SLAC-PUB-14881

Scalla: Structured Cluster Architecture for Low Latency Access

Andrew Hanushevsky and Daniel L. Wang
SLAC National Accelerator Laboratory
Menlo Park, CA, USA
Email: {abh,daniel.wang}@stanford.edu

***Abstract*—Scalla is a distributed low-latency file access system that incorporates novel techniques that minimize latency and maximize scalability over a large distributed system with a distributed namespace. Scalla's techniques have shown to be effective in nearly a decade of service for the high-energy physics community using commodity hardware and interconnects. We describe the two components used in Scalla that are instrumental in its ability to provide low-latency, fault-tolerant name resolution and load distribution, and enable its use as a high-throughput, low-latency communication layer in the Qserv system, the Large Synoptic Survey Telescope's (LSST's) prototype astronomical query system.**

***Keywords*-distributed storage; scalability; high concurrency**

I. Introduction

Scalla has been the primary distributed file access system for the high-energy physics community for nearly ten years, and has allowed the community to provide uniform access to a large distributed federation of large data sets. Its design accounts for key needs of a distributed file access system for large scientific communities. It must allow access from geographically diverse scientists to data servers which are also geographically distributed among many countries. It must provide efficient access to large files through a namespace with minimal central control. It must tolerate network and server failures and be self-healing so it can be managed without a dedicated operations staff.

This paper begins with a brief architectural overview of Scalla to provide some context. It continues with a deep inspection of Scalla's name cache management and name resolution protocol, describing both their design principles and implementations and explaining how low-latency look-up is achieved in the face of failure and without a persistent central directory. The data structure used for caching is described along with the policies that maintain and manipulate it. The resolution protocol is described with attention to how lookups proceed scalably even in high load conditions. Two examples of Scalla usage are then presented, followed by brief treatments of related work and a conclusion.

II. Overview

A. Motivation

In 2001 the BaBar experiment, a collaboration of 400 physicists from over 9 countries studying the relationship between matter and anti-matter, decided to switch their data analysis framework from Objectivity/DB database system to the Root framework. The new analysis framework relied largely on structured flat files either locally accessible to a compute node or served through a network-based file server. Flat files were seen as a great simplification to the experiment's massive data handling and distribution requirements.

While data handling and distribution would be simplified, the problem area shifted into finding a file server solution that could scale to the petabytes of data the experiment would generate and handle peak loads from a thousand or more simultaneous analysis jobs. The nature of the load was driven by the peculiarities of the framework which would perform several meta-data operations on dozens of files per job prior to commencing analysis. This meant that any new file access system needed to sustain thousands of transactions per second, cluster hundreds of physical data servers just to handle the amount of data, and recover gracefully from failures expected when a massive amount of hardware is deployed.

The three main system requirements: low latency, scaling, and recoverability, all needed to be met simultaneously; otherwise, it was clear that the BaBar collaboration would not be able to perform data analysis in a timely manner. Such an event would doom the experiment and the investment of hundreds of millions of dollars.

A search of systems available in 2001 reveals now, as it did then, that no affordable commercial solutions existed that could meet all three requirements. Hence, the stage was set for the development of Scalla.

B. Architecture

Scalla, Structured Clustering for Low Latency Access, is a network-based file access system consisting of one or more low latency data servers, called xrootd, coupled with cluster management services provided by servers called cmsd[1]. The system is symmetric in that for each xrootd there is a corresponding cmsd. Cluster organization is shown in Figure 1.

1) Cluster topology: In Scalla, nodes (i.e., an xrootd paired with a cmsd) are clustered in sets of 64 and the sets

[1]Scalla can be used as an un-clustered system, in which case no cmsd's need be started.

Contributed to IEEE International Parallel and Distributed Processing
Work supported in part by US Department of Energy contract DE-AC02-76SF00515.

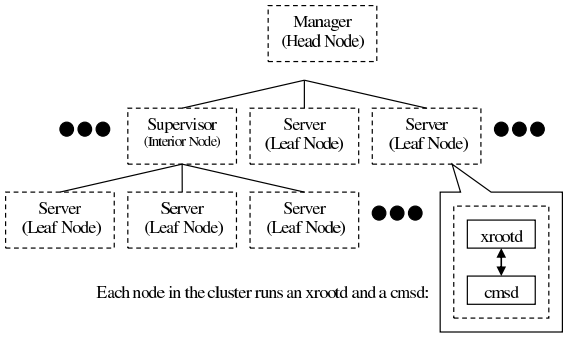

Figure 1: Scalla Cluster Organization

are arranged in a 64-ary tree[2]. As long as linear algorithms are employed, it takes only O(1) time per set or tree node to locate a file. It follows that the upper time limit is in any sized cluster is O($\log_{64}$(number of servers)), which is an exceptionally good value. Every node in the cluster can be replicated to provide an arbitrary level of reliability.

