Working Plan of the Sorrento Project (DRAFT)

Hong Tang, Aziz Gulbeden, and Tao Yang
Department of Computer Science
University of California, Santa Barbara, CA 93106
{htang, gulbeden, tyang}@cs.ucsb.edu

1 Introduction

Sorrento emerges upon two trends. First, Clusters have become a cost-effective computing platform that provides incremental scalability and high availability. Second, future applications will demand more efficient usage of high volume of storage resources and I/O bandwidth.

The goal of the Sorrento project is to build a self-organizing storage system upon the cluster architecture, with emphasis on four aspects: *programmability*, *manageability*, *performance*, and *availability*:

- **Programmability:** Uniform name space and virtualized storage resources. Both features simplify the programming effort to utilize distributed storage resources.
- **Manageability:** Dynamic adaptation to the changing cluster environment, such as node additions and departures. There should be low administration overhead from the human operator's point of view.
- **Performance:** Efficient data location and access for a range of data-intensive applications, including: Internet services, data mining, I/O applications, stream-media services.
- **Availability:** Tolerate node and disk failures and automatically repair the redundancy of affected data.

Sorrento is closely related to but quite different from several research areas – distributed file systems, parallel file systems, NAS/SAN, and wide-area peer-to-peer systems.

Distributed file systems Most of the previous researches on distributed file systems deal with desktop applications. Consequently, several assumptions they took are different from Sorrento:

- They assume a user's working set is small. Thus they typically use aggressive client side caching to improve performance.
- They assume that each file is relatively small and can fit on a single partition or volume. As a result, each server typically manages a relatively independent partition or volume and they do not consider very large files that may not fit on a single volume or partition. Some other systems such as xFS organizes different nodes in different RAID groups, and each group forms a volume. An existing file still cannot span across multiple volumes.
- They also assume a high sharing pattern, i.e., users expect their changes to be visible to others immediately.

Parallel file systems Existing parallel file systems focus on the programmability and performance aspects. They typically assume the set of I/O nodes are fixed, and thus lack the support for dynamic adaptation. Neither do they consider data availability in the events of node or disk failures.

Storage systems: SAN and NAS They assume a set of dedicated servers supplying storage to the rest of the cluster. The main challenge is that all data must ship across the network, and thus it requires expensive hardware support. However, they also offer the advantages of high performance and highly available, and supporting applications that require strong consistency. We believe this direction is complimentary to Sorrento's, with Sorrento as a low-cost option.

Wide-area storage systems (P2P) Some of them lack the conventional name space function (such as the navigation of a directory tree); some of them require a hard-to-use programming primitive (such as predicates); some of them lack the capability of modifying existing files. Performance is typically not a serious concern.

A comparison chart of existing distributed file systems, storage systems comes with this document (`dfs-storage.pdf`).

The rest of this proposal is organized as follows: Section 2 discusses the overall system architecture. Section 3 presents several approaches for kernel-application interaction. Section 4 describes the methodologies of the evaluation. The status of the project and a plan is in Section 5.

2 System Architecture

The system architecture is depicted in Figure 1. On each cluster node there is a Sorrento daemon that serves for two purposes: (1) as a server to export local storage to other nodes (previously called *provider*); (2) as a proxy to locate and access data hosted on other nodes on behalf of the local applications (previously called *aggregator*). Name space management is done by a dedicated namespace server connected to the same network that maintains the global directory tree. Figure 2 shows the software components within a single node. Besides the core functionalities of the daemon, there are various add-on adapter interfaces that allows applications to interact with Sorrento.

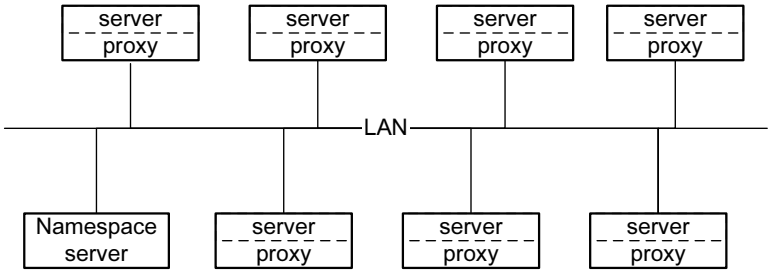

Figure 1: System architecture.

