An Overview of the Sirocco Parallel Storage System

Matthew L. Curry[1](✉), H. Lee Ward[1], Geoff Danielson[2], and Jay Lofstead[1]

[1] Sandia National Laboratories, Albuquerque, NM 87186, USA
{mlcurry,lee,gflofst}@sandia.gov
[2] Hewlett Packard Enterprise, Palo Alto, CA 94304, USA
geoffrey.danielson@hpe.com

Abstract. Sirocco is a massively parallel, high performance storage system that breaks from the classical Zebra-style file system design paradigm. Its architecture is inspired by peer-to-peer and victim-cache architectures, and emphasizes client-to-client coordination, low server-side coupling, and free data movement and placement. By leveraging these ideas, Sirocco natively supports automatic migration between several media types, including RAM, flash, disk, and archival storage.

Sirocco provides advanced storage interfaces, enabling clients to efficiently use key-value storage or block-based storage through a single interface. It also provides several levels of transactional data updates, up to and including ACID-compliant updates across several objects. Further support is provided for concurrency control, enabling greater performance during safe concurrent modification.

By pioneering these and other techniques, Sirocco is well-poised to fulfill a need for a massively scalable, write-optimized storage system. This paper provides an overview of Sirocco's current system design.

Keywords: Parallel file systems · High performance computing · I/O

1 Introduction

Existing parallel file systems, such as Lustre, GPFS, Panasas, and PVFS, all offer storage for very large files with high performance by striping data across devices. Each of these systems have been optimized in different ways, but are at their cores inspired by the Zebra file system [1], which also statically stripes data across servers. For upcoming large scale systems, the explosion of devices (in number and type) presents a challenge to the inherently flat nature of striped organizations.

The aim of the Sirocco project is to completely rethink storage system design, moving away from the current status quo. Instead of offering a rigid striping

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

model with a separate metadata service, Sirocco provides a storage fabric with the assumption that resources are transient and data is fluid. This fundamental rethinking is being done with a nod to backwards compatibility by offering a POSIX client that can work with Sirocco, allowing legacy and next-generation APIs to coexist. The native interface, a low-level I/O API that is designed to be used by both POSIX and non-POSIX clients, is object-based: Containers hold objects, objects hold data forks, and each fork has a key-value store accessed through a 64-bit address space. These four abstract levels offer flexibility to address HPC, cloud, and large scale data analytics needs.

This paper describes the mechanisms and facilities provided by the Sirocco storage system. We begin with a discussion of Sirocco's origin and design principles. We then compare and contrast Sirocco with its inspiration, unstructured peer-to-peer (P2P) file sharing systems. We then discuss reading under free placement. Finally, we describe concurrency control mechanisms in Sirocco.

2 Sirocco's Design Principles

Sirocco's design is focused on fast checkpoints, so the overall philosophy is based on enabling clients to write data into Sirocco quickly. One way to accomplish that is to eschew system-global views of storage. Instead, clients view Sirocco as a federated group of storage servers offering a symmetric API, but different performance and resilience characteristics. Each client is likely to see a different set of servers than other clients, but can discover other servers in the system to improve quality of service. Figure 1 demonstrates how and why data can move from clients through the Sirocco storage system. Based on visibility, durability, and performance, clients can choose the best target for its writes.

The roots of Sirocco originate from the Lightweight File Systems (LWFS) project [2]. LWFS sought to strip down a file system to the bare, required components and allow users to add additional capabilities as needed. This philosophy allows some compute jobs to opt-in to services that are considered a burden to others. The LWFS core consists of an object store with authentication and authorization services only. Other features, such as naming and consistency control, are left to separate services. Sirocco follows this philosophy as well.

To maximize scalability and generality, there are a small number of guiding principles for Sirocco's design.

1. There is no central index that determines where a piece of data must be (or is currently) stored. Clients of the storage system are allowed to place data within any server they can reach. As a consequence, the location of a required piece of data may not be known at the time it is needed.
2. Data will be continually moving within the system to ensure longevity, integrity, and system health. Replicas will be created and destroyed, and servers will eject data into more durable or less burdened stores. Clients will not be notified of these events. Consequently, growing and shrinking a Sirocco store is trivial.