2) Name resolution: Clients first contact the logical head node (which can be one of many) with a request for a file. On the first access to the file, the head node queries its immediate subordinate nodes and asks if they have the file. If a subordinate node has nodes attached to it (i.e., is a supervisor), it asks it its subordinate nodes in turn. The process continues until all leaf nodes (i.e., servers) have been asked. Only those nodes that have the file respond indicating whether the file is online or being prepared to be online (e.g., staging from a Mass Storage System). Multiple responses that are sent to a supervisor are compressed into a single response indicating that the supervisor has the file. Responses are cached by supervisor and manager nodes. Subsequent requests for the file use cached information.

3) Redirection: Once the manager discovers the subordinate node holding the file, it redirects the client to that node. The client then re-issues the request. If the node is a supervisor, the client is also redirected to one of the supervisor's nodes. The process continues until the client reaches a leaf node (i.e., server). If more that one node has the file, a selection is made based on configuration defined criteria (e.g., load, selection frequency, space, etc.).

4) POSIX-like semantics: As a file access system, Scalla provides most, but not all, POSIX file system semantics. Semantics that conflict with the goal of low latency are not natively present (e.g., an ls-type function across all nodes in a cluster). However, full POSIX semantics can be implemented in higher level functions[3] should they be needed.

[2]The choice of cluster size is crucial. [1]

[3]An implementation using native Scalla features exists and is implemented with a Cluster Name Space daemon and the Linux FUSE file system.

Clustering provides clients with a uniform view of a POSIX-like namespace regardless of the number or location of the data servers. The namespace is not exactly POSIX conforming at the manager and supervisor levels since file paths are treated as simple prefixes to a file name; essentially providing a flat namespace. At a data server level, the namespace conforms to full POSIX semantics since each data server uses the host's native file system to implement the data store. The difference in the treatment of namespaces is done for simplicity and generally yields better performance when managing location attributes.

5) Name caching: Recall that file locations are cached by managers and supervisors. It is the cache design that is largely responsible for very low client redirection latency. In fact, requests for files whose information has been cached require less that 50us per tree level. Requests for unknown files incur an additional latency equal to the time it takes a leaf node to respond; increasing the redirection time to about 150us, depending on the network speed. Of course, as more simultaneous requests need to be processed, the average redirection time increases as well. However, the cache uses linear and constant-time algorithms, so the redirection time rises with a very low linear slope as load increases.

III. Techniques for low-latency

This section describes the cache implementation used by Scalla's cmsd and how its algorithms are responsible for the system's low latency.

A. Caching structure

1) The file location cache: Each file is associated with a location object that holds the file's location state. The location state is described by three 64-bit vectors: V_h, V_p, and V_q. Vector V_h describes the set of servers that have the associated file. Vector V_p describes the set of servers that are making the file ready (e.g., staging it from a Mass Storage System). Vector V_q describes the set of servers that need to be queried about the file. Bits in V_q are never present in V_h or V_p.

Each bit in the vector corresponds to a particular server. This is accomplished by assigning each server a number from 0 to 63. Server$_i$ then corresponds to bit (1<<i) in each vector. While this clearly imposes an upper limit of 64 directly-addressable servers, it follows directly from Scalla's cluster organization. Furthermore, the limit allows placing a deterministic upper bound on the amount of time it takes to locate a file in any sized cluster.

Location objects are cached in memory and are accessible by a one-level hash table using linear chaining to resolve collisions. This is illustrated in Figure 2.

