Architecture Overview

First breakdown into the system

Giuseppe Lo Presti, German Cancio, Sebastien Ponce
CERN / IT

Castor Readiness Review – June 2006

Outline

- General picture
  - A first breakdown
  - Architectural features
  - Client deliverables
- Overview of the different parts
  - Old and new components
  - Basics of the new ones

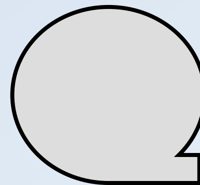
Context View

User

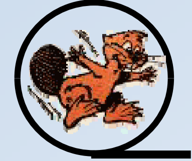
Experiment Framework	Grid enabled applications

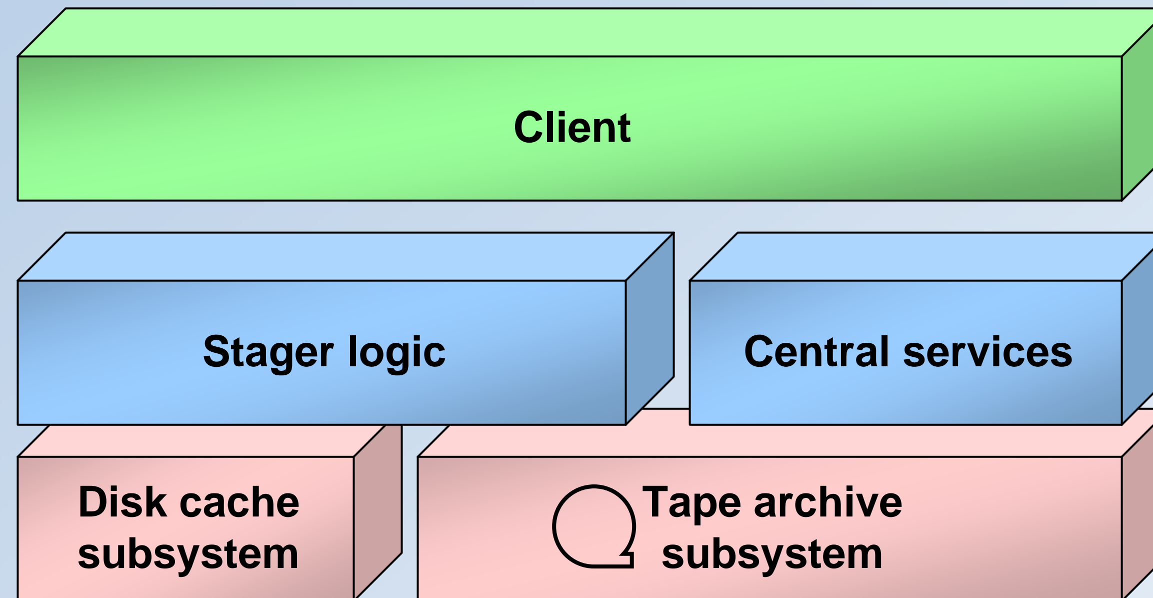
Storage Interface (SRM)

CASTOR



General Picture of Castor



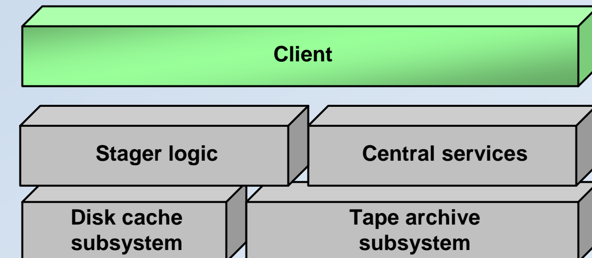


- Shared with Castor 1
  - Central services (e.g. NameServer), tape part
- New in Castor 2
  - Client API, Stager, disk management part

General Picture of Castor

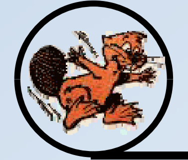
- Key requirements
  - Fault tolerance
  - Scalability
  - Transaction-like behavior
- Design features
  - Distributed system
  - Stateless replicated daemons
  - Database-centric architecture
    - State information stored in a RDBMS
    - Rely on the DBMS performances in terms of fault tolerance

