Delayed instantiation bulk operations for management of distributed, object-based storage systems

Andrew J. Klosterman

August 2009

CMU–PDL–09–108

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis committee

Prof. Greg Ganger, Chair (Carnegie Mellon University)
Mr. Craig Harmer (Symantec)
Dr. Sami Iren (Seagate)
Prof. Dave O'Hallaron (Carnegie Mellon University)

© 2009 Andrew J. Klosterman

Abstract

The basic distributed, object-based storage system model lacks features for storage management. This work presents and analyzes a strategy for using existing facilities to implement atomic operations on sets of objects. These *bulk operations* form the basis for managing snapshots (read-only copies) and forks (read-write copies) of portions of the storage system. Specifically, we propose to leverage the access control capabilities, and annotations at the metadata server, to allow for selective clone and delete operations on sets of objects.

In order to act upon a set of objects, a bulk operation follows these steps. First, the metadata server accepts the operation, contacts the storage nodes to revoke outstanding capabilities on the set of objects, and retains a record of the operation and the affected set of objects. At this point, clients can make no changes to existing objects since any capabilities they hold will be rejected by storage nodes. Second, when clients subsequently contact the metadata server to access affected objects (e.g., acquire fresh capabilities), any records of bulk operations are consulted. Finding that a client is accessing an affected object, the metadata server will take the necessary steps to enact the uninstantiated operation before responding to the client request. This eventual enforcement of operation semantics ensures compliance with the operation's intent but delays the corresponding work until the next client access. With appropriate background instantiation, the work of instantiating bulk operations can be hidden from clients.

In this dissertation, we present algorithms suitable for performing bulk operations over distributed objects using $m-of-n$ encodings. The core logic is concentrated at the metadata server, with minimal support at clients and storage nodes. We quantify the overheads

associated with the implementation and describe schemes for mitigating them. We demonstrate the use of bulk operations to create snapshots in an NFS server running atop distributed, object-based storage.

Acknowledgements

I thank the members and companies of the PDL Consortium throughout my doctoral career (APC, Cisco, EMC, Engenio, Equallogic, Google, HGST, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NetApp, Oracle, Panasas, Seagate, Sun, Symantec and Veritas) for their interest, insights, feedback, and support. Experiments were enabled by hardware donations from APC, IBM, Intel, NetApp, and Seagate. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by the Army Research Office, under agreement number DAAD19–02–1–0389.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**

1.1 Distributed, object-based storage 2

1.2 Storage management and operations upon object sets 3

1.3 Thesis statement 4

1.4 Bulk operations for storage management 4

1.5 Roadmap 6

2 Background and related work **7**

2.1 Distributed, object-based storage 7

2.2 Broader references 8

2.3 Cloning, snapshots and storage management 10

2.4 System components 11

2.4.1 Objects 11

2.4.2 Capabilities 13

2.4.3 Data distribution 14

2.4.4 Client 16

2.4.5 Storage node 17

2.4.6 Metadata server 19

3 Delayed instantiation bulk operations **21**
3.1 Placement of responsibility 21
3.1.1 Bulk operations at the storage nodes 22
3.1.2 Bulk operations at the client 24
3.1.3 Bulk operations at the metadata server 24
3.2 Grouping objects 25
3.3 BulkClone 26
3.4 BulkDelete 28
3.5 Delayed instantiation 29
3.6 Bulk operation tracking 31
3.7 Completion and success criteria 32
3.8 Mitigating costs 34

4 Implementation **35**
4.1 Client 35
4.2 Storage node 36
4.3 Metadata server 37
4.3.1 Front-end 37
4.3.2 Back-end 38
4.3.3 Helper 40
4.4 Background instantiation 40
4.5 NFS server 42
4.6 Protocol 45
4.6.1 Create 46
4.6.2 Lookup 47
4.6.3 Enumerate 47
4.6.4 Delete 51
4.6.5 Write 52
4.6.6 Read 54
4.6.7 BulkDelete 55
4.6.8 BulkClone 56

5 Data structures and algorithms **57**
5.1 Data structures . . . 57
5.1.1 Sequencer . . . 57
5.1.2 Object table . . . 58
5.1.3 BulkDelete table . . . 58
5.1.4 BulkClone table . . . 59
5.1.5 Object metadata tables . . . 59
5.2 Implications of bulk operations . . . 59
5.2.1 BulkClone . . . 60
5.2.2 BulkDelete . . . 62
5.3 Core algorithms . . . 64
5.3.1 GetMDOID . . . 64
5.3.2 InstantiatePassThroughLimits . . . 66
5.3.3 InstantiateHole . . . 68
5.3.4 Divorce . . . 71
5.4 Core operations . . . 72
5.4.1 Enumerate . . . 73
5.4.2 Create . . . 74
5.4.3 Lookup . . . 74
5.5 Correctness . . . 75
5.5.1 Initial bulk operation processing . . . 75
5.5.2 Bulk operation instantiation . . . 77

6 Evaluation **80**
6.1 Experimental setup . . . 81
6.1.1 Data collection and instrumentation . . . 81
6.1.2 Workload scripting . . . 82
6.2 Baseline behavior . . . 84
6.2.1 Database access experiment . . . 84
6.2.2 Capability experiments . . . 85
6.2.3 Create . . . 87

6.2.4 Write . . . 89
6.2.5 Read . . . 98
6.3 BulkDelete . . . 102
6.4 BulkClone . . . 106
6.4.1 Comparing chains-of-clones and prolific clones . . . 106
6.4.2 Access after BulkClone . . . 106
6.5 Background instantiation . . . 109
6.5.1 Non-competitive background instantiation . . . 111
6.5.2 Background instantiation with paced foreground workload . . . 112
6.5.3 Random bulk operation background instantiation . . . 117
6.5.4 FIFO bulk operation background instantiation . . . 119
6.5.5 LIFO bulk operation background instantiation . . . 124
6.5.6 Widest span of objects bulk operation background instantiation . . . 128
6.5.7 Thinnest span of objects bulk operation background instantiation . . . 131
6.6 NFS server . . . 135
6.6.1 Baseline behavior . . . 135
6.6.2 PostMark . . . 137
6.7 Summary . . . 140

7 Conclusions and future work 142

Bibliography 148

A Experimental results 156

A.1 Baseline behavior results . . . 157
A.1.1 Database access experiment results . . . 157
A.1.2 Capability experiment results . . . 158
A.1.3 Create experiment results . . . 160
A.1.4 Write experiment results . . . 164
A.1.5 Read experiment results . . . 179
A.2 BulkDelete experiment results . . . 187

A.3 BulkClone experiment results 189
A.4 Background instantiation experiment results 193
A.5 NFS server baseline results 221
A.6 PostMark experimental results 229

List of Tables

6.1 Write() operation differences with bulk operations . . . 93
6.2 Re-Read() operation differences with bulk operations . . . 99
6.3 Paced Create after BulkDelete with background instantiation . . . 113
6.4 Paced Read after BulkClone with background instantiation . . . 116
6.5 NFS File Create/Clone/Delete benchmark summary . . . 136
6.6 Summary of PostMark results . . . 139

A.1 Ping metadata server back-end database . . . 157
A.2 Acquiring storage node addressing information . . . 158
A.3 Capability revocation . . . 159
A.4 Create(), sequential, no bulk operation code, no background instantiation . . 160
A.5 Create(), random, no bulk operation code, no background instantiation . . . 161
A.6 Create(), sequential, bulk operation code, no background instantiation . . . 162
A.7 Create(), random, bulk operation code, no background instantiation . . . 163
A.8 Lookup(), sequential, no bulk operation code, no background instantiation . 164
A.9 ApproveWrite(), sequential, no bulk operation code, no background instantiation . . . 165
A.10 SSIO_Write(), sequential, no bulk operation code, no background instantiation . . . 166
A.11 Finish_Write(), sequential, no bulk operation code, no background instantiation . . . 167
A.12 Write(), sequential, no bulk operation code, no background instantiation . . 168

A.13 Lookup(), sequential, active bulk operation code, no background instantiation 169
A.14 ApproveWrite(), sequential, bulk operation code, no background instantiation 170
A.15 SSIO_Write(), sequential, active bulk operation code, no background instantiation . . . 171
A.16 Finish_Write(), sequential, bulk operation code, no background instantiation 172
A.17 Write(), sequential, bulk operation code, no background instantiation 173
A.18 Re-Write(), sequential, no bulk operation code, no background instantiation 174
A.19 Re-Write(), revoked caps, sequential, no bulk ops, no background instantiation . . . 175
A.20 Cache-hit re-Lookup(), seq, no bulk ops, no background instantiation 176
A.21 Re-Lookup() to MDS, seq, no bulk ops, no background instantiation 177
A.22 SSIO_Write() after revoke, seq, no bulk ops, no background instantiation . . 178
A.23 Read(), sequential, no bulk operation code, no background instantiation . . 179
A.24 Read(), sequential, no bulk operation code, no background instantiation, invalid caps . . . 180
A.25 Lookup(), fast, bad caps, sequential, no bulk operation code, no background instantiation . . . 181
A.26 SSIO_Read(), fast, bad caps, sequential, no bulk operation code, no background instantiation . . . 182
A.27 Lookup(), slow, bad caps, sequential, no bulk operation code, no background instantiation . . . 183
A.28 SSIO_Read(), slow, bad caps, sequential, no bulk operation code, no background instantiation . . . 184
A.29 Read(), sequential, bulk operation code, no background instantiation, invalid caps . . . 185
A.30 Lookup(), slow, bad caps, sequential, with bulk operation code, no background instantiation . . . 186
A.31 BulkDelete of single objects, sequential, no background instantiation 187
A.32 Create after BulkDelete of single objects, sequential, no background instantiation . . . 188

A.33 Repeated BulkClone of 1000 objects, prolific 189
A.34 Repeated BulkClone of 1000 objects, chain-of-clones 190
A.35 Read of BulkClone source objects 191
A.36 Read of BulkClone destination objects 192
A.37 Create after BulkDelete and completed background instantiation 193
A.38 Read after BulkClone and completed background instantiation 194
A.39 Create after BulkDelete 1:1 with Sleep 195
A.40 Create after BulkDelete 1:3 with Sleep 196
A.41 Create after BulkDelete 1:7 with Sleep 197
A.42 Read after BulkClone 1:1 with Sleep 198
A.43 Read after BulkClone 1:3 with Sleep 199
A.44 Read after BulkClone 1:7 with Sleep 200
A.45 Random Read after BulkClone with Random background instantiation . . . 201
A.46 Sequential Read after BulkClone with Random background instantiation . . 202
A.47 Random Create after BulkDelete with Random background instantiation . . 203
A.48 Sequential Create after BulkDelete with Random background instantiation . 204
A.49 Random Read after BulkClone with FIFO background instantiation 205
A.50 Sequential Read after BulkClone with FIFO background instantiation . . . 206
A.51 Random Create after BulkDelete with background instantiation via FIFO processing 207
A.52 Sequential Create after BulkDelete with background instantiation via FIFO processing 208
A.53 Random Read after BulkClone with LIFO background instantiation 209
A.54 Sequential Read after BulkClone with LIFO background instantiation . . . 210
A.55 Random Create after BulkDelete with background instantiation via LIFO processing 211
A.56 Sequential Create after BulkDelete with background instantiation via LIFO processing 212
A.57 Random Read after BulkClone with background instantiation of widest range213

A.58 Sequential Read after BulkClone with background instantiation of widest range . . . 214
A.59 Random Create after BulkDelete with background instantiation via widest range processing . . . 215
A.60 Sequential Create after BulkDelete with background instantiation via widest range processing . . . 216
A.61 Random Read after BulkClone with thinnest range background instantiation 217
A.62 Sequential Read after BulkClone with thinnest range background instantiation . . . 218
A.63 Random Create after BulkDelete with background instantiation via thinnest range processing . . . 219
A.64 Sequential Create after BulkDelete with background instantiation via thinnest range processing . . . 220
A.65 NFS File Remove . . . 222
A.66 NFS File clone via copy . . . 223
A.67 NFS File Remove after clone via copy . . . 224
A.68 NFS prolific BulkClone . . . 225
A.69 NFS File Remove after prolific BulkClone . . . 226
A.70 NFS chain-of-clones BulkClone . . . 227
A.71 NFS File Remove after chain-of-clones BulkClone . . . 228
A.72 PostMark with no bulk operations . . . 229
A.73 PostMark with Clone via Copy . . . 230
A.74 PostMark with prolific clones . . . 231
A.75 PostMark with chains-of-clones . . . 232
A.76 PostMark with prolific clones and random background instantiation . . . 233
A.77 PostMark with prolific clones and LIFO background instantiation . . . 234
A.78 PostMark with prolific clones and thinnest range background instantiation . 235
A.79 PostMark with chains-of-clones and random background instantiation . . . 236
A.80 PostMark with chains-of-clones and LIFO background instantiation . . . 237

A.81 PostMark with chains-of-clones and thinnest range background instantiation . . . 238

List of Figures

2.1 System model 12
2.2 Capability acquisition and use 14
2.3 Multiple data distributions for a single object 15

3.1 Two manners of using the clone operation 27
3.2 Bulk operation process 30

4.1 NFS server as storage system client 43
4.2 Directory contents after cloning 45
4.3 Create and Re-Create with BulkDelete 48
4.4 Lookup with BulkClone 49
4.5 Protocol for Enumerate 50
4.6 Protocol for Delete 51
4.7 Write protocol with allocation 52
4.8 Write triggering instantiation of a clone 53
4.9 Read triggering instantiation of a clone 54
4.10 BulkDelete 55
4.11 BulkClone 56

5.1 Prolific clones 60
5.2 A chain-of-clones 61
5.3 Pass-through clone example 67
5.4 Use case for InstantiateHole function 69

5.5 Initial bulk operation processing 76
5.6 Instantiation of a bulk operation 77

6.1 Create operation timing comparison 88
6.2 Write protocol with allocation and *extra* Lookup 91
6.3 Write timing comparison 96
6.4 Re-Write timing comparison 97
6.5 Read timing comparison with invalid capabilities 100
6.6 Re-Read timing comparison 101
6.7 BulkDelete and re-Create of pre-existing objects 102
6.8 Components of Create at the Metadata Server 105
6.9 Components of the Instantiate_Hole subroutine 105
6.10 Read of source objects after BulkClone performing on-demand instantiation 108
6.11 Read of destination objects after BulkClone performing on-demand instantiation 108
6.12 Experiment description for create, write, clone, and read of objects. 110
6.13 Experiment description for create, write, delete, and read of objects. 111
6.14 Paced Create after BulkDelete. 114
6.15 Paced Read after BulkClone. 115
6.16 Random Read after BulkClone with Random background instantiation . . . 118
6.17 Sequential Read after BulkClone with Random background instantiation . . 118
6.18 Random Create after BulkDelete with Random background instantiation . . 120
6.19 Sequential Create after BulkDelete with Random background instantiation . 120
6.20 Random Read after BulkClone with FIFO background instantiation 121
6.21 Sequential Read after BulkClone with FIFO background instantiation . . . 121
6.22 Random Create after BulkDelete with FIFO background instantiation . . . 123
6.23 Sequential Create after BulkDelete with FIFO background instantiation . . 123
6.24 Random Read after BulkClone with LIFO background instantiation 125
6.25 Sequential Read after BulkClone with LIFO background instantiation . . . 125
6.26 Random Create after BulkDelete with LIFO background instantiation . . . 127
6.27 Sequential Create after BulkDelete with LIFO background instantiation . . 127

6.28 Random Read after BulkClone with background instantiation of widest range 129
6.29 Sequential Read after BulkClone with background instantiation of widest range 129
6.30 Random Create after BulkDelete with background instantiation of widest range 130
6.31 Sequential Create after BulkDelete with background instantiation of widest range 130
6.32 Random Read after BulkClone with thinnest range background instantiation 132
6.33 Sequential Read after BulkClone with thinnest range background instantiation 132
6.34 Random Create after BulkDelete with background instantiation of thinnest range 134
6.35 Sequential Create after BulkDelete with background instantiation of thinnest range 134
6.36 PostMark configuration for NFS experiments. 138

Chapter 1

Introduction

Distributed, object-based storage has come into its own through the past decade. Pioneered as the NASD project in the mid- to late-1990's [29], various incarnations of this storage model have been discussed in academic [1, 30, 78, 80] and industrial [6, 8, 19, 25, 28, 43, 56, 57, 66] circles. Characteristics of these systems include (1) variably sized objects as containers for data storage, (2) a large, flat namespace for object naming, (3) capabilities for access control, (4) direct access by clients to storage nodes for scalable performance, and (5) a metadata server for control of object metadata and the namespace.

New challenges arise for storage management in this architecture. Management operations in traditional storage systems have a centralized location through which all accesses pass. For instance, a network attached file server serves as a centralized arbiter of access and, during client accesses, can ensure that management operations across all files are enforced. In a distributed, object-based storage system, en masse operations on objects require coordination among the three prime members of the distributed, object-based storage system: clients, storage nodes, and the metadata server. In multi-tenant situations, where many customers simultaneously operate upon a shared storage system, the isolation of management operations for performance and access-control reasons becomes a storage management concern.

1.1 Distributed, object-based storage

This dissertation specifically addresses issues related to distributed, object-based storage systems. These systems are distributed insofar as their components are interconnected by a communication network and need not be physically close together (although proximity is likely). Common distributed systems problems are faced: communication, agreement, authentication, authorization, fault-tolerance, etc. Objects are the containers for storage in these systems. An object has an integer name from a large namespace (e.g., 128 bits) and a byte-addressable data component. Such systems consist of three primary participants: clients, metadata server, and storage nodes. Clients drive data access by creating, deleting, reading and writing objects. The metadata server authorizes client actions by responding to access requests with metadata and capabilities (software tokens denoting access permission) and controlling the existence of objects. Storage nodes respond to client requests submitted with valid capabilities.

Persistent storage of the information in an object is a responsibility split between the separate, but intricately interrelated, components of the distributed, object-based storage system. As a first consideration, the simple existence of an object is controlled. This is the proper job of a namespace server: recording which objects exist, or do not exist, within the namespace. For our purposes, we will combine the functioning of the namespace server and metadata server, referring only to the metadata server throughout the rest of this dissertation. As a second point, the metadata server is responsible for tracking of information about how the data of an object is stored. The metadata consists of such items as the length of the object's data component and the names of the storage nodes upon which portions of that data are stored. The third aspect of persistent storage concerns the storage nodes. They are the ultimate repository of the information in an object's data component and are responsible for persistent storage (i.e., placing the data on a storage device and retrieving it when requested). The clients, of course, populate objects with data based on metadata stored at the metadata server and write that data to storage nodes. Coordinating actions among all of these components complicates efforts to perform an operation upon a set of objects.

1.2 Storage management and operations upon object sets

Storage management involves the care and maintenance of storage systems to meet performance, availability and reliability goals expected for accessing the stored data. Tasks for meeting these goals include, but are not limited to, selection of data storage encodings (e.g., RAID levels), migration of data between storage devices, reclamation of space, management of access, control of quotas, and replication for enhancing read-only performance, disaster tolerance, and archive. In common network-attached file system servers, storage management tasks are concentrated at that file server. The less-centralized nature of distributed, object-based storage systems complicates matters. Without centralized control or complex distributed algorithms, such systems could struggle to perform management tasks.

Applying management tasks to large portions of a storage system is common. For example, an entire file system might be copied to an off-site location for disaster recovery purposes. Or, a block-storage volume might be destroyed to reclaim or re-purpose storage system capacity. A volume-based storage system, like AFS [40], could move a volume between servers for load-balancing. For a distributed, object-based storage system, the set of objects upon which to operate must be an argument to the calls initiating management tasks.

This dissertation explores the approach of bringing centralized control of distributed, object-based storage systems to the metadata server where management tasks can be performed. We refer to the operations supporting storage management tasks as *bulk operations* and target them against sets of objects. We desire the two key characteristics from the bulk operations. First, they should quickly respond to requests for execution. Second, they should operate atomically upon the affected objects. This all-or-nothing semantic simplifies error handling should there be a problem with execution.

1.3 Thesis statement

This dissertation shows that . . .

> *A distributed, object-based storage system can provide atomic bulk operations on compactly described and externally defined object sets using delayed instantiation.*

This is shown by the following sequence of steps:

1. Describing compact object set description approaches and how they can be used for bulk operations in a distributed, object-based storage system.

2. Demonstrating delayed instantiation bulk operations in a prototype system.

3. Showing that background instantiation can be used to reduce the impact of delayed instantiation bulk operations on subsequent accesses.

1.4 Bulk operations for storage management

The aforementioned bulk operations assist with storage management tasks along several dimensions in distributed, object-based storage systems. First, they provide an interface through which groups of objects can be manipulated. Second, they encapsulate the complexity of the tasks in a way similar to a programming library interface: an easy-to-use interface and simple semantics hide complex behind-the-scenes machinations. Third, by grouping related objects together, an administrator can more easily manage portions of a shared storage infrastructure (e.g., shared between distinct administrative or functional entities in a business). Fourth, the atomic action of such operations simplifies understanding of their effects, removes chances of side-effects, and allows for their confident use by storage system administrators.

Given the complexity of storage systems, it behooves the architect to provide simple ways for users to perform their tasks. The availability of bulk operations as a tool for

an administrator fills a sure need for storage management tasks. Without the sort of pre-packaged bulk operations we examine, an administrator might forego the purchase of a storage system or be forced to find another source for their implementation. The source may come in the form of a commercially available software toolkit or home-made scripts – neither of which are likely to be as efficient as if such capabilities were provided within the storage system. With bulk operations included in a storage system at the point of initial design, efficient implementation and best-practices for use can be developed simultaneously and offered to users.

Our implementation of bulk operations exploits a disconnect between the semantics of execution and the instantiation of the intended effects of the operation. At execution time, the storage system promises to honor an operation. However, it is not necessarily required to immediately act on implementing the effects of the operation. The storage system only needs to enforce that the intended effects of the operation are observable when clients investigate. Therefore, the work of actually instantiating the operation upon an object can be delayed until such time as a client would observe the effects of the operation. This strategy of *delayed instantiation* enforces operation semantics from a client-observable viewpoint rather than an *immediate execution* strategy, which would perform all necessary work before returning a “success” code to the caller.

Let us take as an example the interaction of a bulk operation that deletes a group of objects and an object create operation that fails if the target object to be created already exists. Consider this situation: an object exists, is written and read, and then falls within the group affected by a newly executed bulk delete operation. At this point, the storage system may delay the instantiation of the delete operation upon the object; it can wait to free the backing store of the object’s data component. It must only prevent access to the existing data and disallow any listings of the object’s existence in the storage system’s namespace. When a client wishes to create an object with the same name as the deleted object, the storage system must free the backing store of the previous incarnation of the object before allowing the new create operation to succeed. Thus, the semantics of the bulk delete operation have been preserved and the instantiation of the operation delayed until a client would come in conflict with the semantics.

1.5 Roadmap

The remainder of this dissertation is organized as follows. Chapter 2 describes related work and the components of a distributed, object-based storage system. Chapter 3 presents issues related to the design and implementation of bulk operations and describes the two upon which we concentrate. Chapter 4 details the inner workings of our prototype client, storage node, and metadata server that support bulk operations, describes the background instantiation algorithms used to mitigate the costs of bulk operations, describes how an NFS server can work with bulk operations, and presents the operations exposed to clients. Chapter 5 describes the data structures and algorithms that support our implementation of delayed instantiation bulk operations. Chapter 6 presents baseline behavior of the system, calculates costs associated with the particular setup, evaluates the effects of bulk operations and background instantiation, and shows how they are used by our NFS server. Chapter 7 summarizes our findings and describes possible future work. Appendix A includes tables with summary statistics of operation execution times for experimental runs, benchmark configuration, and raw data from experiments involving our NFS server.

Chapter 2

Background and related work

The bulk operations presented here provide support for the creation of selective snapshots and forks within distributed, object-based storage systems. To provide background for understanding of the system, this chapter describes object-based storage and storage management using snapshots and forks. The chapter concludes with descriptions of parts of the particular storage system used for the realization of bulk operations.

First, we relate the system and thesis we are investigating to related work. Then, we describe the system components.

2.1 Distributed, object-based storage

Distributed, object-based storage has evolved since the mid-1990's to address scaling, performance, and management issues with traditional block and file system storage system architectures. Promulgated as the NASD system [29, 31], multiple storage nodes were connected to computers using the storage via a communication network. With the network as a scalable interconnect, storage nodes could be added to scale the capacity and the bandwidth to the aggregate limits of the network and storage nodes [32].

The object model, as a container for data storage, assists this vision by acting as an intermediary data structure between the primitive interface of raw blocks and sophistication of file systems. Block storage is exemplified by the SCSI interface [70], where all storage

locations on a device are blocks (consisting of a number of bytes) addressed by a block number. Access to a block of data reads or writes it in its entirety. File systems provide an abstraction of a file that can be accessed at byte granularity and navigated through directories (containers for files and other directories). Object-based storage fills the gap, pointed out in 1980 [24], in the storage system hierarchy between files and blocks. This is further elaborated upon in the IEEE Mass Storage System reference model [21]. Many examples of file systems have been built to use object-based storage in academia [1, 30, 78, 80] and in industry [6, 8, 19, 25, 28, 56, 57, 65, 66]. The standardization of an interface to object-based storage has assisted efforts at expanding the applicability of the technology [43].

Object-based storage systems share some common characteristics. Unlike traditional hierarchical file systems ([23]), the namespace is flat in object-based storage and consists of a number taken from some set of integers. Allowance is made for direct client access to the storage, taking away the bottleneck of traditional file servers [28–30, 34, 35, 64]. A metadata server takes the place of that traditional file server to control access to storage. To allow the metadata server to control access to the storage without being queried for every access, most systems use capabilities, a type of authorization token popularized in the 1970's [5, 10].

The object-based storage discussed in this dissertation is different from object-oriented programming and persistent object systems as described in these sources [4, 7, 9, 22, 41, 51, 59, 74]. Garbage collection and inheritance of properties between objects, for example, are design goals of object-oriented systems that we are not concerned with.

2.2 Broader references

Various academic or industrial research projects have supported snapshots and forks in various forms; we only present a few of these here although the features are widespread. The *Venti* archival storage system never overwrites data, allowing for snapshot file systems to be easily built on top of it [61]. Snapshots of block-based distributed storage were supported in Petal [50], which was used to build the Frangipani [76] file system. The "Mime" block-based storage system from Hewlett Packard also supported snapshots [16].

The Federated Array of Bricks (FAB [27]) prototype at Hewlett Packard implements a distributed agreement protocol to quickly make snapshots and forks of its distributed block-based storage [3].

Snapshot support in databases evolved from the use of logging for transaction support [2], as the transactions could be rolled-back to any point in time to examine the contents of the database [55, 72]. Databases also inspired the atomic transactional nature of our bulk operations [36].

The atomicity of database transactions transitioned into the realm of data storage systems. Various systems from the 1980s supported atomic actions on groups of files [12, 33, 52, 53, 62, 63, 77]. A summary of many such atomic update storage systems is presented in [75]. Later, atomic updates were applied to logical disks underlying file systems [37].

The roots of the experimental clustered, object-based storage system presented in this dissertation can be found in a distributed block-based storage system. That system, PASIS [1, 81], was built atop versioning storage nodes and presented a protocol that ensured consistent access to clients in the face of access concurrency.

We have also borrowed the concept of access capabilities that were applied to storage systems in the 1970's [5, 10] for controlling access and metadata freshness. They have become fairly standard features of systems where clients can directly access object-based storage [28–30, 34, 35, 64].

Our use of external assignment of object identifiers from a flat namespace fits a possibility for naming outlined in various resource-naming taxonomies [47, 68, 82]. The practice of encoding information in object identifiers, in our case to indicate directory and snapshot/-fork membership, has been used in storage systems for some time to indicate information about the storage locations of data [20, 59]. It is common practice for NFS [14, 58, 73] servers to embed inode numbers in the filehandles (used to uniquely identify files) exchanged with clients; in fact, we plan to use object identifiers as primary components of filehandles.

Similar to the distributed systems problem of providing serialized and consistent access, creating snapshots (distributed agreement of state) is also a general problem [15, 71]. By centralizing the bulk operation instantiation responsibility in the object manager, this

storage system avoids distributed system problems associated with distributed consensus algorithms [13, 38, 48, 49, 60, 69].

2.3 Cloning, snapshots and storage management

For object-based storage to compete in the marketplace, it must approximate the features of established products. One feature that is trickling-down from high-end to mid-range storage systems is the ability to make a point-in-time copy of a storage system [67]. These copies are variously called snapshots, forks and clones. For our purposes, a *snapshot* will be a read-only copy, a *fork* will be a read-write copy, and a *clone* can refer to either.

These storage system clones ease various management tasks associated with business-critical storage systems. Copies of data are commonly used for disaster prevention/recovery, archive, error recovery and system testing. With a clone of storage, a consistent view can be statically maintained for the duration of a copy operation. A clone operation that executes quickly, while storage is online, and atomically in the face of concurrent access, is of great value for these purposes. Early implementation of clone operations occurred in the Andrew File System [40]. Notable commercial success with storage system snapshots was achieved by NetApp, nee Network Appliance [39]. Making system snapshots with write-once storage has also been shown to be feasible [61].

A standard for OSD-2 has propagated through the standards committees of the SNIA Object-Based Storage Devices working group and ANSI T-10 [42, 79]. Standardized commands allow for the creation of "snapshots" and "clones" of partition objects. These objects are linked together with special entries in their "attribute pages" to form chains [11, 43]. When creating these new partitions, options exist for them to be created as copy-on-write versions of their source or as complete byte-by-byte copies. The commands can be long-running with completion noted as a percentage observable in an attribute of the partition. That work differs from this dissertation's approach in four ways. First, we are concentrating on coordinating a "snapshot" or "clone" operation across multiple storage nodes and using heterogeneous data distributions, while their specification applies to a single storage device. Second, we provide for atomicity of a rapidly executed operation, while they spec-

ify a long-running operation that may partially complete. Third, we do not expose tracking between source and destination object sets, while they have attribute pages to indicate that history. Fourth, we allow for operations to occur on any objects in the storage system, while they operate only upon "partition objects" which contain many objects.

2.4 System components

The storage system for which we are constructing bulk operations serves data stored in objects and accessed through cooperation between three primary components: clients, storage nodes and a metadata server. These components are connected by a network and the algorithms for successful operation are distributed across each of the components involved in an operation.

Clients exchange metadata with the metadata server and directly access data on the storage nodes. The metadata server manages metadata and controls access by employing a capability-based access control mechanism. Storage nodes execute client requests to read or write data when presented with valid capabilities.

We next present information about the object model for the system architecture on which we focus, and the data distributions by which the data for an object is stored. What follows are component descriptions addressing the characteristics, responsibilities and functioning of each component, along with example programming interfaces. These descriptions are provided to set the stage for the introduction of bulk operations into a distributed, object-based storage system built around these components.

2.4.1 Objects

An *object* is a named container for a byte-stream. An object has a *name*, or *object identifier* (OID), selected from an *object identifier namespace*. The object identifier namespace, or *namespace*, consists of unsigned integers of some power-of-two number of bits in size. For this work, we assume the object identifiers of 128 bits. For this work, we assume that the object names are selected by clients.The methods described for implementing bulk operations later in the dissertation rely on this aspect of object naming.