The hash key is a CRC32 encoding of the file name. The table itself is sized to be a Fibonacci number of entries. When the number of entries reaches 80% of the table size, a new table is created whose size is the subsequent Fibonacci

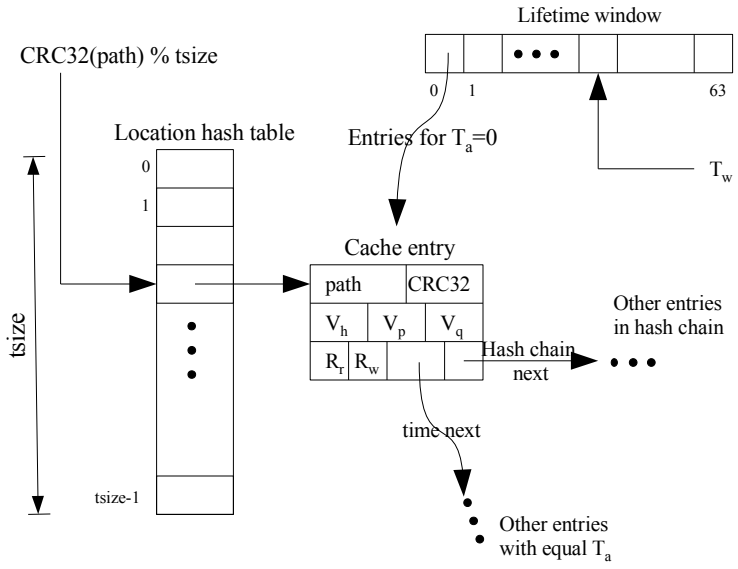

Figure 2: File location hash table and eviction window

number and all of the keys are redistributed. The combination of a CRC32 number modulo a Fibonacci number produces a very uniform dispersion of file names with few collisions[4]. Since the table size grows at a geometrically, the resizing rate decreases as the number of entries increase. In practice, look-up time is constant and resizing ceases in a relatively short time.

2) Global cache object lifetime: Table growth eventually ceases because the maximum number of entries in the table is bounded by an equilibrium reached between the object creation rate and the object lifetime. Each location object has a fixed lifetime, L_t that is configurable but usually set to eight hours. From a practical stand-point, the server has a maximum number of location objects it can create per second, largely determined by its CPU clock rate and network speed. Current state-of-the-art systems, connected via a 1Gb interface, can create about 1,000 location objects per second. Thus, no more than 28,800,000 location objects can exist in the cache over an eight hour period[5]. This also provides an upper bound on memory usage as 28,800,000 location objects represent approximately 16GB of RAM. Since the location object creation rate is far less in practice (e.g., 50-100/second), the memory utilization normally stays well below 1GB.

3) Time-based eviction policy: Location object lifetime is enforced by using a sliding window algorithm that operates in the background. L_t is divided by 64 and a thread ticks an internal window clock, T_w, at a $L_t/64$ rate (e.g., 7.5 minutes). When a location object is added to the cache, it is assigned an add time, T_a, equal to the current T_w modulo 64. All location objects with the same T_a are chained together. This allows the system to find all entries added in a particular window in linear time. Every time T_w ticks, the system hides all entries whose T_a equals[6] the new T_w modulo 64 and schedules a background job to physically remove those entries from the cache. The hiding process is trivial as it simply requires that the text key length in each location object matching T_a to be set to zero preventing the location object from being found in the hash table. As physical removal is a background task, it has minimal interference with cache look-ups. Thus, the cost of cache maintenance is equally spread across L_t and overhead scales linearly with the number of entries; on average only 1.6% of the cache is processed at any one time.

4) Maintaining cache accuracy: The location information is approximate, in that once recorded it is not corrected when the external configuration changes. This is done for scalability reasons. Since millions of files may exist in the cache, keeping the cached information accurate in real time is impractical. Thus, the information is only corrected when it is fetched by applying two correction vectors whose accuracy is maintained in real-time using an O(1) algorithm.

To understand the nuances of location accuracy, several possible occurrences must be considered after location information is cached:

[4]Despite the uniform distribution of CRC32, we found much higher collision rates with power-of-two sized tables compared to Fibonacci-sized.

[5]Based on 1,000 creates/second.

[6]This is not necessarily every object in the chain, as discussed under cache refreshes.

1) a server disconnects,
2) a server is dropped from the cluster,
3) an un-dropped server reconnects, or
4) a new server connects.

In case 1, the server is simply marked as being offline. The server is still considered part of the cluster, though unreachable. In case 2, cached location information referring to the server involved is invalid. In cases 3 and 4, any cached location information that should have considered the server is incomplete.

The first three cases require some further explanation. In order to minimize server queries, the system does not immediately drop a server from the cluster when it disconnects. The hope is that the server is encountering a transient problem and will soon reconnect. When it does reconnect, all existing location information relative to that server remains valid. However, any information cached since the time the server disconnected is incomplete. Normally, this represents a tiny fraction of all cached information. Should the server not reconnect in a configurable amount of time, it is dropped from the cluster and is treated as a new server upon reconnection. If the server reconnects within the drop time limit but has a new set of exported paths the reconnection is also treated as a new connection.

