StoRM 2: architectural overview

Andrea Ceccanti
andrea.ceccanti@cnaf.infn.it

Nov. 2017

Objectives

Make StoRM a really “lightweight” storage manager not bound to a specific management interface

- SRM
- WebDAV
- CDMI
- ?

Reduce maintenance and evolution costs

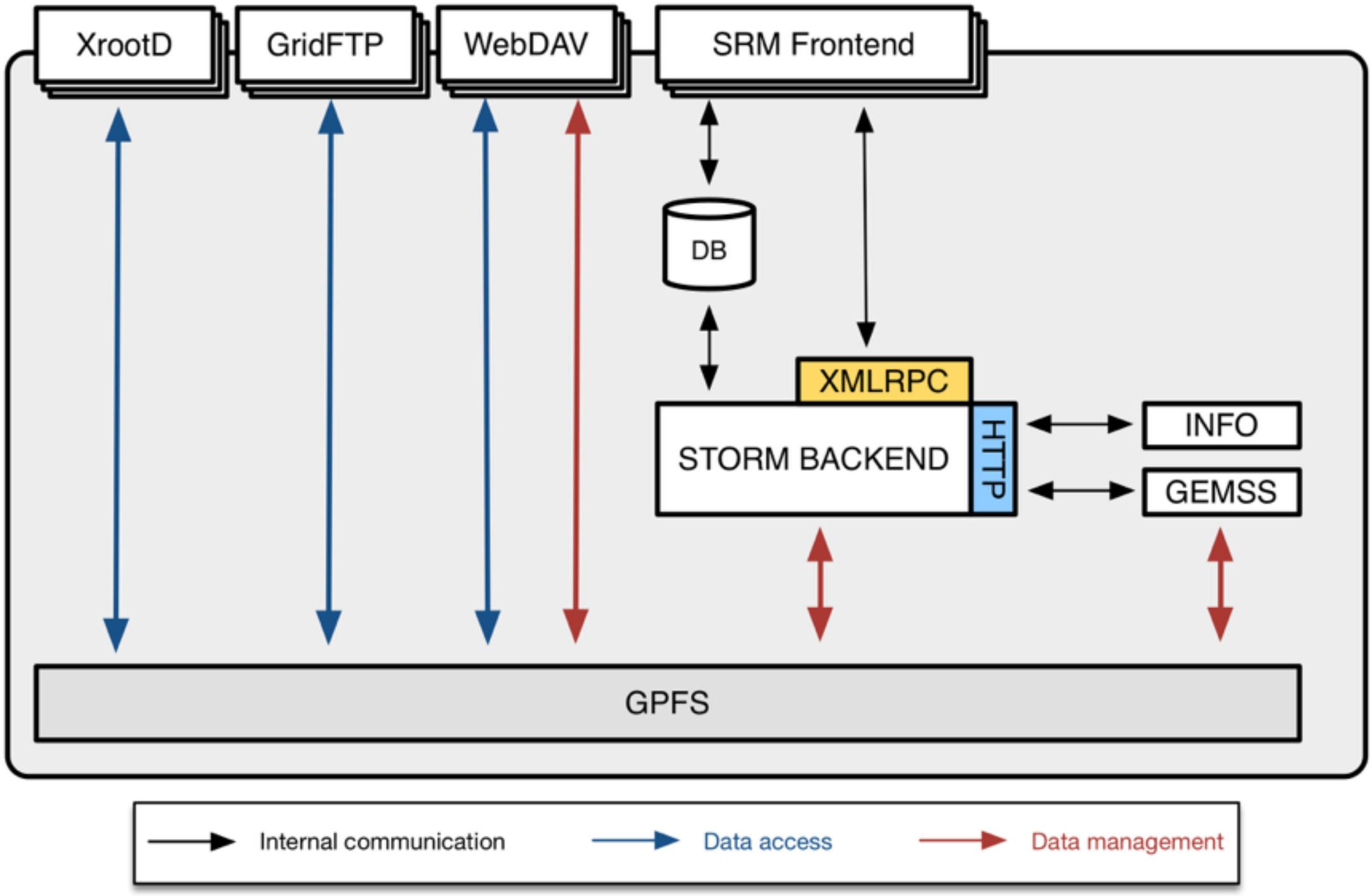
- Current complexity mostly due to unused SRM “features”

