# Extended Filesystem Functionality for Self Managing Storage Systems

Terrence Wong
Advisor: Greg Ganger
Grad Students: Andrew Klosterman, Mike Mesnier

April 30, 2004

**Abstract**

The Parallel Data Lab is in the midst of developing the next generation of distributed storage systems. Dubbed Self-* (Self Star) Storage, this system will improve management efficiency among a cluster of machines at a single location where the machines are connected to one another through a high speed network. One of our goals in building the first instances of this system is to use as much existing code as possible while compromising nothing in terms of desired Self-* Storage functionality. We already have a filesystem, called, S4 that we would like to use as the basis of permanent storage in the Self-* Storage system. This research report describes mechanisms that allow an existing filesystem, like S4, to be coerced into supporting the functionality that Self-* Storage demands.

# Contents

# 1 Introduction

It is a commonly observed trend that data storage requirements are increasing, creating demand for data storage centers. As the density of storage increases and the cost of storage decreases, data centers find themselves managing an increasing amount of storage space. Unfortunately, the cost of management is not getting cheaper. It is estimated that the administrative effort requires one human for every 2 terabytes of storage [1]. With petabyte scale storage coming in the near future, the human element will be a very expensive component of a storage system.

The rest of this section describes two storage systems that are relevant to this research. Section 1.1 introduces a storage system that we are currently building at the Parallel Data Lab. Section 1.2 follows with a description of a storage system that we are building on.

## 1.1 Self-* Storage

We feel the solution to the costly administration problem is to design a new type of storage system from the ground up. Current computer systems force administrators to micro manage resources by requiring that they set parameters for every possible choice. The goal of Self-* Storage [2][3] is to develop a management hierarchy, similar to those found in real life, which allows various parts of the system to be guided by superiors and human goals but free to implement their requirements to the best of their ability.

Self-* Storage is a distributed storage system utilizing large amounts of cheaply available hardware at a single location. It runs several services that will allow it to configure, tune, organize, heal, and manage itself. A human administrator is needed to occasionally replace failed hardware and to input goals that the system should try to achieve. These goals are in the form of bandwidth requirements, availability, storage space, and other performance metrics that the end user desires.

The implementation of Self-* Storage is divided into two phases. The first part, Ursa Minor, is a lightweight version of the design whose purpose is to demonstrate the fundamentals of the system. The second phase, called Ursa Major, will be the full fledged system with all optimizations running and all functionality available.

## 1.2 S4

S4, the Self Securing Storage System [4], is an older project at the Lab. It is a versioning filesystem that doesn't version on every file close; S4 versions every time the data is changed. S4 also validates and stores the name of the client and

each operation that it makes. Unlike conventional versioning filesystems where old versions of files are removed by the cleaner when more space is required, the existence of old versions is guaranteed for a specific amount of time. This time period, called the recovery window, is set to a week to give administrators enough time to detect and understand system compromises. Using these semantics, S4 can help identify a malicious user and the time that the user began altering data. The last valid version of a file can then be recovered. With all data guaranteed for a reasonable amount of time, the malicious user is unable to cover their tracks by hiding the methods they used to compromise the system.

S4 is an object storage system. An object storage system is unlike a conventional filesystems where data is stored and accessed by blocks organized into files which are organized into a directory hierarchy. S4 instead stores and retrieves files based on the file name and the offset into the file. The files are assigned a numbered identifier in a flat namespace. S4 will return a 64 bit number to identify the object that has just been created by a client request. Note that the client must stored this value in order to access the file later and therefore a conventional filesystem layer can be written on top of S4.

In addition to supporting normal objects, S4 also supports a special table object. While a table object is internally similar to a normal object, S4 provides functionality to allow the insertion, lookup, and deletion of key/data pairs. The table information is stored in a B+ tree where internal pointers are indexed by 64 bit hashes of the first key of the block pointed to.

## 1.3 Persistent data for Self-* Storage using S4

Self-* Storage relies primarily on S4's table object to store information. Using erasure codes, Self-* Storage encodes blocks of data into shares which are then stored in S4 tables as key/data pairs. The system can later recall some number of these shares to reconstruct the original block of data. Each Self-* Storage object is represented by shares on some number of S4 drives. These drives have a table object allocated for the Self-* Storage object. The shares are distributed to the various drives and stored in the table using the block number into the object that the share represents as the key, and the share data itself as the data.