Correction of cached location information relies on two pieces of information:

- vector V_m, representing currently known servers that can possibly hold a particular file based on the paths they export, and
- vector V_c, representing servers in the cluster that have recently connected.

V_m and V_c are maintained by the server login and drop methods. Login is the time a server declares the paths it exports. Each exported path is associated with a V_m that defines the servers eligible for that path. The appropriate V_m, relative to the incoming path, is looked up prior and passed to the cache look-up method. V_m is used by the method to limit the servers present in V_h, V_p, and V_q. Login is also the time that the sever is added to V_c. When a server is dropped from the cluster, it is removed from each V_m. where it appears.

Location objects cached prior to a new server connection are incomplete; and V_c must be added to V_q when the location object is looked up. Location objects cached after a server connects are necessarily complete because all servers that can possibly serve the file, based on V_m, are queried. Any servers that are currently offline (i.e., the time between disconnect and drop) are added to the location object's V_q by the method fetching the cached location object and are queried on the next look-up.

While conceptually easy, V_c cannot be maintained as a single vector since it must also capture the time the server connected relative to the time a location object was cached.

$$V_q = (V_q|_{i=0..63}(1 << i \;\; if \;\; C[i] > C_n)) \,\&\, V_m \quad (1)$$
$$V_h = V_h \,\&\; V_q \,\&\, V_m \quad (2)$$
$$V_p = V_p \,\&\; V_q \,\&\, V_m \quad (3)$$
$$C_n = N_c \quad (4)$$

Figure 3: Corrections when $C_n \neq N_c$ in fetched location object

In order to accomplish this, an array of 64 counters, C[], is defined in 1-to-1 correspondence with bit positions in V_c. So, C[i] corresponds to bit (1<<i) in V_c. A master counter, N_c, starts at zero. When server j connects, N_c is increased by one and assigned to C[j]. Hence, C[] keeps track of the time each server connected. Whenever a location object is placed in the cache, N_c is recorded in the location object as C_n. When a location object is fetched, the object's C_n is compared with N_c. If they do not equal, V_c is generated from each C[i] where C[i] $> C_n$. Then, C_n is updated to the current value of N_c. The resulting V_c is added to V_q and is used to remove the corresponding bits from the location object's V_h and V_p. The corrections are shown in Figure 3.

The new V_q represents all of the servers that could possibly serve the file but have not yet been queried about the file. The servers that have the file, V_h, as well as the servers that are preparing the file, V_p, are simply the old value less the servers that need to be queried. Finally, C_n is updated to the current N_c so that the next fetch only corrects the location object if the cluster configuration changes again. The algorithm adds O(1) overhead to each look-up and, in practice, is not often applied over the course of a location object's lifetime.

However, an additional optimization is used to further reduce the overhead to practically constant time regardless of the number of location objects in the cache. Here, each time window maintains a private V_c and C_n, called V_{wc} and C_{wn}, respectively. When a location object is fetched and its $C_n \neq N_c$, a check is made whether an applicable V_{wc} has already been generated for the window in which the location object was added. If so, the window's V_{wc} is used. This avoids having to generate V_c on every look-up. If a V_c must be generated, it is saved in the window's V_{wc} along with the fetched location object's C_{wn}. The optimization works because server connections and location object creation have time locality, making the probability high that most location objects within a time window can re-use a previously computed V_c. At the worst, the system suffers a small degradation for one perhaps two L_t/64 time periods (e.g., 7.5 to 15 minutes).

B. Optimized resolution protocol

Recalling from Section II-B2 that names are resolved by flooding requests downward from the manager, Scalla

employs a request-rarely-respond protocol for server queries. That is, when a server is asked whether it has a file it responds only when it actually has the file. A non-response is treated as a negative response. This protocol is provably the most efficient way of maintaining location information in the event that less than half the servers have the file in question [2]. However, the protocol comes with a latency penalty for the client causing the query. Since the method invoking the query does not know when a server might respond, it must delay the client some reasonable amount time to ensure that a response, if any, is received with a very high probability before telling the client that the file does not exist. By default, the delay is set to 5 seconds. This is sufficient but in most cases arguably much too high.