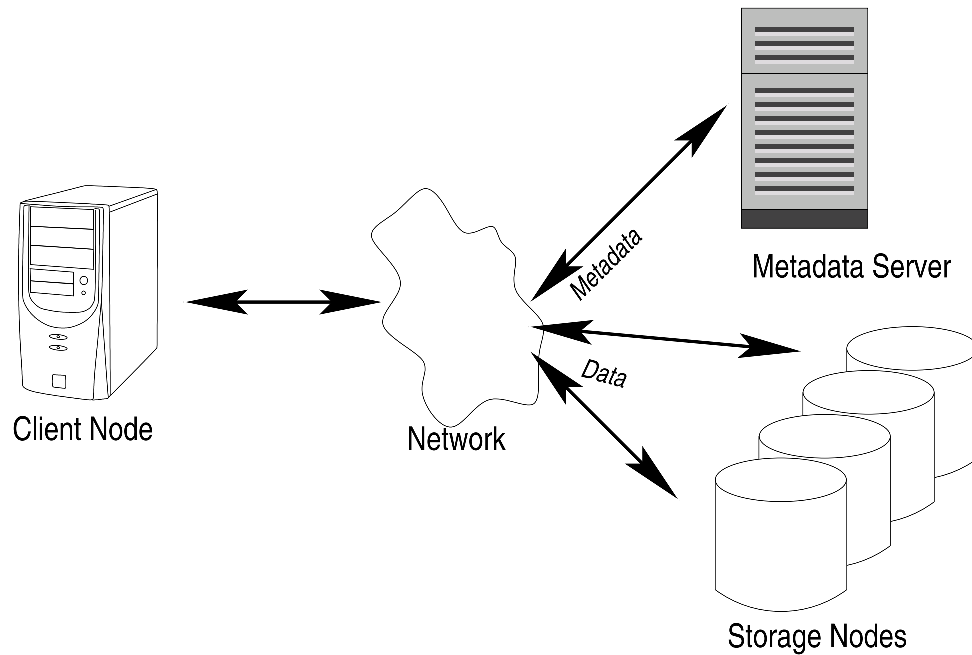


Figure 2.1: System model
A distributed, object-based storage system consists of a metadata server, clients, and storage nodes. Clients exchange metadata with the metadata server before interacting directly with storage nodes to access data.

The *byte stream* component of an object contains raw data that can be written and read. Bytes are addressed starting first at the zero byte and proceeding to the end of the byte-address space. An object's byte stream can be addressed similarly to that of a file, from bytes 0 through $2^{64} - 1$ (for a 64 bit address space). When accessing the byte stream, the object identifier of interest is supplied along with the desired byte stream offset and an indication of the number of bytes of information to be accessed. In this sense, the object identifier serves as a sort of file handle or file descriptor, and addressing of the bytes of the object proceeds just as is done with files.

The attributes of an object may be very limited. We assume that attributes consist of the object identifier, the length of the byte stream, and a logical create time at the very least. The object identifier and byte stream have been discussed in the preceding paragraphs. The

logical create time is maintained to establish a relative ordering for operations that impact the evolution of the object identifier namespace. There is a single logical clock used in the system to timestamp the creation of objects and the execution of bulk operations. Using these timestamps for comparison is a key component of the bulk operation algorithms used to disambiguate cases of object (non-)existence.

2.4.2 Capabilities

Our implementation of bulk operations relies on the use of capabilities to control access to objects. A *capability* is a token (represented by a sequence of bytes) presented along with a request to prove that authorization has been given to perform an action. In the physical world, a movie ticket can be considered a capability: you present the ticket to prove that you have access to a particular theater to view a movie. In the distributed storage system sense, a capability is a data structure authorizing access to an object.

For the purposes of this dissertation, there are two important aspect of capabilities. The first is their ability to authorize clients' access to data on the storage nodes. The second is the ability of the metadata server to revoke that access by contacting the storage nodes.

In distributed, object-based storage, capabilities are completely under the control of the metadata server as shown in Figure 2.2 on page 14. The metadata server is the only entity that may issue and revoke capabilities. Clients request capabilities when they acquire metadata to access objects at the storage nodes. The storage nodes check that capabilities are valid when they receive them along with a read or write request. When necessary, the metadata server contacts storage nodes and informs them that some capabilities should no longer be honored. If a client attempts to use an invalid capability, it is informed that its request cannot be completed because the capability is invalid. The logic of the client library, when receiving such a message, contacts the metadata server for a fresh capability and metadata before retrying an operation.

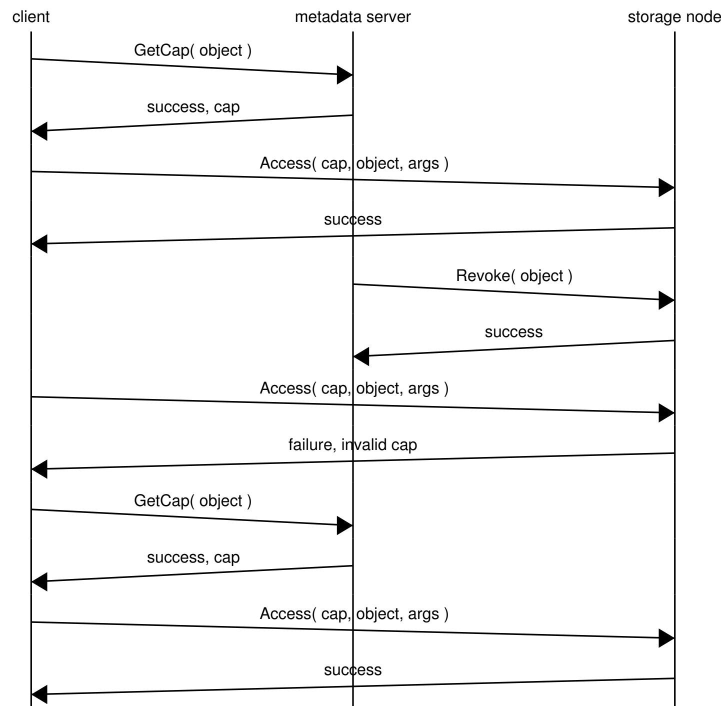


Figure 2.2: Capability acquisition and use
This protocol exchange shows a client acquiring and using capabilities. The metadata server then revokes capabilities. When the client uses invalid capabilities, it re-acquires them and continues with access.

2.4.3 Data distribution

An *extent* is a range of bytes within the byte stream of an object. Each extent can have its own *data distribution* which specifies a block size, a block encoding algorithm, a set of threshold encoding parameters, and an ordered list of storage node addresses. The information of a data distribution is used by the client library to read and write data. It is stored in the metadata server, where it can be accessed by clients. An example data distribution for an object is shown in Figure 2.3 on page 15

The *block size* of a data distribution specifies the number of bytes over which the block encoding algorithm will be applied. The block size is variable so that it might match the common access sizes of client applications.

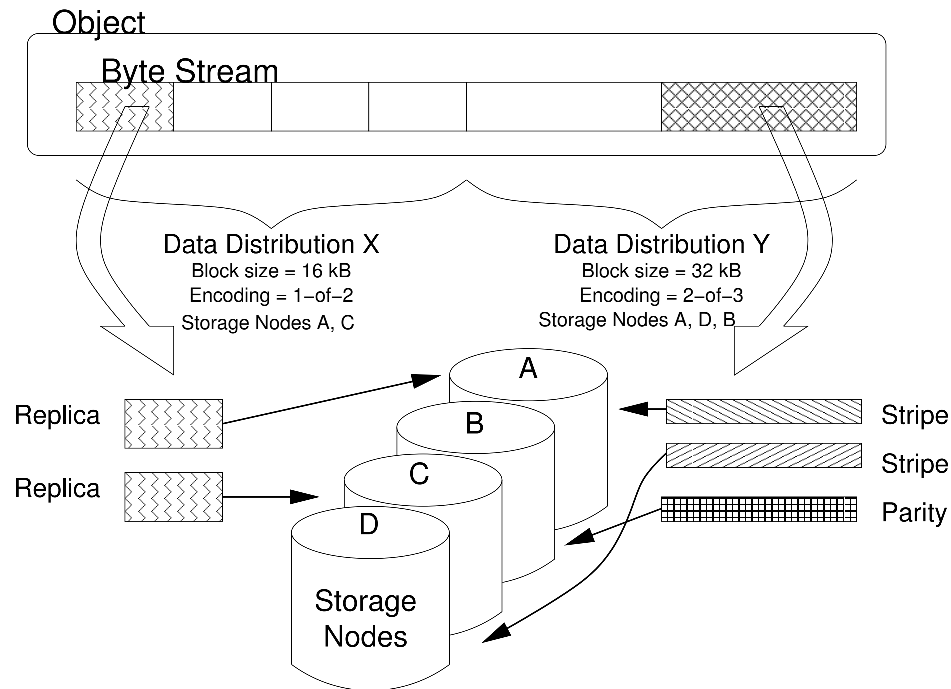


Figure 2.3: Multiple data distributions for a single object
A pictorial example of the data stream of a single object whose data stream has two extents and hence two different data distributions. Of the six total blocks comprising the data stream, four have one distribution and two have another distribution. The resulting fragments for one block of each distribution are shown: distribution X fragments are the same size as the block because each holds a full copy of the data, distribution Y fragments are each one-half the size of the block because of the 2-of-3 encoding. The resulting fragments are spread across the storage nodes of the distributed, object-base storage system.

The *block encoding algorithm* is an enumerated type that acts as an index into a set of functions for encoding and decoding the data. It is parameterized by the threshold encoding parameters. Different algorithms (e.g., replication, striping and parity, and secret sharing) are available to match performance and confidentiality requirements.

The *threshold encoding parameters* describe the number of equal sized *fragments* (the n parameter) into which the block is broken and, in the simplest case, the number of those n fragments necessary to reconstruct the data during a Read operation (the m parameter).

These *m-of-n* encodings, where m is less-than or equal-to n, break a block into n fragments of which m must be fetched during a successful Read operation. For any block of data, there are n fragments of size $\frac{block_size}{m}$ and so $\frac{n*block_size}{m}$ bytes are stored, incurring a *storage blow-up* of $\frac{n}{m}$. As familiar examples of these $m-of-n$ parameters, consider mirroring, striping and parity schemes. Mirroring is a *1-of-n* configuration that has a blow-up of $\frac{n}{1}$ and where any one fragment constitutes a complete copy of the data for a block. Striping is a *n-of-n* configuration, where the data is broken up and spread evenly across storage has a blow-up of $\frac{n}{n} = 1$. Parity schemes are more rightfully termed *m-of-n* schemes and provide redundant information to the tune of the difference between m and n, usually with m fragments of striped data and $(n-m)$ fragments of parity data.

The *list of storage nodes* provides the information necessary for a client to contact the correct storage nodes when reading or writing data. Some block encoding algorithms, like parity schemes that prefer to access the striped fragments and save on parity calculations, use this ordered information to opportunistically take short-cuts through their encode/decode code paths.

2.4.4 Client

Clients are the driving force of the storage system; very little happens that is not directly driven a client. There may be many clients, or few; they could reside on one physical computer or on many. They might use the storage system directly (e.g., a database storing tables and indices in objects) or indirectly on behalf of others (e.g., an NFS server storing files and directories in objects). Clients are assumed to be heterogeneous and are connected to the storage nodes and metadata server via a network.

Clients view the distributed object-based storage system as a set of named objects with each object providing a byte-addressable data portion. They drive the evolution of the object-identifier namespace by creating and deleting objects. When creating objects, the client specifies the object identifier to be used. They can observe the state of the namespace by enumerating its contents. The byte-stream component of objects is modified with write commands and observed with read commands.

Internally, a client translates calls to its own API into calls to the metadata server and to storage nodes. Creating and deleting of objects are operations on the metadata server. When reading or writing objects, metadata manipulation is done via communication with the metadata server and data access, using that metadata, proceeds directly between the client and the appropriate storage nodes. Metadata cached at the client contains a capability that is sent along with requests to storage nodes. Provisions are made for clients to choose the data distribution of the data portion of objects.

API

These are the functions that a client application can call to interact with the storage system.

Create(OID) If the object does not already exist in the object identifier namespace, add it as an empty object of zero-length.

Delete(OID) If the object exists, remove it from the object identifier namespace and make its data inaccessible.

Enumerate(start_OID, end_OID, num_objects) Return a list of at most num_objects starting from start_OID up to end_OID.

Write(OID, offset, length, buffer) Write to the byte-stream component of object OID at the given offset the contents of buffer up to the given length.

Read(OID, offset, length, buffer) Read the byte-stream component of object OID at the given offset and for the given length; return the data in the provided buffer.

2.4.5 Storage node

Storage nodes are, primarily, passive elements in the storage system. They may be heterogeneous in capacity, throughput and responsiveness. A storage node may protect data by storing redundant information, or it may not. A network connects storage nodes to the other system components and to each other. They process data requests received from clients. A

storage node uses information received from the metadata server when making decisions about allowing client access to proceed.

Storage nodes are responsible for storing data. They react to requests received from clients to read or write data. A request to write an object for which a storage node does not already hold data results in an implicit “create” of that object on the storage node. Capabilities sent along with the requests serve as authorization for the operation. If the storage node receives invalid capabilities along with a client request, the operation is not performed and an error message is returned.

The metadata server controls access to data by exercising control over the validity of capabilities and by removing objects from storage nodes. The metadata server may contact storage nodes to inform them that some set of capabilities that may be held by clients are no longer valid. These invalidation requests cause the storage node to update its capability tracking information. Deleting objects from storage nodes is explicitly performed by messages sent by the metadata server.

Internally, a storage node tracks three kinds of information. First, it knows the object identifiers for which it holds data. When a valid write request arrives for an as-yet-unknown object identifier, an implicit “create” of the object will occur. Second, for each data location of the objects, the storage node associates some data. This data is set by write commands and can be retrieved with read commands. Third, storage nodes track information about which capabilities are valid and which are invalid. This information is consulted for each read or write access; any use of invalid capabilities returns an error to the calling client.

API

This is the set of commands addressable to storage nodes. Clients use the SSIO_Write() and SSIO_Read() commands. The MDS uses the Delete() and Revoke() commands.

SSIO_Write(capability, OID, location, buffer) For the object with the given OID, store the contents of buffer at the given location if the capability is valid. This is a low-level command used by the client-visible Write command.

SSIO_Read(capability, OID, location, buffer) For the object with the given OID, return in the given buffer the data maintained for the given location if the capability is valid. This is a low-level command used by the client-visible Read command.

Revoke(OIDs) For the objects with the named OIDs, ignore all capabilities granted previously.

Delete(OID) Make all data locations maintained for the given OID become inaccessible.

2.4.6 Metadata server

The metadata server is a centralized entity that manages the object-identifier namespace and the metadata for each object. There may be many systems cooperating in a cluster to act as a single logical metadata server, or there may be a single system. There is network connectivity between the metadata server and all clients and all storage nodes.

The metadata server is responsible for managing the object-identifier namespace and the metadata for each object. Information about the (non-)existence of objects is maintained. For each existing object, metadata is stored. Clients are forced to use common data distributions when concurrently writing regions of objects through the use of an intention logging system. An intent to write data to a new region of an object using a particular data distribution is logged via the ApproveWrite operation. If successful, this returns a capability. If a client proposes an ApproveWrite with a data distribution that conflicts with an already successful ApproveWrite, the already successful data distribution is returned and the client may then re-try with the new information. After writing to storage nodes, clients execute FinishWrite calls to indicate that data for a region of an object is now available.

The metadata server maintains three sets of information. First, there is an object database that tracks the existence of objects, which is modified whenever objects are created or deleted. Second, there is a metadata database to hold metadata information for those existing objects that contain data. Third, there is a PendingWrite database holding the set of approved metadata for writing to new regions of existing objects. Each ApproveWrite operation can add metadata here, and FinishWrite operations move that metadata to the metadata database. Delete operations result in capability revocation messages being sent to

storage nodes, but actual removal of data at storage nodes can wait until an object identifier is re-used.

API

These are the commands to which the metadata server responds.

Create(OID) Add an object with the given object identifier to the namespace.

Delete(OID) Remove the object with the given object identifier from the namespace, making any of its data inaccessible.

Enumerate(start_OID, end_OID, num_objects) Return a list of at most num_objects starting from start_OID up to end_OID.

ApproveWrite(OID, data_distribution) Log an intent to write data for the object with the given object identifier with the given data distribution.

FinishWrite(OID, data_distribution) Register completion of writing to the object with the given object identifier for the specified data distribution.

Lookup(OID, offset, num_bytes) Retrieve a capability and the data_distribution information necessary to read and write the object with the given object identifier.

Chapter 3

Delayed instantiation bulk operations

This chapter presents features of a delayed instantiation bulk operation design within a distributed, object-based storage system. First, the we consider the options for placement of responsibility for carrying out bulk operations within the system. Second, grouping of objects into units suitable for bulk operations is discussed. Third, we present desired characteristics of two bulk operations: BulkClone and BulkDelete. Fourth, the options for instantiation of bulk operations are considered. Fifth, we offer two strategies for tracking uninstantiated bulk operations. The chapter finishes with considerations for successful completion of bulk operations and ways to mitigate the costs associated with them.

3.1 Placement of responsibility

As we consider the design of bulk operations within a distributed, object-based storage system, we must decide where the responsibility for enforcing semantics will lie. There are three places where responsibility might be assigned based on the need to act simultaneously on many objects: at the client, at the storage nodes or at the metadata server. Each of these locations has its merits, and shortcomings, which we present in the following sections.

There are many practical concerns to address while considering the placement of functionality. Not all of these are addressed in the following discussion, but they serve as guides for the decision making process.

System startup How is the mechanism bootstrapped?

System shutdown How does the system shutdown cleanly?

Recovery and repair If there is an error, how is it detected and corrected?

Performance What are the performance implications?

Administration How is the mechanism managed?

Data sharing What information is shared through the system and how is it shared?

Component join/leave What happens as components join and leave the system?

Externally initiated operation Can an external entity trigger the mechanism and to what effect?

Comprehensibility of semantics Are the semantics of the mechanism easy to reason about?

Abstraction Is the level of abstraction appropriate?

3.1.1 Bulk operations at the storage nodes

Performing bulk operations at the storage nodes concentrates responsibility at the lowest level of a distributed, object-based storage system. This is the approach being pursued by the SNIA OSD working group through the ANSI T-10 committee [11]. As all data is maintained by the storage nodes, they are closer to it than the metadata server or clients when it comes time for a bulk operation. The one distinct advantage is that there is no need for capability revocation from clients when a bulk operation is performed, and performing copy-on-write is straightforward.

There are too many complexities to consider bulk operations at storage nodes as a viable option in our situation. Issues of heterogeneity, permission, namespace management, recovery and feature implementation must be overcome.

A concern with placing responsibility for bulk operations at storage nodes has to do with object existence: since the metadata server tracks object existence, how can storage

nodes know what objects to clone? Besides that, each storage node can have a different view of the contents of the storage system: its local view. Coordinating the set of objects to be cloned (or deleted) across many storage nodes is the issue.

In large distributed systems, the set of active components can be changing from moment to moment. Faulty components, decommissioning of old components and commissioning of new components all change the set of participants at largely unpredictable times. If a storage node misses a message to execute a bulk operation, how will the system handle that? After taking a storage node out of service for repair, it must re-join the system and somehow be populated with information it may have missed. Should logs of missed messages be maintained? Should storage nodes be wiped and re-commissioned any time they re-join? How will this re-commissioning affect the reliability and availability of stored data?

Our system model is designed around an assumption that storage nodes are heterogeneous in their characteristics, in their contents, and in their security. Some storage nodes are expected to have greater processing power and/or capacity, for example. Executing bulk operations at storage nodes, therefore, could lead to drastically different service times observable by the caller. There could also be wide variation in service time given that some storage nodes will hold more objects than others and so could take longer to process a bulk operation. Also, if storage nodes had heterogeneous software for servicing requests, implementation decisions for that software could lead to variation in service time. Varying levels of security throughout a large system lend themselves to the use of $m-of-n$ data encodings, such as information dispersal algorithms, to spread information across storage nodes that may be (or may become) compromised.

The initial storage nodes taken as members for the distributed, object-based storage system were very simple and performed well enough. To add the necessary code to solve the issues presented in the preceding paragraphs would certainly destabilize their position in the system. And, the interfaces would be very difficult to standardize — the OSD-2 standardization effort avoided issues of multi-OSD consistency. As such, it was decided not to pursue the path of making storage nodes responsible for handling bulk operations.

3.1.2 Bulk operations at the client

Having the client library responsible for bulk operations presents challenges in that it creates "silos" of objects. If a client library is tracking bulk operations (BulkClone and BulkDelete) then only that one client knows about the (non-)existence of objects. This complicates sharing of objects between clients as each client would be delegated control over some portion of the object identifier namespace. Yet, such sharing is one of the reasons for object-based storage to exist.

There are two advantages to this approach. First, there is no need for the capability revocation of the metadata server approach. Clients would not have to re-acquire capabilities after bulk operations. Second, only those clients that require bulk operations will have to deal with them. Only they will need to execute code paths concerned with bulk operations.

Client managed bulk operations would be very portable to different storage systems. When they are implemented in a library, there are no particular dependencies on the storage system for assistance in carrying them out. In the absence of a Clone primitive at storage nodes, a client library could make a copy of an object by reading from a source object and writing to a new destination object.

However, the storage system could lose track of quota issues by delegating control to clients. It would need a back-channel to query clients to know how many objects were in-use and what capacity was being consumed.

Also, situations of client failure would require Enumerate() calls to figure out if objects existed in the storage system as compared to what their data structures indicate. Any corruption of state on a client could destroy the tracking structures for bulk operations and place the namespace in an unknown state. And, that approach would clearly not work with untrusted clients.

3.1.3 Bulk operations at the metadata server

Bulk operations as the responsibility of the metadata server can avoid many of the complications of the other two approaches, but comes with performance costs associated with capability revocation and re-acquisition. Both the client library and storage node based ap-

proaches avoid these costs. However, the other characteristics of delayed instantiation bulk operations managed at the metadata server prevail when considering feasibility in practice.

With control at the metadata server, there is a centralized location for managing bulk operation information. This approach avoids having many clients or many storage nodes trying to coordinate bulk operations amongst one another. The metadata server knows what bulk operations have been executed and can handle their instantiation appropriately.

As a necessarily secure and trusted portion of the storage system, the metadata server can be expected to faithfully carry out bulk operations. If bulk operations were managed at untrusted clients, then the storage system may not be stable. Similarly, if a faulty storage node were managing bulk operation information, it could corrupt an entire storage system.

The metadata server is already responsible for managing the object identifier namespace through the Create and Delete operations. The addition of BulkClone and BulkDelete, which also manipulate the namespace, is a logical progression. The metadata about objects affected by bulk operations also places it in an ideal location for handling instantiation: with information already on hand, it is easy to contact the necessary storage nodes to make a clone or delete an object.

3.2 Grouping objects

For the bulk operations proposed in this dissertation, there must be a way to form groups of objects that are specified as arguments. Groups can be formed in many ways. Whatever method is chosen, however, should allow for easy changes to a group's membership. Also, it should be easy to communicate the group as an argument to a function. For the sake of BulkClone, it should be straightforward to map objects from a source group to a destination group.

We propose to group objects as a *range* of object identifiers. Communicating a range-based group is as easy as sending two object identifiers, one representing the start, and another the end, of the range. A range can represent all objects or a single object (start equal to end). Membership of this type of grouping can be readily changed by altering

the objects to which one is referring, and moving objects around in the object identifier namespace (should it be necessary) can be accomplished with bulk operations.

Other ideas for forming groups of objects were considered but rejected.Presenting lists of objects communicated as arguments was deemed too cumbersome for large sets. The possibility of maintaining sets at the metadata server and referencing those sets in function calls was discarded because of the communication necessary to manage the sets, a need for access control for referencing and changing membership, and set-membership issues (could an object be in more than one set?). Forming object sets on the fly by referencing object attributes would be possible if the attribute set on our objects were richer, but such an approach nas many open questions beyond the scope of this dissertation.

3.3 BulkClone

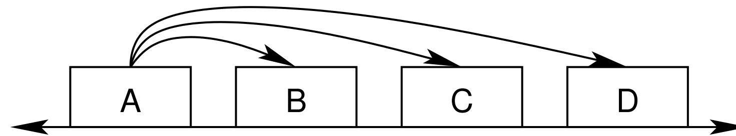
The *BulkClone* operation creates copies of all objects in a source set into a destination set. For our case of range-based bulk operations, objects are mapped one-to-one from a source object identifier range to a destination object identifier range[1]. To be successful, a BulkClone starts with some objects in the source range, and no objects in the destination range; the ranges must not overlap.

BulkClone(source_set, destination_set) Make objects of the destination set appear as copies of corresponding objects in the source set.

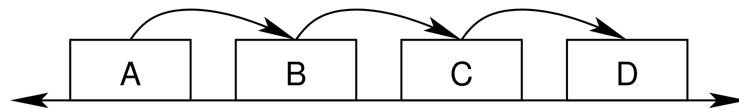
It might be reasonable to restrict the ranges used as arguments for a BulkClone. One might consider disallowing objects in destination ranges from being mentioned in source ranges. Such a restriction would simplify the logic applied to the bulk operation tracking structures, but would also force system users along particular usage paths. With only restrictions on sources being occupied and destinations being empty, two interesting and compatible usage scenarios emerge as shown in Figure 3.1 on page 27: prolific clones and chains-of-clones. A *prolific clone* uses the source range repeatedly as a parent with many destination ranges as its children. A *chain-of-clones* uses a previous destination range as

[1] A *range* is inclusive and consists of a starting object identifier and an ending object identifier.

the source range for subsequent BulkClone operations thus creating a chain where each clone operation forges a new link.



Example of a prolific clone: source remains constant.



Example of a chain-of-clones: prior destination is next source.

Figure 3.1: Two manners of using the clone operation
The original source set of objects is labelled “A”. Each clone operation is represented by an arrow pointing from source to destination set of objects.

The main motivation for providing the BulkClone operation is to support the creation of storage system snapshots. After formatting a file system in a set of objects (a range of objects in our case), repeated application of the BulkClone operation can create new copies of the file system. Any read-only copies become snapshots from which consistent backups can be made and accidentally deleted files can be recovered. Any read-write copies are file system forks that may evolve independently from the original file system image. File system forks provide sandboxes for testing systems on real data or launching near-identical copies off of a “golden master” system image.

Other possible uses of BulkClone include re-arrangement of the object identifier namespace. Such an operation could merge object sets from dis-joint portions of the namespace by cloning them such that they are adjacent and can be managed as a single, range-based set. One could also create *super-objects* by taking objects adjacent in the namespace and using each for specific attributes – this would be like extended attributes in Microsoft Windows NTFS and NFSv4, or resource forks in Apple file systems. Then, by using BulkClone, the entire super-object could be moved through the namespace as necessary. Similarly, a

group of adjacent objects might represent the contents of a directory (or even a directory tree) in a file system and a snapshot of just the one directory could be made.

The instantiation of a BulkClone takes place on those storage nodes holding portions of the source object according to the object's data distribution through the use of a single-object to single-object Clone command. The data distribution of the destination object, therefore, is inherited from that of the source object such that both objects reside on the exact same storage nodes. Future copy or migrate commands (not covered here) might move the objects off of those storage nodes. It is assumed that for speed and space efficiency, a storage node will have a copy-on-write implementation of the one-to-one Clone operation. During the course of access to cloned objects, it is possible that a storage node might exhaust its capacity and require a redistribution of an object to other storage nodes.

3.4 BulkDelete

The *BulkDelete* operation removes a set of objects from the object identifier namespace and, ultimately, frees resources consumed by those objects on storage nodes. For our case of range-based bulk operations, objects within a supplied target range are removed. A range can span all objects. A successful BulkDelete operation has objects in its target range.

BulkDelete(target_set) Delete all objects appearing in the target_set of object identifiers.

A primary motivation for providing the BulkDelete operation is to support the cleaning of objects created by BulkClone operations. Thus, a BulkClone that created a snapshot of a file system can be removed all at once with a single BulkDelete operation.

Additional uses of BulkDelete match other uses for BulkClone. If BulkClone is used to merge object sets by moving them around the namespace, then BulkDelete can clean up the source object sets. If an application were to use super-objects, a BulkDelete would be the most efficient way to delete a super-object. For a file system that groups files within directories as adjacent objects, a range-based BulkDelete operation could quickly delete a directory.

The instantiation of a BulkDelete takes place on those storage nodes holding portions of the object being deleted with a single-object Delete command. The capacity consumed by the deleted object, and its object identifier in the namespace, can then be reclaimed.

3.5 Delayed instantiation

The *instantiation* of a bulk operation is the act of enforcing the effect of that bulk operation upon a single object at the storage nodes. Instantiating a BulkClone entails instructing storage nodes to copy an object. Instantiating a BulkDelete entails instructing storage nodes to delete the object. To preserve the semantics of our bulk operations, a clone must be instantiated before subsequent changes can occur to the object being cloned. A delete must be instantiated before an object with the same object identifier can be created.

There are a few options for when instantiation of a bulk operation might be performed. A system might use *immediate instantiation* and cause a bulk operation to take effect before returning to the caller. This could be troublesome to users of the storage system, because their request could take a long time to return as would be the case when dealing with a large set of objects. A client would be secure, however, in knowing that the requested operation was applied to all targeted objects. An alternative, which we explore in this dissertation, is to use *delayed instantiation* of bulk operations. With this strategy the storage system promises to enforce the effects of the bulk operation but has the option to *delay* the work needed until a client might observe the bulk operation's effect. When using delayed instantiation, a success code can be quickly returned to the caller, while the work of the operation is performed later. Under delayed instantiation, when a client accesses an object affected by a bulk operation, an *on-demand instantiation* is performed before the client's request is allowed to proceed. This adds latency to the client's operation, but has delayed work in the storage system until it is strictly required. If a clone is deleted before it is instantiated, then the work never has to be performed. With delayed instantiation, the storage system may elect to perform *background instantiation* during idle time to save on the added latency of on-demand instantiation.

Examples of how delayed instantiation can work are shown in Figure 3.2 on page 30. There we see that instantiation always occurs before a client observes the effects of a bulk operation. The instantiation can occur before or after a return code is sent back to the caller.

