XTNETFILE, A FAULT TOLERANT EXTENSION OF ROOT TNETFILE

Alvise Dorigo, Peter Elmer, Fabrizio Furano, Andrew Hanushevsky

Physics Department Galileo Galilei
Padova University and INFN Padova, Via Marzolo, 8 35131 Padova ITALY
(alvise.dorigo@pd.infn.it)
Princeton University, Princeton, NJ 08544 USA
(elmer@slac.stanford.edu)
Università Ca' Foscari Venezia and INFN Padova, I-35131 Padova, ITALY
(fabrizio.furano@pd.infn.it)
SLAC, Stanford University 94025, USA
(abh@slac.stanford.edu)

Abstract

When dealing with the concurrent access from a multitude of clients to petabyte-scale data repositories, high performance, fault tolerance, robustness, and scalability are four very important issues. This paper describes the choices and the work done to address the client side of high demand data access needs of modern physics experiments, such as the BaBar experiment at SLAC, and of any other field in which a reliable data access is a primary issue. For this purpose a highly scalable architecture has been designed and deployed which allows thousands of batch jobs and interactive sessions to effectively access the data repositories with as few fails as possible.

ROOT REMOTE DATA ACCESS

ROOT provides a remote file access mechanism via a TCP/IP-based data server daemon known as `rootd`, and its only purpose is to serve opaque data. `rootd` and the ROOT framework allow an analysis job to get access to local or remote files in a transparent way without any change to the source code.

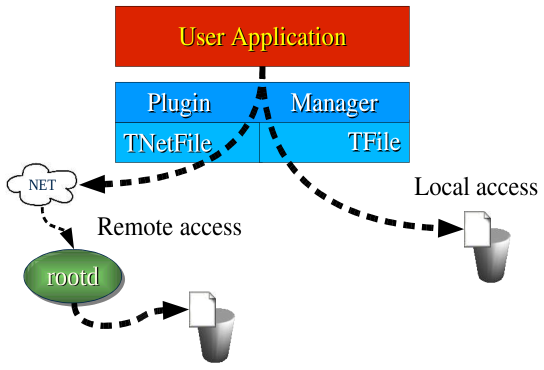

Fig.1 Transparent local and remote file access

In fact thanks to a plugin manager, that recognizes the file URI format, the proper class `TFile` (for local file access) or `TNetFile` (for remote file access) is instantiated and returned to the client as a file handle. Through this file handle the client can read or write slices of data without knowing the actual physical location of the file it isaccessing. Accessing a deployment of interconnected data servers suggests a different paradigm which can be deployed or extended in order to satisfy the heavy requirements of data analysis tasks. At the server side, `rootd` offers the solution to share this big load between many machines keeping the files on their local disks, while at the client side, a specialization of the ROOT's data access classes can provide a way to access the remote data which is transparent to the users of the framework.

PERFORMANCE, SCALABILITY AND FAULT TOLERANCE...

If the `rootd` architecture seems suitable for the purpose, it lacks some functionalities which are crucial for the construction of big processing farms, which must be able to give data processing services to a wide community of users with high availability and performances. Some of these needs are:

- multiple servers have to cooperate with the purpose of:
  - handling huge amounts of data, many times more than the capacity of a single server;
  - making it possible to keep multiple redundant instances of subsets of the data;
- the client applications, given a file to access, should not have to deal with the search of the right server to contact;
- the server has to hide the client applications from its underlying file system types, even if it manages one or more tape units;
- the server has to manage in an efficient way choices about the staging of files from the tape units;
- a load balancing mechanism is needed, in order to efficiently distribute the load between clusters of servers;

- the system resources (sockets, memory, cache, disk accesses, cpu cycles, etc.) have to be used at the best, at both client and server sides;
- a high degree of fault tolerance at the client side is mandatory, to minimize the number of jobs/applications which have to be restarted after a transient or partial server side problem or any kind of network glitch.