## 1.4 Roadmap

Section 2 defines additional requirements of the existing code. Section 3 describes our solution to the requirements listed in section 2 and is followed by preliminary benchmark results in Section 4. Section 5 explains work in progress that will further improve performance.

# 2 Self-* Storage Requirements

As S4 was designed before Self-* Storage was conceived, S4 does not provide all of the functionality that Self-* Storage requires.

## 2.1 Object naming

Since Self-* Storage is a distributed storage system, it will rely on many instances of S4 running on various machines to store data. As a result, we need a global object identifier since the identifier issued by one S4 drive is meaningless to a different drive. We chose a 128 bit object identifier (OID) that the Self-* Storage system will assign when it creates an object.

There is clearly a conflict if S4 chooses an object identifier (S4OID) when a create is called but the Self-* Storage systems is designed to assign the object identifier (SSOID) as well. Thus, we use an S4 table object as a mapping table to translate from the SSOID to the S4OID.

## 2.2 Enumerate

Self-* Storage needs to be able to get a list of OIDs of current objects in the system for its own purposes. A client will never need to call enumerate directly, but the system will need it for the cleaner (clears out versions older than the recovery window to reclaim space) and for crash recovery (verify system metadata to drive metadata). The system may not want to enumerate the entire object space, so the enumeration should allow a specific range to be specified.

## 2.3 Clone

Making copies of objects is essential. A clone of an object is an identical copy of the object's data and metadata assigned to a new identifier. The two objects, while identical, are completely independent, and modifications to one are not reflected in the other. Creating a clone of an object can be done by physically copying the data from the object to its clone, but a copy-on-write scheme is more efficient. In this case, one set of data masquerades as both copies of the cloned object and is only duplicated if one of the objects is modified. If we assume that some subset of objects is never modified, time costs and storage space are reduced. S4 must support a clone operation over a range of objects. A range of objects is defined to be any OID whose value falls between a start OID and end OID inclusive. The caller specifies a range of Self-* Storage OIDs to copy from (source) and the range of OIDs to copy to (destination). When called, the clone operation takes every existing object in the source range and copies it to the appropriate OID in the destination range. Any objects that might exist in the

destination range at the time of the clone call should be deleted. This is done because the clone function at the drive level cannot be called unless validated by a service called the metadata service. If the metadata service determines that the ranges are valid, then the only explanation for the existence of a lingering object in the destination range is that this particular drive did not receive the delete call on those objects or that the objects have not been cleaned.

The range clone operation is useful for creating snapshots of the filesystem for future read only recovery. In addition, clone can also be used for less common operations like a read-write filesystem fork or even the basic "cp" shell command.

### 2.4 Delete

Since the integer based namespace supports operations over ranges of OIDs, it is natural to require a way to delete objects en masse. Furthermore, we assume that the table objects are populated with key/data pairs where the keys are meaningful integers much like the OIDs. Therefore it also makes sense to require a table entry release over a range of keys within a table object.

## 3 Implementation

Since Self-* Storage is a large project with many people working on it, keeping modified code separate is important. As such, our implementation of the requirements took the form of wrapper functions sitting on top of the S4 API layer. Our goal was to provide the necessary functions with as few changes to the S4 drive as possible in order to minimize the amount of code conflict with the team member who was working on the S4 drive code.

### 3.1 Mapping Table

There is clearly a conflict if S4 chooses an object identifier (S4OID) when a create is called but the Self-* Storage systems is designed to assign the object identifier (SSOID) as well. Thus, we use an S4 table object as a mapping table to translate from the SSOID (key) to the S4OID(data). While the generic S4 table is sufficient for the requirements, it is not very efficient. We expect the S4 drive to be heavily populated with objects, so we cannot use 64 bit hashes as indexes to organize the 128 bit Self-* Storage OID since there would be $2^{64}$ collisions per hash. Furthermore, since the keys need to be stored in a sorted fashion (see Enumerate, below), the only hash function that would allow that would be the one that simply returns the high 64 bits of the key. We expect, for simplicity, the Self-* Storage system to create new objects in increasing numerical order, which means that every $2^{64}$ objects created will have the same hash. Searching