Provide horizontal scalability for all StoRM services

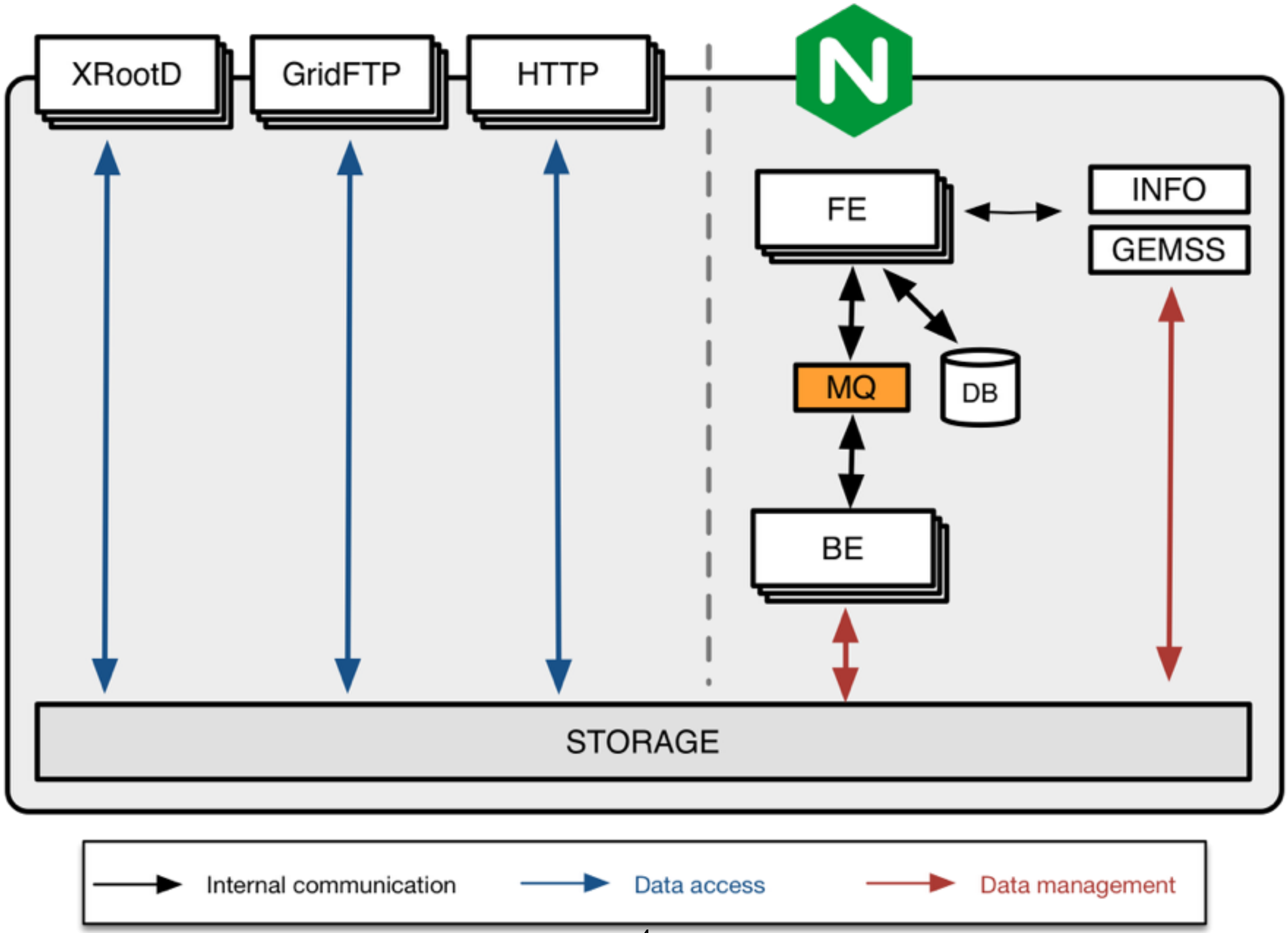
- Currently the StoRM BE cannot be replicated

Simplify service operation and deployment

Current StoRM architecture



Envisioned StoRM architecture (MQ, 2017)



NGINX as TLS terminator and FE load balancer

TLS is handled at the edge:

- decouple TLS load from request management load
- handles VOMS stuff
- keep all TLS/VOMS complications in a single place
  - expiring CRLs etc. etc.