The cache employs a fast response mechanism in order to lower the delay to the minimum time it takes any one server to respond; typically, about 100us, without risking a missed response. Each location object is defined with two response queue indices, R_r and R_w. Index R_r refers to clients waiting for read access to the file associated with the location object while R_w for write access. The response queue is simply an array of 1024 anchors for a list of response objects and the corresponding cache entry. Response objects describe which client needs a response for the location of the file. The response queue is handled by a separate thread that runs asynchronously to cache management.

The response queue is loosely coupled to the cache so that response queue management has no impact on cache look-ups. That is, while a location object refers to a response queue element, that element may be asynchronously removed without any need to correct the reference to it in the cache. Cache methods can trivially check if any existing location object reference to a response queue object is correct at the time the reference needs to be used by checking whether the association is still valid. Hence, cache management and response queue management can independently execute their functions.

1) Resolution steps: The general sequence of events is:

1) A cache entry is looked up. If V_q is not null, a processing deadline of 5 seconds from the current time is set in the location object.
2) If V_h, V_p, and V_q are empty then,
   - if the location object processing deadline has passed, the client is told the file does not exist; otherwise,
   - the cache is asked to add the client to the fast response queue (R_r for read access and R_w for write access) for the location object associated with the file the client wants.
3) If V_h or V_p is not empty the client is directed to one of the online servers represented in V_h or V_p, depending on the type of access required.
4) If V_q is not empty but V_p and V_h are empty or all of the servers are offline, the cache is asked to add the client to the fast response queue (R_r for read access and R_w for write access) for the location object associated with the file the client wants.
5) Each server in V_q is asked whether it has the file in question.
6) The location objects V_q is updated to indicate the set of servers that could not be queried; which usually is null.

In order to provide constant time processing for steps 4 and 6, the cache look-up method returns the reference to the location object and a reference authenticator to the caller in step 1. This allows subsequent cache methods to directly manipulate the location object without additional look-ups. Also, locks need not be maintained across calls to the cache methods. This reduces overhead and increases cache availability. The authenticator allows cache methods to determine whether the reference is still valid. If it is no longer valid, which is rare, a full look-up is performed. References only become invalid when the target location object has been removed from the cache because its lifetime has expired. We say "removed" because once a location object is created it is never deleted though its storage area can be reused for some other location object. This allows references to always point to a valid albeit incorrect location object. The reference authenticator merely verifies that the reference still refers to the same location object for which the reference was generated. This is done by a simple counter in the location object. The counter is increased by one when a location object is removed from the cache. Thus, a reference is valid if its authenticator equals the current counter value in the object it points to. When a reference becomes invalid and the fall-back look-up fails to find a location object for the file, the client requesting the file is asked to retry the operation so that processing can restart from a consistent state.

In step 4 the client's request is added to the fast response queue. If the location object already has a reference to a request queue object and the queue entry is still associated with the location object, the request is added to the existing request. Otherwise, a new request queue entry is obtained and the request added to that entry. If no available entries exist, the client is asked to wait a full time period (i.e., 5 seconds) and retry the operation.

When an entry is added to the fast response queue, the response queue thread is notified that there is an outstanding request. The notification is only performed if the queue was empty implying that the response queue thread is idle.

Once the response queue thread is notified, it starts clocking 133ms time periods. Any request that has been in the queue for longer that 133ms is removed and the cache association is invalidated. Each client request is asked to wait a full time period (i.e., 5 seconds) and then retry the request. Thus, a request is given up to 133ms to be satisfied

before a full wait is imposed. It follows that a request is satisfied if a server responds that it has the requested file within 133ms. Generally, servers respond within 100us so a comfortable margin of safety exists allowing for practically all queries for existing files to be satisfied without imposing a large delay.

When a server responds that it has a requested file, the cache update method is called to indicate in V_h or V_p which server has the file. This process is streamlined in that file names and hash keys are passed along. This eliminates the need to generate the hash key for each response. The cache update method checks whether there is an outstanding client request that needs a response that corresponds to the access mode the server allows to the file (i.e., R_r for read access and R_w for write access). If a response queue reference exists and if it is still associated with the location object, the response objects are moved to the response ready queue with an indication of the server that has the file and the response thread is notified. The reference is then cleared in the location object. The response thread sends the server location to each client waiting for a response and deletes the response object.

2) Mitigating timeout delays: Fast redirection is effective when files already exist on leaf nodes. This is normally not the case when a client tries to create a file. Indeed, the client usually wants to be assured that the file about to be created does not already exist. Because file non-existence is based on no server responding that it has the file, the client is necessarily forced to wait a full time period (i.e., 5 seconds) when creating a file. The same is true when a client tries to access an offline file (i.e., one in a Mass Storage System). In the latter case, the full delay usually represents a small fraction of the time it takes to stage a file; which is typically on the order of minutes.