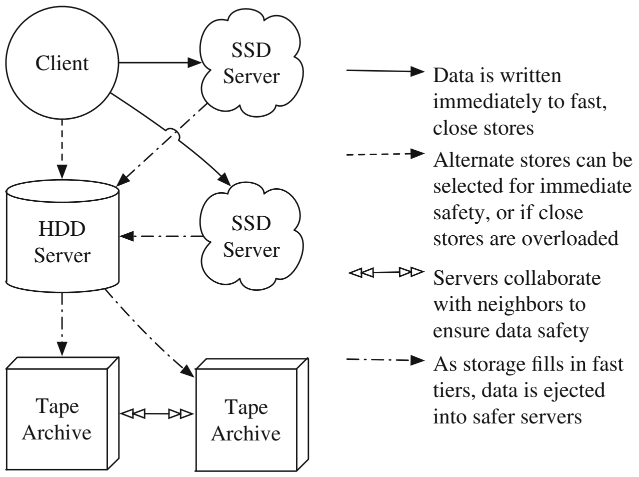
Client
SSD Server
HDD Server
SSD Server
Tape Archive
Tape Archive
Data is written immediately to fast, close stores
Alternate stores can be selected for immediate safety, or if close stores are overloaded
Servers collaborate with neighbors to ensure data safety
As storage fills in fast tiers, data is ejected into safer servers

Fig. 1. Data moves from clients through Sirocco based on load, capacity, and desired safety. Note that these behaviors are based on local decisions; there are no explicit tiers, but tiered behavior follows from safety-motivated victim caching.

3. Sirocco's design emphasizes scalability over legacy concerns. Support for legacy storage system semantics, like POSIX, are required; however, scalability should not be harmed by POSIX considerations. A client implementing more scalable semantics than POSIX should see better scalability on Sirocco.
4. Heterogeneous media (including temporary RAM-based stores, flash-based burst buffers, disk, tape, and others as they become available) should be supported transparently, with symmetric APIs for data access.
5. All data are not created equal, and some need more resilience. Clients should be able to define the level at which Sirocco protects specific data.
6. Server-side operations should be as scalable as possible, particularly when running on faulty hardware. Servers should not couple during any operation.

3 Comparison with Unstructured P2P Systems

The simplest way to illustrate how Sirocco works is by contrasting it with an architecture it was inspired by, unstructured P2P systems. Unstructured P2P systems are well-regarded for their ability to handle web-scale file sharing [3]. While Sirocco is P2P-inspired, it must function in a completely different environment with different requirements. This creates some significant similarities and differences between the two types of system.

Note that this comparison is specifically with unstructured P2P systems. Structured P2P systems, such as those based on distributed hash tables [4], have a different set of constraints that do not conform to all points in the following discussion. Note that this discussion uses the term "servers" instead of "seeds" in the P2P context, to draw a more direct comparison.

3.1 Similarities

Both systems support ephemeral servers (i.e., churn). P2P systems have to support ephemeral servers because servers are not typically under the system's administrative control. Therefore, servers may go offline for any reason (including whim). This capability provides two benefits to Sirocco: Low-level fault tolerance, and elastic allocation of resources (e.g., compute nodes as RAM-based caches) by the storage system based on demand.

Both systems decouple data from location - Any data can exist anywhere, at any time. The reasons each architecture follows this principle are different: P2P servers localize and remove data based on local demand, while Sirocco hosts can migrate or evict data to manage space and resilience.

Both systems use greedy approaches to optimize quality of service. Both systems tend toward this approach for the same reason: Central coordination for performance management does not work at all scales, and may cost more than they gain. For very large scale systems, it can be better to make local, sub-optimal decisions than to coordinate and make a globally optimal decision.

Both systems use popularity to drive copy creation, enhancing performance. Here, the motivation is the same, but the mechanism is different. In P2P systems, popularity automatically drives copy creation, as servers create local copies upon user request. In contrast, Sirocco servers reactively create copies on other nodes in response to high demand from clients.