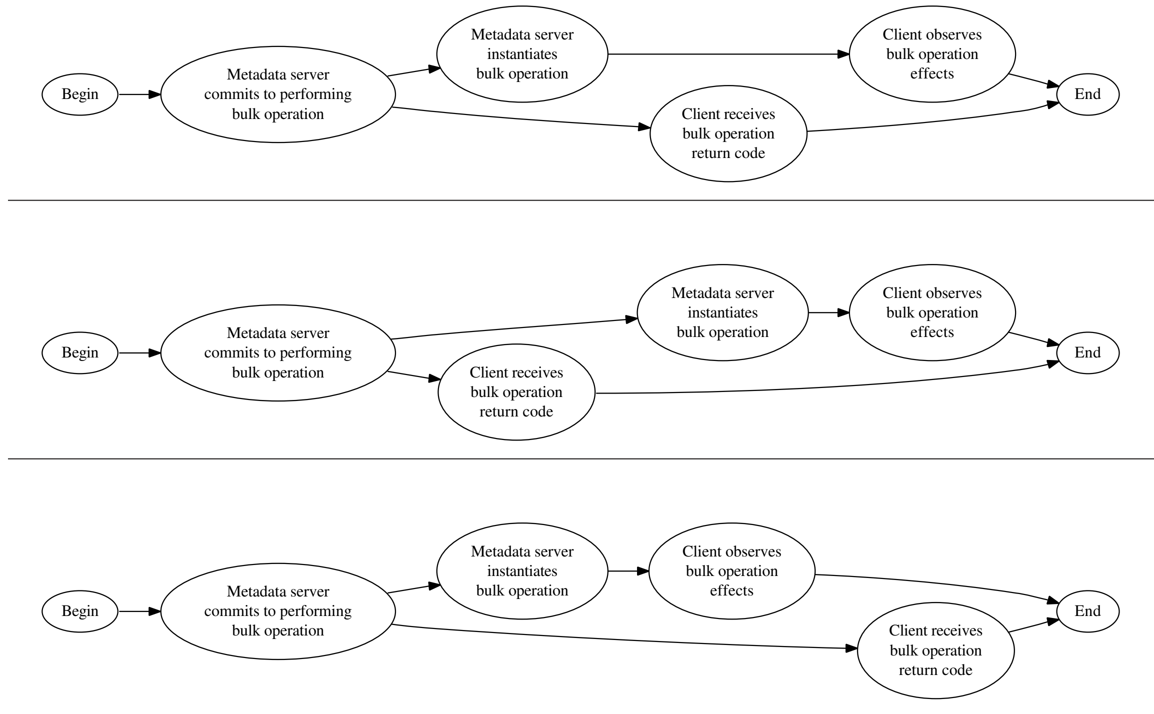


Figure 3.2: Bulk operation process

These are three variations on when bulk operations can be processed and observed by clients. The two paths correspond to two semantic guarantees associated with operation execution: an operation is applied sometime between when it is requested and it is observable as having taken place; operation execution confirmation must follow the request for an operation.

Using delayed instantiation in a distributed, object-based storage system provides several benefits. First, the system can quickly return a success code to the caller since the work will be delayed. Second, the system can save work, since bulk operations are not required to be instantiated if a client never accesses the affected objects.

Delayed instantiation is used throughout computing. One example is *copy-on-write* where multiple references to a single data structure split to become references to multiple data structures when a modification is made to the single data structure. This allows for independent evolution of the data referred to by those initial references and conserves space and time while the data structure remains unchanged. Similarly, *lazy evaluation* of expressions until the moment they are required saves on the time required for the evaluation.

The capability-based access control system used in object-based storage can provide a window of time during which the instantiation of a bulk operation can be delayed. By revoking capabilities to objects affected by a bulk operation and then accepting it for execution, the metadata service can delay instantiation until a client requests access to an affected object. At the point where a client accesses an object affected by a bulk operation, an *on-demand instantiation* must take place.

3.6 Bulk operation tracking

We have considered two primary approaches to tracking delayed instantiation bulk operations: range splitting and timestamp tracking. The *range splitting* approach would record bulk operations in tracking table entries and modify entries that covered objects undergoing instantiation by splitting the range mentioning them. The *timestamp tracking* approach would record all object identifier namespace operations along with logical timestamps to understand their ordering. Ultimately, a combination of range splitting and sequence tracking was necessary to correctly track bulk operations and perform required instantiations.

To record the execution of delayed instantiation bulk operations, some information about them must be retained by the storage system. The arguments to the operation, namely the set (or sets) of objects affected, are the minimum information. Since we have settled on using ranges to represent object sets, the starting and ending object identifiers are recorded in the bulk operation tracking tables. A BulkClone table tracks the source and destination ranges of clones. A BulkDelete table tracks the target range for deletion.

Range splitting modifies the bulk operation tracking tables as instantiation is performed. The idea is that, when an operation causes a clone to be instantiated upon an object, the

bulk operation table entry that tracked the object is removed, split (or trimmed if the object is at the start or end of the range), and the entries (or entry) re-inserted. Thus, the operation on that object is no longer tracked by the bulk operation table and so will not be instantiated again. This approach leads to significant fragmentation of the tracking table as instantiations cause the proliferation of table entries.

Timestamp tracking adds more information to the bulk operation tracking tables and the object table. The idea behind timestamp tracking is that the ordering of events can be used to determine what instantiation(s), if any, must be performed. A logical clock is maintained by the metadata server, and each Create operation and each bulk operation store a logical timestamp with the record of their execution. When a Create adds an object to the object database, a timestamp is stored. When a BulkClone or BulkDelete is executed, their arguments are saved along with a timestamp. This approach suffers from requiring long-term retention of bulk operation information and potentially lengthy calculations to determine if an instantiation must be made.

By combining range splitting and timestamp tracking, a comprehensive bulk operation management system can be created – we describe one such solution in Chapter 4 starting on page 35. Also, ambiguous situations encountered by each individual mechanism can be reconciled with their combination. Range splitting is used to track which bulk operations remain to be instantiated. Timestamp tracking allows the overlap of bulk operation ranges to be tracked while allowing for reasoning about which objects are affected by which operations.

3.7 Completion and success criteria

For a bulk operation to complete successfully, a number of conditions must be satisfied. First, successful capability revocation at the bulk operation execution time must be performed. Second, at instantiation time, a sufficient number of storage nodes must be contacted to enact the effects of the bulk operation.

Successful capability revocation is an interesting issue when *threshold based encodings* are used in data distributions. Revocation involves information about encodings and

requires that the correct set of storage nodes be contacted. The complexity of revocation increases as bulk operations are added to a storage system where data distributions are customizable on extents of objects.

Given an $m-of-n$ encoding for an extent of an object[2], a successful revocation must be performed at $n-m+1$ (where $n > m$) storage nodes to prevent a client from accessing the data of that extent. By performing revocation at $n-m+1$ storage nodes, a correctly behaving client cannot access data because it will encounter more than m capability errors when reading or writing data fragments. Consider as an example the case of a $1-of-3$ encoding (three-way mirroring). Any one storage node holds a complete copy of the data. In this case $m = 1$ and $n = 3$, so $n-m+1 = 3-1+1 = 3$, and we see that we must revoke capabilities from all three storage nodes to prevent a client from reading or writing fragments (replicas in this case) at storage nodes. The situation holds for other encodings, such as a $5-of-7$ which affords protection from double-faults. In the $5-of-7$ case, $m = 5$ and $n = 7$, so $n-m+1 = 7-5+1 = 3$, and we see that we must revoke capabilities from three storage nodes. This keeps a client from being able to successfully reach a set of five storage nodes necessary to read or write fragments.

Applying the limits for capability revocation for a single bulk operation can be done with knowledge of all objects involved, all of the extents of the objects, all of the encodings and their thresholds and knowing the storage nodes holding fragments for each of those encodings. This information creates *revocation sets* over the storage nodes that must be collated and then applied by sending revocation messages and awaiting sufficient responses to prevent client access. The generation of these revocation sets is complex without maintaining additional indices within the metadata server databases. A sufficient revocation can be accomplished by keeping track of the *least* fault tolerant encoding in the system and revoking capabilities from the set of all storage nodes. As long as revocation is acknowledged by $all - least + 1$ storage nodes, then we can be content that no client can access any fragments of data for the objects affected by a bulk operation.

When it becomes time to instantiate a bulk operation, at least m storage nodes must

[2]This treatment of revocation and instantiation limits applies to comprehensively versioning storage (such as are described in [1] and [81]) using quorum based encodings where the quorum is of size m.

acknowledge the instantiation request for an extent. This is a situation analogous to revocation but for the single object upon which a bulk operation is being instantiated, we need to consider its storage nodes, encodings and thresholds for all extents. As long as m storage nodes get the instantiation message for each extent (where m is a per-extent value), the instantiation succeeds.

3.8 Mitigating costs

Delayed instantiation bulk operations obviously come with costs. One can see that, based on range splitting, fragmentation of the bulk operation tracking tables will be an issue. The re-acquisition of revoked capabilities by clients adds latency to the next access after a bulk operation executes. And, the on-demand instantiation of bulk operations at storage nodes further slows that next access.

Two of the three costs can be addressed with a single mechanism with which we experiment: background instantiation. A *background instantiation* is an instantiation of a bulk operation upon an object that happens some time between operation execution and the first time a client requests access to it. This can help with cleaning up fragmented bulk operation tables and save on contacting storage nodes to perform on-demand instantiation. The protocol steps of a client realizing that a capability is invalid and re-acquiring a new capability are not ameliorated. A background instantiation comes at little cost if it is done during idle time at the metadata server.

Chapter 4

Implementation

This chapter describes the implementation of the various components of the distributed, object-based storage system into which we have engineered support for delayed instantiation bulk operations. This implementation provides us a platform for evaluating bulk operations and delayed instantiation. The software components of client, storage node, and metadata server are described as they relate to bulk operations. Background instantiation strategies for mitigating the costs of delayed instantiation are presented. An NFS server is described that uses the BulkClone operation for creating snapshots and forks of its served file system. The protocol steps for operations and their interactions with bulk operations conclude the chapter.

4.1 Client

The client library requires little specialization for bulk operations. It passes commands to the metadata server and storage nodes. It caches metadata and capabilities, as appropriate. It handles the cases where an attempted operation to a storage node returns errors regarding invalid capabilities. The only extension needed was an interface for invoking bulk operations.

BulkClone(src_start, src_end, dst_start) A command sent from the client to the metadata server that will cause all objects in the source range to be copied, one-to-one,

into the destination range. The end of the destination range can be calculated from the size of the source range.

BulkDelete(target_range_start, target_range_end) A command sent from the client to the metadata server that will cause all objects in the target range (from target_range_start to target_range_end, inclusive) to be removed from access by clients of the storage system.

4.2 Storage node

The storage node also requires minimal extension.The base storage node evolved from the S^4 project and the PASIS project. It provides a block-based interface to objects where, for a given block number, it stores data provided during a write and retrieves any data for a block during a read.

Individual storage nodes are instructed to revoke capabilities in bulk, to clone individual objects and to delete individual objects. Capabilities and revocations are tracked as integers: if a capability is less than the tracked revocation, it is rejected as invalid. Bulk revocation of capabilities sets the tracked revocation for affected objects, causing existing capabilities to be rejected. Bulk capability revocation is used by the metadata server in the course of bulk operations and instructs the storage node to reject a range of capabilities. To process the clone and delete commands, a storage node first locates the particular object in its data structures. When deleting an object, the storage node marks the data structures used by that object for reclamation and removes its object identifier from service. When cloning a source object, the storage node duplicates the current view of an object as the destination object.

The storage nodes used in our prototype system already had the ability to delete single objects. We added a BulkRevoke operation that expanded upon an existing single-object capability revocation call. We added a Clone operation that did not require externally reading a source object and writing it to a destination object. The Clone operation internally copies the source object to the destination object. A more ideal implementation would em-

ploy copy-on-write between the source and destination object data, but this implementation short-cut should suffice when objects are small and can be wholly contained in memory.

BulkRevoke(capability_sequence_number, start_OID, end_OID) Reject capabilities with a capability_sequence_number less than that given as an argument for objects between start_OID and end_OID, inclusive.

Clone(source_OID, destination_OID) Make a copy of the current view of the source object under the name of the destination object.

4.3 Metadata server

The bulk operations capable metadata server consists of three primary components: a client-facing front-end, a back-end database, and a helper application for instantiation.

4.3.1 Front-end

The front-end to the metadata server is the point-of-entry for remote procedure calls (RPCs) originating from clients. It is a C++ program that mostly acts as a translation layer from the RPC format for client dialogue to the PostgreSQL database where server-side functions implement the service logic for the requests. The translation is done automatically using the SQL standard for "embedded SQL in C"; compilation of an annotated C/C++ program is preceded by an "embedded SQL in C" pre-processor that replaces annotations with calls to library functions and standard C pre-processor macros.

When started, the front-end joins itself to the storage system by registering as a metadata server. Various checks of the back-end database are performed and, if necessary, it is initialized. A service loop receives client requests and hands them off to worker threads for servicing. When this service loop is idle, attempts at instantiating bulk operations in the background are made (see Section 4.4 on page 40).

Capability revocation is performed by this front-end program and uses the BulkRevoke RPC to storage nodes. Revocation is accomplished by informing storage nodes to reject

capabilities (integers) lower than a particular value which is controlled by the metadata server. Since the database does not directly participate in the RPC dialogue of the storage system, as bulk operations enter the front-end, the necessary storage nodes are contacted with Revoke messages. If there is a problem with revocation, then there is no need to go to the back-end — in this case an error message returned to the client. If there is a problem at the back-end that causes a bulk operation to abort, then the capabilities have already been revoked and clients will need to re-contact the metadata server.

There were two additional functions that needed to be added to the metadata server interface, one for each bulk operation supported.

BulkClone(src_start, src_end, dst_start) Cause all objects in the source range to be copied, one-to-one, into the destination range. The destination range is calculated from dst_start and the size of the source range.

BulkDelete(target_range_start, target_range_end) Cause all objects in the target range to be removed from access by clients of the storage system.

4.3.2 Back-end

The back-end of the metadata server holds the data structures containing metadata, implements the algorithms to satisfy client requests, and performs delayed instantiation bulk operations. It is built around a PostgreSQL database with tables for storing metadata and server-side functions written in the pl/pgsql scripting language to handle RPC requests. The use of an off-the-shelf database gives built-in transaction support and high-level programming in a declarative language, SQL.

On a clean start of the metadata server, the front-end creates the tables and indices in the metadata database, some tables with values acting as stand-ins for C-language enumerated types, and installs the server-side functions for handling client requests.

A configuration table holds variables that control the runtime behavior of the server-side database functions. One setting forces immediate instantiation of bulk operations. Another setting enables or disables all bulk operation code paths.

There are two tables for tracking bulk operations: one for BulkClone and one for BulkDelete. These tables track the arguments to their respective bulk operations and a sequence number indicating the point in the evolution of the system at which the bulk operation occurred. A BulkDelete table entry tracks the starting and ending object identifiers of the range being deleted. A BulkClone table entry tracks the starting and ending object identifiers of the source and destination ranges. Trigger functions, database functions invoked under certain conditions, are executed upon insertion into the tables. The trigger functions make sure that if a row is inserted without a sequence number, the latest sequence number is incremented and assigned to the row. If a row is being inserted that already has a sequence number, then it is accepted without change

A database table tracks objects that exist on storage nodes. This object table tracks the object identifier, length of the byte stream, and a sequence number representing the point at which the object was created. The object identifier for each entry must be unique.The length for each object corresponds to the highest offset reported by a FinishWrite command. The sequence number is not necessarily unique across all rows of this table. If an object is created on its own, then it gets a unique sequence number. When an object enters the object identifier namespace by virtue of a BulkClone operation, all objects within the destination range of that clone will share the same sequence number as they were all created at the same instant.

There are two additional tables, each for storing metadata describing the backing store for byte-ranges of individual objects. One of these tables, the MetadataExtent table, tracks ranges that have already had a FinishWrite call on them. The other table, the PendingWrite table, tracks the pending writes submitted during ApproveWrite calls by clients. Any byte range of an object is only tracked in one of these tables at a time. If compatible metadata is inserted adjacent to existing metadata, the ranges are merged into a single row of the table. The metadata is moved from the pending write table to the tracking table, and the object length may be updated, when FinishWrite calls are made. Data from the tracking table is returned when client libraries make Lookup calls.

Server-side functions implement the logic of the metadata server necessary for completing client-initiated RPCs. For each of the interface functions, Lookup, Enumerate, Ap-

proveWrite, FinishWrite, BulkClone and BulkDelete, there is a primary service function. A number of other functions implement common code called by these primary functions. The functions access the database tables as necessary and execute the loops, control statements, and SQL queries that embody the necessary algorithms for service in a bulk operations capable metadata server.

The primary, and most of the secondary, functions are written in pl/pgsql. Two functions are written in C, with appropriate wrappers for use by pl/pgsql. One of these C functions calculates the capabilities returned by a Lookup. The other C function acts as a gateway for communication with the "helper" application.

4.3.3 Helper

The helper application is used by the metadata server back-end for on-demand instantiation of bulk operations. Since there is no direct way for pl/pgsql functions to communicate externally with the storage system, a pl/pgsql stub function is provided. This stub communicates via IPC with the helper application. When the helper application receives a command, it forwards that command to the addressed storage nodes. As information is returned by the storage nodes, the helper application collates the replies and directs them back across IPC to the pl/pgsql function.

4.4 Background instantiation

Background instantiation of bulk operations may be performed by the front-end of the metadata server to reduce the record-keeping overhead of bulk operations and to speed subsequent client access to affected objects. Record-keeping overhead grows with almost every bulk operation instantiation, causing searches of bulk operation tables to take longer. Also, when clients trigger on-demand instantiation, they incur additional latency as their requested operation is performed. Background instantiation, occurring during metadata server idle time between bulk operation acceptance and an on-demand instantiation, cleans up tracking information and eliminates instantiation in a client's critical path. Idle detection

is set to be very aggressive: if the main loop of the metadata server front-end is idle for 1 ms, it may trigger background instantiation operations.

For experimentation purposes, background instantiation can be (de-)activated through the use of a command line argument presented when the metadata server is started. When the metadata server processing loop times-out waiting for work, it may initiate a background instantiation. There are five algorithms that may be set from the command line (or interactively) for choosing an operation and an object upon which to instantiate it.

1. **Random bulk operation, random object**
   Randomly pick the BulkClone or BulkDelete table for processing. Select a random table entry. Instantiate the bulk operation on a randomly selected object from the first 100 objects affected by the selected bulk operation table entry.

2. **FIFO bulk operation, lowest sequence number**
   Between the BulkClone table and BulkDelete table, select the entry with the lowest sequence number[1]. Process the bulk operation on the object with the lowest object identifier.

3. **LIFO bulk operation, highest sequence number**
   Process the highest object identifier from the bulk operation table entry (either BulkClone or BulkDelete) that has the highest sequence number.

4. **Largest span, lowest object identifier**
   Process a bulk operation from the bulk operation table entry that spans the most object space (maximum difference between starting and ending object identifiers, which does not necessarily correspond to the most objects since a range could span millions of objects but be very sparsely populated). Break ties by processing the bulk operation affecting the lowest numbered object identifier. Process the lowest numbered object affected by the bulk operation.

[1] There is no need for a tiebreaker since the same sequence number could not be shared by a BulkClone and a BulkDelete. For the case of a previously split bulk operation table entry, the lowest sequence number decides the entry to process.

5. **Smallest span, lowest object identifier**
   Process a bulk operation from the bulk operation table entry that spans the least object space (minimum difference between starting and ending object identifiers). Break ties by processing the bulk operation affecting the lowest numbered object identifier. Process the lowest numbered object affected by the bulk operation.

The efficacy of these algorithms is compared in Section 6.5 on page 109 and in Section 6.6.2 on page 137

4.5 NFS server

An NFS server that uses the BulkClone operation to create snapshots and forks of its file system was built for experimentation and to demonstrate feasibility. Slight changes to a pre-existing NFS server that stores its directory and file information in objects are made to introduce this new functionality.

The NFS server is an application that acts as a client of the distributed, object-based storage system.. It does so on behalf of NFS clients. As such, it is acting as a sort of proxy, or gateway, for NFS client access to the objects by overlaying a pathname and filename hierarchy atop the flat object namespace and using individual objects as containers for file and directory data. This relationship between NFS clients, the NFS server, metadata server and storage nodes is shown in Figure 4.1 on page 43.

The NFS server operates over TCP/IP and follows most of the NFSv3 specification [14]. We diverge in that writes are not guaranteed to reach stable storage before returning a success code to a client. The storage nodes use a shared memory area to emulate NVRAM, and this area serves as a staging area for dirty data. This model follows that of high-performance production systems that actually contain NVRAM (with data persistence across reboots) for staging dirty data [26, 54]. The NFS server considers data stably stored once a write operation has been acknowledged by a storage node, with no guarantee of the data being able to survive a reboot.

The exported NFS file system is rooted at a *root-object* (similar to a super-block of a block-based file system). This object contains information about the formatting of the

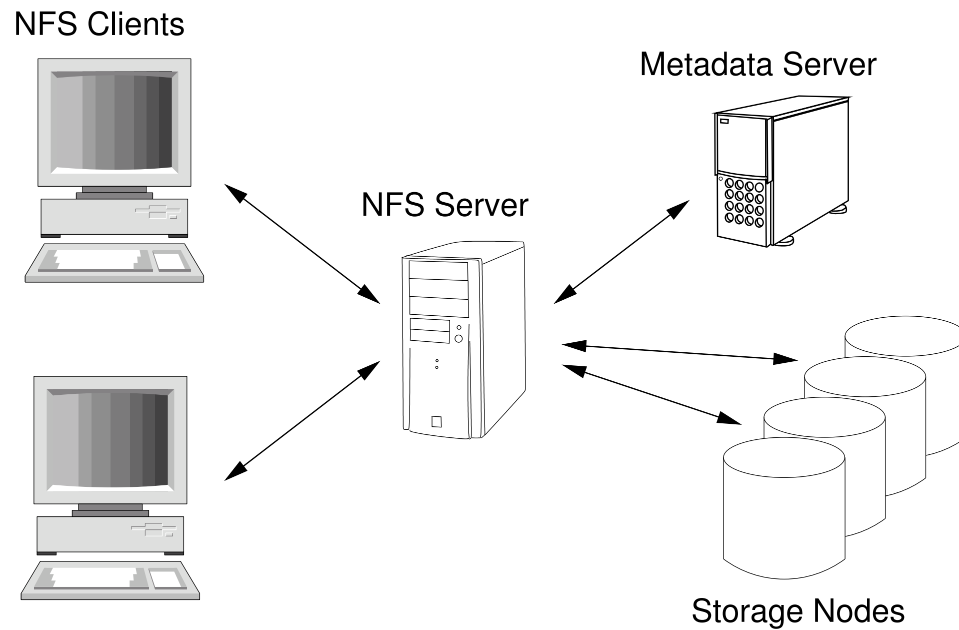


Figure 4.1: NFS server as storage system client
The NFS server is a client of the object-based storage system. It translates NFS client file and directory requests into object accesses.

file system. The root directory object is at a well-known location based on an allocation scheme stored as part of the root-object. All directory objects contain mapping information from file and directory names to object identifiers.

To track the snapshots (and forks) of the NFS file system, structure is applied to the object identifiers used. This structuring is not done to work around any limitations of the implementation; it is simply a breakdown that appears to work well. The 128 bit object identifier is separated into three components. The uppermost 32 bits are static and represent the instance of an NFS server within the storage system — many NFS servers can be running and using the object-based storage system simultaneously. The following 32 bits represent the particular instance of a clone of the file system. The lowest 64 bits represent the file or directory within a particular clone. A file system served by an NFS server has its top 32 bits always the same. A clone of a file system has the top 64 bits always the same.

To create snapshot of the NFS file system, the server goes through a series of three steps. First it suspends access by NFS clients and flushes all caches of dirty data. By suspending access, no other NFS client requests can make forward progress until the snapshot has been made. The push of all dirty data to the storage system ensures that a consistent set of client-written information is captured in the snapshot. Second, it makes the snapshot by invoking BulkClone through the storage system's client library API. The source range argument to the BulkClone operation is drawn from the current range of objects used by the NFS file system. The destination range is specified in the command triggering the operation. Once acknowledgement of the successful completion of the BulkClone is received, tracking information is written to the root-object of the destination range of the clone. Third, the caches are re-enabled and NFS client accesses are allowed to resume.

One issue with making file system snapshots in this manner is the contents of the objects holding directory entries. Since a directory tracks a mapping of file names (which do not change during a snapshot) to object identifiers (which *do* change), all directory entries in the snapshot will refer to objects in the original file system. An example of this sort of situation is shown in Figure 4.2 on page 45.

To address this situation, the post-snapshot destination range file system has a note placed in its root-object indicating the range of objects contained by the file system, and directory entries are mapped into this range upon access. When the file system starts, the root-object is read. Whenever directory entries are read, the objects that correspond to file names are checked. If the object identifier is outside the range of objects tracked, then it is mapped into the correct range. Only objects cloned from the previous incarnation of the file system will need to be re-mapped, and only until their directory entry is accessed and written out to storage with corrected values. Any newly created objects will have object identifiers in the "current" object range from the start; their directory entries will not need interpretation.

Using this re-mapping on the fly, the contents of the directory entries of the file system do not need to be re-written at snapshot time. The correct interpretation can be performed (a simple comparison check and bit substitution) at access time. As directories are accessed, the mapping of file names to object identifiers can be lazily updated.

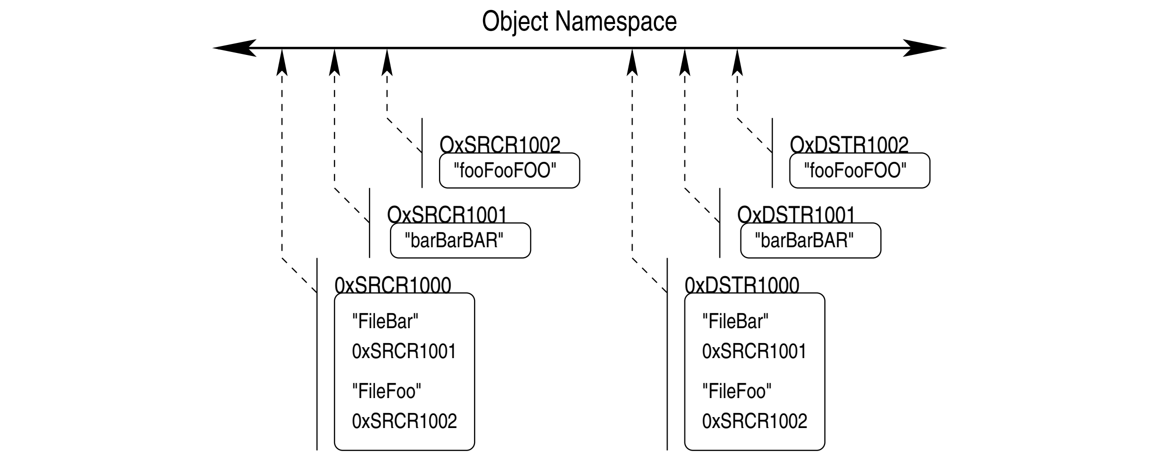


Figure 4.2: Directory contents after cloning

After executing a BulkClone operation the contents of a destination directory object associate file names with objects in the source set. In this example, a BulkClone of objects from the source range $(0xSRCR0000, 0xSRCRffff)$ have been cloned into the destination range $(0xDSTR0000, 0xDSTRffff)$. The directory object $0xSRCR0000$ has been cloned to $0xDSTR0000$, and the contents of $0xDSTR0000$ point to objects in the range $(0xSRCR0000, 0xSRCRffff)$. For clients to access the correct objects when traversing the destination objects, the file server must only replace the "*SRCR*" with "*DSTR*" in the object identifiers that it encounters in directory entries.

An alternative implementation could skip storing the remapped portion of object identifiers in the directory entries altogether. Then, when accessing an object, the correct information about the mapping and the remaining object information in the directory entry could be combined to form an object identifier. This would obviate the re-mapping done with the implemented approach. We used the re-mapping approach to minimize the amount of code that needed to be touched in the NFS server.

4.6 Protocol

There are seven basic operations available to clients through the client library. The API for the basic operations was sketched out in Section 2.4.4 on page 17 and for the bulk

operations in Sections 3.3 and 3.4 on pages 26 and 28.

This section presents those operations in the context of the RPC protocol between client, metadata server, and storage node. The instantiation of bulk operations can occur during the normal execution of the protocol and is shown in the following diagrams.

A system can have many clients, each of which can be issuing commands concurrently in the storage system. The metadata server receives the requests from these clients and processes them. There are potentially many storage nodes within the system. Subsets of these storage nodes are contacted by clients to perform reads and writes of object data. Storage nodes are also contacted to revoke capabilities and instantiate bulk operations. In the diagrams shown in this section, it should be understood that the column representing the storage nodes, while only labeled as "storage node", really applies to any number of storage nodes.

4.6.1 Create

The Create() command adds an object to the object namespace. There is one argument: an object identifier. If the named object does not already exist, then it will be added to the namespace.

With delayed instantiation bulk operations, the Create of an object may cause the metadata server to contact storage nodes to delete a previously existing incarnation of an object. Consider, for instance, the following sequence of events from the client application's perspective:

1. Create(OID = 100)
2. Write(OID = 100, offset, data)
3. BulkDelete(range = 50 … 150)
4. Create(OID = 100)

A client creates an object, writes it, executes a BulkDelete that affects it, and then recreates the object. During the given sequence, the protocol exchange as shown in Figure 4.3

on page 48 occurs. Because of the delayed instantiation strategy for implementing bulk operations, the second Create takes longer as the metadata server must inform the storage node(s) that a Delete must be instantiated.

4.6.2 Lookup

Although strictly an internal command, since it is a sub-step of Read and Write, Lookup is the primary trigger for causing the instantiation of clones.

The Lookup() command returns metadata for an object to the caller. There is one argument: the object identifier for the object being queried. This command is used by the client library to retrieve metadata and capabilities from the metadata server. The client library uses this function as part of Read and Write operations, if it does not have cached capabilities to access an object.

The Lookup operation can trigger BulkClone instantiation. If it is executed upon an object that is either the source or destination of a BulkClone, then both objects must be created so that they may evolve independently. A Lookup that triggers an instantiation takes longer than one that does not, because of the extra step where the metadata server contacts the necessary storage node(s). Such an interaction is shown in Figure 4.4 on page 49.

If the object does not exist, an error is returned. If the object does exist, any metadata extents for the object will be returned together with capabilities for accessing any existing data on storage nodes.

4.6.3 Enumerate

The Enumerate() command returns a list of object identifiers that exist in the namespace at a point in time. There are three arguments: a starting object identifier, an ending object identifier, and a number of objects to return (up to some maximum, set to 100 objects in our implementation).

A client sends an Enumerate command to the metadata server. All processing occurs at the metadata server; no storage nodes are involved.