FE load balancing

- NGINX knows how to LB http services very well

Issues

- NGINX VOMS module to handle VOMS credential validation
  - we needed it yesterday
- Support for the infamous GSI delegation “0/D” byte ?
  - This is always 0 in our case (no delegation support on StoRM anyway)
  - Supporting this allows deployment without changing requirements on the clients
- Otherwise, do L4 LB and let FEs do TLS/VOMS stuff (missed opportunity)

NGINX VOMS module

We need to allocate one-two sprints to have a working, reliable solution

- can be based on the current VOMS APIs

GSI delegation support

- I think we can be disruptive on this, and require HTTPS
- but it’s worth checking the ratio of GHTTPS vs HTTPS requests that reach the CNAF production servers, and ask FTS developers (the main SRM client) if it’s a problem to require plain HTTPs

The message queue

RabbitMQ seems a battle-tested and reasonable solution

- Good support in Java and C++

MQ used for all communication among services

Deployment needs to be scalable and reliable

- but likely a single RabbitMQ instance will go a long way in handling the communication patterns between StoRM microservices
- Expertise for a HA deployment already present @ CNAF (rabbitmq is used extensively in cloud@cnaf & bebop monitoring infrastructure)

Issues

- Understand how to best implement messaging, not much experience here (but this is the fun part)

The database

MariaDB seems a reasonable choice

- PostgreSQL would be fine as well

StoRM needs the database to maintain some state

- requests and SURLs status in the case of SRM that would be unmanageable if kept on the FS (and we don't want to be bound to a posix fs as storage backend)

The database will also hold storage area space information and the tape recall table used by GEMSS

Only the FEs talk to the DB

Deployment scalable and reliable

- also here a single instance will go a long way

The new StoRM frontend

Implements management protocols endpoints

- SRM, WebDAV, CDMI, (whatever may become fashionable)

Implements authorization & mapping

- VOMS, OAuth, etc.

Implements validation on requests

- e.g., space availability checks, conflicts situation (a PtG on a SURL that has a PtP ongoing)

It does not interact directly with the storage

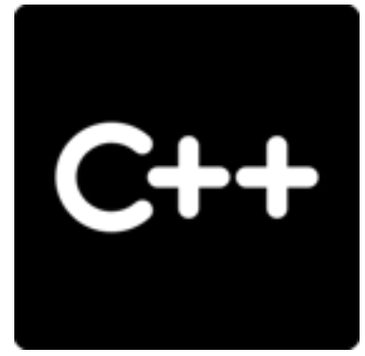
- the BE does that, the FE creates tasks for the BE to execute

Communicates asynchronously with other services via the MQ

- FE & BE can scale independently from each other

Stateless Spring Boot application (Java)

The new StoRM backend



Implements management operations on the storage

- Storage is now a posix FS but could be an object store

Stateless worker that executes tasks fetched from the MQ and reports about the outcome (also on the MQ)

Completely decoupled from the FE

Ideally the only component that directly interacts for management operations with the storage

Implemented in C++

- iff testable and iff Francesco commits reasonable effort on this, otherwise Spring Boot like the FE

Needs good library support for:

- mocking, testing (& coverage), logging, metrics, messaging

Storage management operations

The BE will implement the storage management logic interacting directly with the storage

Storage management operations (SMOs) are orthogonal to the specific storage management protocol used to manage the storage (e.g., SRM or WebDAV)

Examples:

- Data object lifecycle operations (create/remove file, or object in object store)
- Data object metadata operations (touch, get size, get/set ACLs or other authz permissions)

A management protocol operation (e.g., srmPtP) is the composition of several SMOs

The FE builds the list of SMOs to be executed for each request and delegates the execution to the BE

Storage management operations (POSIX FS)

Create/delete/move file & directory

Get/set ownership & ACLs

Get file metadata

- file size, availability (is it online?), modification time etc…

Get directory contents

We have to define the minimum set of SMOs that allows us to support SRM and WebDAV

SRM PtG example

PrepareToGet is used to prepare a read transfer for a set of files available on the SE