3.2 Differences

P2P systems publish constant data. Sirocco allows data to be modified. This embodies the distinction between a content addressable store, as most P2P systems implement, and a general-purpose storage system like Sirocco. This implies that different revisions of the same data may exist in multiple locations, requiring some effort to determine which portions of data are current.

P2P systems disseminate data with pulling. Sirocco relies on pushing. A client will store data within a P2P system by simply publishing its presence, allowing other clients download it as they wish. Sirocco enables clients to push data into the system, while also specifying a resilience level for the data. This causes servers within the system to further push data to other stores to attain and maintain resilience.

P2P systems use centralized directories or structured subnetworks to find servers and files [5]**. Sirocco relies on searching.** BitTorrent employs trackers to enable a system to quickly find seeds holding a particular file. Sirocco does not include these types of facilities, as they can harm ultimate write scalability of the system.

P2P systems do not function well with excessive numbers of leeches. Sirocco must support large numbers of leeches (i.e., clients). Significant research in the P2P community focuses on thwarting leeches, also known as free-riders [6]. Leeches do not contribute resources to the system, but instead act in selfish ways, imposing load. In contrast, a parallel file system client does not typically have storage to offer, so nearly any interaction it has with the system is considered to be leeching. As parallel file system clients tend to outnumber servers at a ratio of at least 10-to-1, Sirocco must effectively cope.

P2P systems are not concerned with the lifetime of a file. Sirocco must actively preserve data. P2P systems are not considered archival, and are not designed to preserve unpopular data. Sirocco is intended for data that is to be kept indefinitely, so space management is crucial.

4 Logical Structure of Storage

Sirocco's logical storage organization is based on the Advanced Storage Group (ASG) interface [7], which was partially developed for and motivated by Sirocco. The address space is made up of four 64-bit values. These values denote a container ID, object ID, fork ID, and record ID, which can be expressed as $\langle \text{container}, \text{object}, \text{fork}, \text{record} \rangle$ (Fig. 2). Loosely, container IDs usually map to file systems within the storage system, object IDs map to files, and fork IDs map to data forks within a file. The hierarchy and relationships are static; forks cannot move between objects, for example. A record is a variable length atomic unit of up to 2^{64} bytes, and represents an "atom" of storage. This may be a single byte of a flat file, a floating point number, a text string, etc.

Sirocco reserves forks within the name space for security information. Each container x such that $x \neq 0$ has security information recorded in the KV store located in $\langle 0, 0, 0 \rangle$, record x. Each object y in container x has security information stored in the KV store $\langle x, 0, 0 \rangle$, record y. Each fork z in object y in container

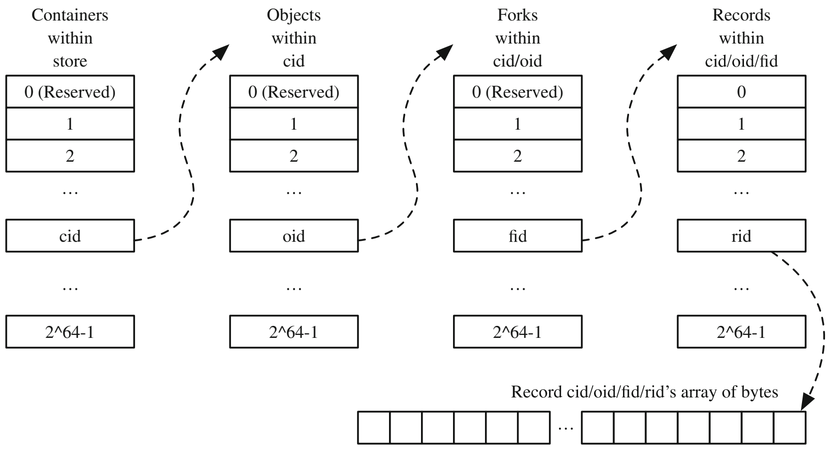

Fig. 2. An illustration of address space organization

x has security information stored in record z of $\langle x, y, 0 \rangle$. Access to the records within a fork is protected by the security attributes of the fork. If not present, attributes are inherited from the object or container.