Scalla mitigates this side-effect with a parallel prepare operation. Here, a client can provide a list of files that will be needed, regardless of access mode, ahead of any individual file request. The list spawns parallel look-ups in the background. While each background look-up suffers a full delay; externally, at most a single full delay is encountered by the client. While this does not address ad hoc requests, it is effective for production type processing of multiple files (e.g., large scale data analysis, bulk transfers, etc.). This is a major trade-off in the design. Scalla is specifically designed to efficiently handle read and update processing modes and bulk file creations. These are the modes most often used by research analysis frameworks.

C. Other techniques

1) Cache Refresh Processing: Occasionally, a cached location object must be refreshed. The refresh is driven by a client request and is triggered when the client is vectored to a server that, in fact, cannot serve the requested file. Such an event can be caused by an I/O error, a Mass Storage System failure, or by timing edge effects. The last possibility is an anticipated after-effect of the Scalla model. In order to provide low latency, synchronization points are kept at a minimum. For instance, a file may be deleted by one client at the same time another client is requesting access to that file. In such a case, the requesting client may be directed to a server that no longer has the file. From the requesting client's stand-point erroneous location information was provided.

The general client recovery mechanism from failing access situations is to reissue the request asking for a cache refresh along with the name of the host that failed to provide access to the file. When such a request is received, the location object is refreshed by asking all relevant servers whether they have the file and avoiding the failing server when vectoring the client request.

A location object refresh is logically treated as a new un-cached request. However, the overhead of placing the location object in the cache is eliminated. Since fresh location information is obtained, the location object's T_a is updated to the current time (i.e., window). A significant optimization is that even though T_a is updated, the location object is not placed in the corresponding window chain of objects as this would require too much processing time. Instead, the task is left to a future thread that will be spawned to delete expired location objects in the window chain where the refreshed location object currently resides.

Deferring re-chaining is a significant optimization strategy. Since the deletion thread processes the complete window chain, it can trivially recognize location objects that no longer belong in the window about to be deleted. It is equally trivial to place such objects in the correct chain at this point. By deferring the re-chaining operation, a single linear-cost task can re-chain all objects whose T_a has changed, where re-chaining each object individually results in a more quadratic cost.

2) Deadline-based synchronization: When a location object is newly cached or when it is refreshed, a processing deadline equal to the current time plus 5 seconds[7] is set for the object. The purpose of the deadline is to provide processing synchronization for the location object. It is quite possible that two or more clients cause the same location object to be fetched. Should the location object have a non-null V_q, the servers in V_q are queried. Only one thread should issue the queries. The deadline effectively prohibits multiple threads from issuing queries regardless of the state of V_q. An active deadline implies that some thread is in the process of issuing queries. Thus, threads that fetch location objects whose processing deadline has not passed simply defer[8] the client past the deadline, should V_h and V_p be null, to provide enough time for accurate location information to be collected by some other thread.

[7]The time is configurable.

[8]Clients are deferred using the fast response queue.

Deadlines greatly simplify query synchronization. No additional locks or queues are required.

IV. Applications

A. Production usage

Scalla is available as a packaged system under a BSD license and has been widely deployed, either in total or as a critical component, across the High Energy Physics and Astrophysics community. We describe a few notable deployments.

The ALICE LHC [3] experiment uses Scalla to provide world-wide file access by clustering storage over 60 sites in 20 countries. The US Atlas [4] and CMS LHC [5] experiments are using Scalla to create regional clustered data repositories consisting of dozens of sites to make petabytes of data available for on-demand copying as well as real-time WAN file access.

The Fermi-GLAST [6] astrophysics experiment is using Scalla as a key component to perform timely data analysis and data reconstruction at SLAC and simulation at IN2P3 of data down-linked from its gamma-ray satellite observatory.

The Star experiment [7] at Brookhaven National Laboratory is using Scalla to augment data storage by clustering over 600 batch server nodes and making their storage uniformly available to all batch jobs.

The widely used Parallel Root Facility (PROOF) [8], a Hadoop-like system for the Root framework, uses Scalla as a fundamental part of its data access infrastructure.

