candidates for migration.

Neverthless, garbage collection is usally a more sensitive subject because it is related with the lifetime of files on the disk cache. So it has an immediate impact on the user's job performance.

As for the migrator and the recaller, garbage collector will be driven by policies and rules. This is where an experiment administrator will explicitely put its own preferences.

Tape Mover

The production version of the tape mover has been modified to circumvent a known deficiency in older versions of CASTOR;

A *stream*, i.e. an aggregation of files to be read or written to tape, could never be modified as soon as the request was starting on tape servers.

The immediate consequence is that, if another stream was to use the same tape, this was seen as two different processes, not being able to share the same tape device at the same time. The best that could be done was then to not unmount the tape, but a rewind was still needed for technical reasons.

This inefficient way of dealing with tapes has been addressed, and we implemented the possibility to *append* to an existing tape stream a unlimited number of requests.

TESTING

We put together all the ready components and built a prototype of CASTOR, that will be used for the Alice Data Challenge at the end of year 2004.

The diskservers used in the tests were not very well tuned for CASTOR. They consisted of RAID arrays presenting a single XFS filesystem, with h/w level 5 and s/w level 1. The servers were tuned to give very good performance for streams in same direction. However, for competing streams in opposite direction, the read performance dropped substantially while the write performance stayed constant (see figure 3).

An application like CASTOR has a different requirement, in particular it expects one read and write streams to have a similar rate when they are alone on a machine. This has to do with performance for migration to tape (read from disk) together with performance of production (write to disk).

Nevertheless, it was an useful exercise to see how CASTOR would perform with such suboptimal filesystem tuning.

Another tuning of the diskservers is an iteration process between us and our filesystem experts, the results are not yet ready.

Goal

We want to test the full chain, i.e. involving memory, network, disk and tape. So the measurement is the rate when writing to tape.

We use 5 disk servers, where the nominal write speed through RFIO (involving network, memory and disk) has been measured around 25-30 MB/s.

The read-from-disk speed is the same if there is not other competing write stream, otherwise around 5 to 12 MB/s.

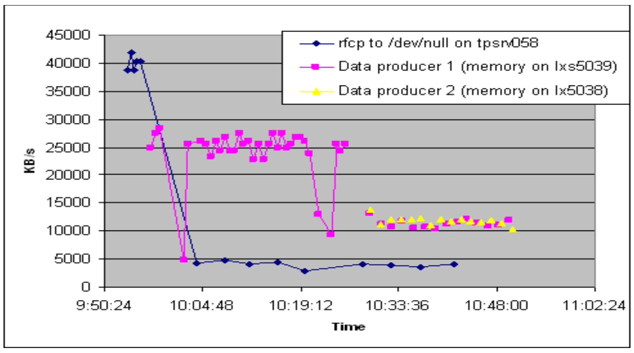

Figure 3

We scheduled six producers, so that the probability to always have one producer per machine was high.

Only time windows introduced by the scheduling decisions could lead to a machine with no producer in the case it is idle.

The produced data was written to tape (i.e. read from disk) using two streams to 9940B drivers, for which the nominal speed is around 25 to 30 MB/s. The network overhead does not matter too much in our case because the CASTOR tape mover handles the network and tape I/O in parallel provided there is enough data in the mover's internal buffers for continue streaming the tape.

To make sure that CASTOR would migrate fine in a non-hostile version of that environment, we made sure that there was no data producers running concurrently at the very beginning and and the end of tape stream.

As shown in the figure 4, when there was no producer the rate when writing on tape was good as expected. The interesting case is in the middle: it was guaranteed that globally read-from-disk performance is bad – neverthless, thanks to the migration policy rule, written in CLIPS, the system always selected the *best* filesystem at any time.

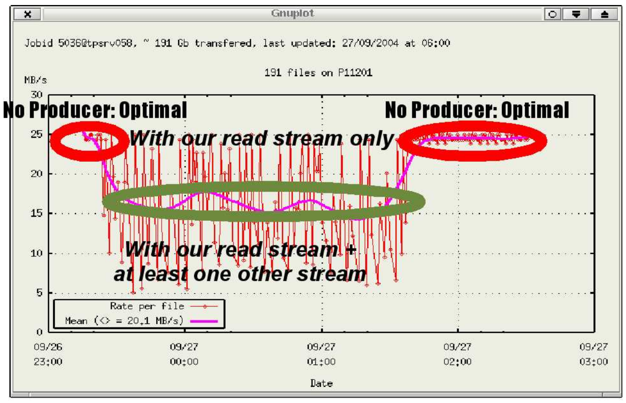

Figure 4

We got into the small time windows when nothing was scheduled. In such a case, the remaining producers not yet scheduled did not go to the machine where we were reading from disk, because the monitoring agent said to the Resource Manager there was activity, and the Resource Manager forwarded this information to the MAUI scheduler, used in this test.

Thanks to the request scheduling and the just-in-time selection of the best files to migrate, the system managed to perform better than what one would expect from the measured filesystem performance.

Next step is of course to re-run the test in a completely different environment, hopefully more in favour of CASTOR.

MANAGEMENT

The CERN Operational Team took over the totality of CASTOR service management since end of year 2003.

This triggered the adoption of the Lemon and Quattor frameworks for monitoring and configuration the castor services.

In *Lemon*, monitoring metrics are added to handle our s/w survey, together with specific h/w survey (tape drives for example).

Quattor is used to maintain the s/w in synchronization across the machines, maintaining and distributing consistent configuration information from central definitions.

CONCLUSION

A hybrid (with some old pieces left) stager prototype has been setted up for the Alice Mock Data Challenge. Despite of a suboptimal filesystem tuning the first results with this prototype are very promising.

We expect the meet the goal of 450 MB/s aggregrate rate.

The final new stager system has its design ready and the implementation of the remaining components of the central framework has started.

REFERENCES

[1] http://cern.ch/castor
[2] http://cern.ch/quattor
[3] http://cern.ch/lemon
[4] http://www.uml.org
[5] http://www.platform.com/products/LSF
[6] http://supercluster.org/maui
[7] http://www.ghg.net/clips/CLIPS.html
[8] http://www-didc.lbl.gov/NetLogger/