

Multi Back-end architecture

A proposal on how to implement a multiple back-end system for load-balancing and high-availability in the StoRM server.

An outline for Multiple Concurrent Backends in StoRM

Author: Riccardo Murri

Version: \$Revision\$

Date: \$Date\$

This document outlines an implementation strategy for having multiple concurrent backends in a StoRM SRM server, thus providing high availability and improved performance. The proposed changes to the currently implemented architecture are very small.

Table of contents

- [Introduction](#)
- [Handling of client requests](#)
 - ◆ [Synchronous requests](#)
 - ◆ [Asynchronous requests](#)

Introduction

The StoRM SRM server is currently implemented by two separate processes: a *front-end server*, that receives client requests, checks some authentication issues, forwards a pre-processed form of the request to the back-end server, and hands responses back to the user client program; and a *backend server*, that actually executes the tasks that require operating on the filesystem.

The frontend server has been adapted from the DPM frontend server; by design, it can run in multiple concurrent instances, that can share a single common backend.

The issue has been raised that the architecture with a single backend might be inefficient and sensitive to failures, as the backend is a single point of failure and work concentrator of the whole StoRM system.

The StoRM architecture may support replication of the backend server with minimal modifications.

Handling of client requests

Communication between the frontend and the backend happens through two distinct channels:

- ◆ for synchronous requests (that require some filesystem operation to be performed before a response is sent back to the client), the frontend issues a request to the backend through XML-RPC messaging;
- ◆ for asynchronous requests (those for which the client will monitor the status by issuing another request), the frontend stores the request data in a certain database table, the backend picks the new request from this table (currently, by polling the DB at defined intervals) and then executes the requests, periodically updating its status on the DB;
- ◆ for status requests (the client is asking the status of a previously issued asynchronous request), the frontend reads the data directly off the DB, with no interaction with the backend server.

Synchronous requests

The backend listens for synchronous requests coming from a frontend server on some fixed TCP port. Connections from the FE to the BE are routed through an LVS director, which forwards the actual connection data to one backend, picked according to some algorithm chosen by the systems administrator among those implemented by LVS (round-robin, least connected, etc.).

This schema inflicts actually no change to the StoRM code base: LVS is an *external* component, which can be deployed at sysadmin's will: if one wishes to use a (1 frontend) – (1 backend) configuration, then no LVS setup is necessary.

Issues to look out

- If the same XML-RPC/HTTP connection is reused for many requests (which should be done for efficiency reasons, to avoid the overhead of setting up a TCP connection), then the FE should automatically and seamlessly reconnect to the BE. This would provide high availability, as the LVS director will forward the new connection to another backend server, if the previous one is not responding anymore, but this can be done only at the TCP connection setup phase, and not *after* a connection has been established.

Asynchronous requests

The FEs write requests to a single DB table (presently called `dpm_pending_req`); as the requests are appended to `dpm_pending_req` by some SQL INSERT statement with server-side generation of the unique row id (a feature which MySQL and other major DB servers currently implement), there is no concurrency issue.

From FE to BE

The BE read new requests off the `dpm_pending_requests` table; if there are multiple BEs, there are two ways to avoid different BEs picking the same request:

- ◆ either every BE locks the `dpm_pending_request` table for reading when polling the DB,
- ◆ or the FE server alerts the BE of a new request via a synchronous call [1]: therefore the actual BE picking the request will be chosen by LVS.

[1] This can be implemented as a special

The first solution features a few shortcomings: no two BEs can read the pending requests table concurrently (can be a scalability problem), and if a BE crashes while reading the pending requests table, it may leave the table locked.

The second solution, while requiring more changes to the existing code base (especially in the FE), has also another advantage: a BE will pick a request as soon as it is written to the DB by the FE, thus having no need to poll the DB in a short loop as it currently happens in the Picker object in the BE.

From BE to FE (status and updates)

The BE presently keeps a request status up-to-date on the records in some `dpm_*_filerreq` DB table. A column is to be added to these tables, for specifying *which* BE is taking care of the requests; the content of this field should be any tag uniquely identifying the BE, for instance, its IP address.

The only production use of this BE tag is for the FE to know, which BE to contact if the client requests early abortion or suspension of a running request.

BE status

If a BE crashes or is anyway taken off-line, other BEs should take in charge its requests.

Therefore, a new table `storm_be_ids` should be created, with two columns: the BE id (see above), and a timestamp. The timestamp is periodically updated by all running BEs; if a BE notices another one is not updating its timestamp, it should take in charge all of its requests (or ask some other alive BE to do so, with a synchronous request).

High-availability and performance of external components

LVS

The LVS system needs a "director" host, which forwards the incoming connections to server hosts in the pool.

The director component may be replicated in a HA setup, by using the heartbeat software: heartbeat and LVS are very well integrated and it's easy to setup both to cooperate (see <http://www.linuxvirtualserver.org/HighAvailability.html>).

The LVS has different setups and modes of operation to accomodate for different performance needs (follow the links on LVS/NAT, LVS/DR and LVS/TUN in <http://www.linuxvirtualserver.org/Documents.html>).

The Data Base server

This actually depends very much on the actual DB server used; notably, Oracle provides clustering solutions for HA and increased performance.

For MySQL, one can duplicate the server and use heartbeat for HA (see, for instance <http://www.karkomaonline.com/article.php?story=2004012416185184>). Performance tuning of the DB server is a task that any DBA is able to carry on successfully ;-).

System Message: INFO/1 (<string>, line 216); [backlink](#)

Duplicate implicit target name: "lvs".

Docutils System Messages

System Message: INFO/1 (<string>, line 217)

External hyperlink target "xml-rpc" is not referenced.

System Message: INFO/1 (<string>, line 218)

External hyperlink target "linux-ha" is not referenced.