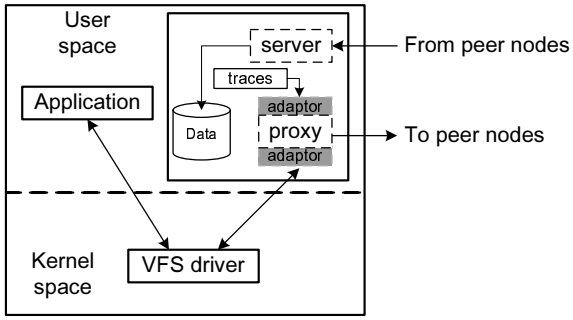

Figure 2: Components in a single Sorrento node.

1. Traditional applications can access the Sorrento file system through standard `mount/unmount` interface. Once a Sorrento file system is mounted to a local mount-point, kernel remembers the type of the file system and will relay file system operations to the Sorrento VFS driver. The VFS driver further calls back into the user-land Sorrento daemon to serve the requests. The VFS driver and the kernel-user interaction is currently being investigated by Nick Lane-smith. Additional information will be found in Section 3.
2. Applications that require richer functionalities can do so through an extended API library. Nothing has been planed on this front yet. Similar work can be found in PVFS's integration with ROMIO and MPI-IO.

3. To facilitate trace-driven evaluation, a trace replay interface is also integrated into the Sorrento daemon. More details on this topic is in Section 4.

2.1 Proxy Module

The proxy module contains the following active entities (threads):

- **Dispatcher:** The proxy-side dispatcher listens on a domain socket for incoming requests and insert them to a local request queue.
- **Thread pool:** The proxy-side thread pool handles the actual processing of the requests from local applications. They dequeue requests from the request queue, handle them (possibly contacting remote servers), and send result back to clients.
- **Subscriber:** The subscriber monitors the multicast channel, picks up the status updates from other nodes and modify the data location protocol states.

The state information maintained by the proxy module includes:

- A set of live hosts, each of which has the following attributes:
  - **Host ID:** Each host is identified by a 128-bits Host ID. There are several reasons for not using IP address to identify a host: (a) a host can have multiple IP; (b) We can quickly migrate an instance to another server without worrying about changing the IP address (e.g., if a storage device is multi-ported, I can quickly do hand-off).
  - **Host address:** IP+port.
  - **Host load:** a history of load samples of previous T seconds.
  - **Available storage:** A variable-sized array of storage devices. Each device has performance parameters (based on vendor tag and use a capability database which stores performance data through offline profiling), total space, available space.
- The namespace server address.
- The set of opened file handles.
- A local file data cache (optional).

2.2 Server Module

The server module is responsible for handling the requests coming from other hosts in the system. Its design is shown in Figure 3.

The server module contains the following active entities (threads):

- **Server dispatcher:** The server-side dispatcher listens on a UDP port and a multicast port for incoming requests, and deposits them to the request queue of thread pool. Optionally, the dispatcher may implement admission control schemes that rejects or drops the requests when the local node is overloaded.
- **Thread pool:** The server-side thread pool handles the requests for local file block creation, read/write, open/close and deletion. After the request is completed, it responds to the request through UDP specifying whether the operation was successful or not; also in the case of a read operation, the requested data is sent back to the client through the response message.
- **Publisher:** The publisher periodically sends local status updates to the multicast channel.

The state information maintained by the server module includes:

- Load condition.

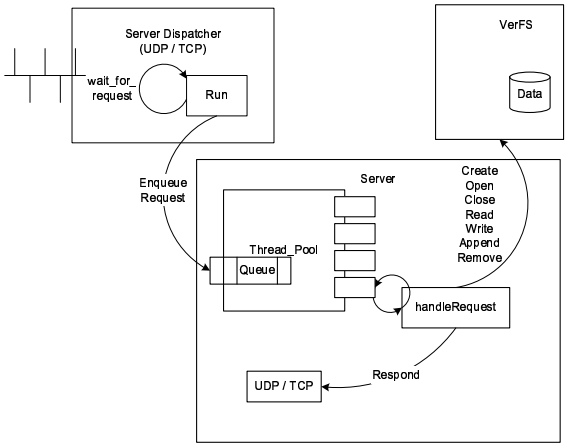

Figure 3: Server architecture.

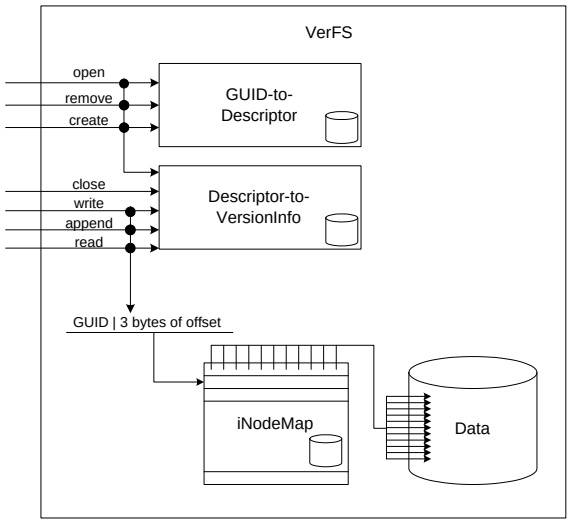

Figure 4: VerFS module.