B. Distributed dispatch

Scalla is used as a distributed communications layer for the Large Synoptic Survey Telescope's (LSST's) [9] prototype query access system, Qserv. LSST's astronomical catalog, in its final data release, will contain records of billions of celestial bodies in trillions of observations, supporting both quick retrieval (retrieve all facts for a single object) and longer analysis (pair-wise combinations, summaries over all records) efficiently. Such scale both in raw data size and computation was not supported by any off-the-shelf software at an affordable price, so Qserv was built [10].

Limited resources were available for building Qserv, so its design re-used MySQL and Scalla as building blocks for its shared-nothing parallel architecture. MySQL was used as lower-level query engine, and Scalla provided a means of tracking large numbers of worker nodes, handling when they joined or left, communicating with them, and sending work to them. Gearman [11] was considered as an alternative distributed dispatch system, but was not well-matched–for example, its method for returning results was not scalable to large data sizes. Scalla's maturity in handling of node registration/de-registration, node/network faults, and stability was a key factor in its selection.

A Qserv master needs to communicate with its workers in order to transmit work (queries) and retrieve results. Masters dispatch work to nodes hosting the data of interest, and retrieve results similarly. Since Scalla, at its heart, maintains a filesystem abstraction, Qserv masters communicate with workers by opening, reading, writing, and closing files in Scalla. Workers (Scalla servers) in a Qserv Scalla system report their data availability by "publishing" or "exporting" paths that include a partition number. When a master opens a path for a particular partition number, Scalla guarantees that it has a communications channel to a worker hosting that particular partition. In this way, Qserv leverages Scalla's mapping between data and host, enforcing a strong separation between master and worker that simplifies fault-tolerance, replication, and load balancing. Indeed, in Qserv's current implementation, there is no configuration for the number of nodes in the cluster.

More information about Qserv may be found in [10] or at http://dev.lsstcorp.org/trac/wiki/dbQservOverview.

V. Related work

Scalla is one of many distributed file access systems that have stood the test of time. However, it differs from most others by its choice to track only file paths requested by clients. In the Andrew file system (AFS) [12], Vice servers must each maintain a consistent replica of the volume location database, which must maintain locations for all volumes (regardless of actual use). Changes are expected to be infrequent. Cluster masters in the Google File System (GFS) [13] maintain locations of all files in a cluster regardless of use. For Scalla, this means that node registration and de-registration are extremely light operations[9]. In GFS, node registration is more expensive since the incoming server must transmit its entire manifest to the master. Scalla's design choice means its scalability is weakly dependent on the number of currently popular files (see Section III-A2) but completely independent of the number of files available. Its disadvantage is that obtaining global lists of files is not implemented except through a separate Cluster Name Space Daemon. In practice, this is not problematic for users accessing scientific files since they will determine the files of interest by searching metadata databases that catalog the science data within those files.

Few other systems use distributed lookup to locate names among servers that each host a subset of the namespace. In many cases, especially when updates are infrequent, lookups are more efficiently executed on a single machine. The Domain Name System (DNS) [14] arose out of a need to distribute the administration, distribution, and management of a host-to-address mapping since the previous technique, maintaining and exchanging a single mapping file (`HOSTS.TXT`), could not cope with a growing number of mappings from

[9]Nodes need only identify path prefixes for their hosted data, without guaranteeing that they host all files on those prefixes. Servers in a cluster generally export the same prefixes even though they generally host different files.

distributed entities. Distributed lookup in DNS meant that a large mapping file no longer had to be exchanged and kept consistent. In Scalla, distributed lookup performs a similar function in preventing the need for the bulk exchange of file location mappings. Early in development, it was found that an incoming server's submission of its file list caused long delays (minutes for a single server) that not only slowed lookups through not only data structure updates but heavy network traffic. This was an intolerable delay. By foregoing persistent state and only caching file recently-requested, Scalla clusters of hundreds of nodes can begin serve files within seconds of restarting.

VI. Conclusion

Scalla arguably exceeded its three main design objectives: low latency, scaling, and recoverability. In retrospect, these objectives were met using a simple but effective design.

- Low latency was met by uniformly using linear or constant time algorithms in all high-use paths, avoiding locks whenever possible, and using compact data structures to maximize the memory caching efficiency.
- Scaling was achieved by architecting the system as a 64-ary tree. Nodes can be added easily and as the number of nodes increases, search performance increases at an exponential rate.
- Recoverability is inherent in that no permanent state information is maintained and whatever state information is needed it can be quickly constructed or reconstructed in real time. This allows dynamic changes in a cluster of servers with little impact on over-all performance or usability.