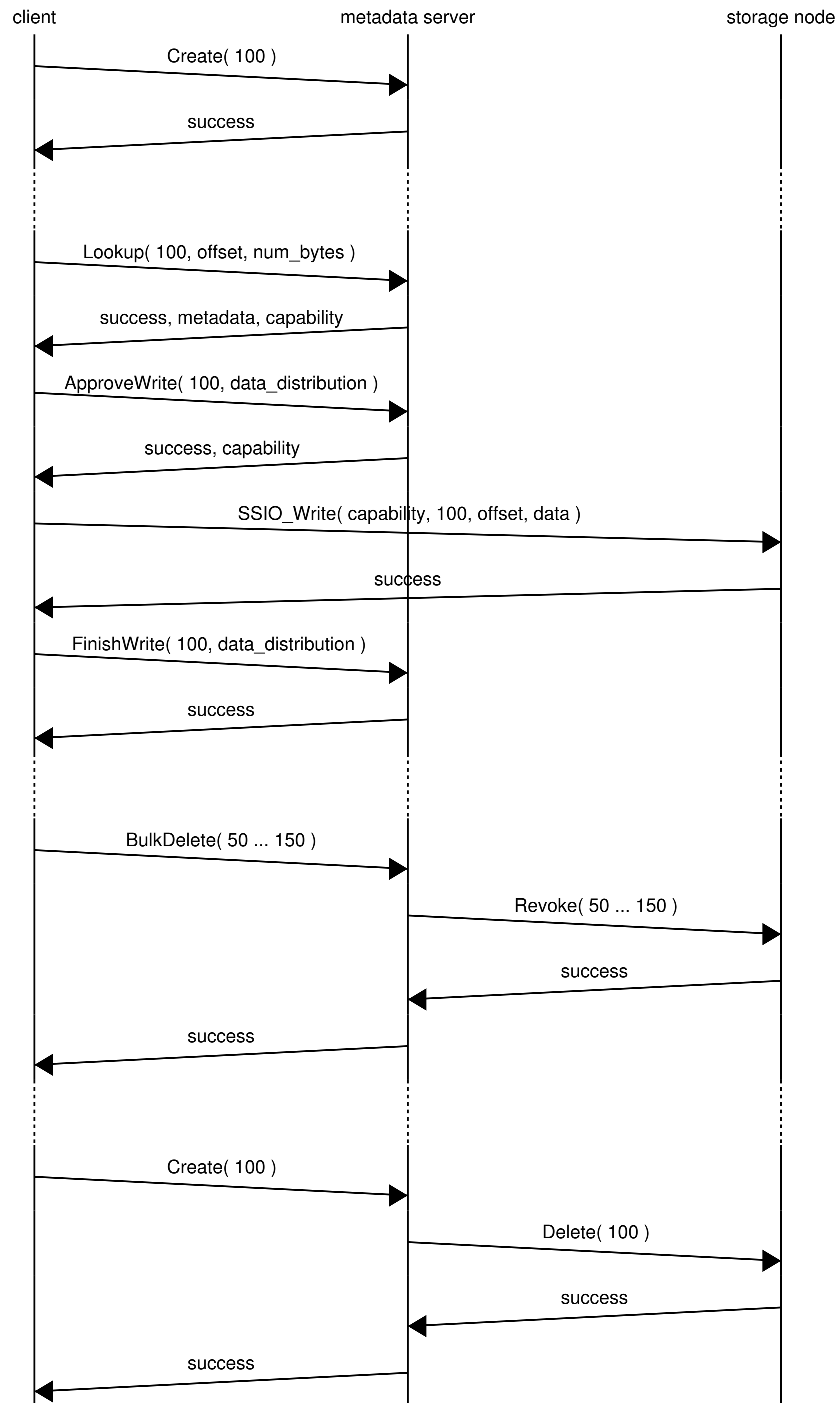


Figure 4.3: Create and Re-Create with BulkDelete

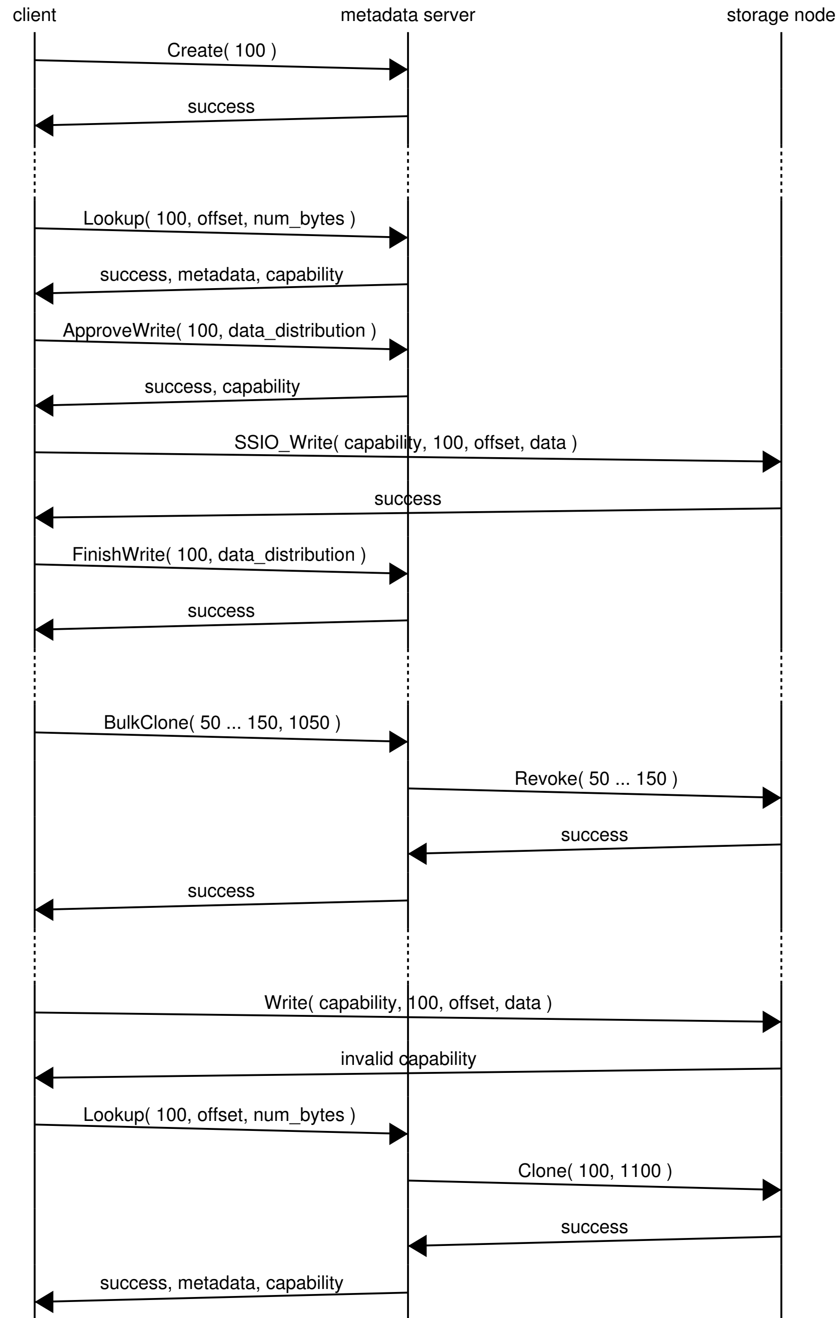


Figure 4.4: Lookup with BulkClone

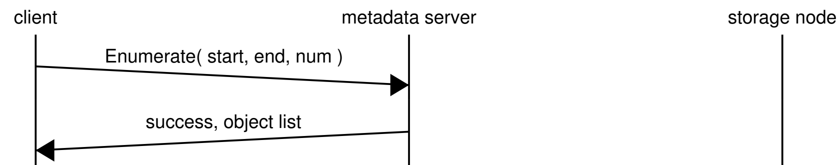


Figure 4.5: Protocol for Enumerate

A successful return from an Enumerate() command provides the caller with a number of objects in the given start–end range, starting with the lowest object identifier (the start) proceeding towards higher object identifiers (the end).

The presence of bulk operations in the storage system does not have an impact on the protocol for the Enumerate command. The effect is felt in the algorithms necessary to reason out the actual contents of the namespace when bulk operations are creating and removing objects. The algorithms are described in Section 5.4.1 on page 73.

If there are fewer than the requested number of objects in the range, then all of them will be returned. There is no indication to the caller that they have received all object identifiers, if the last object in the returned list is less than the ending object provided as an argument. Likewise, there is no way for the caller to know that there are more objects in the requested range if the returned list contains the requested number of objects and the last identifier in the list is less than the ending object.

A client can indirectly get a consistent enumeration of a range of objects. Given that there is a limit on how many objects can be returned by an Enumerate command at a time, it is impossible to directly get the state of a large number of objects at one instant. But, if used in conjunction with the BulkClone command and some object identifier translation, it is possible to get a consistent view of portions of the storage system. By first cloning the namespace of interest, and then iteratively enumerating over the destination range, a client sees a consistent view of the source range since the BulkClone operation is atomic. All that is required is a little math on object identifiers to translate them back from the destination range into the source range. When the enumeration is complete, the destination range and its objects can be removed with a BulkDelete command. A disadvantage of this approach is that BulkClone instantiation will occur, if clients access objects in the source range while

the enumeration is ongoing.

4.6.4 Delete

The Delete() command removes a single object from the object namespace, causing the resources of that object to be freed. There is one argument: the object identifier of the object to be deleted.

A client issues a Delete command to the metadata server. Internally, the metadata server may choose to represent the Delete as a BulkDelete and delay its instantiation or it may choose to immediately instantiate the operation by contacting storage nodes. Regardless of this choice, the storage nodes must be contacted to revoke capabilities. With the option of deleting immediately, or later, the metadata server can delay work at storage nodes required to free large objects.

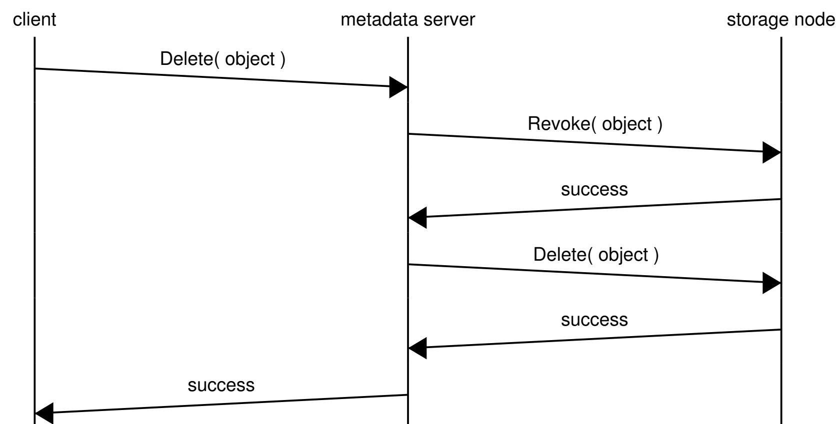


Figure 4.6: Protocol for Delete

It is obvious, given the protocol as shown in Figure 4.6, that some optimization could be performed. Batching the Delete and Revoke commands sent from the metadata server to the storage node would save a network round-trip.

The Delete() command will succeed unless there is trouble encountered within the MDS. No guarantees are made as to when the resources of the object will be freed, other than that it will happen before another object with the same object identifier can be created.

4.6.5 Write

The Write() command places data into the byte-stream component of an object. There are four arguments: the object identifier, the byte-offset at which to start writing, a buffer of data to be written and the length of that buffer.

The Write command issued by a client into the system goes through several steps. First, the client library must acquire metadata and capabilities. A Lookup from the client to the metadata server acquires metadata and capabilities sufficient for handling over-writes of data in existing portions of an object. If it is a write to an unallocated portion of an object, the client library must log its intent to write through the use of the ApproveWrite command and must complete its write using the FinishWrite command. The actual transfer of data between the client and the storage nodes is accomplished by the SSIO_Write message. A diagram of this protocol, without bulk operations, in show in Figure 4.7 on page 52.

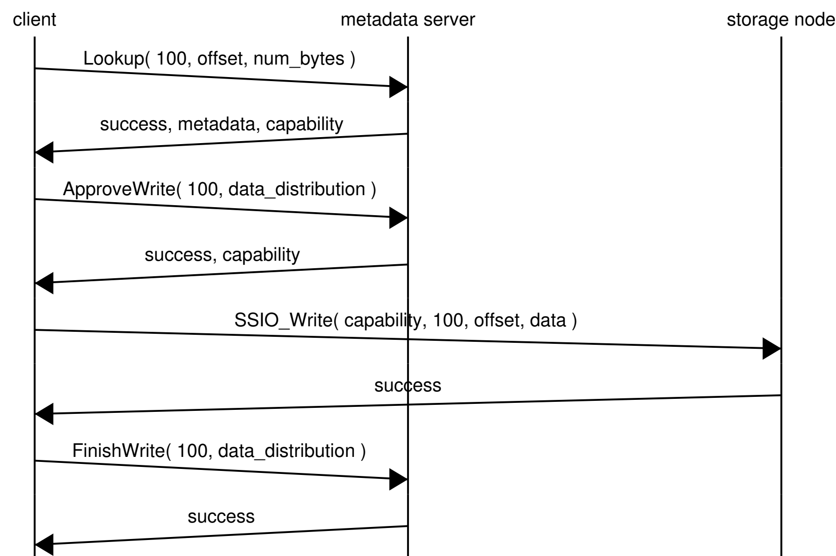


Figure 4.7: Write protocol with allocation

During the protocol with bulk operations, the metadata server may have to instantiate clones that are children of the object being accessed. Or, it may have to instantiate the object being accessed as a child of a clone. This latter situation is shown in Figure 4.8

on page 53. The former situation only differs in the arguments of the Clone sent from the metadata server to the storage node.

When a write triggers an instantiation, that instantiation only needs to occur for the first access to the object. Once the instantiation has taken place, future write operations (assuming no further bulk operations are issued) proceed quickly and without the penalties associated with bulk operations (capability revocation and bulk operation instantiation).

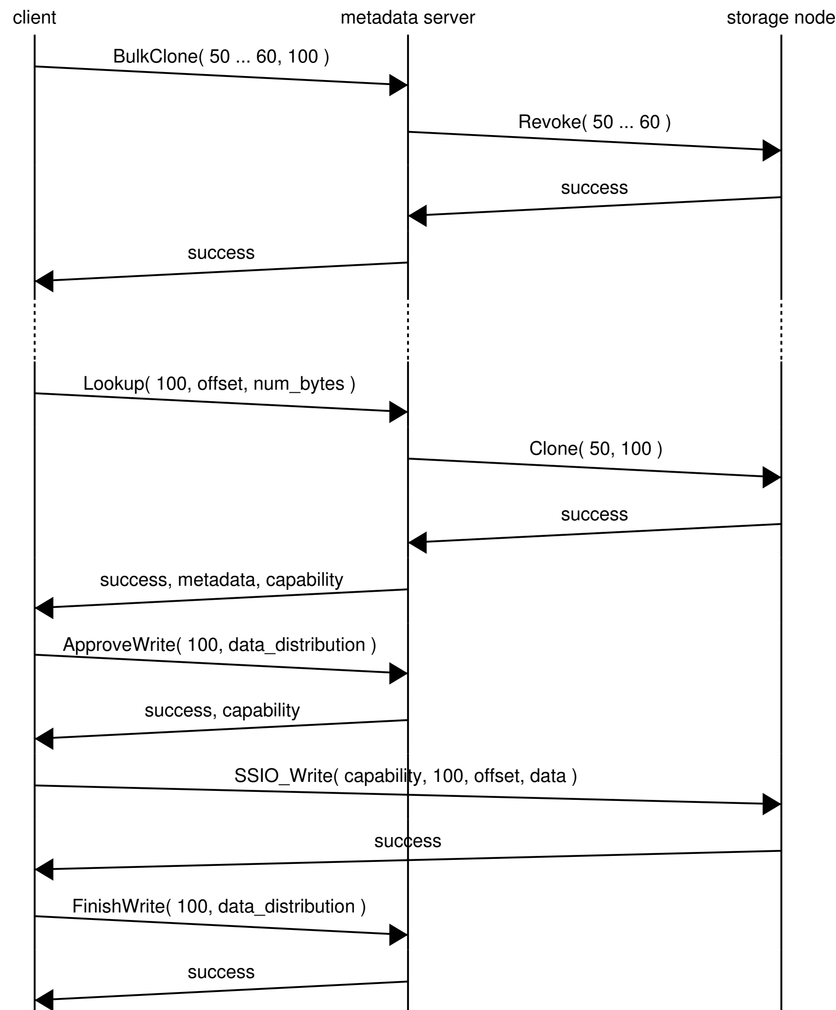


Figure 4.8: Write triggering instantiation of a clone

4.6.6 Read

The Read() command returns data from the byte-stream component of an object. There are four arguments: the object identifier, the byte-offset at which the read begins, a buffer to receive the data and the length of the buffer.

Before data can be read directly from storage nodes, the client library must acquire metadata and capabilities to the object. This is done through the Lookup command sent to the metadata server. The actual transfer of data between the client and the storage nodes is accomplished by the SSIO_Read message.

A Read can trigger clone instantiation in the same ways as the Write command. The object being read, or objects cloned from it, might need to be instantiated. The former situation is shown in Figure 4.9 on page 54. The latter situation only differs in the arguments of the Clone sent from the metadata server to the storage node.

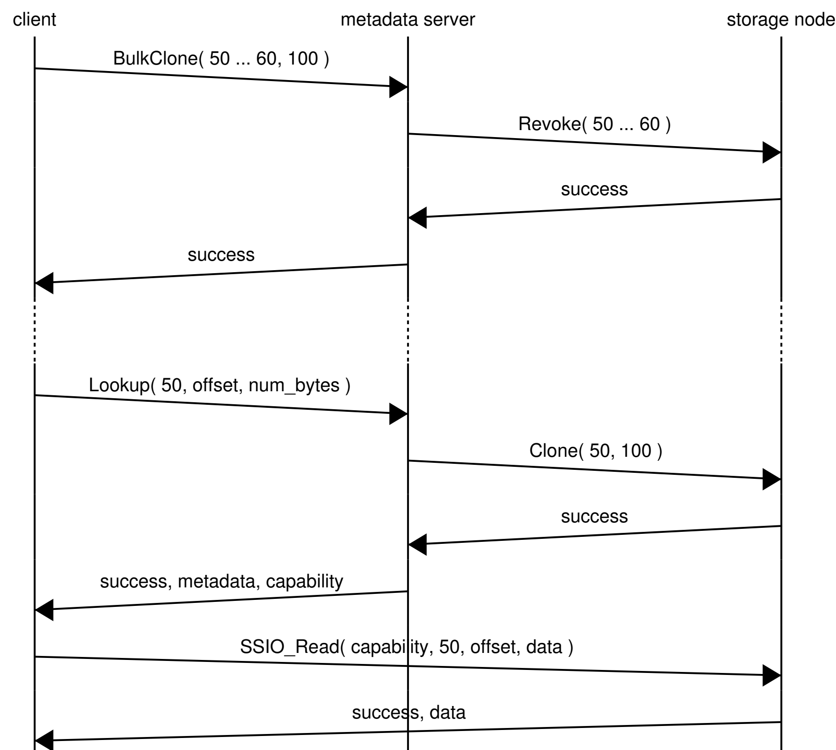


Figure 4.9: Read triggering instantiation of a clone

When a read triggers an instantiation, that instantiation only needs to occur for the first access to the object. Once the instantiation has taken place, future read operations (assuming no further bulk operations are issued) proceed quickly and without the penalties associated with bulk operations (capability revocation and bulk operation instantiation).

4.6.7 BulkDelete

The BulkDelete() command removes objects within its target range from the object namespace. There are two arguments that comprise the target range for the delete: the starting object identifier and the ending object identifier .

Like with the single-object Delete() command, the objects covered by the target range of a BulkDelete() will be made unavailable for access and any resources occupied by the objects will be freed before another object with the same object identifier can be created.

If the target range is not occupied (contains no objects), then the operation will fail as there is nothing to delete. If there is an internal error, then the operation will fail.

Figure 4.10 on page 55 shows the protocol for executing a BulkDelete operation. To restrict access prior to instantiation, the metadata server only needs to revoke capabilities at the storage nodes.

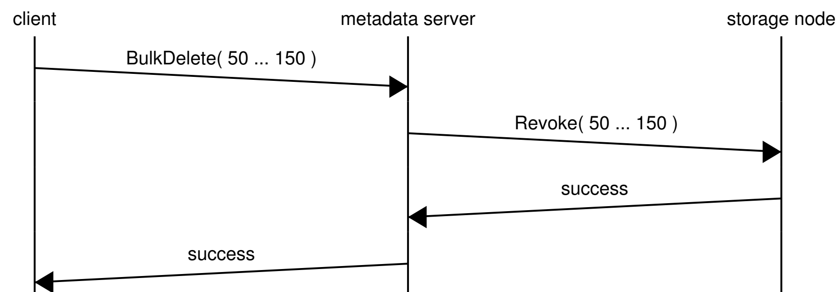


Figure 4.10: BulkDelete

4.6.8 BulkClone

The BulkClone() command creates new objects in the object namespace by making one-to-one copies of all objects in a source range to corresponding objects in a destination range. There are three arguments: the beginning and ending object identifiers for the source range, and the beginning object identifiers for the destination range. The ending object identifier of the destination range can be calculated from the span of the source range.

If the source range is not occupied (contains no objects), then the operation returns an error. If the destination range is occupied by any objects the operation fails and returns an error. This is done so that our implementation will not have to perform an exhaustive check for one-to-one correspondence between objects already existing in the source and destination ranges. If the source and destination ranges do not cover/span the same number of object identifiers, or if they overlap, an error is returned. Other internal errors (e.g., inability to revoke capabilities at sufficient storage nodes) might also cause the operation to fail.

Figure 4.11 on page 56 shows the protocol for executing a BulkClone operation. In the protocol's Revoke step, there is no need to revoke capabilities for the destination range of the clone as those object are guaranteed not to exist by the semantics of the operation.

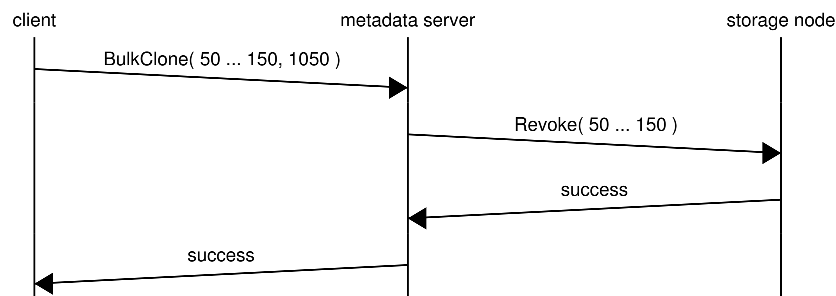


Figure 4.11: BulkClone

Chapter 5

Data structures and algorithms

This chapter provides details of how bulk operations are implemented within our distributed, object-based storage system prototype. First, the data structures are described. They are the repositories of information about the existence of objects, their corresponding metadata, and the state of unapplied bulk operations within the system. Second, the core algorithms for coping with bulk operations are described. Third, the basic operations and their integration with bulk operations are described. Fourth, the flexibility of bulk operation semantics for BulkClone and BulkDelete and the correctness of the system in the face of failures are discussed.

5.1 Data structures

Various data structures are maintained by the metadata service in order to track information necessary to determine the existence of objects and to read and write their data. For the most part, the data structures consist of tables for tracking information.

5.1.1 Sequencer

The logical clock of the metadata server is maintained as a SQL SEQUENCE object. Each time a logical timestamp is requested of the object, it increments. Rather than maintaining

a counter in a table, this feature of SQL was used.

5.1.2 Object table

The object table tracks objects that have been individually created and/or those that have metadata associated with them[1]. Information maintained includes the object identifier, the length of the byte-stream component of the object, and a logical timestamp indicating the creation time relative to other object creation/deletion in the system. Some objects that are tracked in this table may not exist because they are covered by a BulkDelete operation, but they are maintained here until the delete is instantiated.

The object identifiers are unique across all entries. The timestamp is *not* unique across all entries as a BulkClone operation can create many objects at the same logical time.

5.1.3 BulkDelete table

The BulkDelete table is the first of the two bulk operation tracking tables. All delete operations awaiting instantiation are recorded in this table. As a BulkDelete operation is issued, the target range supplied as an argument is trimmed to only represent existing objects. This step involves internal Enumerate operations to discover the true limits to the deleted range. The trimmed range and a logical timestamp are entered in the table when a BulkDelete is executed.

As deletes are instantiated, the BulkDelete table entries are split and trimmed. The split occurs at the object identifier of the delete that is instantiated. Similarly to the way the target range is trimmed at initial execution time, when instantiations occur the resulting ranges are also trimmed through the use of internal Enumerate operations. The resulting ranges are re-inserted with the original timestamp representing the logical time at which the BulkDelete occurred.

[1] When metadata is associated with an object, there is data on the storage nodes.

5.1.4 BulkClone table

The BulkClone table is the second of two bulk operation tracking tables. All clone operations awaiting instantiation are recorded in this table. The BulkClone table holds source and destination ranges for cloned objects. As a BulkClone operation is issued, the source and destination ranges supplied as an argument are trimmed to only represent existing objects. The source range is used to guide the process and internal Enumerate operations are used to trim a split range such that existing objects are at its limits. The corresponding destination range is created based on the source range. The resulting ranges are re-inserted with the original timestamp that represents the logical time at which the BulkClone occurred.

5.1.5 Object metadata tables

There are two tables that track object metadata. The PendingWrite table tracks metadata extents for which clients have performed ApproveWrite commands, but have not yet issued FinishWrite commands. The ExtentMetadata table tracks those portions of objects that have been the targets of FinishWrite commands. Other than the usage of the tables, they are identical.

Both tables track information about the byte range and data distribution for objects. The data distribution information includes block size, encoding algorithm, threshold encoding parameters and an ordered list of storage nodes. These items are described in Section 2.4.3 on page 14.

5.2 Implications of bulk operations

The two bulk operations, BulkClone and BulkDelete, were designed and built to be of general use. This means that there are fewer restrictions on their use than could have been made in a simpler system. Correspondingly, their construction is complicated. This section describes some of the interesting interactions possible with these bulk operations.

5.2.1 BulkClone

The *BulkClone* operation accepts two arguments: a source range and a destination range of objects. The source range must describe a portion of the object identifier namespace that contains objects. The destination range must be empty. Objects are made to appear in the destination range, on a one-to-one basis with the object in the source range, as a result of execution.

The storage system does not care how clients use the BulkClone operation, as long as the initial conditions are met and the post-conditions can be satisfied. It is apparent that there are two patterns in which clients could use repeated BulkClone operations (e.g., for scheduled, periodic file system snapshots).

In the first pattern, a single source range is repeatedly used along with different destination ranges. This is a *prolific clone* situation in which a single "parent" source range has many "children" that are the destination ranges. If we allow capital letters to represent non-overlapping ranges of objects, the following sequence illustrates the prolific clone idea through the sequential execution of BulkClone operations:

1. BulkClone(A, B)
2. BulkClone(A, C)
3. BulkClone(A, D)

A result of the execution of this series of commands is shown in Figure 5.1 on page 60. In the illustration, the double-headed straight line represents the object identifier namespace as a number line. Each of the lettered boxes represents a range of objects. The three curved arrows represent the three BulkClone operations.

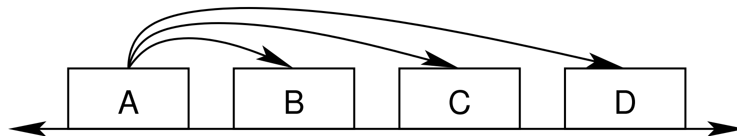


Figure 5.1: Prolific clones

In the second pattern, a destination range of a previous BulkClone is used as the source range for a subsequent BulkClone (along with a fresh destination range). This is a *chain-of-clones* in which the BulkClone operation is the "chain link" that connects the ranges of destination → source → destination. If we allow capital letters to represent non-overlapping ranges of objects, the following sequence illustrates the prolific clone idea through the sequential execution of BulkClone operations:

1. BulkClone(A, B)
2. BulkClone(B, C)
3. BulkClone(C, D)

A result of the execution of this series of commands is shown in Figure 5.2 on page 61. In the illustration, the double-headed straight line represents the object identifier namespace as a number line. Each of the lettered boxes represents a range of objects. The three curved arrows represent the three BulkClone operations.

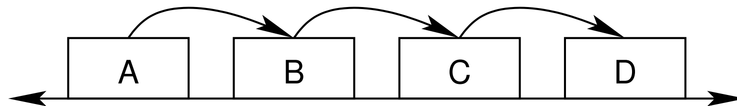


Figure 5.2: A chain-of-clones

When considering the delayed instantiation of BulkClone operations, these two usage patterns have implications for the triggered instantiations done in-line with client requests. This comes about because instantiating a clone must sever its ties with other objects in a copy-on-write manner.

So, for an access to a parent object in a prolific clone situation, all child objects must be instantiated[2]. This could potentially be many instantiations. While they could mostly

[2] Instead of instantiating all children of the parent, a single child could be instantiated and then the entries within the BulkClone table altered to use the instantiated child as the new parent for the other clones. For example, consider object A as a clone parent for objects B, C, and D. When accessing object A, the simplest instantiation (which we perform) is to instantiate B, C, and D as clones of A. One could instantiate only B as a clone of A, and then alter the BulkClone table such that B is the parent object for C and D without violation BulkClone operation semantics.

proceed in parallel, variation in response times will lengthen the overall instantiation time, further delaying the triggering request made by a client. For a chain-of-clones, only a single child clone is ever instantiated. This is a factor to be taken into consideration by a client performing BulkClone operations.

5.2.2 BulkDelete

The *BulkDelete* operation accepts a single argument: a target range of objects to delete. The target range must have "live" objects in it; i.e., there must be something to delete. The objects to be deleted are made to immediately disappear from the object identifier namespace and can be re-used.

The expected common use of BulkDelete is to quickly wipe out the clones that result from a BulkClone operation. The clones likely represent a snapshot or fork of a file system. These file system copies must be periodically re-claimed to recover storage capacity and portions of the object identifier namespace.

There are a few pathological cases of the use of BulkDelete and its interaction with BulkClone. The first case is that of thrashing the namespace: making a clone and then immediately deleting its source range. The second case is that of *pass-through clones* where un-instantiated clones, bracketed by deletes, require special handling. The algorithm that handles pass-through clones has already been presented, in Section 5.3.2 on page 66.

Sequential use of two object identifier ranges for BulkClone source and destination is possible with intervening BulkDelete operations. This use of the namespace is possible because BulkDelete can clear out a previous source range so that it can be re-used as a destination range while a previous destination range is used as a source range. As an example, consider the following sequence of bulk operations where each capital letter represents a range of objects; initially range A has objects in it and range B does not.

BulkClone(A, B) Populate range B with cloned objects from range A.

BulkDelete(A) Delete all objects in range A; only range B has objects.

BulkClone(B, A) Populate range A with cloned objects from range B.

BulkDelete(B) Delete all objects in range B; only range A has objects.

At the end of this sequence, assuming that no other operations triggered bulk operation instantiation, range A has the same namespace contents as it started out with. Objects exist in range A; range B is empty. The metadata server is tracking the bulk operations even though they have no overall effect (other than making all objects now in range A have identical sequence numbers recording their logical time of creation). Therefore, at the next access to the objects, instantiation will be required to clear out the bulk operation table. Depending on how often this series of operations is performed, a great many bulk operation table entries will be visited to decipher the current state of the storage system.

Pass-through clones

The storage system must be able to handle the case where a BulkClone destination range has a preceding BulkDelete that clears out the range and a subsequent BulkDelete that does the same. It is valid for there to be other BulkClones with source ranges that were executed between the initial BulkClone and the second BulkDelete. This situation is shown in the following series of bulk operations where there are initially objects in ranges A and B.