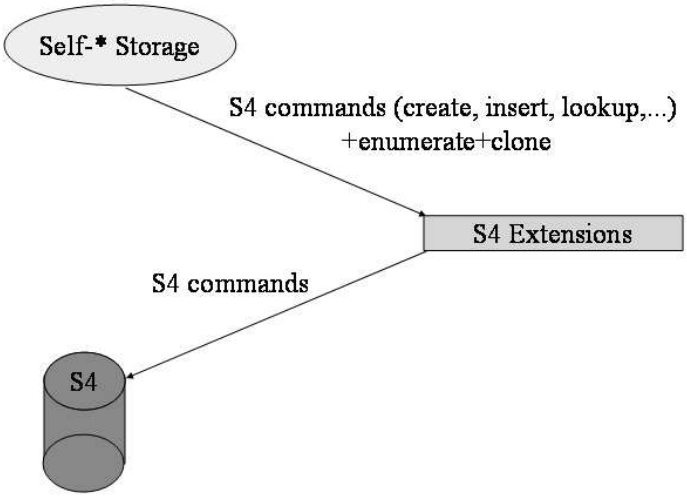


Figure 1: The location of our code as extensions to the S4 API

a B+ tree with a high number of collisions will be very expensive, and must be avoided.

In order to combat this problem, we created a new table type by adding a flag bit to the table's metadata to signify it as a mapping table. We then modified the relevant S4 functions to understand that this particular table type has 128 bit indexes, i.e. the key itself unhashed, for internal block pointers. As a fortunate consequence of removing the hashed key indexes, the mapping table stores the key/data pairs sorted by key which provides for a very efficient enumerate.

### 3.2 Enumerate

Enumerating over the S4 table type is easy because of the nature of a B+ tree; the data is all stored in a row across the very bottom of a B tree. Enumerating a particular range simply requires starting at the beginning of the data and walking across while checking each key, returning those that fall in the specified range. This isn't really efficient however, if we consider a very dense S4 object space and a very small enumeration range. The enumeration essentially has to look at each object in the S4 drive no matter how small the range is.

We altered the table to keep the table sorted by key which allowed for enumerations to run in time relative to the size of the enumeration range rather than the size of the table. Instead of enumerate being required to walk across the bottom of the entire tree, it does something more closely resembling a lookup of the first SSOID in the enumeration range. Rather than returning the data as a

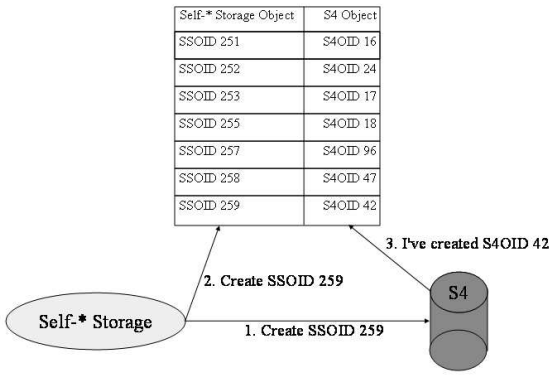


Figure 2: Creating an object inserts an entry into the mapping table

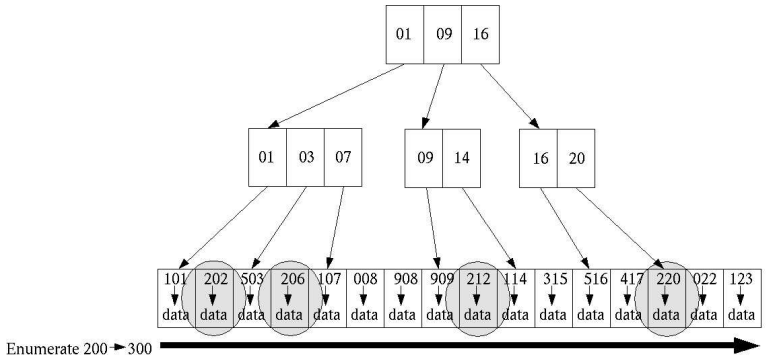


Figure 3: Inefficient enumeration over a generic S4 table object

normal lookup would do, enumerate simply walks across the bottom from there copying everything into a buffer until it reaches the end of the enumeration range, reaches the end of the B+ tree data, or fills the buffer.