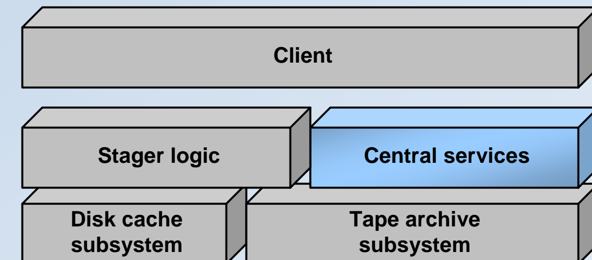
Client deliverables



- Command line interface for end users
  - stager commands (`stager_xxx`)
  - RFIO commands (`rfxxx`)
- Client API
  - only C
  - internal API in C++
- Supported Protocols
  - Integrated Protocols
    - RFIO: rfio://server:port//castor/cern.ch/...
    - ROOT: root://server:port//castor/cern.ch/...
    - XROOT soon to come
  - External Protocols
    - GridFTP: gsiftp://server:port//local/mnt/point/...
- SRM v1 and v2 interface

Central Services

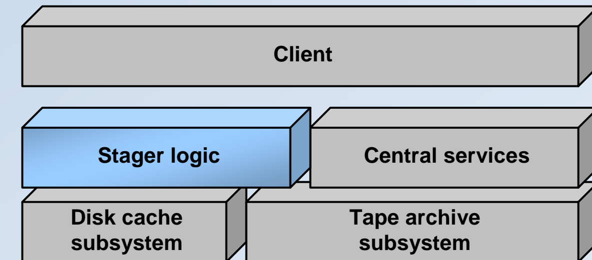




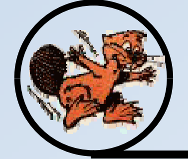
- NameServer
  - Database for the Castor “FileSystem”
  - Stores tape-related info as well
- Volume and Drive Queue Manager (VDQM)
  - Daemon for tape queue management
- Volume Manager (VMGR)
  - Archive of all tapes available in the libraries
- Castor User Privileges (CUPV)
  - Authorization daemon: provides rights to users and admins for tape related operations

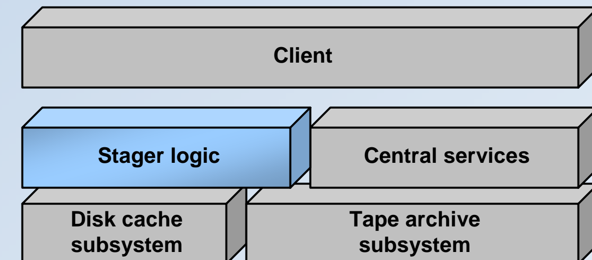
Stager Logic

- Design principles
  - Decisions taken:
    - at DB level (stored procedures), or
    - by external plugins (scheduler, expert system)
  - Typical decisions
    - Preparation of ***migration*** or ***recall*** streams
    - Weighting of file systems used for migration/recall
    - Garbage collection decisions
  - Actions performed by dedicated daemons



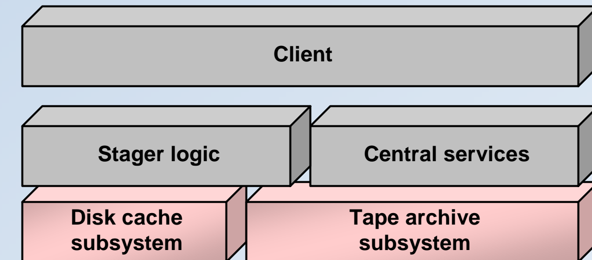
Stager Architecture





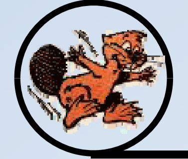
- Database centric architecture
  - “Surrounding” daemons are stateless
  - Well defined database interfaces separated from the rest of the code
  - **Oracle** fully supported, **PostgreSQL/MySQL** partially implemented
- Stateless components
  - can be restarted/parallelized easily $\Rightarrow$ no single point of failure
  - Stager split in many independent services
    - distinction between queries, user requests and admin requests
    - fully scalable
- Minimal footprint of inactive requests
  - Requests are not instantiated in terms of processes until they run
    - Stored in DB and/or scheduler while waiting for resources
    - Number of *migrator / recaller* instances ~ nb drives (no process instances while waiting for drive)

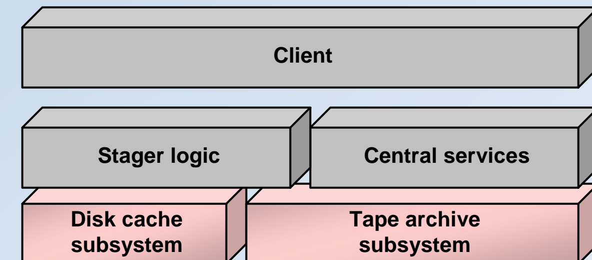
Disk Cache and Tape Archive



- Scheduled disk access
  - All user requests are scheduled
  - Advanced scheduling features for ‘free’ (e.g. fair-share)
  - 2 schedulers supported
    - **LSF**, expensive, fully supported and recommended for Tier1 size setups
    - **Maui**, open-source, development frozen in Mar 2005
- Dynamic *migration / recall* streams to / from tape
  - Multiple concurrent requests for same volume will be processed together
  - New requests arriving after the stream has started are automatically added to the stream