BulkDelete(B) Remove the initial objects from range B.

BulkClone(A, B) Populate range B.

BulkClone(B, C) Clone into range C.

BulkDelete(B) Clear objects from range B.

At the end of this series of operations, objects in range C are clones of the objects in range A. When an instantiation is next triggered in range B, a clone will have to be instantiated from an object in range A to range C. Additionally, as the metadata server searches for the metadata necessary to instantiate an object in range C, it must pass-through the deleted range B to get to range A: this is another complication that the algorithms handle. Please see Section 5.3.2 on page 66 for information on the low-level procedure that implements this logic.

One can also imagine more tortuous combinations of BulkClones and BulkDeletes that cycle over the same ranges of the object identifier namespace. For instance, namespace thrashing could be occurring along with pass-through clones. The metadata server's algorithms must be able to handle these situations correctly as the behavior of a client cannot be predicted and general bulk operations are available to them.

5.3 Core algorithms

The algorithms described in this section are used internally by the metadata server as it processes client requests.

5.3.1 GetMDOID

The GetMDOID function is used in the process of retrieving metadata for an uninstantiated clone. Given an object identifier, the function looks through the bulk operation tables until it finds the instantiated object identifier that has associated metadata describing what should be the contents of the argument object identifier.

The GetMDOID function takes two arguments, a target object identifier and a sequence number. The ultimate ancestor object identifier of the target object identifier is the object which holds metadata describing the data that the target object identifier holds. Given a simple case of an object, A, that is created, written and then cloned, the ancestor object for A is itself. If the clone of object A is A', then the ancestor object of A' is A since the contents of the two objects are the same by virtue of the clone.

The GetMDOID function returns the object identifier of the entry in the Object table that is the ancestor of the target object identifier supplied as an argument. A second argument, a sequence number that acts as a horizon at which point searching through the metadata server's tables stops, limits the scope of database table searches. This function looks back along a chain of clones and returns the ultimate ancestor object of its argument, or indicates that no such object exists.

The tortuous logic of the function can return any of three values. First, it may return "impossible" when the system cannot (or should not) have reached the state where the

search was conducted. Such a result represents a serious error that is noted in log files. Second, it may return "no metadata object" when there is no metadata object for the supplied target object identifier. Third, it may return an object identifier corresponding to an entry in the Object table from which metadata (if any) can be found for the target object.

The function first performs three queries, each of which may or may not return a valid row of data. One query returns a row resulting from a search of the Object table for the target object identifier. The returned row is the target object identifier if it has already been instantiated or otherwise created, and detectable NULL value otherwise. A second query returns the most recent BulkDelete table entry that covers the target and has a sequence number less than or equal to the search horizon. This result might later be interpreted to show that there is no ancestor object for the given target object identifier. The third query returns the most recent BulkClone table entry with a destination range that covers the target and has a sequence number less than or equal to the search horizon. This result might be used to recurse the GetMDOID function and search for a more distant ancestor of the target object identifier.

The first cases checked among the results of the three queries are those where all three return valid data. If the BulkDelete entry's sequence number is between[3] the sequence numbers for the Object entry and the BulkClone entry, then a recursive call is made to GetMDOID with arguments of the Object entry translated into the source range of the BulkClone entry, and a horizon being the BulkClone entry's sequence number. The return value of the recursive call is returned. If the BulkClone entry's sequence number is between the sequence numbers of the Object entry and the BulkDelete entry, then "no metadata object" is returned. If the Object entry's sequence number is between those of the BulkDelete and BulkClone entries, then "impossible" is returned. If the Object entry's sequence number is between those of the BulkClone and BulkDelete entries, then "no metadata object" is returned. If the BulkClone entry's sequence number is between the BulkDelete and object sequence numbers, then "impossible" is returned. If the BulkDelete entry's sequence number is between the BulkClone and object sequence numbers, then "impossible" is returned.

[3]This "between" relation is the SQL BETWEEN relation where "X BETWEEN Y AND Z" translates to "X >= Y *and* X <= Z".

If the Object and BulkDelete entries exist, but the BulkClone entry does not then there are two cases to handle. If the Object entry's sequence number is greater than the BulkDelete entry's, then return "impossible". Otherwise, "no metadata object" is returned.

If the Object and BulkClone entries exist, but the BulkDelete entry does not exist, then "impossible" is returned.

If the Object entry exists, and the BulkDelete and Bulk clone entries do not exist, then the object identifier for the Object entry is returned. This is the ultimate ancestor at the end of the original search.

If the Object entry does not exist and the BulkDelete and BulkClone entries do exist, then there are two cases to handle. If the BulkDelete entry's sequence number is greater than that of the BulkClone entry, then "no metadata object" is returned. Otherwise, a recursive call is made to GetMDOID with arguments of the Object entry translated into the source range of the BulkClone entry, and a horizon being the BulkClone entry's sequence number. The return value of the recursive call is returned.

If the Object entry and BulkClone entries do not exist, but the BulkDelete entry does exist, then "no metadata object" is returned.

If the Object entry and BulkDelete entries do not exist, but the BulkClone entry does exist, then a recursive call is made to GetMDOID with arguments of the Object entry translated into the source range of the BulkClone entry, and a horizon being the BulkClone entry's sequence number. The return value of the recursive call is returned.

If all three entries are found to be empty, then a result of "no metadata object" is returned.

5.3.2 InstantiatePassThroughLimits

The InstantiatePassThroughLimits function instantiates clones for its target object between the logical clock sequence numbers provided as arguments (a lower and an upper bound). The intended use is that clones of the target object are instantiated between BulkDeletes whose sequence numbers are provided as arguments to the function[4]. A description of

[4]Similar to the unimplemented alternative instantiation idea for chains-of-clones, in this situation we could re-write the entries in the BulkClone table for subsequent clones instead of instantiating them.

the situation in which this algorithm is applied can be found in Section 5.2.2 on page 63. An example situation which this function was design to handle is shown in Figure 5.3 on page 67.

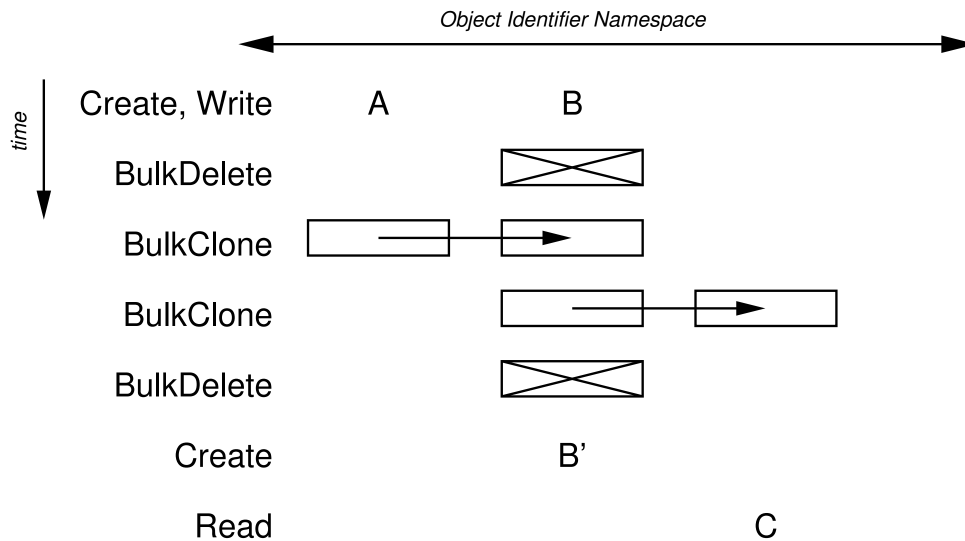


Figure 5.3: Pass-through clone example
Instantiation of the clone from object A to object C is done when object B' is created. Creating object B' empties that portion of the object identifier namespace, forcing the instantiation according to our implemented algorithms.

The first step of this function searches for a BulkClone table entry (there can be only one) with a destination range that contains the target object and has sequence number between the lower and upper bounds provided to the function. This is the “destination clone”. If there is no such BulkClone table entry, then processing continues by splitting any other BulkClone entries with source ranges containing the target object and with sequence numbers between the lower and upper bounds. After these splits are performed, the function can return successfully.

If there is a destination clone, then additional searches are performed. A set of “source clones” are sought which have source ranges that contain the target object and sequence numbers between the lower bound and the destination clone. Each of these clones are split and re-inserted into the BulkClone table.

Next, a metadata object for the target object is sought through the GetMDOID function. If no metadata object is found, then there is no object from which to make a pass-through clone (from the destination clone through to any source clones). A search is made for BulkClone table entries that contain the target object in their source ranges and have sequence numbers between the destination clone and the upper bound. These entries are split and re-inserted into the BulkClone table. Then, the destination clone is split and we return successfully from the function.

If a metadata object is found, then clones are instantiated that contain the target object in their source range and have sequence numbers between the destination clone and the upper bound. Instantiation involves first translating the target object into the destination range of the clone. Then, the translated object is inserted into the Object table. The clone is instantiated at storage nodes. The metadata object's metadata extents are copied and inserted into the MetadataExtent table for the translated object. Then, the clone is split and re-inserted into the BulkClone table. Finally, the destination clone is split and re-inserted, and the function can return successfully.

5.3.3 InstantiateHole

The InstantiateHole function is used internally to create a hole in the object identifier namespace. The function is called with a single object identifier as its argument. A variant function adds a sequence number as an argument and does not process operations with sequence numbers greater than that. It is most commonly used when instantiating BulkDeletes to remove all bulk operation dependencies on a particular object identifier. It can create child clones to break chains of clones and initiate BulkDelete instantiation. After calling InstantiateHole the object identifier namespace has no entry for the target object identifier.

Consider the use case in Figure 5.4 on page 69. We show the situation in the object identifier namespace in an initial condition in the top half of the figure. The bottom half of the figure represents the namespace after the InstantiateHole function has performed its job of clearing out an object identifier entry. It has split BulkDelete and BulkClone table entries after instantiating those operation on the necessary objects.

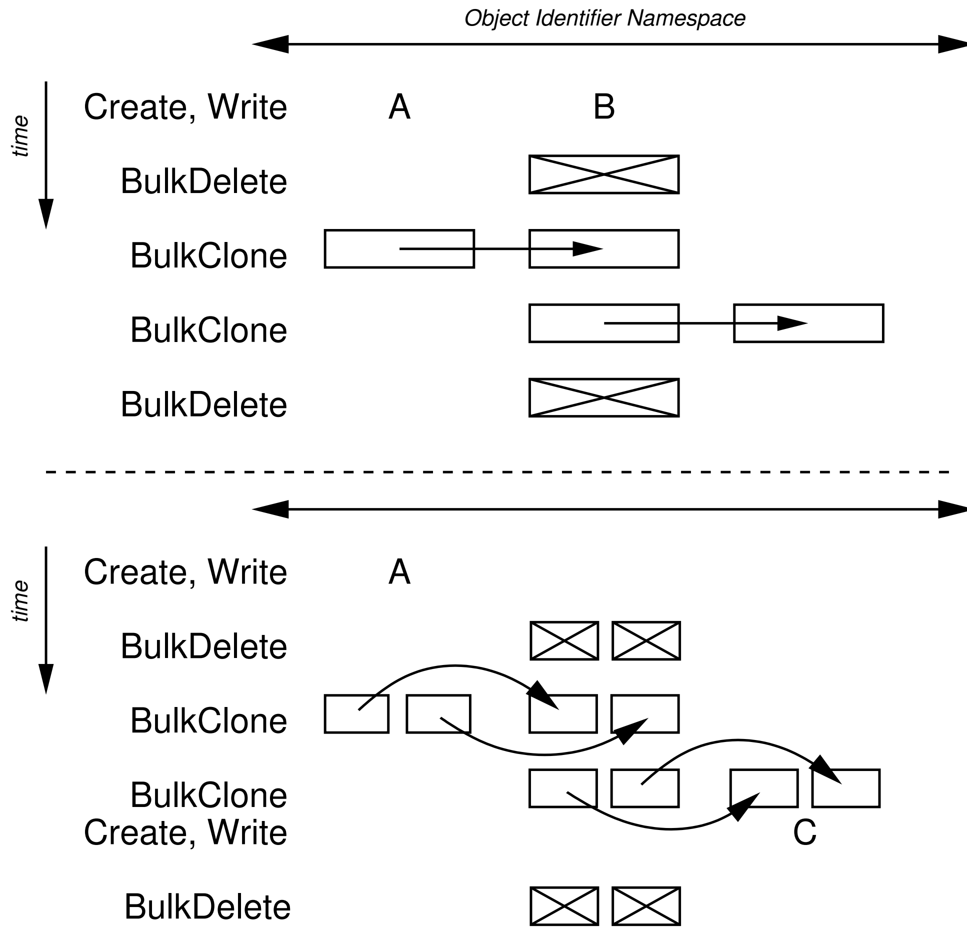


Figure 5.4: Use case for InstantiateHole function
The top half of the figure represents the state of the object identifier namespace before a call is made to InstantiateHole(B). The bottom half shows the situation after the call. Note the delete of object B, the creation of object C with the timestamp of the BulkClone from which it was instantiated, and the splitting of the bulk operation representations.

If a BulkClone table entry with destination range containing the target object is found between the limits, then the logic splits around that BulkClone entry's sequence number. Other BulkClone entries that have source ranges covering the target object and occurred between the lower bound sequence number and the BulkClone with the destination range are simply split and re-inserted. BulkClone table entries that happened after the destination clone and before the upper limit are instantiated. For each instantiation, a metadata object is searched for through the use of the GetMDOID function. If there is no source metadata object is found, then the remaining BulkClone entries are simply split and we return successfully from the function. If there is a source metadata object, then clones are instantiated. First, the target object is translated into the destination range of the BulkClone. Second, the destination object is inserted into the Object table. Third, the clone is instantiated at storage nodes. Fourth, metadata extents are copied and inserted into the MetadataExtent Table for the destination object. Fifth, the BulkClone being instantiated has its entry in the BulkClone table split and re-inserted. Sixth, the destination BulkClone from which the pass-through clones were made is split and re-inserted into the BulkClone table.

The InstantiateHole function starts out by getting the entry from the Object table for the target object. From there, one of two paths is taken, depending on whether the search of the Object table was successful.

If the target object is found in the Object table, the earliest BulkDelete that covers it is searched for. If a BulkDelete is found, then, for each BulkClone that has the target object in its source range and that has a sequence number less than that of the earliest BulkDelete (i.e., the BulkClone happened before the earliest BulkDelete), a series of steps are taken. First, the target object identifier is translated into the destination range of the BulkClone. Second, a call is made to instantiate a hole at that translated object identifier but only concerned with events that happened before the BulkClone currently being processed. Third, the translated object is inserted into the Object table. Fourth, the metadata extents for the target object are copied to the translated object. Fifth, the BulkClone being processed is removed from the BulkClone table, split around the target object identifier, and re-inserted. Sixth, the clone of the translated object is instantiated. After this BulkClone operation

processing, attention turns to that first BulkDelete. Seventh, the target object is deleted from the object table, as part of creating a “hole” in the object identifier namespace. The delete of the target object is instantiated. And, finally, the entries for the target object are removed from the MetadataExtent table[5].

If no first BulkDelete was found, then processing of the BulkClones still has to happen. This proceeds exactly as outlined in the previous paragraph. Then, the target object is removed from the system as was also described above. The function can return successfully at this point.

If processing falls through the above conditions, then any remaining BulkDelete table entries that apply to the target object are processed in order from oldest to most recent. For each of these BulkDelete entries, the subsequent BulkDelete (if any) is found. Any pass-through clones between the BulkDeletes are processed with a call to InstantiatePassThroughLimits on the target object with limits of the two BulkDelete sequence numbers (the BulkDelete being processed and the subsequent one). If there is no subsequent BulkDelete to bracket a possible pass-through clone, then the next logical timestamp is used for the upper limit of the call to InstantiatePassThroughLimits. The BulkDelete entry being processed is then split around the target object before the next one is processed.

If the target object of this call to InstantiateHole is not in the Object table, then processing proceeds similarly to the above outlined procedure. However, the necessary searches for pass-through clones make calls to InstantiatePassThroughLimits with a lower limit of 1, being the earliest logical timestamp.

5.3.4 Divorce

The Divorce function instantiates bulk operations on objects that exist. There is a single argument to the function: the object that should be divorced from any associated bulk operations. At the end of execution, the target object is no longer associated with any bulk operations.

---

[5]They are removed after instantiation of a delete since the delete needs to know which storage nodes to contact, and that information is in the metadata extents.

The Divorce function starts by calling GetMDOID for its target object. There are two cases to investigate: either the metadata object is the same as the target object, or it is not.

If the metadata object is the same as the target object, then the first step is to retrieve the object's information from the Object table. Then, child clones are instantiated for each BulkClone table entry where the target object is in the source range and the sequence number of the BulkClone is greater than that of the target object.

These clone instantiations are done in a series of steps. First, the target object is translated into the destination range. Second, the InstantiateHole function is called for that translated object identifier. Third, the translated object is entered into the Object table with the length of the target object and the sequence number of the BulkClone entry from which the instantiation is being performed. Fourth, the clone is then instantiated at storage nodes and the metadata extents are copied from the target object to the translated object.

If the metadata object is different from the target object, then the target object is simply instantiated. This process uses the steps outlined in the previous paragraph but uses the metadata object identifier in place of the target object identifier. The call to InstantiateHole takes care of making any necessary child clones of the target object.

5.4 Core operations

This section describes the algorithms behind the core metadata server operations that can trigger on-demand bulk operation instantiation. A Create operation can trigger a BulkDelete instantiation. A Delete operation can trigger a BulkClone instantiation. A Lookup operation can trigger a BulkClone instantiation. An Enumerate operation must be able to navigate the bulk operation tracking tables to determine which objects exist.

Any time an operation tests to determine if an object exists, it invokes portions of the Enumerate operation and must consult the bulk operation tracking tables. Whenever an operation requires access to the metadata for an object, logic of the Lookup operation is used. Instantiation of BulkDelete operations can ensure that the namespace is empty and the system purged of backing store for an object when a new object is being created. Such instantiations take place whenever a Create operation is invoked with the name of an object

affected by a BulkDelete. Instantiation of a BulkClone operation ensures that a source object and destination object can evolve independently. Whenever a client attempts to access a source or destination object of a BulkClone, instantiation must take place.

5.4.1 Enumerate

The Enumerate command allows for the (iterative) retrieval of the contents of the object namespace. It supports retrieval of up to 100 object identifiers at a time. A *min* and *max* parameter are supplied as arguments along with a count of objects to return.

To check for the existence of an object, multiple pieces of information must be brought together. At first glance, one might expect that if an object is mentioned in the object table, then it exists. But, it might be covered by an entry in the BulkDelete table and so *not* exist. Similarly, even if an object is not mentioned in the object table, it might actually exist due to a chain of clones that can be tracked back to another object in the object table. And, for that chain of clones to be complete, it cannot have any intervening BulkDelete operations that cut the chain.

A recursively called SQL statement merges the information described above to produce the results needed to satisfy an Enumerate operation. For the target range of the Enumerate, all objects in the Object table that are within the range, except those covered by a BulkDelete, are placed into the set of object identifiers to return. That set is merged with cloned objects that have a destination within the target range. To pull in the cloned objects, the entries of the BulkClone table with destination ranges overlapping the target are involved.

Each of the overlapping clone destinations is first trimmed to match the enumeration range. Then, objects in the source range of each clone within the enumeration range are picked from the Object table, unless the source or destination range of the clone is covered by a BulkDelete. These objects pulled from the clone's source range are recursively merged using a UNION construct with objects pulled from the source range of the clone.

The set of objects pulled forward into the enumeration range are then checked against any BulkDeletes that might affect them. This check must be made since BulkDeletes might cover objects pulled in.

5.4.2 Create

The Create command causes a target object's object identifier to appear in the namespace. Just after a Create, there is no backing-store for the byte-addressable contents of the object. Only a single object can be created with each Create command, and a command fails if the object already exists.

A Create operation can trigger an on-demand instantiation of a BulkDelete. Since identifiers can be re-used, this situation must be handled. When re-creating an object, the Create will fail if the delete on the object cannot be instantiated.

Before a Create can proceed, the object identifier being created must not exist. When processing a Create, the first step is to Enumerate the target object. If it already exists, then the operation fails. If the enumeration returns that it does not exist, then a "hole" must be created that will create any necessary clones of the previously deleted object and then instantiate the delete.

5.4.3 Lookup

The Lookup command returns metadata and capabilities for a byte-range of an object. The object must already exist, but there may or may not be metadata for the requested byte range.

While processing a Lookup, the metadata server first performs an enumeration to verify that the target object exists. If the object exists but is not explicitly mentioned in the Object table, then the existence was caused by a BulkClone operation. To create an independent object, the target object is "divorced" from its parent and any child objects. This divorce also happens if the object does not exist because of a clone, in order to allow any child clones to evolve independently of their parent.

After the object has been freed of its parent and all children, metadata for the requested byte range is returned.

5.5 Correctness

This section informally argues the correctness of our design in the face of failures. Components can fail, and messages can be lost, and the bulk operation semantics are obeyed.

The behavior expected of bulk operations are as follows. They are atomic; either they are applied to all relevant objects or none of the objects are affected. All clients observe the same state of the objects affected by bulk operations. Instantiation of a bulk operation follows the execution of that bulk operation and delayed instantiation must be indistinguishable from an apparent immediate instantiation. A BulkClone operation makes copies of all objects in the source range into the destination range. A BulkDelete operation removes the affected objects from the object identifier namespace.

There are two parts to bulk operation processing. First is the initial processing of the bulk operation by the metadata server. Second is the instantiation of the bulk operation upon affected objects. The consequences of failure at each stage in these two protocol exchanges are outlined in the following sections.

5.5.1 Initial bulk operation processing

The initial processing of a bulk operation involves just two steps. First, the bulk operation command is sent by the client to the metadata server. Second, the metadata server revokes capabilities by contacting storage nodes. These steps are shown in Figure 5.5 on page 76. The protocol involves a client, metadata server, and at least one storage node. Errors can occur during communication or within any of the protocol participants.

Case: Metadata server does not receive client command

Bulk operations are initiated by clients that send a bulk operation command to the metadata server. If a network failure causes a client's bulk operation command to not make it to the metadata server, then no bulk operation will be executed. A client will receive an error message from its own networking stack indicating that the message was not sent. The client can observe the status of the system by using the Enumerate command to verify that the bulk operation did not occur. For a BulkClone, an enumeration of the destination range

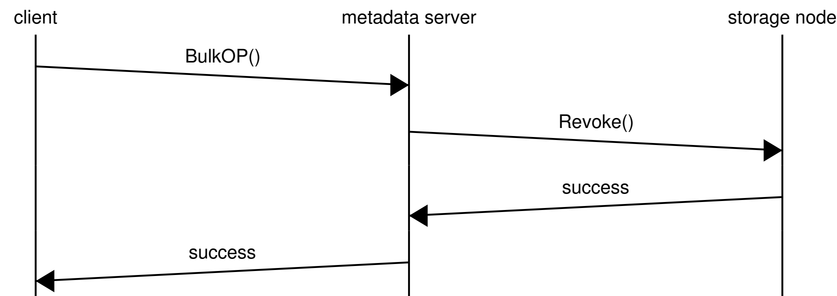


Figure 5.5: Initial bulk operation processing

should return empty. For a BulkDelete, an enumeration of the target range will return some objects if it was not already empty.

Case: Internal metadata server error

Should the metadata server crash before it can complete the processing of a bulk operation, the client will not receive a completion notification. Because the database uses transactions to update its tables, the bulk operation table updates will either be completed fully, or not at all. The client can use Enumerate to determine whether or not the operation completed and, if not, retry the operation.

While a metadata server is down, clients will still be able to use cached capabilities (if they have not been revoked) to read and write data at storage nodes. They will not be able to execute a FinishWrite operation or create or delete objects.

Case: Capabilities not revoked

To complete a bulk operation, the metadata server must be able to revoke capabilities for affected objects. If a communication error between the metadata server and storage nodes prevents this from happening, then the bulk operation will be aborted and the client informed of failure.

If some storage nodes can be contacted to perform revocation, but others cannot, then the success of the operation is dependent on the data encoding threshold of the affected objects. If enough storage nodes can be contacted to ensure that no client can successfully

read or write data, then the bulk operation can succeed. If enough storage nodes do not respond to capability revocation, then the bulk operation will fail. Clients can recover after some storage nodes reject their cached capabilities by re-contacting the metadata server to acquire fresh capabilities.

Case: Client never receives "success" from MDS

If the client that initiated a bulk operation never receives a "success" message back from the metadata server, there are a number of things that could have gone wrong (as mentioned above). The client can determine if the bulk operation succeeded by using the Enumerate operation in the manner previously described.

5.5.2 Bulk operation instantiation

The instantiation of a bulk operation is triggered by a command sent by a client to the metadata server. As examples, a Lookup command can trigger BulkClone instantiation, and a Create command can trigger a BulkDelete instantiation. In each case, the metadata server must instantiate the bulk operation at the storage nodes before returning a "success" code to the client. The steps for generic bulk operation instantiation are shown in Figure 5.6 on page 77. The protocol involves a client, metadata server, and at least one storage node. Errors can occur during communication or within any of the protocol participants.

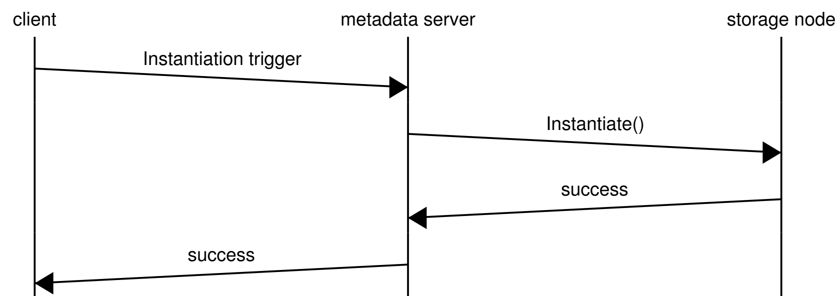


Figure 5.6: Instantiation of a bulk operation

Case: Metadata server internal error

If the metadata server fails after receiving the operation that should trigger instantiation, but before performing the instantiation, then the client will not be able to perform the requested operation. At this point, there are no valid capabilities for affected objects so bulk operation semantics will be preserved.

The client can retry the triggering operation after the metadata server recovers.

Case: Unsuccessful instantiation

If the metadata server cannot successfully instantiate the bulk operation at the storage nodes, then the client's access cannot be allowed to proceed. A successful instantiation involves contacting enough storage nodes, based on the encoding of the data distributions used in the target object. Internal errors at storage nodes or communication difficulties between the metadata server and storage nodes can thwart instantiation.

Partial instantiation can be tolerated. A partial instantiation of a clone can be resolved by deleting the destination objects. A partial delete can be resolved by completing the delete at a later time. Client access, however, must continue to be restricted until an instantiation is completed. This might appear to be a slight confusion in the semantics of bulk operations: the instantiation was promised to have been made visible to clients at execution time, but now access can be denied. However, if one assumes that storage nodes are as available to clients as they are to the metadata server, then, if the metadata server cannot get to them, neither can the clients. This makes the denial of access by the metadata server moot, since the client would not be able to access the object anyway.

Case: Client does not receive response

A client might not receive a response from the metadata server if the metadata server fails, or if the client fails. The client does not know whether or not the instantiation completed and can judge that by the return code of a re-tried operation.

Upon re-try, the client will observe different return codes based on the completion of the instantiation. If a BulkClone was instantiated or not, the re-tried triggering operation

should succeed. If a Create command triggered the instantiation of a BulkDelete, then if the Create was successful, it will fail when re-tried since the object already exists. If that initial create failed, then the re-tried Create can succeed after the instantiation is performed. (If it was simply and RPC interruption, then an RPC replay cache could return the expected result to the client.)

Chapter 6

Evaluation

The evaluation of bulk operations demonstrates three points. First, the bulk operation implementation described in this dissertation comes at a cost, which is quantified. Second, background instantiation of bulk operations can be used to reduce that cost. Third, they can be practically used while providing access to clients of the storage system.

The costs of bulk operations are measured using differences in experiment execution times. The storage system implementation includes switches for disabling bulk operation routines. The cost of bulk operations can thus be measured as the difference in execution time for an experiment with, and without, bulk operations enabled. These timings are be built up to an understanding of where time is spent during the servicing of bulk operations.

The effect of the background instantiation algorithms are measured by experiment execution times. By running an experiment with, and without, background instantiations being performed, the savings of running those algorithms are calculated. The costs are eliminated, of course, if a system remains idle for long enough. When background bulk operation instantiations compete with foreground workloads and on-demand instantiation, the benefits will be reduced.

Demonstrating the practicality of bulk operations is done using an NFS server. The files and directories of the NFS server are stored within the distributed object-based storage system. Standard NFS clients access data through the NFS server. The NFS server uses BulkClone operations to generate snapshots of the file system. The effects of those

snapshots on client access are measured.

This chapter ends with a short summary of the experimental results.

6.1 Experimental setup

Experiments are run across a set of identical machines. Each is a rack-mount server with a pair of Intel Xeon processors running at 3.0 GHz with 2 GB of RAM. The systems are connected via a switched 1 Gb/s Ethernet network. All experiments are run in-memory so the hard disks should not affect the outcome of experiments.

The software used is as follows. The operating system is Debian Linux with a version 2.6.16 kernel and swapping disabled. The metadata server uses PostgreSQL 8.1 configured with fsync of the write-ahead log turned off.A version of the Ursa Minor storage system [1] is used as the distributed, object-based storage system. The storage system is run only under fault-free circumstances.

The launching of experiments and various software components of the distributed, object-based storage system is controlled by a single process. Actions on separate machines are coordinated via a set of `ssh` control connections that spawn and monitor local processes. Once the storage system has been started, any experiments can be run against it. The storage system is restarted cleanly between experiments, unless otherwise noted.

6.1.1 Data collection and instrumentation

The time required to execute an operation is measured as the difference between a starting and an ending time for a function call. Various functions measured are "wrapped" by a pass-through function that starts a timer, calls the function, stops a timer, and records the difference between the stop and start in a data structure. Other functions are explicitly annotated for timing. A trigger dumps the data structure at the end of a run. By saving the data in-memory and writing it later, minimal overhead is introduced into the running program. Post-processing is used to extract statistics and generate graphs of the time-series data.

Averages and standard deviations of experimental results are presented in the text and tables of this chapter. Additional statistical summaries of the results can be found in the Appendix, which starts on page 156.

6.1.2 Workload scripting

A scripting program is used to generate the necessary function calls in the experiments making "raw" calls to client library functions. The scripting program takes no longer to execute than a custom C/C++ program to execute the same calls. During execution, the scripting program stores information about storage system objects that have been created and written so that it can know what objects remain to be created, can be deleted, and can be read. The timing of each operation invocation is recorded, including both the time since the start of the script at which the operation was issued and the duration of execution. BulkClone operations can, of course, create objects and BulkDelete operations can delete them.