Each record has an update ID, a 64-bit user-modifiable attribute. It is a logical clock that is expected to increase by at least one after each write, and is used to determine the most recently written instance of a particular record. Specifically, Sirocco will use the update ID to reconcile two instances of a record during migration from one server to another, or when multiple copies of data are located for reading. If the update IDs are different, the largest update ID determines which record is used. The client is expected to know (or determine) an appropriate update ID to use for a record for each write.

4.1 Data Interfaces

Sirocco provides the following types of operations:

Write: Given a data buffer, container ID, extent (i.e., range of records), record length, and update ID, the range will be overwritten on the server transactionally (i.e., all data is written to the record range, or none of it is). Optionally, one can provide an update condition (see Sect. 6) and an update ID to use with it. A user can omit the update ID, and instead use an automatically incremented update ID on that store.

Read: Sirocco supports sparse data within forks, creating a need for obtaining a map that describes the data present. Such a map describes the extents returned. The read operation allows for the user to obtain the map, the data within the requested range, or both. The map will, for all extents present, indicate the size and update ID of each record. Optionally, one can provide an update condition and an update ID to use with it. One can provide special IDs for the fork, object, and/or container ID to obtain a map of an object, a container, or containers, respectively. Another optional flag, the location flag, can specify that the server should invoke location protocols to find the most recently written copies of the record. This can be an expensive operation, but its expense can be mitigated. See Sect. 5 for more information.

To increase network efficiency, it is possible to send several commands to a server at once in a batch. These can be used to implement list I/O [8] or other non-contiguous I/O operations, even across different objects. Sirocco also allows batches to be specified as transactional, enabling fully ACID updates to records. Paired with concurrency control (see Sect. 6), transactional batches can be an extremely powerful construct.

An interesting feature of this API is that there are no "create" or "delete" operations. One way to think about this is to reason that all objects exist at all times, they just do not have any data within them. Likewise, a deletion of data is logically punching the extent, which is procedurally accomplished by writing zero-length records over a range.

5 Where Is the Data?

One side effect of the free placement enabled by the Sirocco model is that data can be initially written to a location that is not ultimately considered safe enough to hold the data. This is a typical behavior in a write-back caching model. For instance, in a local POSIX file system, bulk I/O contents can be temporarily held in memory. The user is only guaranteed that I/O requests will reach full durability if she calls "sync." Sirocco enables a similar technique. However, the location of data written in this way is not predictable, and makes reads challenging. We do not consider read-heavy workloads a solved problem. However, Sirocco provides two facilities to make reads possible.

The first facility enables reading data that the client did not predict it would need, i.e. a random, one-off read. Such a read requires an extensive search over the population of storage servers, which Sirocco servers will perform on behalf of the client. We are conducting research on how best to reduce the expense of such searches [9], improving the efficiency over exhaustive broadcast searches.

The second facility, proxying, enables efficient reading in the case of data that we can predict will be read, especially by multiple clients. Data like file system structures and metadata can be quite efficiently supported through this mechanism. Another potential application is to use proxying to pre-locate data that will be required for a job to launch (e.g., input decks).

Each server, when initially started, considers itself non-authoritative for all records in the store. "Non-authoritative" simply means that the server can make no assumptions about the freshness or location of any data. Therefore, if the client is requesting data, the full location process must be executed to provide a guarantee of up-to-date data. However, in cooperation with other clients, a server may be deemed authoritative for ranges of records. Once authoritative, clients must direct all write requests for that range to that server. This enables the server to cache data and/or other metadata, including location.

6 Concurrency Control

Concurrency control is the ability to ensure a correct outcome to concurrent updates to shared data by multiple clients. There are two well-known methods for accomplishing concurrency control. Pessimistic concurrency control (i.e., locking) is the most common form found in storage systems. Other systems, including database management systems, employ optimistic concurrency control, where operations can be attempted, then rolled back in the event of an invalid concurrent modification [10]. Optimistic concurrency control is beneficial in cases where the likelihood of conflicting operations is low, and locking would incur a significant overhead on the operation. An example is a read-modify-write operation on a remote store. Taking and releasing a lock would double the number of network round trips to complete the operation.