Disk Cache and Tape Archive





- Pluggable policies
  - For recall, migration, I/O scheduling, GC
  - Defined per disk pool but centrally written
  - Allows support for
    - volatile storage (GC, no migration)
    - durable storage (no GC, no migration)
    - permanent storage (GC, migration)
- "Pluggable" protocols
  - RFIO and ROOT internal, XROOT coming, GridFTP external
  - "Easy" addition of new protocols
    - but no clean interface

Security and safety aspects

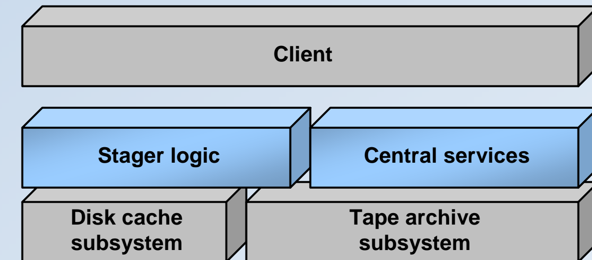
- Authorization
  - per file ACLs at the namespace level
  - restricted access to diskservers (rootd, rfio, xrootd)
- Authentication
  - strong authentication under work
    - running currently at prototype level
- Resiliency against hardware failures
  - any node can die with no major impact
    - relying on the DB for data, all deamons can be replicated
- Disaster recovery
  - regular backups of the DBs

Outline

- General picture
  - A first breakdown
  - Architectural features
  - Client deliverables
- Overview of the different parts
  - Old and new components
  - Basics of the new ones

Old and new components

- Castor 2 has its roots in mid 1990s
  - Several components have been left almost unchanged
    - E.g. the Central Services
  - New components have been written from scratch
  - Happy mix of old and new code
    - Different styles due to different developers and (mostly) different languages
    - Most C++ code is interfaced in C
    - *more tomorrow*

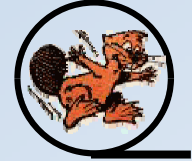


- …not a surprise taking into account time extension of the project and number of involved people!

Overview of the new parts

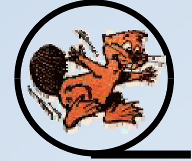
- Among the new components…
  - Stager, Request Handler, core framework and db access have been written from scratch for Castor 2
- A complex architecture
  - Several constraints and requirements
- Need for a proper software design process
  - Use of UML
- Need for a db access layer
- Need for external plugins
  - Dedicated tasks (e.g. scheduling)

Software design process



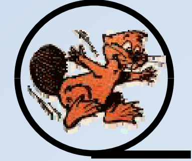
- Use of UML
  - *Sequence and activity* diagrams to describe the workflow and the information flow
  - *State* diagrams to describe the status lifecycle and evolution of all the stateful entities
    - e.g. a request, a disk copy, a tape copy…
  - *Class* diagrams to design in details the software components (for the C++ part)
  - *Class* diagrams to *also* describe the database model
    - special case of an E-R model mapped to a set of classes

DB access layer



- DB centric architecture
  - All entities are stored in a DB with their status
- Need for a DB access abstraction layer
  - Generic enough to homogeneously handle all needed entities, including future ones
  - A very special case of reflection/introspection "the Java way", but to be implemented in C++!
  - Adopted solution: use of a **code generation facility** from UML class diagrams
    - Autogenerated code provides standard interface to the database and to streaming for **any class** of a given UML Class Diagram
    - *more tomorrow*

Extensibility



- External *plugins* to delegate specific tasks
  - Scheduler
    - I/O job dispatching
  - Expert system
    - Allows external components to take decisions
      - External perl scripts
      - CLIPS Rule-based engine
    - Current implemented policies
      - filesystem selection
      - migration and replication policies
      - recall will come
  - Garbage Collecting
    - Customizable policy in SQL

Configurability

- Different requirements from different deployments
  - E.g. Tier0 vs. Tier1s vs. …
  - Enhance configurability vs. hard-coded logic
- Concept of ***Service Class***
  - Group a set of policies to be applied for a class of users to provide a given service level
    - *more on the Castor operation*
  - Customization by admin scripts
    - Related metadata stored as well in the database
  - *more tomorrow*

Comments, questions?