### 3.3 Clone, phase 0

A basic clone can be implemented using the mapping table and the enumeration function. The first step is to enumerate the destination range and delete any objects that are there as per the requirements. The second step is to enumerate the source range. Every object in the source range gets copied directly to a new S4 object and the new SSOID to S4OID mapping gets inserted into the mapping table. It is clear that this method is not efficient. Cloning on a large range of objects means that the caller has to wait for each object to be copied which is potentially very time consuming.

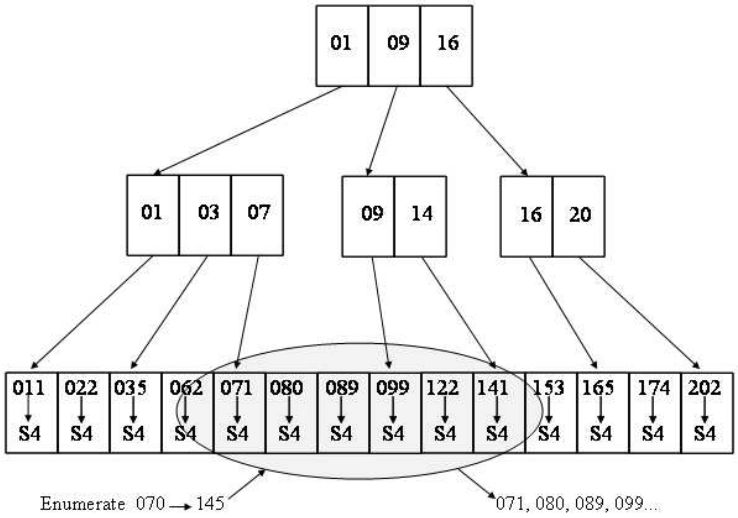


Figure 4: A better enumeration over a S4 table with sorted keys

### 3.4 Copy-on-Write Clone, phase 1

A better solution to the problem is a simple copy-on-write scheme that adds minimal complexity to the system. This method is implemented in Ursa Minor and is modeled in a fashion similar to a hard link in an inode based filesystem. Instead of immediately copying the file as was done in phase 0, a new entry is simply inserted into the mapping table to translate the new clone's SSOID to the original object's S4OID. A table storing reference counts for S4 OIDs is added to track the number of SSIODs that are related to the particular S4OID in order to determine if a modification to that object requires it to first be copied. All functions that modify objects are updated to understand the need to copy S4 objects that have reference counts higher than 1 before modification. Read functions do not require any modifications at all since the copy on write information is contained in the mapping table that they already use.

### 3.5 Delete

Since there are ways to create and list objects over ranges of SSOIDs, it is also useful to delete objects over ranges. In addition to allowing deletion of objects over ranges, we allow deletion of key/data pairs by key within a table object. Deleting a range of objects is very similar to an enumeration over the same range. The delete function traverses the table looking for the beginning of the range, but instead of copying found objects into a buffer as in the case of enumerate, the function releases the objects by marking them as deleted. The

| Self-* Object | S4 Object |
|---|---|
| SSOID 251 | S4OID 16 |
| SSOID 252 | S4OID 24 |
| SSOID 253 | S4OID 17 |
| SSOID 255 | S4OID 18 |
| SSOID 257 | S4OID 16 |
| SSOID 258 | S4OID 17 |
| SSOID 259 | S4OID 16 |

| S4 Object | Ref Count |
|---|---|
| S4OID 16 | 3 |
| S4OID 17 | 2 |
| S4OID 18 | 1 |
| S4OID 24 | 1 |

Figure 5: Example of the mapping table (left) and reference count table (right) after a clone operation

S4 cleaner eventually frees the associated resources for reuse. In the same way that deleting ranges of objects removes the appropriate key/data pairs from the mapping table, deleting a range of keys from within a table object removes the appropriate key/data pairs from the table associated with the particular object. However since the shares stored in a generic S4 table object are not guaranteed to have sorted keys, the range delete will have to walk the entire bottom of the B+ tree in order to ensure all of the keys in the range are released.