Sirocco implements optimistic concurrency control for a more general workload. Instead of inspecting data, the server executing the operations compare the incoming update IDs with those already present. If the transaction contains a *conditional operation*, the server will ensure that the stored update ID conforms to the expectations specified by the conditional operation. Conditional operations work within transactional batches, extending their applicability to more complex workloads. If a conditional write fails, the enclosing transaction is also failed, rolling back any changes. More information on uses and performance of conditional updates are available [11].

Sirocco also provides a mechanism that a client can leverage to implement pessimistic concurrency control via traditional leased locks, the trigger. Triggers are similar to conditional operations, but with three important differences. First, a trigger is only able to be registered on a single record. Second, a trigger does not modify data, but is instead associated with a batch that is executed when the trigger is activated. Third, a trigger does not fail when its condition is not true. Instead, the operation is placed on a per-record queue. Operations are removed from the head of the queue and executed when that operation's trigger becomes true. This happens when another non-triggered write operation modifies the update ID of the record.

A few additional considerations are allowed for triggers to enable failure recovery if a client fails to release a lock. During a triggered operation, the client is notified of progress: First when the operation is deferred for later execution, and then when the queued operation is next in line to be executed. This allows the client with an operation at the head of a trigger queue to detect when progress is not made in a timely manner, which can be interpreted as a lease expiration in a locking protocol.

Triggered operations allow clients implement locking protocols against storage servers without requiring discrete lock services. Further, a variety of locking schemes can be implemented in clients and libraries without increasing the complexity of the lock service itself. In the current prototype, migration and reconciliation can potentially cause unwelcome changes to the update ID that can obviate its utility for locking. We are investigating a variety of ways to overcome this limitation.

7 Conclusions

Sirocco is a fundamental departure from traditional storage system designs for high end computing environments. By rejecting the current Zebra model for a P2P-style model, resilience features can be incorporated more easily. While some of the limitations of this approach may complicate the file system interaction built on top of Sirocco, the flexibility and features make considerations worth the trouble.

Future materials will be made available at http://www.cs.sandia.gov/Scalable_IO/sirocco.

References

1. Hartman, J.H., Ousterhout, J.K.: The Zebra striped network file system. ACM Trans. Comput. Syst. **13**(3), 274–310 (1995). http://doi.acm.org/10.1145/210126.210131
2. Oldfield, R., Ward, L., Riesen, R., Maccabe, A., Widener, P., Kordenbrock, T.: Lightweight I/O for scientific applications. In: 2006 IEEE International Conference on in Cluster Computing, pp. 1–11, September 2006
3. BitTorrent.org
4. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: SIGCOMM 2001, pp. 149–160 (2001)
5. Timpanaro, J., Cholez, T., Chrisment, I., Festor, O.: Bittorrent's mainline DHT security assessment. In: 2011 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5, February 2011
6. Feldman, M., Papadimitriou, C., Chuang, J., Stoica, I.: Free-riding and whitewashing in peer-to-peer systems. IEEE J. Sel. Areas Commun. **24**(5), 1010–1019 (2006)
7. Karakoyunlu, C., Kimpe, D., Carns, P., Harms, K., Ross, R., Ward, L.: Toward a unified object storage foundation for scalable storage systems. In: Proceedings of the 5th Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS 2013) (2013)
8. Ching, A., Choudhary, A., Liao, W.-K., Ross, R., Gropp, W.: Noncontiguous I/O through PVFS. In: Proceedings of 2002 IEEE International Conference on Cluster Computing, 2002, pp. 405–414 (2002)
9. Sun, Z., Skjellum, A., Ward, L., Curry, M.L.: A lightweight data location service for nondeterministic exascale storage systems. Trans. Storage **10**(3), 1–22 (2014). http://doi.acm.org/10.1145/2629451
10. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981). http://doi.acm.org/10.1145/319566.319567
11. Carns, P., Harms, K., Kimpe, D., Wozniak, J., Ross, R., Ward, L., Curry, M., Klundt, R., Danielson, G., Karakoyunlu, C., Chandy, J., Settlemyer, B., Gropp, W.: A case for optimistic coordination in HPC storage systems,. In: 2012 SC Companion in High Performance Computing, Networking, Storage and Analysis (SCC), pp. 48–53, November 2012