The script files optionally specify a random seed number used with a random number generator and otherwise contain a number of stanzas describing workloads. The description of a workload has six components: a capability revocation schedule, an object identifier generator, an access generator, a termination condition, a set of operations, and a data distribution.

Revocation schedule The revocation schedule will revoke capabilities every so many seconds during the execution of the current workload. It is used to study the effects of capability revocation on client accesses. This particular revocation function is especially made available for use by this scripting program; it is not available to clients and is used internally by the metadata server. In order to perform revocation, the function uses cached information about the storage nodes in the system and contacts them directly to revoke capabilities for all object identifiers covered by the object generator.

Object generator The object generator operates over a range of objects, specified by starting and ending object identifiers. Within that range of objects, the object generator

processes the object identifiers sequentially or randomly as specified. When an operation is executed, it requests an object identifier from the object generator to use as an argument.

Access generator The access generator operates similarly to the object generator. Given starting and ending bytes and an access size, the access generator selects either sequential or random addresses for requests. It is used by read and write operations to pick bytes as the target of input or output.

Termination condition A termination condition exits the current workload stanza when one of three conditions is met. Only one condition can be specified for each stanza. Termination can occur when a particular number of operations have been issued. This is useful when creating, writing, and reading particular objects as there can be three workload stanzas, each terminating after executing 1000 operations, thus creating 1000 objects, writing them and then reading them. Termination can occur when a certain number of bytes has been accessed. This allows a workload stanza to conclude after reading and/or writing a certain amount of data. Termination can occur when a certain number of seconds has passed. This allows for experiments limited by time.

Operations The operations are specified by name and a fraction of the whole workload that their execution should comprise. The operations are Create, Delete, Read, Write, BulkDelete, BulkClone, and Sleep. The BulkClone operation has additional parameters: a probability for performing a chain-of-clones and a probability of changing the object generator range to the source range of a random clone along a chain-of-clones. The Sleep operation has a parameter of how many seconds to sleep and allows for slack time during which the metadata server can perform background instantiations.

Data distribution description A workload stanza contains a data distribution description. This includes, among other things, the threshold parameters for $m-of-n$ encodings and a block size (that can be different from the access generator block size). The storage nodes available for use are obtained from the storage system.

The workload scripting program records timing information for each operation invocation, including the time since the start of the script at which the operation was issued and the duration of execution. Since a workload can execute an operation in different circumstances, the timings of an individual operation can vary widely. Consider, for instance, the case of workload stanzas that create and write to 1000 objects, followed by a BulkDelete of those 1000 objects, and finally re-creates those 1000 objects. The timing of the first 1000 creates would be expected to be faster than the second 1000, which must instantiate a delete. In this example, the average and median execution times, the minimum and maximum execution times, and the distribution of the data points are of interest. The average and median provide expected operation durations. The minimum and maximum give a sense for the best and worst case execution. The distribution of data points allows one to interpret any skew away from the mean and median.

6.2 Baseline behavior

The baseline experiments determine expected values for operation execution. This information provides context for the costs associated with the use of bulk operations in the prototype storage system.

This section contains four categories of experiments. The database access experiment establishes the minimum cost of performing an operation at the metadata server. The capabilities experiments establish timings for various aspects of capability generation and revocation. The Create baseline experiments expose the overheads of bulk operation algorithms. The Write and Read baseline experiments measure other overheads of bulk operations and show the effects of cached capabilities and metadata.

6.2.1 Database access experiment

The implementation of a metadata server in a PostgreSQL database was undertaken with full knowledge that it would not perform as well as a custom solution. There are remote procedure call (RPC) overheads associated with calling from C/C++ code, executing the

standardized "embedded SQL in C" interface, and invoking the database where server-side functions execute the operation logic and query various tables. These overheads are a basic part of every interaction with the back-end of the metadata server.

To measure this cost, a custom "ping" function was implemented along the execution path of back-end metadata server operations. The function is installed into the database just like other back-end functions, makes no calls of SQL operations, and simply returns immediately (i.e., `BEGIN RETURN(0); END;`). From the client library, this ping function is invoked, and the execution time is measured.

The experimental setup involves two computers. One is the client issuing the calls to the ping function. The other is the metadata server, with a local instance of PostgreSQL running the entire metadata server (front-end, back-end, and helper). The results of repeated invocations of the ping function are shown in Table A.1 on page 157. We see that any access to the database at the back-end of the metadata server takes 5.1 ms on average.

6.2.2 Capability experiments

The capability system is key to our bulk operation algorithms. All bulk operations revoke capabilities to force clients to contact the metadata server, at which point on-demand instantiation takes place.

There are three aspects of capabilities exercised experimentally here. The first is the time it takes to generate a capability. The second is the time needed to acquire a list of all storage nodes in the system, a necessary part of a successful large-scale capability revocation. The third is the actual revocation time observed at the metadata server as it contacts storage nodes.

The metadata server generates capabilities with a short sequence of C code. This C code is invoked by a pl/pgsql server-side function in normal operation and in our experiment. The experiment runs in the PostgreSQL database and simply calls the capability generation function 100,000 times and returns the total execution time. We divide this total execution time by the number of operations performed to find that the average time necessary to generate a capability is 62.5 μs. As this is orders of magnitude faster than other operations,

we will not consider it as contributing to the overall execution of bulk operation related processing in the future.

In order to revoke capabilities at storage nodes, the metadata service must first have a means of contacting those storage nodes. This must be done every time the set of storage nodes changes[1]. The acquisition of a list of storage node addresses queries an internal interface within the prototype storage system. This interface communicates with an in-memory database of all registered “components” in the storage system where each component represents a contactable entity (e.g., program) that provides some service. Storage nodes are such entities, as is the metadata service. In this experiment, there are two storage nodes that are enumerated. The information returned enables the metadata service to contact the storage nodes through a communication interface and instruct them to revoke capabilities.

There are only two computers taking part in this experiment. One holds the storage system component addressing information The other queries that information to retrieve the list of storage nodes. There are also the two storage nodes running, but they are not contacted during this experiment. They are only present to register their endpoints.

The average time to acquire the list of storage nodes is 1.35 ms.

Capability revocation is driven by the front-end to the metadata service. When an operation is decoded, if it is a bulk operation, the front-end to the metadata service revokes capabilities before sending the operation to the back-end PostgreSQL server-side functions for additional processing. This experiment repeatedly calls the same revocation function used by the front-end. The front-end informs all active storage nodes of the need to revoke capabilities. In this experiment, two storage nodes are active in the system. Cached information about storage nodes in the system is used for communication addressing (acquisition of a list of storage nodes is not part of this experiment).

Three computers take part in this experiment. One is the metadata server making revocation calls. The other two are the storage nodes receiving and processing revocations.

The average time to revoke capabilities at two storage nodes is 1.24 ms.

[1]The prototype system does not contain logic for informing components of changes to the set of storage nodes. For correctness and even though we operate in fault-free mode, we request the list of storage node addresses every time we perform a bulk revocation of capabilities.

Summary

It takes about 62.5 μs to generate a capability. It takes 1.35 ms, on average, to acquire addressing information for two storage nodes.It takes 1.24 ms, on average, to revoke capabilities at two storage nodes. These are basic costs associated with the capability component of bulk operations.

6.2.3 Create

The Create operation populates the storage system with addressable objects. The algorithms associated with bulk operations add overhead to the basic process of making sure an object being created does not already exist. The purpose of these experiments is to determine that overhead.

In these experiments, the Create command is used to create objects in a range of objects, sequentially or randomly, with or without bulk operation algorithms active. The bulk operation code is always present in the metadata server, but a switch has been installed to choose whether those code paths are followed. Background instantiation code is similarly de-activated, as it is irrelevant to basic Create operations.

The Create operations are issued by a program making calls into the client library. The library forwards the requests across the network to the metadata server. At the metadata server, the front-end passes the operation to the back-end, where a PostgreSQL server-side function processes the operation. Then, the operation's result is returned to the client. We measure the client observed time for the Create operation.

The first experiment is of Create operations called with object identifiers selected sequentially through a range of objects without bulk operation code active. The average time to complete a Create operation in this case is 10.0 ms.

The second experiment is of Create operations called with object identifiers selected randomly across a range of objects without bulk operation code active. The average time to complete a Create operation in this case is 10.3 ms.

The third experiment is of Create operations called with object identifiers selected sequentially through a range of objects with bulk operation code active. The average time to complete a Create operation in this case is 15.6 ms (with a standard deviation of 3.00 ms).

The fourth experiment is of Create operations called with object identifiers selected randomly across a range of objects with bulk operation code active. The average time to complete a Create operation in this case is 16.0 ms (with a standard deviation of 1.3 ms).

Summary

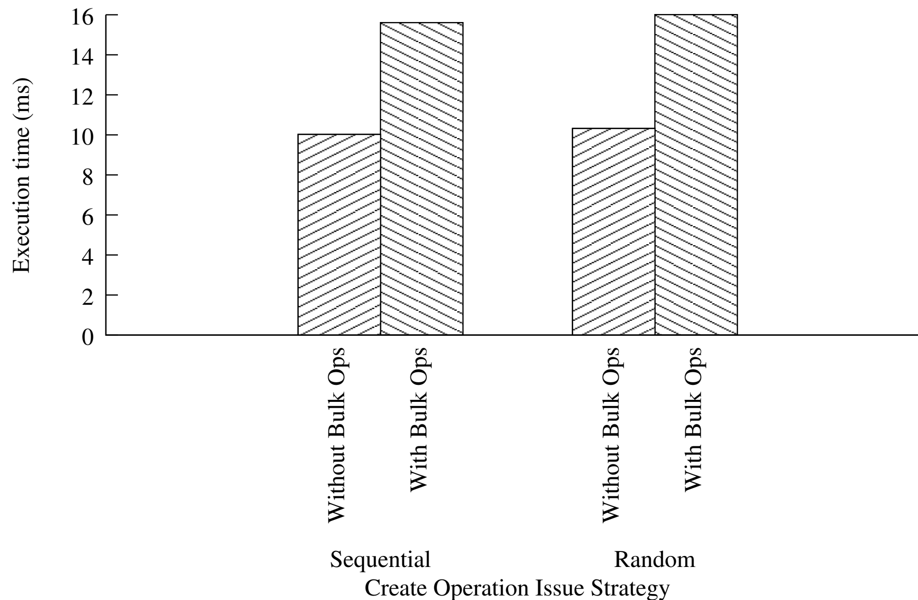


Figure 6.1: Create operation timing comparison
A graphical display of the comparative average timing of Create operations issued sequentially and random both with, and without, bulk operation code active. Thus the average overhead induced by the bulk operation algorithms is quantified at 5.6 ms for sequential Create and 5.7 ms for random.

The chart in Figure 6.1 on page 88 summarizes the results of our Create experiments. Without bulk operation code active, it takes about 10.0 ms to create a single object sequentially. With bulk operation code active, it takes about 15.6 ms to create a single object

sequentially. The expected overhead for sequential create of bulk operations, is $15.6 - 10.0 = 5.6$ ms, or 56.0%. Without bulk operation code active, it takes about 10.3 ms to create a single object randomly. With bulk operation code active, it takes about 16.0 ms to create a single object randomly. The expected overhead for random create with bulk operations code active, is $16.0 - 10.3 = 5.7$ ms, or 55.3%. As more and more objects are added to the metadata server, it was verified (by examination of the time-series data) that there is no increase in the time required to create an object.

6.2.4 Write

The Write operation places data into objects of the storage system. The algorithms associated with bulk operations add overhead to the basic process of making sure that an object being written exists, checking for a need to instantiate a clone, and checking for existing metadata (or metadata that will exist upon clone instantiation). The purpose of these experiments is to determine that overhead and point to its source.

The basic Write protocol was shown in Figure 4.7 on page 52. In this experiment, we will measure timings of the protocol as part of a Write of 64 kB of data using an underlying data distribution with 16 kB block sizes and a single storage node. There are three computers involved in this experiment: the client, the metadata server, and a lone storage node. The experimental run first creates objects sequentially in a fresh storage system and then sequentially processes the objects again, issuing a 64 kB write to each of them. A run is made with bulk operation code *disabled*. Then, a separate experiment (on a fresh system) is made with bulk operation code *enabled*.

An unexpected discovery was made during these experiments. When a program calls the client library Write function, there are actually *two* Lookup RPCs performed. The first Lookup discovers that the object has not been written to and has no metadata associated with it. The client library logic then invokes a function to choose a data distribution. That function *also* calls Lookup to check for any pre-existing metadata that it could be compatible with when it chooses a distribution for a neighboring and unallocated portion of the object's data stream. So, the protocol underlying the Write operation looks like that shown in Figure 6.2 on page 91.

These experiments are designed to exercise the protocol and break out the various steps of Lookup, ApproveWrite, SSIO_Write (low-level write that transfers data), and Finish-Write. When portions of the protocol execute with cached capabilities and metadata, timings are also presented. When capabilities are rejected by storage nodes, timings are also measured and presented.

One could take these results for 64 kB Writes and extrapolate results to larger Write operations. This would be done by holding constant all timings but that for the low-level SSIO_Write protocol phase. As SSIO_Write is the only data transfer operation, adding time here for multiples of 64 kB would amortize the setup costs of the other protocol components for an overall Write.

Write without bulk operations code

Our first experiment with the Write operation provides us with a baseline for timing. The system is setup with *no* bulk operations code paths active and *no* background instantiations being performed. The client library Write call is used with a timing wrapper and other functions have been annotated to report execution timing.

The experiment proceeds in two phases on the client. In the first phase, the client program issues Create operations sequentially to a set of 1000 objects. In the second phase, the client program issues sequential Write operations sequentially to the 1000 objects. The Write operations are for the first 64 kB of data in the objects and the data distribution is 1-of-1 with 16 kB blocks. The Write operation goes through the protocol steps show in Figure 6.2 on page 91.

The first component of the Write issued by the client is the Lookup operation. For each of the 1000 Write operations, there are two Lookup operations, totaling 2000 Lookup operations in this experiment. The first Lookup for an object is used to discover if there is any existing metadata for the object that should be used for the Write operation. The second Lookup is performed by the client library as it attempts to choose a data distribution for the Write. The Lookup operations take, on average, 7.63 ms (with a standard deviation of 1.2 ms). There is no substantial difference in timing for the two calls made in each

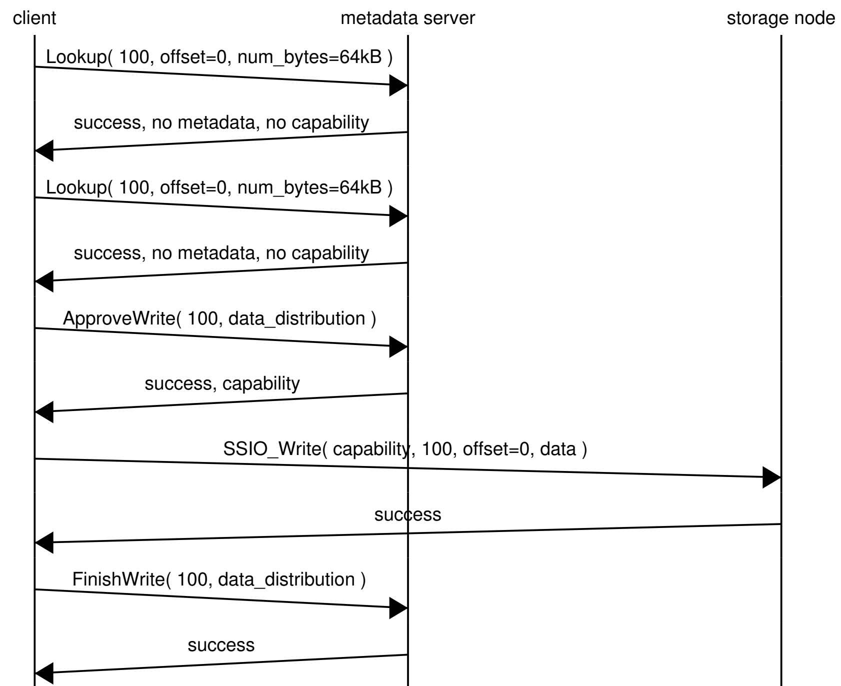


Figure 6.2: Write protocol with allocation and *extra* Lookup

protocol interaction. These calls do extend to the back-end of the metadata server and therefore incur the 5.1 ms cost of going to the back-end.

The second component of the Write issued by the client is the ApproveWrite operation. The ApproveWrite conveys to the metadata server the proposed data distribution for the overall Write operation on a particular object. The ApproveWrite operations take, on average, 11.1 ms (with a standard deviation of 1.92 ms), and since these calls do extend to the back-end of the metadata server, they also incur the included 5.1 ms cost of going to the back-end.

The third component of the Write issued by the client is the SSIO_Write operation. This accomplishes the low-level transfer of the data from the client to the storage node, using metadata and capabilities from the ApproveWrite operation. The SSIO_Write operation does not involve the metadata server and, on average, it takes 6.31 ms (with a standard deviation of 1.15 ms) to write 64 kB of data. This translates into an average data transfer rate of 10.1 MB/s.

The fourth and final component of the Write issued by the client is the FinishWrite operation. This makes the written data available to be read by other clients. The FinishWrite operations take, on average, 14.7 ms (with a standard deviation of 3.85 ms). These calls do extend through the front-end of the metadata server to the back-end for processing by a PostgreSQL server-side function and therefore incur the 5.1 ms cost of going to the back-end.

With bulk operation code disabled the average time to Write 64 kB of data from the perspective of a process using the client library is 47.4 ms (with a standard deviation of 7.67 ms). This translates into an average transfer rate of 1.35 MB/s for the overall operation. This is quite a bit different from the actual transfer of data performed by the SSIO_Write operation at 10.1 MB/s. So, the actual data transfer is quite quick, but the surrounding protocol operations drags down the timing for the overall operation.

Write with bulk operations code

The second experiment is identical to the first, except that bulk operations code is active for the operations executed. Operation timings are recorded as objects are created and then

written. The timing information acquired in this experiment illustrates the overhead of bulk operations when compared against timing information from execution without bulk operations code gathered in the previous section.

With bulk operation code enabled, the SSIO_Write operations take, on average, 6.41 ms (with a standard deviation of 0.562 ms) to write 64 kB of data. This translates into an average data transfer rate of 9.98 MB/s.

With bulk operation code enabled, the average time to Write 64 kB of data from the perspective of a process using the client library is 54.3 ms (with a standard deviation of 8.27 ms). This translates into an average transfer rate of 1.18 MB/s.

Table 6.1: Write() operation differences with bulk operations – Timing for the components of a Write() operation with and without bulk operation code active.

Operation	With bulk ops (ms)	Without bulk ops (ms)	Difference (ms)
Write()	54.317	47.377	6.940
Approve_Write()	13.652	11.052	2.600
Finish_Write()	15.802	14.662	1.140
Lookup()	9.184	7.635	1.549
SSIO_Write()	6.413	6.308	

The results for Write with, and without, bulk operation code active are summarized in Table 6.1 on page 93. The table shows that bulk operation algorithms increase the time necessary to perform basic Write operations by 6.94 ms (14.6%) compared with running the system without bulk operations. And this is just in code paths followed to exercise the bulk operation logic as no bulk operations have been called during these experiments. The SSIO_Write code does not involve the processing of bulk operations, so the average execution time of those operations does not significantly change and is virtually unchanged.

Re-Write with good cached capabilities, without bulk operations code

The third experiment exposes the importance of capabilities to performance. There is a significant difference in the protocol steps and time necessary for a client to complete a Write operation when cached capabilities are already held as compared to when revoked capabilities are held. Since bulk operations revoke capabilities, this information is relevant to understanding the impact of capability revocation on client accesses after bulk operation execution. The experiment is conducted without bulk operations code enabled. Background instantiation is irrelevant here since no bulk operations are issued.

The operations of the experiment are issued in three phases. In the first phase, the client program issues Create operations sequentially to a set of 1000 objects. In the second phase, the client program issues sequential Write operations sequentially to the 1000 objects. The Write operations are for the first 64 kB of data in the objects and the data distribution is 1-of-1 with 16 kB blocks. In the third phase, the client program re-issues sequential Write operations sequentially to the 1000 objects. This third phase is the item of interest for this experiment.

The re-Write of the third phase of the experiment will proceed quickly since the client has cached capabilities and metadata for the byte range to be written. There is no need to perform the ApproveWrite and FinishWrite stages of the protocol as the byte range already has metadata describing its data distribution. The timing for the re-Write operation is, therefore, expected to compare favorably with the previous SSIO_Write timing which had an average execution time of 6.31 ms.

Over-writes of 64 kB of data in 16 kB blocks with no bulk operation code active and no background instantiation takes, on average, 5.82 ms (with a standard deviation of 0.211 ms) to re-Write data. The data compares favorably with the average timing for the SSIO_Write timing (6.31 ms) of the initial data.

Re-Write with invalid cached capabilities, without bulk operations

The previous experiment showed the effect of over-writing with valid cached capabilities. This fourth experiment demonstrates the effect of over-writing with invalid cached capa-

bilities. This is the situation that a client would encounter after a bulk operation has been executed and capabilities have been invalidated.

The experiment is very similar to the third experiment presented with one difference. Between initial Write to the objects and the re-Write, a capability revocation message is sent to the storage node. Therefore, the protocol operation of the re-Write will have the client first trying the Write and being rejected because of invalid capabilities. The client will then re-contact the metadata server and perform a Lookup to get new capabilities. Using the new capabilities, the Write to the storage node will succeed.

In this experiment, the overall Write operation on the client averages 31.3 ms (with a standard deviation of 6.27 ms).

The first Lookup operation performed by the client finds cached capabilities and metadata within the client library. This operation is quick and takes an average of 0.094 ms (with a standard deviation of 0.027 ms).

The Lookup operation that re-acquires capabilities must travel across the network to the metadata server. This operation also retrieves and returns stored metadata for the object. The average time required to perform that Lookup is 19.4 ms (with a standard deviation of 6.24 ms).

The two low-level SSIO_Write operations that occur for each high-level Write during this experiment each take the same amount of time. The first instance of SSIO_Write to the storage node fails because of the use of invalid capabilities by the client. The second attempt succeeds after the client acquires fresh capabilities. The average time to perform each of the low-level writes (two of them per high-level write) is 5.79 ms (with a standard deviation of 0.273 ms).

Data for Write with, and without, bulk operation code active is displayed in Figure 6.3 on page 96 and shows the 6.94 ms overhead of bulk operations code.

Client library execution of a Write operation with cached metadata and revoked capabilities takes over 5 times longer (31.3 ms) than with cached metadata and valid capabilities (5.82 ms) when bulk operations code is disabled, as shown in Figure 6.4 on page 97. This is a penalty that will be paid by clients after the capability revocation step of a bulk operation.

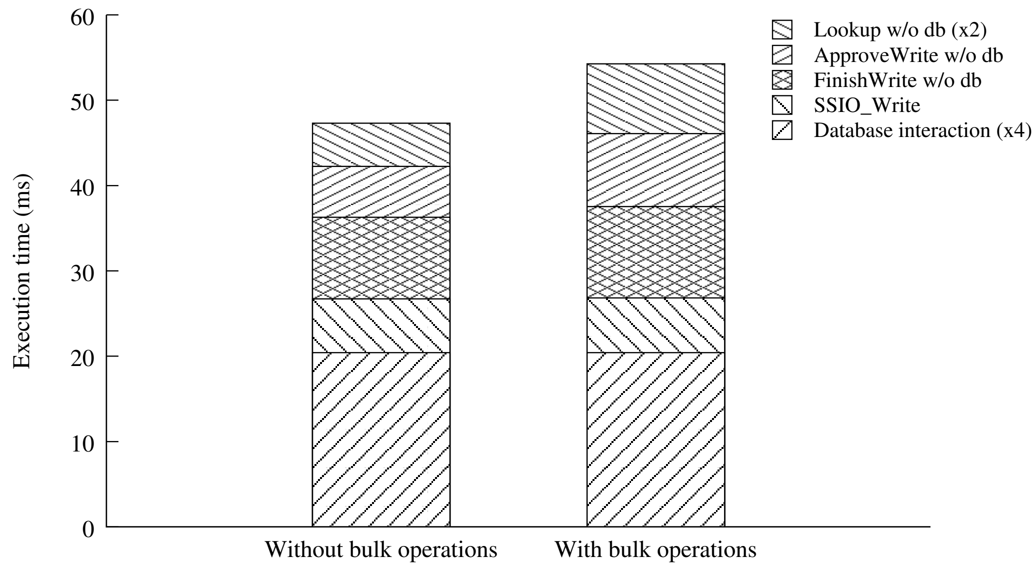


Figure 6.3: Write timing comparison
These are the average timings for the components of the Write operation at the client with database interaction times pulled out (each database interaction add an average overhead of 5.1 ms). A Write is composed of two Lookup operations, an ApproveWrite, and a FinishWrite, and an SSIO_Write. Without bulk operations, it takes 47.4 ms on average to perform a Write of 64 kB in 16 kB blocks. With bulk operations, it takes 54.3 ms on average. This is a difference of 6.94 ms which represents the overhead of bulk operations.

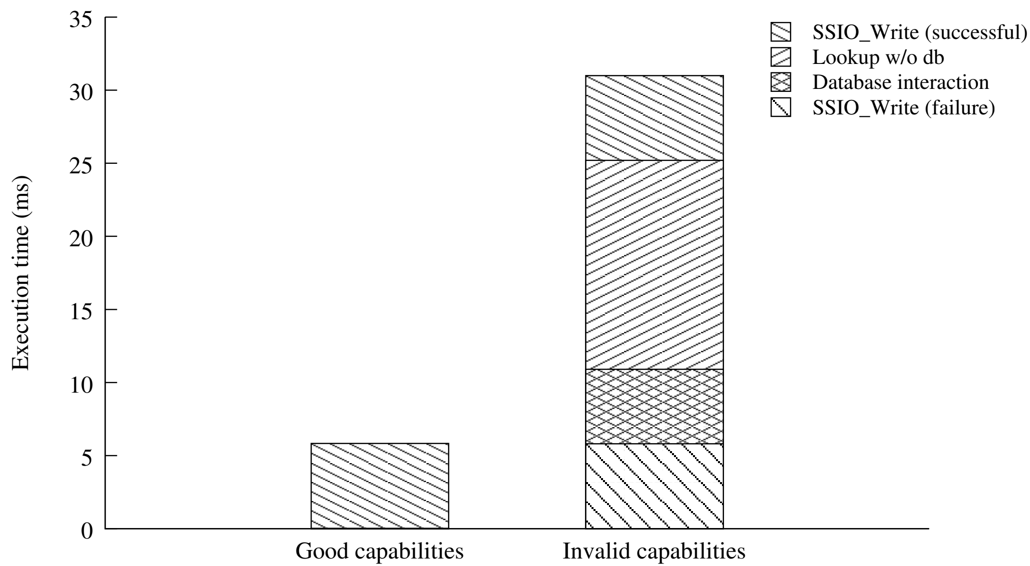


Figure 6.4: Re-Write timing comparison
These are the average timings for the components of the Write operation when it is over-writing data with valid, and invalid, capabilities as observed at the client and bulk operation code is inactive. The average database interaction time of the Lookup operation has been pulled out (5.1 ms). An over-write of 64 kB in 16 kB blocks does not need to go through the ApproveWrite and FinishWrite steps. With valid cached capabilities, it proceeds quickly. With invalid cached capabilities, as would happen after capability revocation performed in the course of a bulk operation, additional steps take place.

6.2.5 Read

The Read operation retrieves previously written data from objects in the storage system. The algorithms associated with bulk operations add overhead to the basic process of making sure that an object being read exists, checking for a need to instantiate a clone, and checking for existing metadata (or metadata that will exist upon clone instantiation). The purpose of these experiments is to determine that overhead and point to its source.

The protocol for a client to perform a Read of an object is as follows. A client performs a Lookup operation to the metadata server to acquire metadata and capabilities to access the object (assuming that the object exists), unless it has cached metadata and capabilities for the object. The client then contacts the necessary storage nodes to retrieve data using a low-level SSIO_Read operation. If the SSIO_Read request presents storage nodes with invalid capabilities, the operation is rejected, otherwise the storage node returns the requested information. If the SSIO_Read is rejected because of invalid capabilities, the client library will perform a Lookup operation and retry the SSIO_Read with new metadata and capabilities. This completes the Read operation.

To investigate the baseline behavior of the client library Read operation, we perform a series of experiments.

The first experiment measures the time to Read 64 kB of an object in 16 kB blocks with cached metadata and valid capabilities; bulk operations code is not active. The average time for a Read, observable by the process invoking the client library function, is 5.9 ms (with a standard deviation of 0.335 ms). Since there is no need for the client to interact with the metadata service in this case, the timing is the same if bulk operation code is enabled.

To understand operation timing when a Read is performed by a client using revoked capabilities, the prior experiment is extended. After reading the data in the objects, capabilities are revoked (without activating bulk operation code), and the data is re-read with invalid capabilities. It takes an average of 27.9 ms (with a standard deviation of 5.92 ms) to read 64 kB in 16 kB blocks.

The protocol behind this re-reading has four steps. Average timing for operations, as determined experimentally, are presented in parentheses as the protocol is described. First is a quick Lookup (0.092 ms with a standard deviation of 0.021 ms) that finds cached

metadata and capabilities to use for the Read. Second is a call to the low-level read function, SSIO_Read (3.13 ms with a standard deviation of 0.128 ms), that attempts to read data from the storage node but cannot because the presented capabilities are invalid; no data is returned. This read returns quickly because it does not transport any data. Third is a Lookup (19.5 ms with a standard deviation of 5.89 ms) to the metadata server to re-acquire capabilities and metadata. Fourth is a SSIO_Read (4.94 ms with a standard deviation of 0.141 ms) that succeeds and transmits the data from the storage node to the client.

When we activate the bulk operation code paths, the average time for a client to re-read data while it holds cached capabilities is 32.0 ms (with a standard deviation of 6.34 ms).

The second Lookup operation takes longer with bulk operation code active. Its execution averages 23.5 ms (with a standard deviation of 6.32 ms).

Summary:

Without bulk operation code active, it takes 5.9 ms on average to perform a Read of 64 kB with 16 kB blocks with cached capabilities and metadata. With invalid capabilities, it takes 27.9 ms on average.

Table 6.2: Re-Read() operation differences with bulk operations – Average timing for the components of a Read() operation using invalid, cached capabilities, with and without bulk operation code active.