The file system operations are performed through a module called VerFS. VerFS manages the objects and metadata. Its design and components are shown in Figure 4.

Server interacts with this module by making calls to `Create`, `Open`, `Close`, `Read`, `Write`, `Append`, and `Remove` functions. Open, Remove and Create calls take GUID as parameter. Create call needs to insert the VersionInfo (metadata for version based file system) of the specified object into the Descriptor-to-VersionInfo table, and insert the GUID to descriptor mapping to the GUID-to-Descriptor table. Open and Remove operations get the descriptor from the GUID-to-Descriptor table and access Descriptor-to-VersionInfo table to read the VersionInfo of the object

Read, Write, Append and Close calls take descriptor as argument, they read the VersionInfo and access the INodeMap. VersionInfo's are indexed by Descriptors rather than GUIDs in order to make Read, Write and Append operations fast. INodeMap stores pointers to the physical block store for each block of the objects.

The state information stored by VerFS consists of:

- **INode Map:** It keeps the location of the blocks that take this server as the *home host* based on consistent hashing.
- **GUID to Descriptor Table:** A hash table that stores mapping from GUIDs to descriptors. Descriptors are used as local identifiers to the objects and each object has a different descriptor.
- **Descriptor to VersionInfo Table:** A hashtable that stores metadata for each object. It is indexed by the object descriptors.
- **Physical block store:** The file system that keeps the objects.

2.3 Locating and Accessing Data

A cluster node that needs to perform an operation on a file first has to find the corresponding GUID by contacting the Namespace server. This is shown by arrow 1 in Figure 5 where the proxy module sends a message to the Namespace server. Then the Namespace server sends the GUID of the object back to the proxy (2). Proxy hashes the GUID using *consistent hashing* and determines the node which is responsible for storing that object, and sends the request to the server module of that node (3). If the object is found on the node, request is processed and response is sent (4).

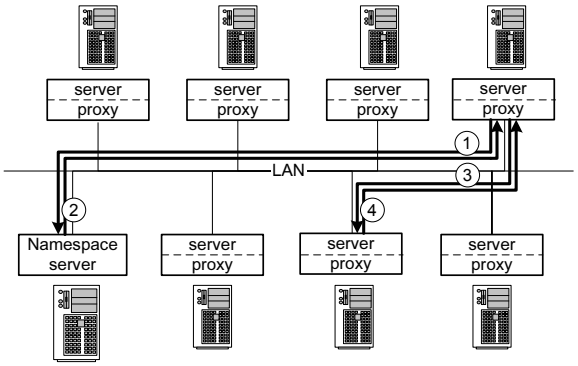

Figure 5: Accessing a file on a remote node.

It is also possible that the object might not reside in the contacted server. However the server has to store location of the objects for which it is the home host. If the object is not found, the server returns the node that the proxy should contact to and proxy module sends the request to the node it received.

3 Kernel-user Interfacing

There are three approaches to allow user programs to access the Sorrento system with the conventional system call interface: (1) system call interception; (2) kernel VFS extension; (3) NFS loopback server.

- **System call interception.** System call interception can be done either through kernel modules or modifying the glibc library. There are two problems of this approach: (1) It cannot support mmapped files. (2) It incurs overhead of checking even for files that are not stored in the Sorrento system.

- **VFS extension.** VFS extension also requires a kernel module that registers a new file system type. The kernel remembers the type of each mounted file system, and redirect each file system operation to the corresponding VFS extension. This approach is completely transparent and compatible with existing file system. The kernel communicates with a user file system daemon through a special character device.

- **NFS loopback server.** NFS loopback server does not require any kernel modification. The extended file system is mounted as an NFS file system, while the NFS server listen at a local address. Since current NFS client does not propagate the close() call to the NFS server, kernel may still need to be modified to change such a behavior. Potential problem with this approach is the performance due to the RPC overhead.

Currently Nick is researching on the last two options and plans to leverage the work by the open source project called AVFS.