# 4 Results

We ran our tests on a single Pentium 4 2.66ghz machine with 1GB of RAM. The S4 drive ran on the local machine which stored the data on a Seagate Cheetah 10K.6. The 36GB disk drive spins at 10,000 RPM and utilizes the Ultra320 SCSI interface.

## 4.1 Enumerate

We tested the speed of an enumeration over a variable range of objects. As figure 6 shows, the speed of an enumeration is linear with respect to the number of objects being returned. This is expected because the time cost is dominated by the data collection at the bottom of the tree. The cost of the enumerate is acceptable even for large numbers of objects because large enumerations will be broken up into segments by the system in order to maintain a manageable buffer in which to store enumerated objects. We expect the system to perform some operation on the subset of objects before continuing the enumeration; the

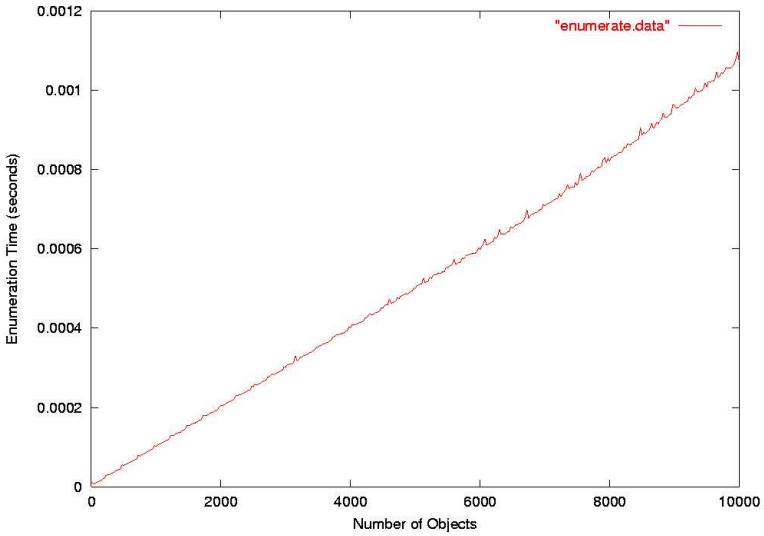


Figure 6: Enumeration time over variable number of objects

enumeration will have negligible time cost relative to the system operation on the returned object list.

### 4.2 Clone

We also did speed tests on the clone operation over a variable number of objects. The graph in figure 7 shows that the clone operation is more expensive than an enumeration over a comparable range. While it is linear for most of the test, there is an unusual increase in the amount of time required to clone at approximately 8500 objects. Although we were able to determine that the cause is in the block allocation portion of the S4 drive's table insert function, further work will have to be done to understand exactly why the insertion time is so high and whether or not it can be reduced.

## 5 Future Work

### 5.1 Copy on Write Clone, phase 2

While the version of clone for Ursa Minor is functional, it can be improved. Even though the copy on write scheme saves time and space, in that the object copy is delayed, there are still metadata operations required for each cloned object to be reflected in the mapping table and range table. As the number of

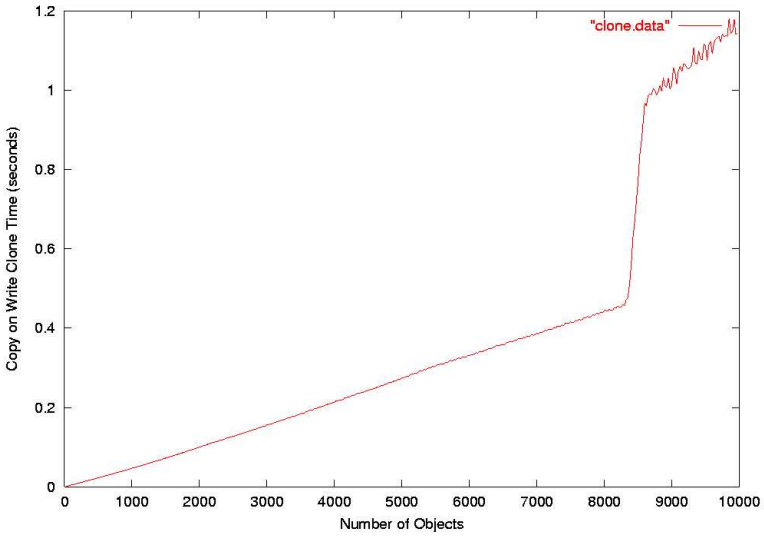


Figure 7: Clone time over variable number of objects

objects in S4 grows, a clone operation could potentially encompass a range of objects so large that inserting one entry into the mapping table for each object is too time consuming, as figure 7 shows.