The current version of `rootd` doesn't support this set of features; in particular it is not designed to cooperate with other `rootd` daemons; furthermore it uses a forking mechanism to satisfy multiple concurrent requests, consuming lots of system resources. Hence, it is not usable to satisfy loads of 500-1000 clients or more.

The new server designed by the BaBar software specialists to achieve all the requirements listed above has been called **`xrootd`** ("eXtended" `rootd`).

As `TNetFile` is the `rootd`'s client, so `xrootd` needs its specific client that supports an improved communication protocol [1]. This is the reason why, in parallel with the development of the `xrootd` daemon, the `TNetFile` class has been extended, deriving from it the new class `TXNetFile`, formerly `XTNetFile`. Some of the design choices which give the needed functionalities are:

- the communication protocol, which embeds two kinds of redirection mechanism:
  - synchronous: a request can obtain a "redirect" response, which means that the client has to disconnect, connect to another server and continue the processing at the new site;
  - asynchronous: a server can send, as an unsolicited response, the "redirect" command to one or more clients. This is useful if an administrator wants to shutdown a machine, with a gentle way to make all the clients continue their job;
- an architecture in which servers can redirect the clients to other ones, which interact in order to:
  - redirect clients where the files are;
  - balance the load of multiple servers keeping redundant data;
  - give to the clients the possibility of trying to continue their jobs by auto redirecting to another server if the former server crashes or becomes unavailable;
- sophisticated communication policies at the client side, able to handle any kind of communication errors. The failing requests are retried until:
  - another working server is found;
  - the same server becomes available again;
  - a specified maximum number of retries is reached;
- multiplexed persistent connections. This means that a single TCP connection from a client to a server can carry multiple independent data streams. Also, TCP connections are persistent for a short period if connected to a data server, for a long time if connected to a load balancer. This helps in lowering the system resource consumption and the network overheads due to repeated multiple connections to the same host.

RELATED WORK

Much work can be found in the literature about fault tolerant data servers and their performances; it's less easy to find out works dealing with the robustness of the communication and with the purpose of permitting a client application to have such a high level of fault tolerance in a highly loaded and available system.

Many works are available dealing with distributed file systems, but we have to point out that the current feature level for the xrootd/TXNetFile project is not entirely oriented to the distributed file system direction at the moment. In fact, a consistent part of the design work for distributed file systems lies in treating path and filename semantics and cache policies and coherency. These are some reasons why a distributed file system usually causes a consistent network and cpu overhead when dealing with read/write operations on the file it manages [2] [3].

A continuous load of a thousand clients per server is not so usual in the distributed file systems world, but it could be satisfied by using particular configurations. Again, one of the problems which may arise is given by the network overhead due to the synchronization of the internal caches and buffers. This can be a serious issue when dealing with petabyte-scale data repositories and a great number of clients which process the data.

However, such a perspective is common in many the organizations that rely on massive data sharing, not only depending from the ROOT package. For instance, such a robust file server facility could be integrated in the many Grid [4][5][6] initiatives that will support the analysis on next generation physics experiments or other fields.

Scalability is another critical aspect considered in literature. Many existing systems are scalable in some way, typically by passing through a system administration approach (e.g. mounting many remote directories under NFS) or by implementing custom software layers giving abstractions for multiple servers [7], but what can be noted is that very often the scaling measurements are done with numbers of 16 to 50 clients per server. What can be done if we multiply by 100 the number of the connecting clients, and we do not want to force complicated administrative policies and create potential sources of lockups? This is another reason for thinking about an architecture for data access, trying to reach a "nearly linear" scaling performance, limited only by the data throughput and latency of both disks and networks.

This work has many common points with the one described about the Google File System [8]. A difference with the Google File System is that it considers parts of files (chunks) as its data unit, while xrootd has the single file. Also, at this moment, the xrootd system does not have a mechanism to keep the coherence between multiple copies of a file which might be modified by an application, since it's not needed by its current deployment. It seems also that the xrootd/TXNetFile project put a bigger effort in refining the communication policies and the resource consumption.