4 Evaluation Methodology

The biggest challenge of the evaluation is essentially the chicken-and-egg problem. Without having the system deployed, we cannot run actual benchmark on it. While without benchmarking the system, we are unable to effectively make design choices.

We chose to use trace-driven evaluation for three reasons. First, it allows us to get around the need for the kernel-user interfacing module, which could be time-consuming to implement and test without adding much insights toward the research topics we focus on. Second, it is easier for us to quickly benchmarking various applications without actually porting the applications to the target environment. Third, potential problems in the system design are easier to reproduce and identify.

There are also a number of questions to solve to make trace-driven evaluation feasible. First, we need to answer at which level should we collect traces. The I/O activities at the disk block level are too low-level and may not be representative if the same application runs upon a different file system implementation (For example, cache hit rates and metadata access patterns would be two things totally different.). On the other hand, the I/O activities at the file level would be more meaningful, but it does not capture the I/O activities upon mmapped files. The current choice is to collect traces from applications that do not heavily involve mmaped file operations.

The second problem is related to the fact that many existing applications manage distributed storage resources by themselves through data partitioning, and we need to find a reasonable way to transform the I/O activities of those applications to what it would be like if a global virtual disk were in place.

The third problem is how do we use a trace collected with a fixed hardware setting (say N nodes), to generate synthetic traces which preserve the characteristics of the application's I/O activities and allow us to evaluate the performance with a different hardware setting. Similar work has been done on the disk level I/O which demonstrates that it is a very hard problem in general.

The following is a list of the applications that can be used in the evaluation:

- **Crawler.** Each crawler writes (appends) to a file exclusively owned by itself. Each node runs a fixed number of crawlers, and the files may expand at different rates for different crawlers. This application can be used to evaluate how our first-touch placement policy can be used to heuristically place data on the local disks, and a background daemon can move old data to other nodes to maintain the storage utilization and to prevent storage overflow.

- **Protein sequence matching.** We may build the service by using a centralized storage system with close to 1 terabytes disk space. We sample the file I/O activities for a large number of queries. We then store the protein database in the Sorrento system, and split the traces of these queries and run them on distributed nodes. This allows us to evaluate the data access performance without any locality. We may also divide the database into many small partitions (the number of partitions much larger than the number of cluster nodes), and we sample the file I/O activities of a parallel search algorithm – each process searches a subset of partitions and then aggregates the results. We then distribute the partitions without being aware of

which partition will be accessed by which process initially. Replay the traces on different nodes, and over the time, the partitions accessed by a process should gradually migrate to the local partition.

- **Parallel I/O applications.** We may profile the activities of applications with special calls to parallel I/O interfaces (such as programs using MPI-IO). And we can run different MPI nodes on different machines and replay the data access activities of that node. This allows us to evaluate the performance of the shared access (although with different byte-range) of the same file. It is even better if different nodes may change the regions they access from one phase to the next.

The other dimension of the evaluation is how the system responds to the changes of the cluster environment. To evaluate the effect of automatic integration of new storage resources, we may use the crawler application, and show that existing files will expand to the new node. In terms of node failures, there are two aspects to evaluate: (1) Existing files are still addressable; (2) Lost files will have their redundancy restored if replication is used.

5 Project Status and Development Plan

Currently the namespace server and most of the software modules are implemented, including: load statistics collection; consistent hashing; UDP request/response management with timeout and retrial; persistent hash table (for object catalog); file block cache (for physical block manipulation); GUID (for block and host ID management); thread pool.

The milestones of the project development are:

1. Barebone runtime system. No version-based data management. No data replication. No dynamic migration. No system call level transparency or parallel I/O interface.
2. Add dynamic data migration.
3. Add version-based data management and data replication.
4. Add kernel VFS switch.

The research issues that may lead to publications are:

1. Trace analysis of large-scale data-intensive applications (SIGMETRICS). It would be good to have both I/O activity traces and node addition/departure traces. Focusing on the comparison with traces from desktop applications.
2. Dynamic service migration with data migration support (system or Internet technology conferences). Allow services to gradually improve performance through online profiling and data migration.
3. Support quick check pointing through log-structured version-based file management (Parallel computing conferences). We can provide two types of check-pointing mechanism. Hard check-pointing pauses the program until a version is created and copied to a different physical location. Soft check-pointing just creates a new version of a file, and the old version will be gradually moved to the other nodes in the background. Hard check pointing is slow but can tolerate both node failures and disk failures, soft check pointing is fast (just like traditional copy-on-write technique) and can tolerate node failures but does not guarantee the recovery after disk failures.
4. Overall system implementation, performance evaluation (system conference).