The FE will handle a PtG as follows

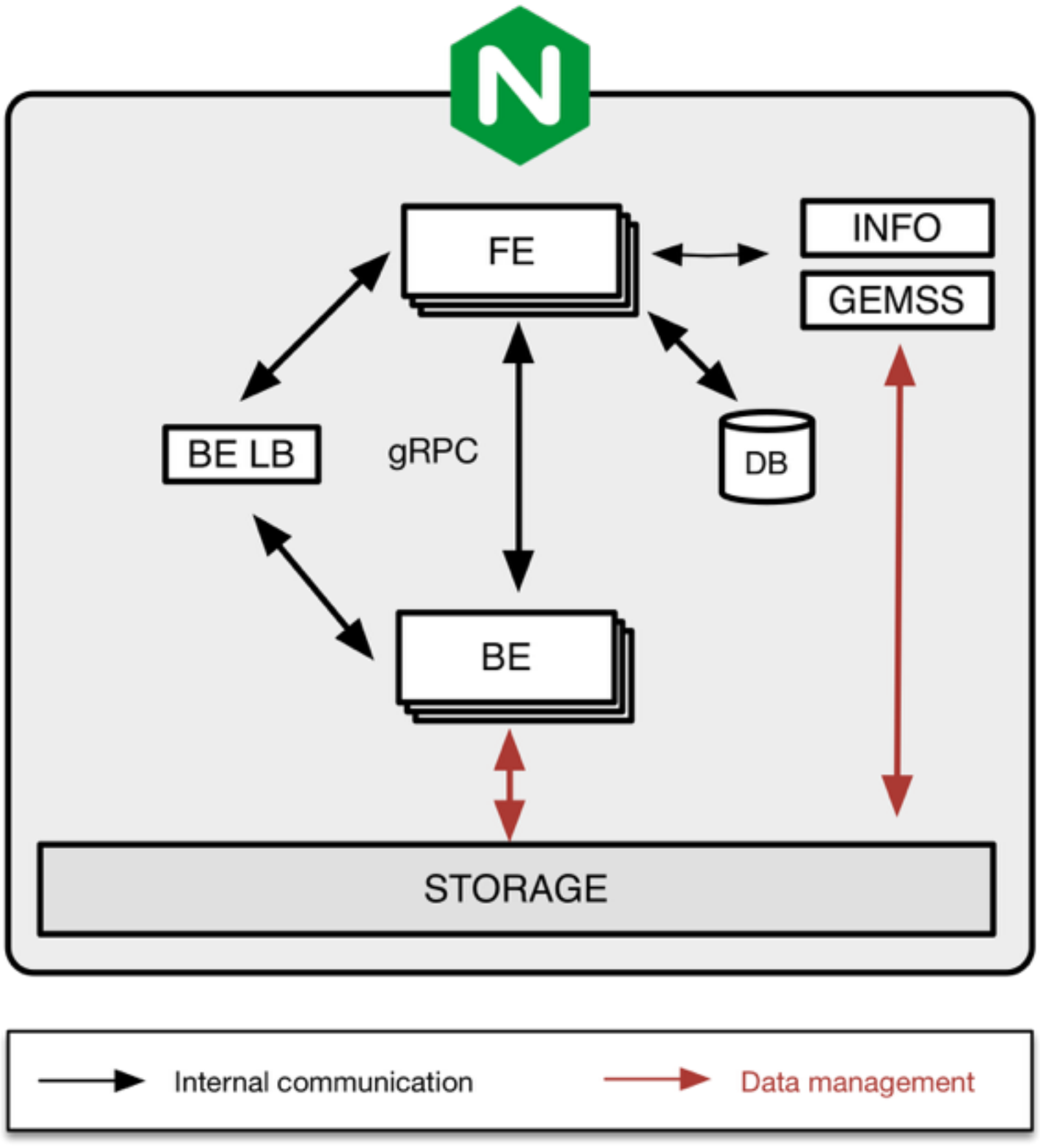
- Create a request uuid; this uuid will be included in all logging and communication related to the request
- PtG validation:
  - do we have that storage area? are parameters meaningful?
- Authorization checks
  - is this VO supported? Can a user with these attributes access the storage area?
- Conflict checks:
  - is a PtG on those SURLs allowed by the system right now? (check if no PtPs are active on the same set of SURLs etc.)
- File checks: do the requested SURLs exist? to answer this question the FE will create a task for a BE to know the answer