Operation	With bulk ops (ms)	Without bulk ops (ms)	Difference (ms)
Read()	32.009	27.873	4.136
Fast Lookup()	0.093	0.092	0.001
Fast SSIO_Read()	3.174	3.129	0.045
Slow Lookup()	23.548	19.482	4.066
Slow SSIO_Read()	4.963	4.941	0.022

A summary of the average timings for the component operations of a client performing a Read with cached (but invalid) capabilities are shown in Table 6.2 on page 99, and in

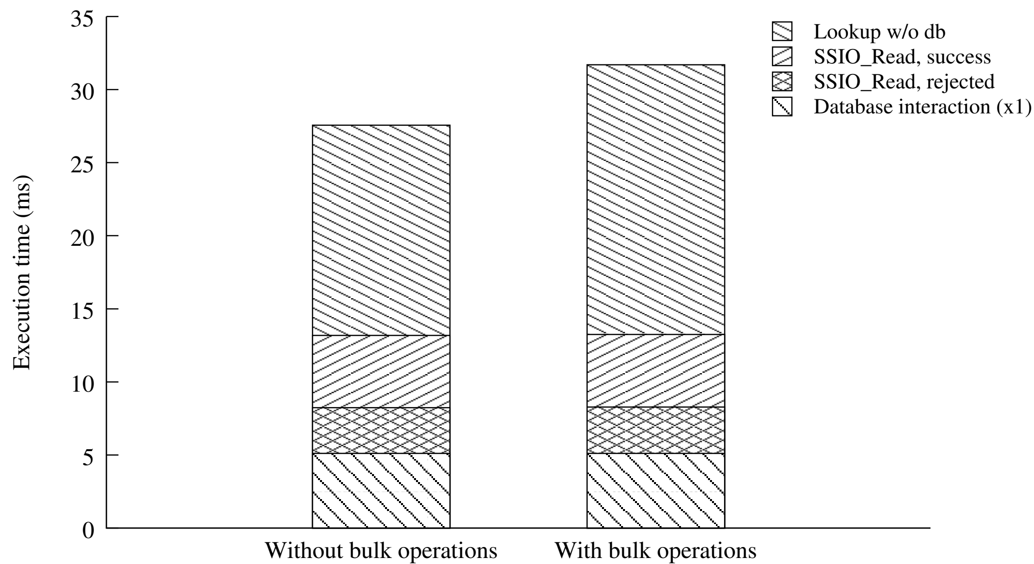


Figure 6.5: Read timing comparison with invalid capabilities
These are the average timings for the components of the Read operation with invalid capabilities at the client with database interaction times (average of 5.1 ms and an overhead of our implementation) pulled out. This arrangement of the data highlighs that the Lookup operation contributes the most to the timing difference. The difference in overall average execution timing is 4.14 ms which represents the overhead of bulk operations.

Figure 6.5 on page 100, for cases with, and without, bulk operation code active. The bulk operation code executed by the metadata server contributes an overhead of 4.14 ms. Read timing results are summarized graphically in Figure 6.6 on page 101.

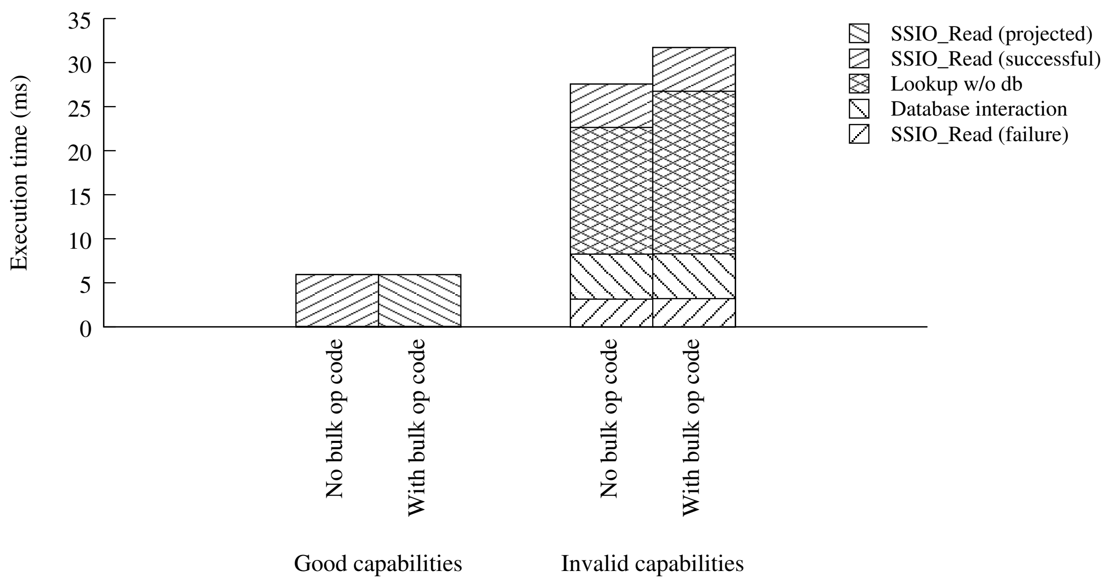


Figure 6.6: Re-Read timing comparison
These are the average timings for the components of the Read operation when it is re-reading data with valid, and invalid, capabilities as observed at the client. The average database interaction time of the Lookup operation has been pulled out (5.1 ms).

6.3 BulkDelete

The BulkDelete operation affects the object identifier namespace by removing many objects at once. Our delayed instantiation method ensures that any previous incarnation of an object is removed from the storage system before a new instance can be successfully created.

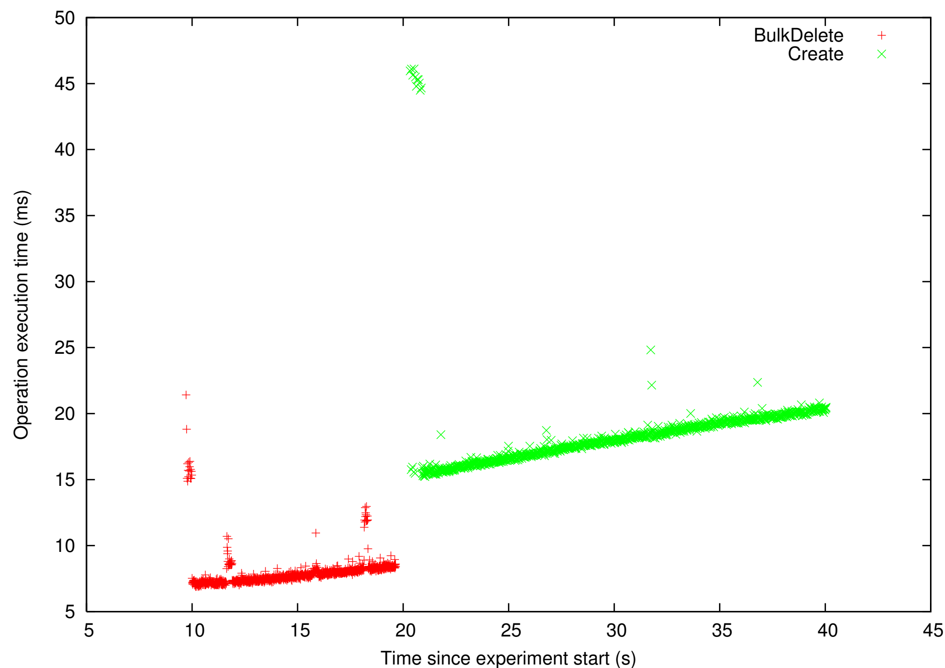


Figure 6.7: BulkDelete and re-Create of pre-existing objects

To learn about the performance of BulkDelete, we first investigate its use on a single object. An experiment is set up with a client and a metadata server on separate machines; no storage node is directly involved. Background instantiation of bulk operations is disabled. The client first uses Create to sequentially create a number of objects. Then, BulkDelete is called to individually remove each object (i.e., the target range for delete covers a single object identifier). Finally, we use Create to re-create the objects. The data points for the BulkDelete and re-Create operation timings are shown in Figure 6.7 on page 102 and described on the following pages.

The average time needed to BulkDelete a single object is 7.95 ms (starting with a few errant points and having a standard deviation of 1.28 ms) and shows a linear dependence on the number of objects that have been deleted. This dependence also corresponds with the number of entries in the BulkDelete table. The average timing of Create operations on recently deleted objects is 18.2 ms (starting with a few errant points and having a standard deviation of 3.22 ms) and shows a linear increase with the number of objects re-created (the number of objects in the Object table). These second calls to Create objects take longer than those executed on an untouched storage system running bulk operation code (15.6 ms). This result is expected, as the bulk operations algorithms induce additional work (e.g., must consult additional tables and execute logic to make instantiation decisions).

Digging deeper into the workings of the Create operation, we break down the internal timing. The Create operation consists of three primary subroutines. The first performs an enumeration of the object identifier namespace to make sure that the object does not already exist. The second subroutine ensures that a "hole" exists in the namespace by instantiating any BulkClone and BulkDelete operations for the object being created. The third routine inserts the record of the creation into the Object Table.

Since the back-end of the metadata server is not part of our normal tracing infrastructure, we made special measurements for this case. Messages were written to the PostgreSQL log file with millisecond granularity. One message was output as the Create function started. Another message marked the start of the enumeration step. Following this is a message marking the end of the enumeration step and the beginning of the "instantiate hole" step. Next is a message marking the end of the "instantiate hole" step and the beginning of the insert step. Finally comes a message marking the end of the function.

Taking the difference between the various timestamps presents us with millisecond granularity information about the subroutines. The data points plotted in Figure 6.8 on page 105 show the timing information for the components of the Create after individual BulkDelete of objects at the back-end of the metadata server (within the PostgreSQL database).

The quickest of the subroutines is the insert subroutine with an average execution time of 0.59 ms. The enumerate subroutine is the next fastest (average of 3.12 ms) and ex-

hibits a linear dependence on the number of objects. The bunching of data points at the discrete millisecond marks is due to the timing resolution. The InstantiateHole subroutine (described in Section 5.3.3 on page 68) consistently takes the longest (average of 10.70 ms and is therefore the major contributor to the execution time of the Create operation in this case. It also shows a linear dependence on the number of objects, and the dependence has a steeper slope.

A closer look at the internal operations of the InstantiateHole function shows two routines contributing to the increased latency as the number of objects grows. Those routines are searching the object table (average of 1.68 ms) and looping over BulkDelete operations (average of 1.09 ms). The contributions of these routines are shown in Figure 6.9 on page 105. It makes sense that the searching of the object table should be dependent on the number of objects, if the underlying database is performing a database table scan.

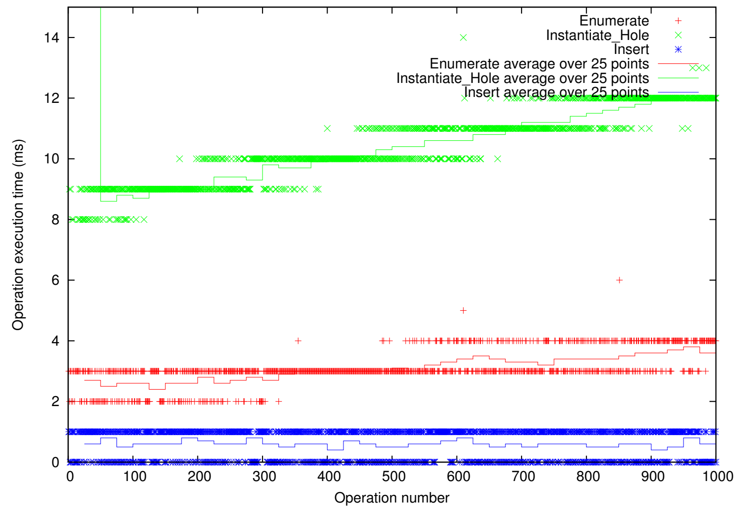


Figure 6.8: Components of Create at the Metadata Server
Sample points and averages for three components of Create at the metadata server. A few errant initial points for "Instantiate_Hole" are close to 30 ms and help explain the errant points in Figure 6.7.

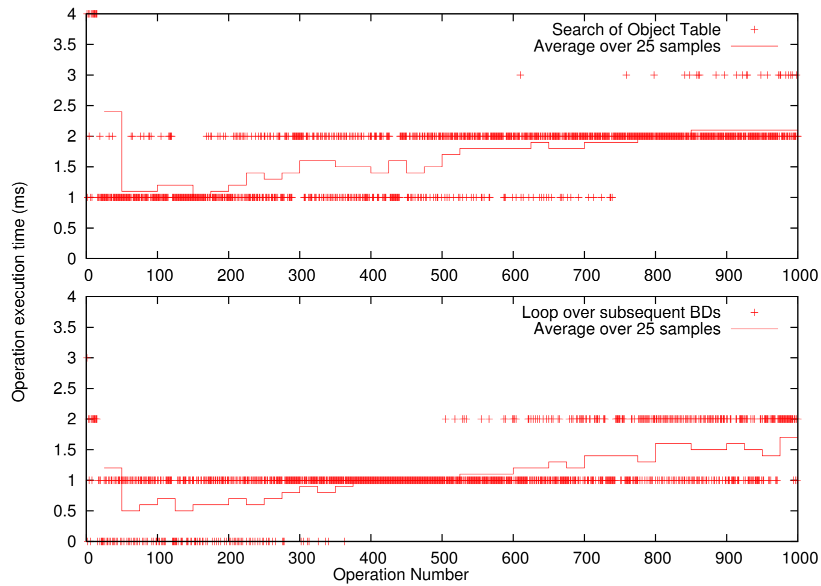


Figure 6.9: Components of the Instantiate_Hole subroutine

6.4 BulkClone

The BulkClone operation affects the object identifier namespace by creating copies of existing objects. Our delayed instantiation method ensures that any clones of an object are instantiated before client access is permitted, so that accurate copies can be created.

6.4.1 Comparing chains-of-clones and prolific clones

To learn about the behavior of BulkClone, we first investigate its two anticipated modes of employ: chains-of-clones and prolific clones. These experiments are set up with a client and a metadata server running on separate computers. No storage nodes are involved in these initial experiments. The sequence of events for the experiments proceeds as follows. The client issues Create operations for 1000 objects in the object identifier namespace. Then, the client calls BulkClone and the timing of those calls is observed. The results provide information to use in planning how best to use the BulkClone operation.

The results indicate that prolific clones are much more sustainable than chains-of-clones.

For prolific clones, BulkClone is called 2000 times. This results in 2 million objects being tracked at the end of the experiment. The average timing for a BulkClone was 252.0 ms (with a standard deviation of 42.0 ms).

For chains-of-clones, BulkClone is called 25 times. Only 25 BulkClone operations were performed for this test because the time required for each operation increased with each call, to the point where the operation timed-out. The average timing for a BulkClone was 1780.0 ms (with a standard deviation of 970.0 ms).

6.4.2 Access after BulkClone

To learn about the influence of BulkClone on subsequent client accesses, we Read objects after a BulkClone operation. An experiment is set up with a client, a metadata server and a single storage node, all on separate computers. Background instantiation of bulk operations is disabled. The experiment has four phases. In the first phase, a set of 1000

source objects is created. In the second phase, 64 kB of data in 16 kB blocks is written to the source objects. In the third phase, the objects are cloned to form a destination set with a single BulkClone operation. In the fourth phase, one of the object sets is read and timing reported. For one instance of the experiment, the source objects of the BulkClone are read. For another instance, the destination objects are read.

In the case of the source objects being read, the client will attempt to use its cached capabilities, fail in its first low-level SSIO_Read because of invalid capabilities, re-acquire capabilities with a Lookup operation, and then succeed in reading the data with a second SSIO_Read. When the destination objects are being read, the client will not have cached capabilities and will perform a Lookup to acquire metadata and capabilities. Both situations will trigger the instantiation of a clone. We expect that the reading of the source objects will take slightly longer due to the attempted use of invalid capabilities and the necessary capability re-acquisition.

Figures 6.10 and 6.11 provide time series data for the experiments. reading each source object after cloning takes, on average, 74.3 ms (with a standard deviation of 15.7 ms). Reading each destination object after cloning takes, on average, 73.0 ms (with a standard deviation of 20.8 ms).

In contrast to expectations, there is not a significant difference in the averages of the Read times, and the standard deviations are quite large (15.7 ms for reading source objects and 20.8 ms for reading destination objects), We see that, in general, it takes longer to read the first few cloned objects than the later ones. This points to a dependence on the size of the BulkClone tracking table as accesses subsequent to the execution of a BulkClone are made. This dependence is entirely contained within the Lookup operation executed as part of the overall Read.

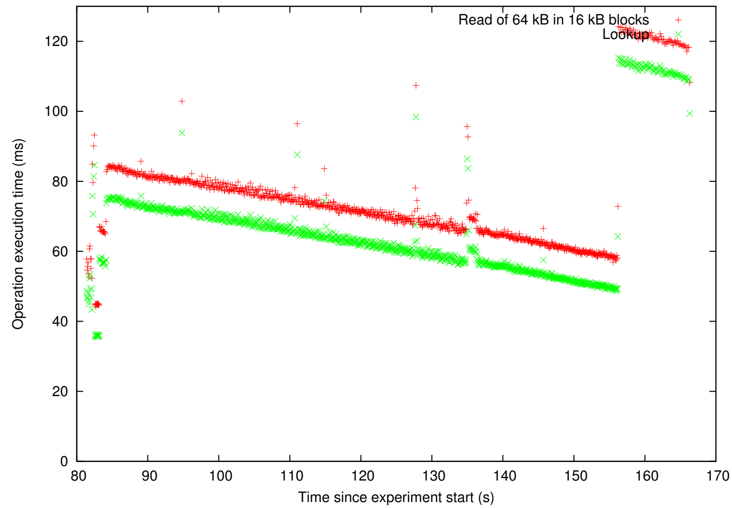


Figure 6.10: Read of source objects after BulkClone performing on-demand instantiation

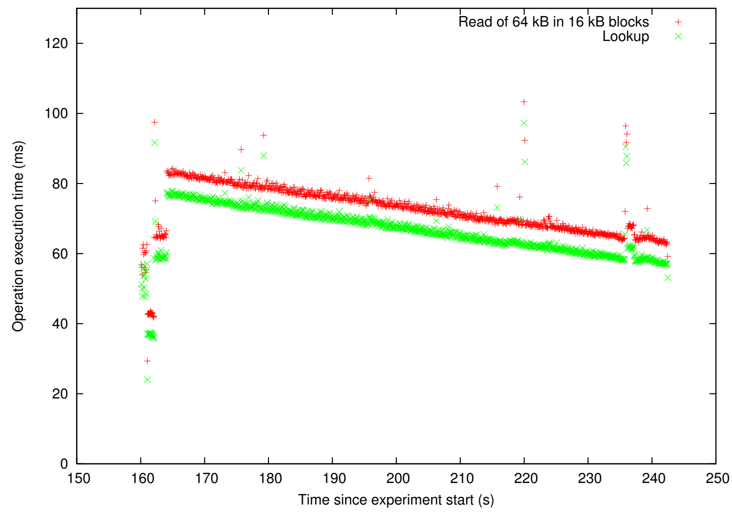


Figure 6.11: Read of destination objects after BulkClone performing on-demand instantiation

A few spurious initial data points are not displayed on this graph as the y–range has been trimmed to match that for reading source objects after BulkClone.

6.5 Background instantiation

This section explores background instantiation as a way to eliminate the instantiation step from on-demand access to objects affected by bulk operations.

Various background instantiation algorithms have been implemented to choose which bulk operation table record and which object to process. These are described in Section 4.4 starting on page 40. The following experiments evaluate and compare these algorithms in non-competitive and competitive situations where background instantiation does not, or does, compete with the foreground experimental workload. The experiments involve three computers, a client, a metadata server, and a single storage node.

The first set of experiments illustrates background instantiation results when performed entirely in the background. We show that background instantiation, in the absence of a competing foreground workload, is "free". This is done by executing a bulk operation and waiting for it to be completely instantiated in the background before resuming a foreground workload. The foreground workload then proceeds with execution timing similar to when capability revocation has occurred. These experimental results are described in Section 6.5.1 beginning on page 111.

The second set of experiments illustrates background instantiation results when lightly competing with a foreground workload. They show that background instantiation saves work by instantiating bulk operations before the foreground workload accesses affected objects. There is competition between the foreground and background workloads in these cases. The experiments consists of a foreground workload, executed after a bulk operation, that consists of either a Create or Read operation executed in a mixed workload with some number of Sleep operations. For example, when executing a Read operation in a 1:1 ratio with Sleep, there are, on average, 50% Read operations and 50% Sleep operations in the client's workload after a BulkClone operation. If the ratio were 1:3, then 25% of the operations would be Reads and the remaining 75% would be Sleep operations, on average. These experimental results are described in Section 6.5.2 beginning on page 112.

The remainder of the experiments run have foreground workloads executing closed-loop and directly competing with ongoing background instantiation.

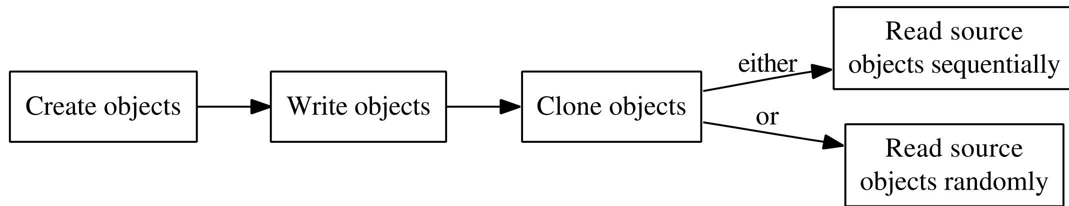


Figure 6.12: Experiment description for create, write, clone, and read of objects.

The third and fourth experiments are run for each of the background instantiation selector algorithms and can be found in the subsections named after each of those algorithms.

The third set of experiments involve Read of source objects after a BulkClone with competitive background instantiation. The BulkClone operation must be instantiated before Read is allowed to succeed. The results from these BulkClone experiments with background instantiation may be compared to the results reported for the case where no background instantiation is performed (see Section 6.4.2 on page 106). The experiments proceed in four phases, illustrated in Figure 6.12 on page 110. Phase one creates 1000 objects in sequential order. Phase two sequentially writes 64 kB in 16 kB blocks to those objects in sequential order. Phase three clones those objects with a single BulkClone operation. Phase four has two possibilities: random or sequential Read of the source object set concurrent with background instantiation according to the algorithm being tested. We will refer to *phase four-A* as reading the source object set sequentially, at least twice.[2] Reference to *phase four-B* will be reading the source object set randomly. During phase four, the foreground workload and the background instantiation workload compete with one another inside the system.

The fourth experiment involves Create of objects after a BulkDelete with competitive background instantiation. The BulkDelete operation on the objects must be instantiated before Create is allowed to succeed. The experiments proceed in four phases, illustrated in Figure 6.13 on page 111. Phase one creates 1000 objects in sequential order. Phase two sequentially writes 64 kB in 16 kB blocks to those objects in sequential order. Phase three

[2]We read the objects at least twice, for a total of at least 2000 Read operations after the BulkClone, to force at least half of the Read operations to execute after the BulkClone has been instantiated.

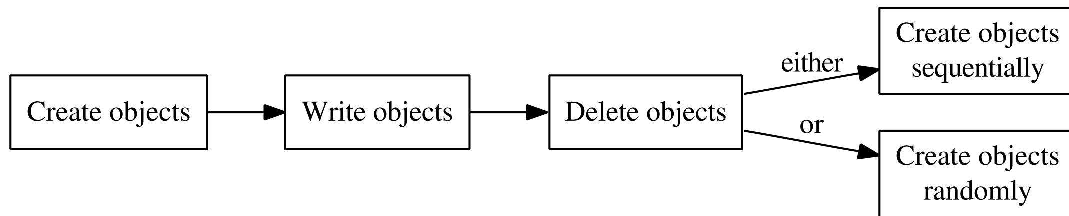


Figure 6.13: Experiment description for create, write, delete, and read of objects.

executes a single BulkDelete operation on those objects. Phase four has two possibilities: *random* or *sequential* Create of the deleted object set concurrent with background instantiation according to the algorithm being tested. *Phase four A* refers to re-creating objects sequentially. *Phase four B* refers to re-creating objects randomly.

6.5.1 Non-competitive background instantiation

For our non-competitive background instantiation experiments, we allow background instantiation to complete after a bulk operation before resuming access. The background instantiation selector algorithm used is "random", but could be any other as all have the same eventual effect.

In the BulkDelete experiment, we Create objects, execute a single BulkDelete operation that affects all created objects, sleep for long enough that background instantiation completes, then re-Create the objects. The second Create operation takes 16.0 ms on average (with a standard deviation of 0.725 ms). Create with bulk operation code active (reported in Section 6.2.3 on page 87) takes an average of 15.6 ms, putting the results of this experiment slightly slower but within a standard deviation.

In the BulkClone experiment, we Create objects, Write to their first 64 kB in 16 kB blocks, execute a single BulkClone operation that affects all of the objects, sleep for long enough that background instantiation completes, then Read the objects (64 kB in 16 kB blocks). In this experiment, the Read operation takes 36.4 ms on average (with a standard deviation of 6.82 ms). This is within a standard deviation of the results from re-reading an

object with invalid capabilities and bulk operations code activated: 32.0 ms (with a standard deviation of 6.34 ms)

6.5.2 Background instantiation with paced foreground workload

This section describes the results of experiments where a foreground workload that executes infrequent operations interacts with bulk operations. The frequency of foreground operations after the bulk operation is modulated by sleep operations that emulate "think-time" of the client.

When we mix operations in a 1:1 ratio, 50% of the operations are the foreground workload's operation (Create or Read) and the remaining 50% are sleep operations. The mixture of operations is random, but controlled such that the approximate ratios of operations are maintained. We tested with the "random" background instantiation selector algorithm, only.

Create after BulkDelete

Our first paced workload involves Create of objects after a BulkDelete. Within a range of 1000 objects, we sequentially issue Create operations, then use a single BulkDelete to remove the objects, and then begin recreating the objects with Create operations intermixed with 16 ms sleep operations. We choose 16 ms as it closely matches the timing reported in Section 6.2.3 on page 87 for sequential Create with bulk operation code active.

The summarized results of three experimental runs are shown in Table 6.3 on page 113. In the tabular data, we observe that the average time for performing a Create decreases as the ratio of Sleep operations increases. This is expected since the background instantiation can run more often as more Sleep operations occur. Thus, the background instantiation completes sooner, relative to the foreground Create operation, and more of the Create operations can execute quicker, since there is no BulkDelete instantiation necessary. We also observe the standard deviation decreasing as the ratio of Sleep operations increases. This is due to the longer Create times induced by competition between the background instantiation workload and the foreground Create workload, occurring until the bulk operations

table is clean. When there are more Sleep operations executed, relative to Create operations, instantiation completes sooner and there are fewer data points off of the average.

Table 6.3: Paced Create after BulkDelete with background instantiation – Results for the Create operation executed after a BulkDelete. The ratio of Create to 16 ms sleep operations for each of the three experiments is as shown.

Create:Sleep ratio	Create time (ms)	
	Average execution time	Standard deviation
1:1	33.5	47.9
1:3	23.8	28.8
1:7	18.7	13.7

The time-series data for the runs are shown in Figure 6.14 on page 114. In these graphs showing Create operation timing, we observe two aspects of the data. First, as there are more Sleep operations executed, relative to Create, the time required for background instantiation decreases. This is evident in the shorter band of data points off of the average in the graphs. Second, the overall time required for the Create operations varies with the operation ratio. It is entirely expected that, when Create operations execute approximately 50% of the time, you will reach 1000 executions faster than when it only executes 12.5% of the time (with the 1:7 ratio).

Read after BulkClone

The second paced workload involves Read of objects after a BulkClone. Within a range of 1000 objects, we sequentially issue Create operations, then Write 64 kB in 16 kB blocks, issue a single BulkClone operation, and immediately begin reading the objects intermixed with 32 ms sleep operations. We choose 32 ms as it closely matches the timing of a Read with invalid capabilities as reported in Section 6.2.5 on page 98.

The summarized results of three experimental runs are shown in Table 6.4 on page 116. In the tabular data, we observe effects mirroring those for Create after BulkDelete, and

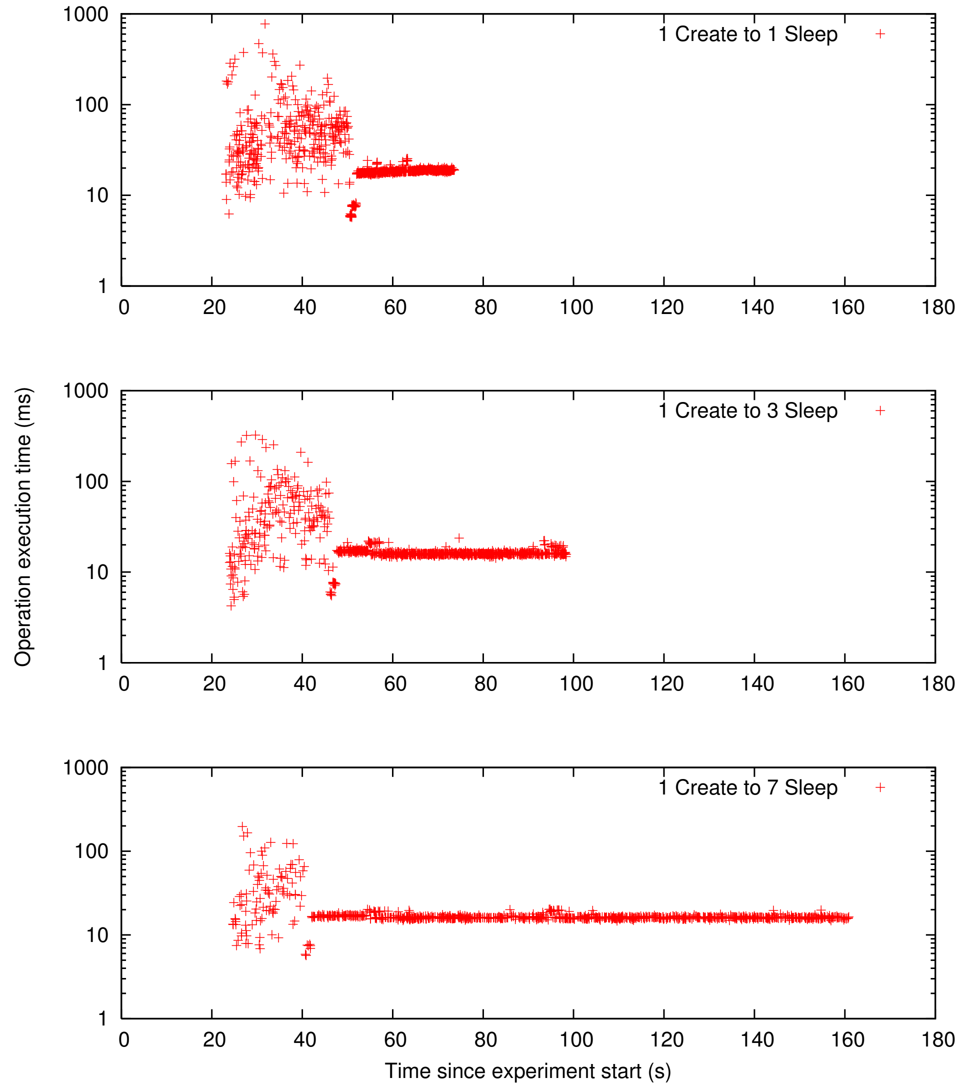


Figure 6.14: Paced Create after BulkDelete.