TXNETFILE – THE CLIENT SIDE

The base of fault tolerance and reliability of the system is built on some important features implemented in the client and defined in the communication protocol. The protocol defines the behavior of the client in the case of explicit redirection requested by the server (for example it can redirect somewhere else because it is going off-line for maintenance) or communication error (a particular data server crashed or unexpectedly closed a connection). In both cases the client has to come back to the load balancer that can redirect it to another available data server.

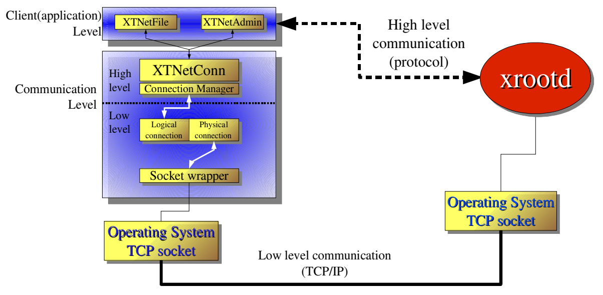
Client(application) Level
XTNetFile
XTNetAdmin
Communication Level
High level
XTNetConn
Connection Manager
Low level
Logical connection
Physical connection
Socket wrapper
Operating System TCP socket
High level communication (protocol)
xrootd
Operating System TCP socket
Low level communication (TCP/IP)

Fig.2 Architecture overview

Even in the case of absence of load balancer, a retry mechanisms ensures that if the communication (or a crashed server) is restored soon, the client is able to recover its operations and the job won't loose any data. Another interesting fault tolerant behavior is when a file that is supposed to be in a particular host has been actually moved somewhere else (by the administrator): again the protocol defines that the client has to go back to the load balancer and send a “refresh” request to it in order to re-locate the moved files and memorize their path through the data servers.

This client is composed by three layers with different tasks:

- the interface layer: here the `TXNetFile` class re-implements all the virtual methods inherited from ROOT's `TNetFile`; another class `TXNetAdmin` has been built at this level in order to perform administration operations like file copy/moving/stat-ing/deletion, file retrieve from tape systems, file system space check, etc.
- the high level communication layer: here the protocol directives are implemented, as well as the load balancing and the policies related to the fault tolerance;
- the low level communication layer: here the protocol packet structure is known, in order to give the functionalities of:
  - ➢ connection multiplexing;
  - ➢ raw data receiving and un-marshaling into an internal message stream, in a parameterizable asynchronous (using a queue and a reader thread per physical connection) or synchronous way;
  - ➢ raw data marshaling and writing to the connections through a socket wrapper layer. It is such a kind of physical layer that performs all socket-related operations like read/write, socket polling to handle read/write timeouts, connection and disconnection, connection timeout detection;
  - ➢ Socket errors handling.

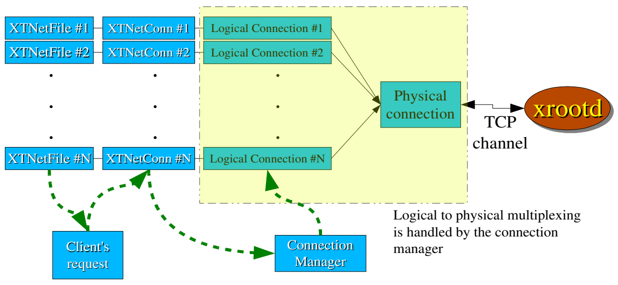

Fig.3 - Connection multiplexing and path of a message

The first important feature of a physical connection is the ability to handle connection timeouts (different than that one implemented in the TCP stack) and also read/write timeouts; if a time limit is reached for one of the connection or communication primitives, an error code is propagated to the higher levels in order to avoid very long hungs of clients in those cases in which a server become unexpectedly unreacheable for ethernet-related troubles or is unable to promptly respond because it's too much busy.