The copy on write clone that we are working on for Ursa Major will have another table, which we call the "range table". When a clone is called in Ursa Major, the source and target ranges are simply inserted as keys and data into the range table, allowing the function to return quickly. All of the S4 functions that access objects will have to be updated to understand that the mapping table is not the only source of information as to the existence of objects. This will increase complexity, but we believe the increased overhead of checking both tables will be negligible and that the amount of time saved when clone is called over a large densely populated range will be considerable.

The consequence of the range table is that the mapping table will no longer be the authoritative source of whether or not a particular object exists. The range table may describe a particular object to be a chain of clones of previously cloned objects. The system would have to do several table operations and compares to follow the chain, which eventually leads back to the original object in the mapping table. We are still exploring the tradeoffs associated with storing information in the range table as opposed to the mapping table. It seems to be the case that the tradeoffs are directly associated to the expected usage patterns that the system will see. If large dense ranges are cloned, it is certainly a good idea to quickly insert them into the range table. However if very small or sparse ranges are cloned, it makes sense to simply evaluate them right away to save the range table from getting bogged down with trivial entries. The solution to this

is not trivial, though, because we do not know how the client will use the system and therefore the terms "large dense ranges" and "small or sparse ranges" are poorly defined. Since we don't want to have to limit the user's options, we have to consider all possibilities. With so many variables to consider, however, we feel the next step is to describe particular usage patterns that will be most popular and optimize the range table for those. Other less common clone operations will work, but the user will be advised that system performance may not be optimal in those cases.

### 5.2 Benchmarks

Although we were able to obtain data on the time required to enumerate and clone objects, we were unable to determine the impact of the copy-on-write scheme compared to the immediate copy scheme. We were also unable to calculate how much of an impact the copy-on-write scheme has on performance when a modified object is actually copied in a table insert or delete call due to the lack of real world workloads. Unfortunately, the rest of the Self-* Storage system for Ursa Minor is not yet complete, and until it is, our benchmarking is limited to functionality independent of the usage pattern..

## 6 Conclusions

This work proposes mechanisms to allow Self-* Storage to be built on top of an existing filesystem called S4. After exploring the requirements that Self-* Storage puts on S4, we found that wrapper code for the S4 API was sufficient in providing functionality such as object enumeration, range clone, and range delete. While the new functionality of S4 is useful for Self-* Storage, however, the potentially large number of mapping table operations required for a large clone can make the implementation for Ursa Minor insufficient for performance reasons. A proposed range table based copy-on-write clone for Ursa Major may reduce clone overhead to an acceptable level.

## 7 Acknowledgements

I would like to thank my advisor, Greg Ganger, for giving me this opportunity. I would also like to thank Greg, Andrew Klosterman, John Strunk, and Mike Mesnier for guiding me in this project.

# References


[1] Total Cost of Storage Ownership - A User-oriented Approach, Gartner-Consulting, 15 February 2000

[2] G. Ganger, J. Strunk, A. Klosterman, "Ursa Major Design Document," Parallel Data Lab.

[3] Ganger, G.R., Strunk, J.D., Klosterman, A.J Self-* Storage: Brick-based storage with automated administration, Published as Carnegie Mellon University Technical Report, CMU-CS-03-178, August 2003.

[4] Self-Securing Storage: Protecting Data in Compromised Systems. Strunk, J.D., Goodson, G.R., Scheinholtz, M.L., Soules, C.A.N. and Ganger, G.R. Appears in Proc. of the 4th Symposium on Operating Systems Design and Implementation (San Diego, CA, 23-25 October 2000), pages 165-180. USENIX Association, 2000.