ASG Storage Model

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Introduction

This document describes the storage model that will be implemented by each Triton and Sirocco.

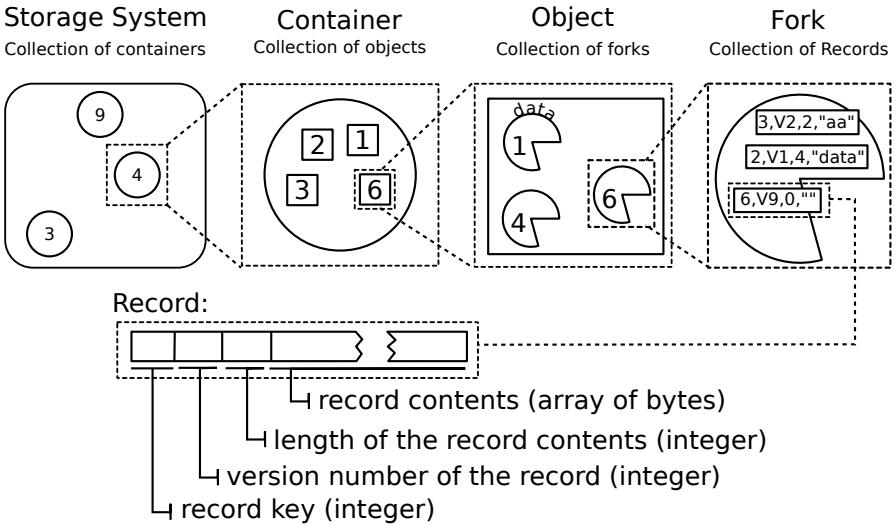
The goal of this document is not to define a single header which an be used with both implementations, but to make sure the model implemented by the header/API is compatible.

Note
This is a working document - not intended as a deliverable or distribution at this point. The goal is to support the model discussion and to capture issues raised during the discussions.

Chapter 2

Model

2.1 Overview



Note
TODO: need to bring location into this figure.

2.1.1 Terminology

Record
: a tuple containing key, version, and data. The key and version are integers, while data is a variable length array of bytes.

Fork
: A collection of records.

Object
: A collection of forks.

Container
: A collection of objects.

Location
: An identifier for a logical instance of a storage server.

2.2 Components

The storage system consists out of a set of locations, where each location can store records, forks, objects and containers.

Each entity (record, fork, object, container) has a numeric identifier. Containers, objects and forks each form a distinct namespace for the entities they contain. For example, more than one object can contain a distinct fork with id 1.

Note
Need to explain distribution. Containers, objects and forks can be distributed over multiple servers.

2.2.1 Record

A record is the basic storage block for the storage system. A record consists out of a numeric key or record id, a number indicating the version number of the record, and the record's data. The data is a variable length array of bytes. Empty records, records for which the data has length 0, are allowed.

2.2.2 Fork

Each fork within an object has a fork id: a fixed length integer number. A fork contains zero or more records, and forms a distinct key space for the records it contains.

2.2.3 Object

An object is a collection of forks. Objects form a unique name space for the enclosed forks, i.e. the same fork id within a different object represents a different fork.

The fork with identifier 0 is special and represents the enclosing object.

Note
Need to clarify the entity 0 rule.

2.2.4 Container

A container consists out of a collection of objects. Each container forms a unique name space for the objects it encloses.

The object with id 0 is special and represents the container.

Containers can be used to partition the storage system into logical units. For example, each container could hold its own file system. Containers provide isolation.

Containers cannot be nested.

2.3 Operations

This section describes how a client can interact with the storage system.

Note
TODO: Need to add parameters to the descriptions below. Not the full API detail, but more as a clarification on how to specify what to act on etc.

2.3.1 write

A write operation stores data in a sequential range of records in the specified fork. The fork is identified by listing its fork id, and the identifiers of the enclosing container and object. The range is described by a starting record key, and the number of records in the range. For example, range 1,3, indicates records 1, 2 and 3.

Together with the range, the number of bytes going to each record in the range also needs to be specified. The write operation expects the user to provide nr_of_bytes_per_record times nr_of_records_in_rage bytes of data.

A number of optional write modes are possible:

- A flag indicating that the write should only succeed if the version number specified is strictly higher than the existing version (*store conditional*). By specifying a special *null* version, this flag can be used to indicate a write should only succeed if the entity has not yet been written to.
- A flag indicating that the new version number of the entity should be the highest existing one plus one.

A write without specifying any of these flags makes it possible to force a version number on an entity.

A write operation can optionally be directed to a specific location.