The Low-level communication layer

The low-level communication layer deals with logical connections and physical connections. It is responsible for raw data sending and receiving and is able to detect all the errors directly related to the socket descriptors.

As previously mentioned, multiple logical connections (each used by a single instance of `XTNetFile`) can be mapped to a single physical connection. This kind of connection multiplexing is managed by a connection manager object in this software layer.

The physical connections also handle connect/read/write timeouts; if a time limit is reached for one of the communication primitives, an error code is propagated to the higher levels. This prevents the client to be stuck for long periods if the network has problems or the server is so busy that is unable to promptly respond to the requests it receives.

The low level communication layer can work in two ways: synchronous and asynchronous. In the synchronous way, the physical TCP channel is locked around a full request-response cycle. In the asynchronous scenario each physical connection is bound to a thread which reads messages from the TCP channel and put them into a queue. In this case, when an `XTNetFile` instance wants to send a message to the server, it simply writes it to the TCP channel; when it needs a response to its request, it queries the queue for the next message belonging to its logical connection. This way of working allows the server to send unsolicited responses to the clients. These messages, which don't pair up with any request, when received, are propagated up to the higher levels of the architecture for proper processing.

The high-level communication layer

The high level communication layer has the purpose of handling the content of the messages exchanged with the server it is connected to. It also applies various policies to any communication error reported by the lower level, and take the proper action. For instance, if a client first contacted a load balancer that redirected it to a data server, a communication error (generated, for example, by a socket-related error or a by read/write timeout) is treated as a redirection to the former load balancer; if the client didn't contact any load balancer, but went directly to a data server, it will retry a connection to the same machine for a specified maximum number of attempts.

In any case, when a new working connection can be established, the client can continue the processing transparently, even if this implies a new login or a new file open request at the new host.

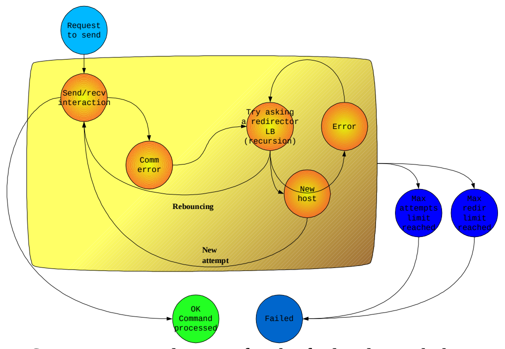

Figure 4 - State transition diagram for the fault tolerant behavior of a client

All these connection and communication policies are handled by an internal communication class, called `XTNetConn`, instantiated by the interfaces of `XTNetFile/XTNetAdmin`. The protocol requests are passed to a unique connection manager that forwards them to the logical/physical connections. To keep care of the fact that many clients can use the same physical TCP channel, the physical channel is locked around writes, otherwise, in the case of a multithreaded application, writes could overlap.

REFERENCES

[1] Andrew Hanushevsky, XRootd Protocol Version 2, 2003, http://www.slac.stanford.edu/~abh/xrootd

[2] John H. Howard et al., Scale and performance in a distributed file system,ACM Transactions on Computer Systems, Feb, 1988

[3] Cary Whitney, Comparing Different File Systems' NFS Performance. A cluster File System and a couple of NAS Servers thrown in,The sixth SCICOMP Meeting, SCICOMP 6 (Univ. of Berkeley), IBM System Scientific Computing User Group, 2002

[4] Official European Grid web page (2004) http://eu-datagrid.web.cern.ch/eu-datagrid/

[5] Official Grid-Egee web page (2004) http://egee-intranet.web.cern.ch/egee-intranet/gateway.html

[6] Ian Foster, Grid Technologies & Applications: Architecture & Achievements, CHEP'01, 2001

[7] F.Garcia, A.Calderon, J.Carretero, J.M.Perez, J.Fernandez, A Parallel and Fault Tolerant File System based on NFS Servers, Euro-PDP'03, IEEE Computer Society, 2003

[8] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, The Google file system, Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM press, 2003