Once the checks are over, the PtG handling can begin

SRM PtG example (II)

- Create an SRM request token
- Save the request status and parameters in the database
- Create a sequence of operations, some of which are SMOs, all linked to the request uuid, to actually perform the PtG:
  - compute the unix account mapping, if needed/requested
  - setup ACLs to enable direct filesystem access, if needed/requested
  - set pin lifetime on requested files
  - if there are files that are offline
    - enqueue a recall for those files
  - generate TURLs as requested

StoRM 2 gRPC envisioned architecture



gRPC instead of RabbitMQ message bus

The interaction between FE and BE is a typical remote procedure call

- which can, in most cases, be handled asynchronously

An efficient, popular, highly scalable RPC mechanism in 2017 is Google RPC (gRPC)

- based on HTTP/2 and Google Protocol Buffers
- very well supported in Java and C++

The protocol supports LB among servers, so we can decouple client from servers and let them scale independently

- it can be done by proxying the BE or by doing LB client-side (with the help of a service discovery registry)
- advantages/disadvantages of both approaches described here
  - https://github.com/grpc/grpc/blob/master/doc/load-balancing.md

gRPC approach: pros and cons

I don’t really know how to answer this question, as I have no experience with gRPC, but just picking my mind on it

Pros:

- Probably simpler to code
  - no need to handle the interaction with the MQ, auto-generated stubs handle the communication
- Efficiency
- Simpler deployment
  - no need to have the external MQ, but you will need an LB anyway to scale up/have BE HA

Cons:

- RPC LB: for best performance the advice is to use client-side LB (i.e. no proxying) and rely on a Look-aside Load Balancer; there are existing examples, but apparently not a mature product
  - but we could live with L4 LB, or build our own simple LB

Requirements for StoRM 2

Testing

- >90% coverage on *all* code

Monitoring & Metrics

- services provide /health endpoints to report status information that can be used to monitor the service health (and provide hints to LBs)
- services measure and expose metrics that can be used to track down performance bootlenecks

Draining & graceful shutdown

- services support the concept of graceful shutdown and draining, i.e. provide an endpoint/RPC to request the draining and graceful shutdown

With spring-boot we know how to meet the above requirements

In C++, we will have to learn

Interesting C++ libraries

Testing

- https://github.com/philsquared/Catch
- https://github.com/google/googletest (also provides a mocking library)
- https://github.com/cpputest/cpputest
- https://github.com/eranpeer/FakeIt (mocking library)
- https://github.com/rollbear/trompeloeil (mocking library)

Logging:

- https://github.com/gabime/spdlog

Metrics:

- The libraries I've found are projects with few contributors:
  - https://github.com/ultradns/cppmetrics
  - https://github.com/dln/medida

Development organization

Code repository

- On baltig.infn.it, storm2 group, private
- Single repo for all the code (FE,BE,CLIs,etc)
  - easier building, packaging, versioning
- Git flow branching model “simplified”
  - Already in use for VOMS, StoRM, IAM
- README.md describes content of each directory
- We keep a CHANGELOG.md following these rules

Development organization (II)

Dockerized development/testing environment

- Avoid “compiles/works/tests are green in my box” scenarios
- Compose to ramp up services
- Ideally also IDEs and devel tools could be dockerized

Code of conduct = good developer common sense

- Write tests for everything
- Do not break the build
- Do not push stuff until all tests are green
  - exceptions apply to personal dev branches/repositories

Development organization & cycle

SCRUM-like

- aim at one month sprints
- what is done in each sprint is defined at the start of the sprint
- version increased at each sprint

Pre-Sprint 0

- Converge on an architecture
- Break big development items into stories
- Agreement on the “Definition of done”

Sprint 0

- setup CI, dev environment, repo, issue tracker

Sprint 1-n

- Development!

Issue tracking

I would keep everything in a single place: Gitlab

- Pros: simplicity
- Cons: JIRA is powerful and flexible

StoRM 2 versioning

A single version for all the components

- avoid this version of BE works with this other version of FE etc.
- experience tells us one version is the way to go

Documentation

Gitbook seems a very good option that we know how to use

- The strong requirement we have is on Markdown