Today, Scalla is being deployed in environments and for uses that were never conceived in 2001. This speaks well for the systems adaptability but the underlying reason is that the system can meet its three fundamental objectives at the same time.

Acknowledgment

This work supported in part by the U.S. Department of Energy under contract number DE-AC02-76SF00515.

References

[1] B. Horling, R. Mailler, and V. Lesser, "A case study of organizational effects in a distributed sensor network," in *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004 (IAT 2004)*, Sept. 2004, pp. 51–57.

[2] F. Furano and A. Hanushevsky, "Managing commitments in a multi agent system using passive bids," *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2005*, pp. 698–701, 2005.

[3] The ALICE Collaboration, K. Aamodt, A. Abrahantes Quintana, R. Achenbach, S. Acounis, D. Adamová, C. Adler, M. Aggarwal, F. Agnese, G. Aglieri Rinella, and et al., "The ALICE experiment at the CERN LHC," *Journal of Instrumentation*, vol. 3, pp. 8002+, Aug. 2008.

[4] G. Azuelos, K. Benslama, D. Costanzo, G. Couture, J. E. Garcia, I. Hinchliffe, N. Kanaya, M. Lechowski, R. Mehdiyev, G. Polesello, E. Ros, and D. Rousseau, "Exploring Little Higgs models with ATLAS at the LHC-," *European Physical Journal C*, vol. 39, pp. 13–24, Feb. 2005.

[5] The CMS Collaboration, S. Chatrchyan, G. Hmayakyan, V. Khachatryan, A. M. Sirunyan, W. Adam, T. Bauer, T. Bergauer, H. Bergauer, M. Dragicevic, and et al., "The CMS experiment at the CERN LHC," *Journal of Instrumentation*, vol. 3, pp. 8004–+, Aug. 2008.

[6] N. Gehrels and P. Michelson, "GLAST: the next-generation high energy gamma-ray astronomy mission," *Astroparticle Physics*, vol. 11, pp. 277–282, Jun. 1999.

[7] M. Aggarwal, S. Badyal, P. Bhaskar, V. Bhatia, S. Chattopadhyay, S. Das, R. Datta, A. Dubey, M. D. Majumdar, M. Ganti, P. Ghosh, A. Gupta, M. Gupta, R. Gupta, I. Kaur, A. Kumar, S. Mahajan, D. Mahapatra, L. Mangotra, D. Mishra, B. Mohanty, S. Nayak, T. Nayak, S. Pal, S. Phatak, B. Potukuchi, R. Raniwala, S. Raniwala, R. Sahoo, A. Sharma, R. Singaraju, G. Sood, M. Trivedi, R. Varma, and Y. Viyogi, "The star photon multiplicity detector," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 499, no. 2-3, pp. 751 – 761, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900202019721

[8] M. Ballintijn, R. Brun, F. Rademakers, and G. Roland, "The PROOF Distributed Parallel Analysis Framework based on ROOT," *ArXiv Physics e-prints*, Jun. 2003.

[9] Z. Ivezic, J. A. Tyson, T. Axelrod, D. Burke, C. F. Claver, K. H. Cook, S. M. Kahn, R. H. Lupton, D. G. Monet, P. A. Pinto, M. A. Strauss, C. W. Stubbs, L. Jones, A. Saha, R. Scranton, C. Smith, and LSST Collaboration, "LSST: From Science Drivers To Reference Design And Anticipated Data Products," in *American Astronomical Society Meeting Abstracts #213*, ser. Bulletin of the American Astronomical Society, vol. 41, Jan. 2009, http://arxiv.org/abs/0805.2366v2.

[10] D. L. Wang, S. M. Monkewitz, K.-T. Lim, and J. Becla, "Qserv: A distributed shared-nothing database for the LSST catalog ," in *Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. Washington, DC, USA: IEEE Computer Society, 2011.

[11] (2011, Sep.) Gearman. [Online]. Available: http://gearman.org

[12] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West, "The itc distributed file system: principles and design," *SIGOPS Oper. Syst. Rev.*, vol. 19, pp. 35–50, December 1985. [Online]. Available: http://doi.acm.org/10.1145/323627.323633

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29–43, October 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945450

[14] P. V. Mockapetris and K. J. Dunlap, "Development of the domain name system," *SIGCOMM Comput. Commun. Rev.*, vol. 25, pp. 112–122, January 1995. [Online]. Available: http://doi.acm.org/10.1145/205447.205459