Writing a record completely replaces any existing data. Partial writes are not possible.

Every write operation, including when specifying a range containing than one record, is atomic.

Note
TODO: Need to clarify how the conditional works for a range of version numbers.

2.3.2 read

Read retrieves a range of records from a specified fork.

A read operation can optionally be directed to a specific location.

Similar to the write operation, a read can be conditional upon the value of the version.

Note
TODO: need to clarify how the conditional works for a range of version numbers.

2.3.3 punch

A punch operation removes all data from the specified record range. Punch is similar to write (in that it supports conditional execution based on the existing version number) with the exception that instead of adding data, the matching data is removed.

Note
Can punch be applied to a fork, object or container?

Note
Is punch the equivalent to writing 0 bytes to a range?

Note
Is punch different from reset in that it can not write the 0 (non-existing) version?

2.3.4 reset

Reset resets an entity (container, object, fork or record) to the its original condition. It differs from punch in that it does not specify a data range and does not take a version number.

2.3.5 probe

Probe returns a set of matching items. A probe searches within the entity specified. This entity can be a container, object or fork. If no fork is specified, probe searches the indicated object. Likewise, if both object and fork are left unspecified, probe searches the container.

Entities need to be fully specified. It is not legal to specify a fork id but no object or container id.

A probe can optionally be directed to a specific location.

2.4 Clarifications & Notes

2.4.1 Determining Object ID

The system does not help in finding an unused entity (object, container, fork). The user is responsible for picking (and claiming) an id.

2.4.2 Creating and Deleting Entities

There is no need to explicitly create an entity (object, container, fork). All entities exist at all times, in a default configuration. Punch and reset restore the default for respectively part of or the whole entity.

2.5 API Details

This section contains advice and notes on how the model can be implemented by an API.

- It should be possible to reuse functions for all entities;

Chapter 3

Use Cases

This section shows how the model can be used to support a variety storage libraries and file systems.

3.1 Supporting POSIX

3.1.1 POSIX Directories (namespace)

How to implement traditional directories.

If needed, the POSIX clients will have to perform locking (outside of the storage system) or otherwise collaborate to force data consistency and strict posix compliancy.

3.1.2 Implementing POSIX extended attributes

Extended attributes are variable length keys with an associated variable length value, and can be attached to a POSIX file (directory entry).

3.1.3 POSIX file consistency

Need to clarify

3.2 Data Provenance

One example is to be able to find all objects that containt content originating from a specific source. The *data* is being tracked, not files or objects. It would be acceptable to loose some information when merging. So, when copying data from one object in another object, it would be sufficient to remember that the second object contains data from the first, without being able to say exactly which data comes from which object.

The current way of handling this is by using many small files (from 4 to 32 MB in size).

Note
This can most likely be implemented more generally by enabling queries based on record (content). Needs to be discussed. See open questions.

3.3 High-Level I/O Libraries

Map HDF5/pNetCDF/Damsel directly on top of objects.

3.4 Extending Memory Address Space

1. For out of core computation
2. For GUPS (Giga-Updates per second?) accesses

Note that the traditional mmap is using read/write and so does not require special consideration.

Brad: need to clarify/update (see ORNL Milestone)

3.5 Hierarchical Storage

Brad: need to clarify/update (see ORNL Milestone)

3.6 MPI-I/O

An example of non-POSIX consistency: user-controlled sync through MPI_File_open/MPI_File_close/MPI_File_sync.

Chapter 4

Rationale

Explanations on why we made certain decisions.

- System picks OID vs client picks.

Chapter 5

Open Questions / Discussion Points / Notes

This chapter collects various points that have been brought up but not (sufficiently) answered yet.

5.1 Compatibility with legacy systems

5.1.1 Compatibility with POSIX

5.1.1.1 Namespaces/directories

SNL	strict posix compliancy. Example: atomic move of a file between two directories.
ANL	Consider PVFS approach: compliancy where it matters, not guaranteed for some rare border cases.

5.1.1.2 Data Consistency

Need clarify

5.2 Various

Left here to ensure that everything is clear in the single-fork type model.

- Extents: is it a model concept or purely an optimization?
  - What happens with an extent if a hole in punched in the middle?
  - Do all bytes in an extent have the same version number?
  - Are extents preserved and can they be retrieved as first-class entities?
    - What happens with an extent a new extent (with higher version number) overlaps?
    - Is it possible to enumerate all extents?
- If a container is mapped to multiple storage servers, where are its records stored? Do you have to locate the server that has object 0 in the container and find the records there?
- Is indexing done in the system or on top of it?
  - Out of scope. Ignore for now.