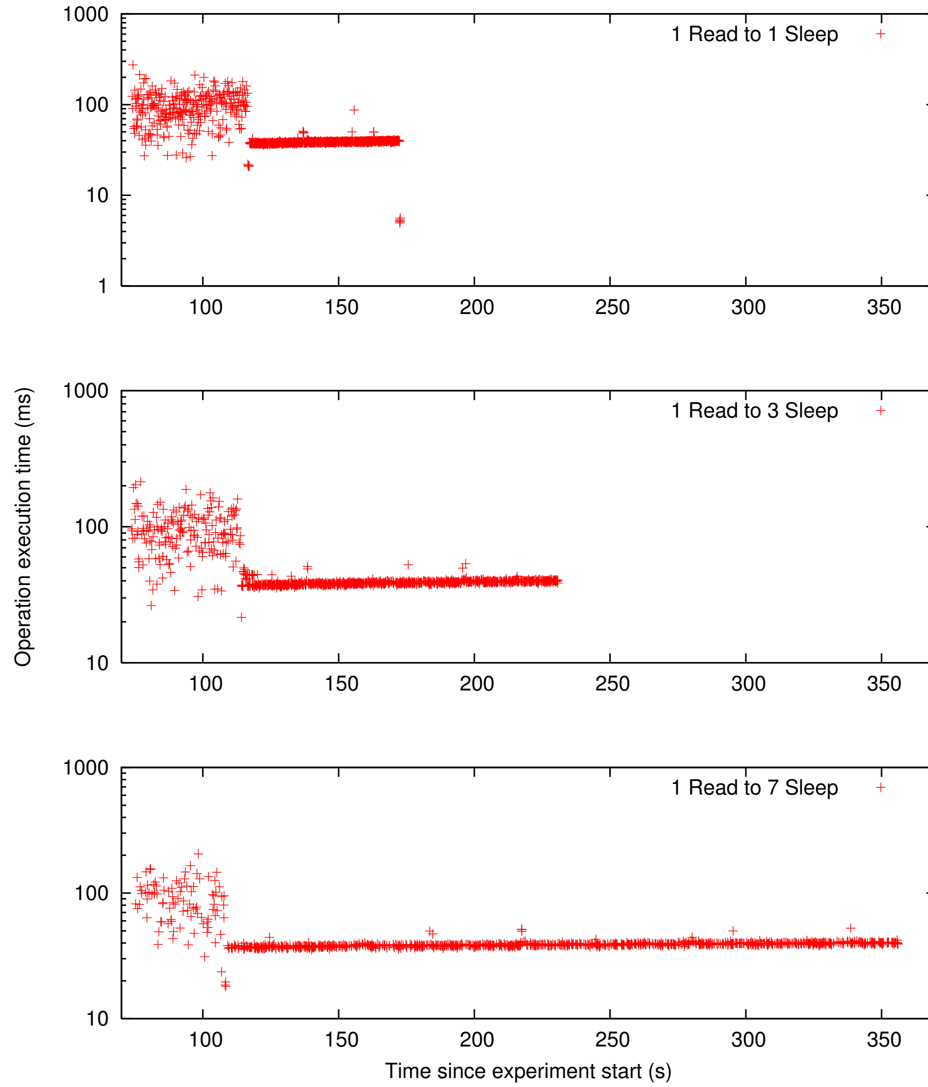


Figure 6.15: Paced Read after BulkClone.

Table 6.4: Paced Read after BulkClone with background instantiation – Results for the Read operation executed after a BulkClone. The ratio of Read to 32 ms sleep operations for each of the three experiments is as shown. The Read is of 64 kB in 16 kB blocks.

Read:Sleep ratio	Read time (ms)	
	Average execution time	Standard deviation
1:1	56.1	34.4
1:3	49.9	27.0
1:7	43.7	18.6

the effects are for similar reasons. As there are more Sleep operations compared to Read operations, the average and standard deviation of the Read execution times decrease. This is because the background instantiation clears out the bulk operation table quicker, thereby allowing for more operations to complete in the expected time.

The time-series data for the runs are shown in Figure 6.15 on page 115. Again, the interpretation of the results mirrors that for Create after BulkDelete. Less overall time is required as more Sleep operations execute relative to Read operations. Also, the period of interference is reduced, since more background instantiations occur during the more frequent Sleep operations.

Overall, we see that when background instantiation has a chance to execute, it improves the performance of the foreground workload by removing a portion of the penalty incurred by our bulk operation system.

6.5.3 Random bulk operation background instantiation

The first background instantiation selector algorithm is called “random.” It randomly picks a bulk operation table (BulkClone or BulkDelete) from which to select a bulk operation. If the chosen table is empty, no instantiation is performed. Otherwise, the algorithm picks a random row from the selected table. The bulk operation is instantiated upon one randomly selected object from the first 100 objects in the range affected by the bulk operation; the discovery is done with a call to Enumerate.

Figure 6.16 on page 118 shows results for Read operations to randomly selected source objects after a BulkClone. There are a number of features of the graphed data to point out.

The initial random Read operations exhibit relatively unpredictable execution times ranging from the low 10’s of milliseconds to over 100 milliseconds. This occurs while the foreground Read workload is competing with the background instantiation of the clones and persists so long as there are clones to instantiate. The two “lines” of instantiation times are caused by the Read accesses that must re-acquire capabilities for the cloned objects and by the Read accesses that re-Read objects that have already been randomly accessed during the experiment. The higher line, around 40 ms, corresponds to Read accesses that the foreground workload has not yet accessed so it must perform a Lookup to the metadata server before proceeding with the low-level Read at the storage node. This compares favorably with the data from Section 6.4.2 where the quickest instantiations take about 45 ms.The lower line, around 5 ms, corresponds to Read accesses for which the client has already acquired metadata and capabilities and is using that cached information. This compares favorably with the data from Section 6.2.5 where a Read with cached capabilities and metadata takes an average of 5.9 ms. The lower line of data points appears slowly as objects are re-accessed. The higher line of data points begins to show evidence of fewer data points as the experiment progresses. The stairstep line showing average Read time over 1 s trends downward during the experiment as more and more accesses use cached metadata and capabilities to access instantiated clones. Both of these trends are expected consequences of the random access pattern and the associated re-accessing of objects. Indeed, over half of the accesses use cached metadata and capabilities since 2000 accesses are performed and only 1000 objects were cloned.

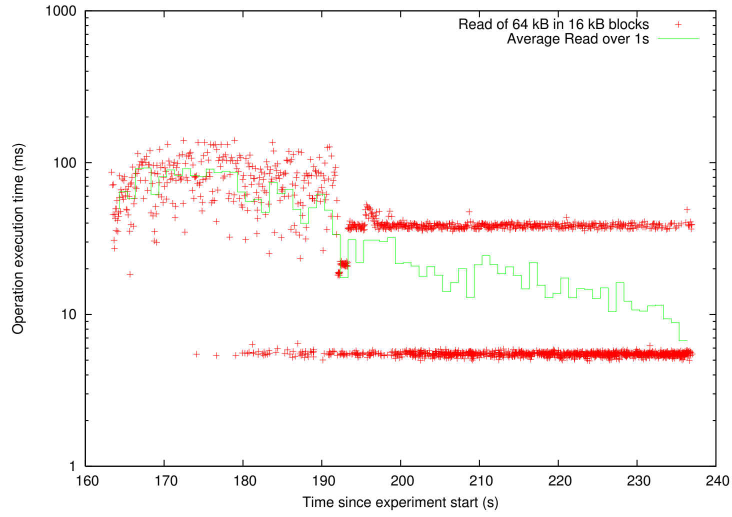


Figure 6.16: Random Read after BulkClone with Random background instantiation

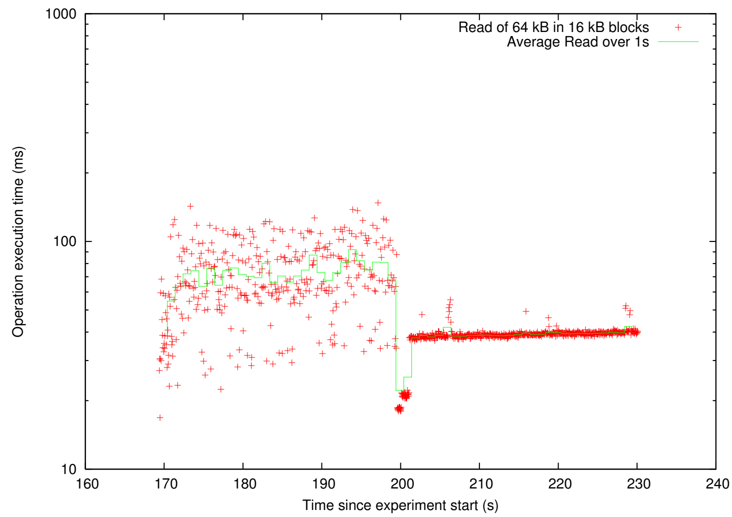


Figure 6.17: Sequential Read after BulkClone with Random background instantiation

Figure 6.17 on page 118 shows, initially, the Read operations exhibiting unpredictable execution times ranging from the low 10's of milliseconds to over 100 milliseconds. Again, this occurs while the foreground Read workload is competing with the background instantiation of the clones. Interference persists so long as there are clones to instantiate. Only 1000 Read operations are issued after the BulkClone, so we do not observe the approximately 5 ms Read operations when valid, cached capabilities are used.

Figure 6.18 on page 120 shows the results of random Create after BulkDelete of 1000 objects, again showing the effects of the foreground workload competing with the background instantiation workload. At least 50% of the operations completed quickly, coming in under 10 ms and comparing favorably with the baseline Create timing (average of 10.3 ms)

Figure 6.19 on page 120 shows results for sequential Create after BulkDelete, again showing the effects of workload competition. Since the background instantiation is occurring randomly, the end result is quite similar to that of random Create after BulkDelete. Again, over 50% of the operations completed quickly, coming in under 10 ms.

Summary

The experiments for the random bulk operation instantiation selector algorithm produced results as expected. When performing a Read operation after a BulkClone, the clone must be instantiated before access can proceed, and an initial attempt to use cached metadata and capabilities will prolong the operation. When performing a Create operation after a BulkDelete of a written object, the delete must be instantiated before the Create can succeed. Once all instantiations have been performed, the Create time returns to the baseline.

6.5.4 FIFO bulk operation background instantiation

The second background instantiation selector algorithm used in our experiments is called "FIFO." It processes bulk operation table entries in FIFO order, instantiating the operation on the lowest numbered object identifier affected by the entry with the lowest sequence number.

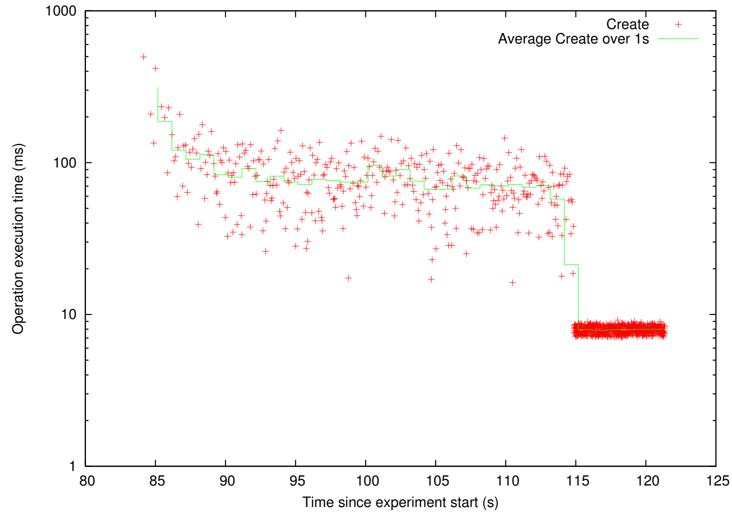


Figure 6.18: Random Create after BulkDelete with Random background instantiation

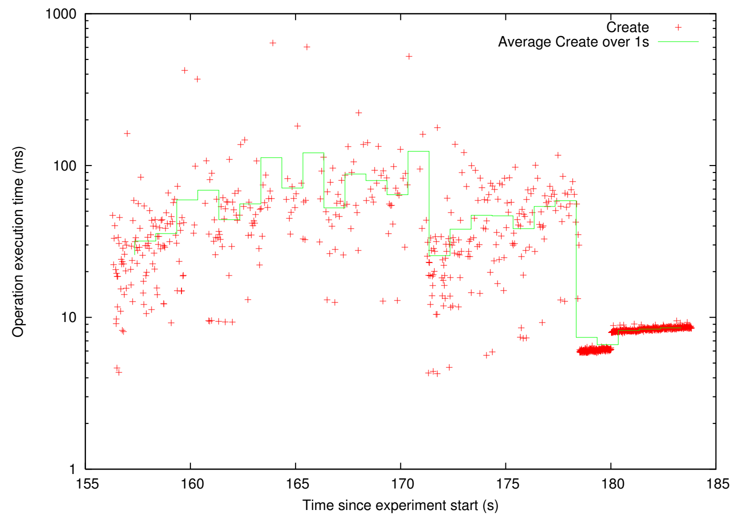


Figure 6.19: Sequential Create after BulkDelete with Random background instantiation

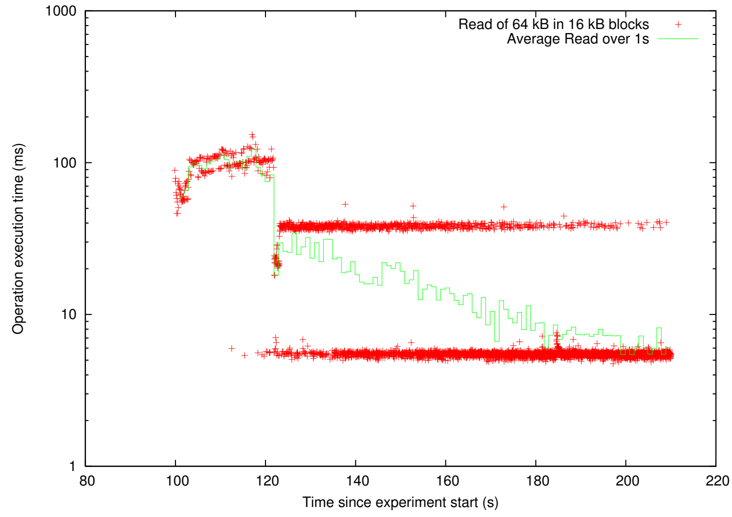


Figure 6.20: Random Read after BulkClone with FIFO background instantiation

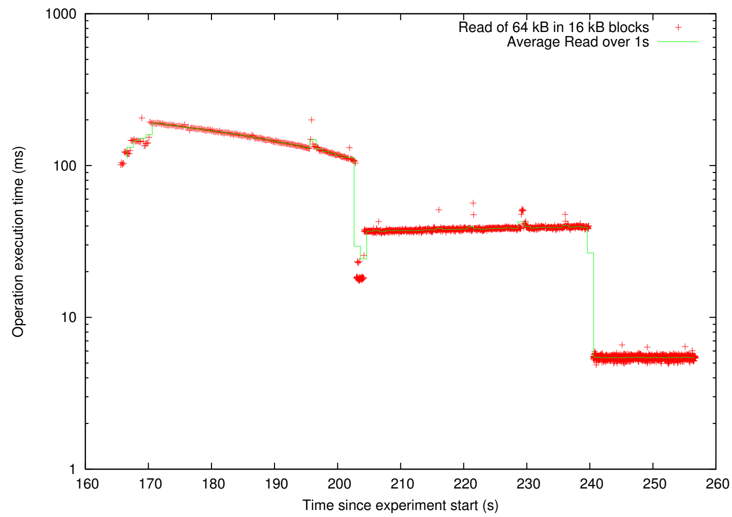


Figure 6.21: Sequential Read after BulkClone with FIFO background instantiation

Figure 6.20 on page 121 shows timing for Read operations to source objects after a BulkClone. There are a number of features of the graphed data to point out.

First, we note the period of competition between the foreground Read workload and the background instantiations at the beginning of the experiment. Then, as the random choice of objects from which to Read begins to cycle through objects that have already been touched, access proceeds more quickly. At about the same time, the background instantiation workload completes. There is a clear binary distribution of Read times after the background instantiation workload completes. The longer time is for Read operations that are for objects that the client has not already Read and therefore must try to Read from the storage node, be rejected for invalid capabilities, re-acquire fresh capabilities from the metadata server, and execute the successful Read. The shorter time corresponds to Reads executed with valid cached capabilities and metadata. Both times are as expected and match with those for the "random" algorithm previously presented. The one-second average line trends to faster Read execution time as more and more of the accesses are to objects for which the client has cached and valid capabilities.

Figure 6.21 on page 121 shows operation execution timing for sequential Read operations being performed after a BulkClone. There are three clear and distinct regions of the timings. The first region mirrors the trend shown in Figure 6.10 on page 108 but here it is slower. The sequential foreground workload is colliding, near perfectly, with the background instantiation workload that is also proceeding sequentially. The second region finishes out the first pass of accesses to the 1000 objects with timing for a protocol interaction that attempts to use cached capabilities to Read, is rejected by storage nodes, re-acquires capabilities, and then succeeds. The third region shows timing for re-reading the 1000 objects when the client holds cached metadata and valid capabilities, and so proceeds quickly with the Read.

Figure 6.22 on page 123 shows timings for creating objects randomly after BulkDelete using FIFO selection for background instantiation. For the first few seconds, the background workload slows the foreground workload as they compete for resources. This results in a near 100 ms average Create time that is then followed by the expected sub-10 ms average Create time after all instantiations have taken place. At least 75% of the accesses

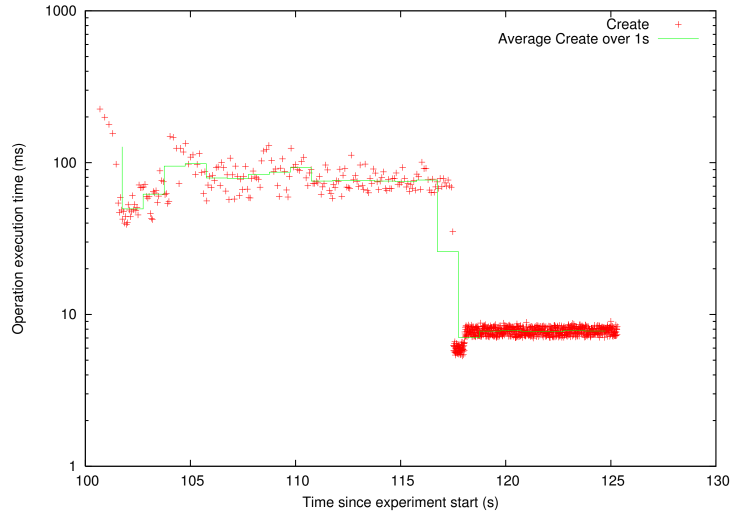


Figure 6.22: Random Create after BulkDelete with FIFO background instantiation

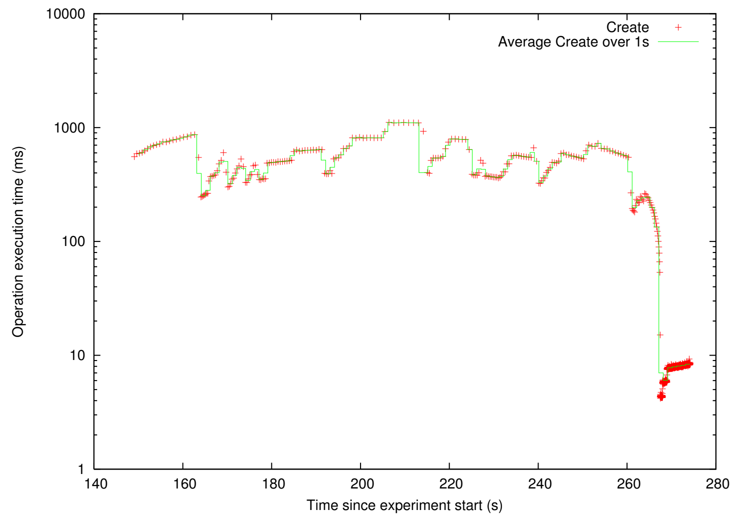


Figure 6.23: Sequential Create after BulkDelete with FIFO background instantiation

occurred with sub-10 ms average Create time.

Figure 6.23 on page 123 shows timings for sequential Create after BulkClone with FIFO selection of background instantiation. There is a much longer period of time during which the foreground workload is adversely effected by the background instantiation work. This dramatic (approaching an average of 1 s per Create) conflict is not unexpected. Because of the sequential Create and sequential instantiation occurring, the two workloads are competing for access to the same objects and same bulk operation table entries. Once the instantiations are complete, the expected sub-10 ms average Create time finishes off the accesses. As before, at least 75% of the accesses occurred with sub-10 ms average Create time.

Summary

With random foreground workloads, we see the effects of competition slowing the operations. Once the instantiations are complete, the foreground workload's behavior meets expectations for execution without the need for on-demand instantiation. The sequential foreground workloads show more extreme effects upon their execution with direct competition for access to individual objects as the FIFO background instantiation. The sequential Read after BulkClone displays three phases to the execution of the experiment: competition with background instantiation, re-acquisition of capabilities, and use of cached metadata and capabilities for access.

6.5.5 LIFO bulk operation background instantiation

The third background instantiation selector algorithm used is called "LIFO." It processes bulk operation table entries in LIFO order, instantiating the operation on the highest numbered object identifier affected by the entry with the highest sequence number.

The results for the two experiments performing Read after BulkClone are very similar to the results as shown for the "FIFO" background instantiation selector algorithm. With random object Read there is a period of interference while clones are instantiated, then progressively fewer client accesses need to re-acquire capabilities, and progressively more

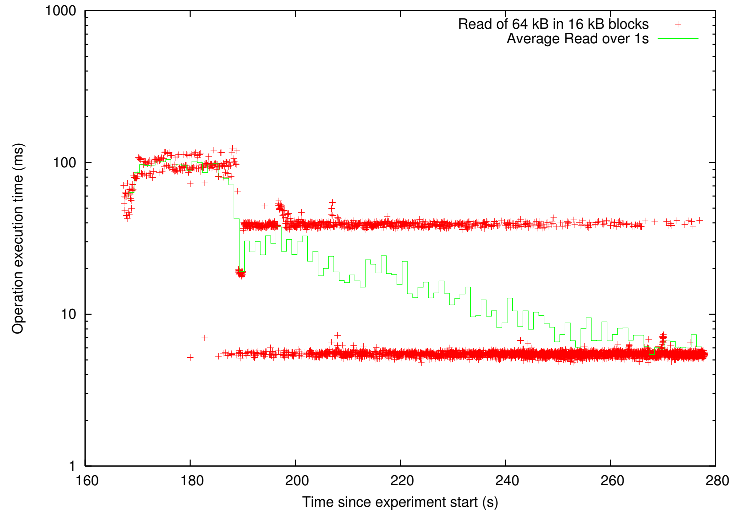


Figure 6.24: Random Read after BulkClone with LIFO background instantiation

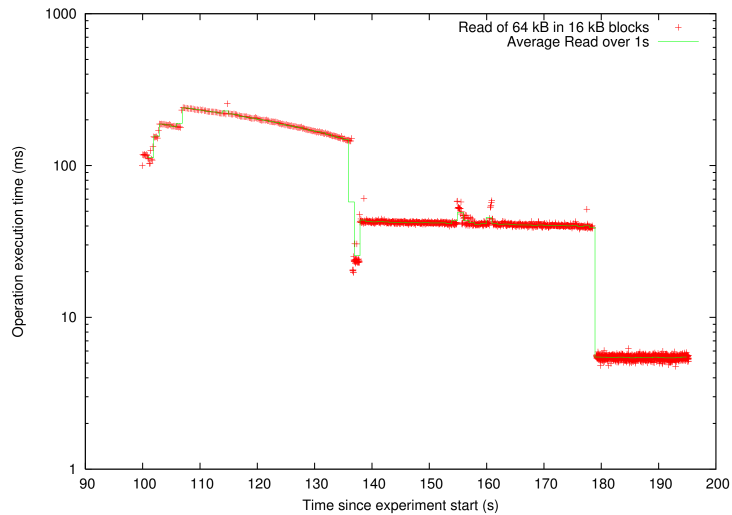


Figure 6.25: Sequential Read after BulkClone with LIFO background instantiation

random client accesses are using re-acquired, valid, cached capabilities. The results for random object Read are shown in Figure 6.24 on page 125. With sequential object Read the same pattern of interference, capability re-acquisition, and cached capability use shows in the data. The data for random object read are shown in Figure 6.25 on page 125.

The LIFO selector is similar to the FIFO selector when a single bulk operation has been issued. As random foreground operations fragment what was once a single bulk operation table entry, the background instantiation processing proceeds in an orderly fashion performing operations on either the lowest or highest object identifier referenced. As sequential foreground operations work, they compete with the background instantiation for database access to the same single bulk operation table entry. The FIFO processing is working on the same objects (lowest object identifier) as the sequential foreground workload and at the same end of the bulk operation table entry. The LIFO background processing is working on the highest object identifier of the single bulk operation table entry while the foreground workload is chipping away at the lowest object identifier of the single table entry.

For Creates after a BulkDelete, the results for LIFO are very similar to those for FIFO. Once the background instantiation workload has exhausted all instantiation possibilities, the foreground workload proceeds quickly. Results for random Create are shown in Figure 6.26 on page 127. The sequential Create after BulkDelete continues to compete for access to the same bulk operation table entry and thereby extends the time for experiment completion. Results for sequential Create are shown in Figure 6.27 on page 127.

For both sequential and random sequential Creates, at least 75% of the Create operations executed in less than 10 ms.

Summary

We see results for LIFO that are very similar to those for FIFO. This can be attributed to the random workloads' unpredictability with sequential processing. Also, the sequential foreground workloads' competition with background instantiation for the same bulk operation table entry.

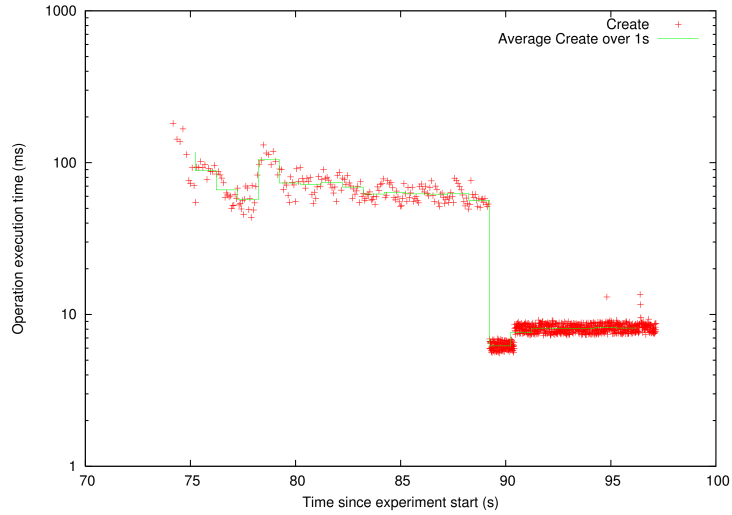


Figure 6.26: Random Create after BulkDelete with LIFO background instantiation

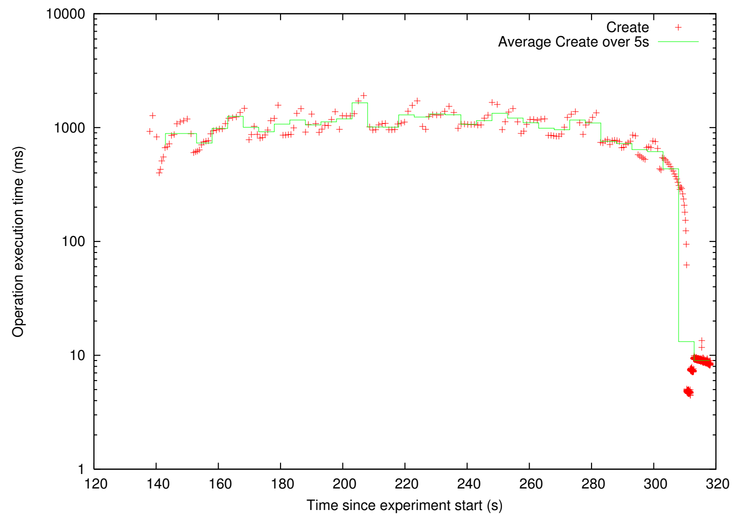


Figure 6.27: Sequential Create after BulkDelete with LIFO background instantiation

6.5.6 Widest span of objects bulk operation background instantiation

The fourth background instantiation selector algorithm is called "widest". It processes bulk operation table entries starting with the table entry that spans the widest range of objects. In the case of a tie, it prefers the table entry with the lowest object identifier for BulkClone, and highest object identifier for BulkDelete. For BulkClone and for BulkDelete, the lowest object identifier in the source object range is instantiated.

The results for the two experiments performing Read after BulkClone are very similar to the results as shown for the "FIFO" and "LIFO" background instantiation selector algorithms. With random object Read, there is a period of interference while clones are instantiated, then progressively fewer client accesses need to re-acquire capabilities, and progressively more random client accesses are using re-acquired, valid, cached capabilities. The results for random object Read are shown in Figure 6.28 on page 129. With sequential object Read, the same pattern of interference, capability re-acquisition, and cached capability use shows in the data. The data for random object read are shown in Figure 6.29 on page 129.

As random foreground operations fragment what was once a single bulk operation table entry, the background instantiation processing proceeds in an orderly fashion performing operations on the widest range of objects. As sequential foreground operations work, they compete with the background instantiation for database access to the same single bulk operation table entry. The "widest" processing is working on the same objects (lowest object identifier) as the sequential foreground workload and at the same end of the bulk operation table entry.

For Create after BulkDelete, the results for "widest" are very similar to those for FIFO and LIFO. With random Create after BulkDelete, once the background instantiation workload has exhausted all instantiation possibilities, the foreground workload proceeds quickly. Results for random Create are shown in Figure 6.30 on page 130. The sequential Create after BulkDelete continues to compete for access to the same bulk operation table entry and thereby extends the time for experiment completion. Results for sequential Create are shown in Figure 6.31 on page 130. For both, at least 75% of the Create operations executed in less than 10 ms.

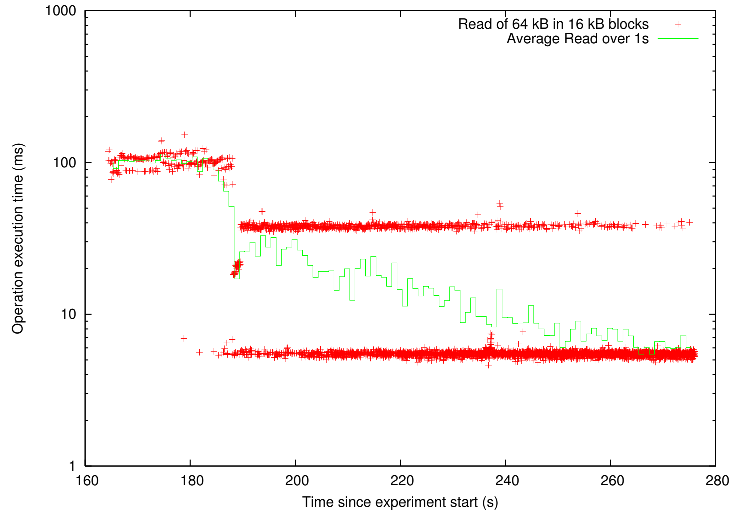


Figure 6.28: Random Read after BulkClone with background instantiation of widest range

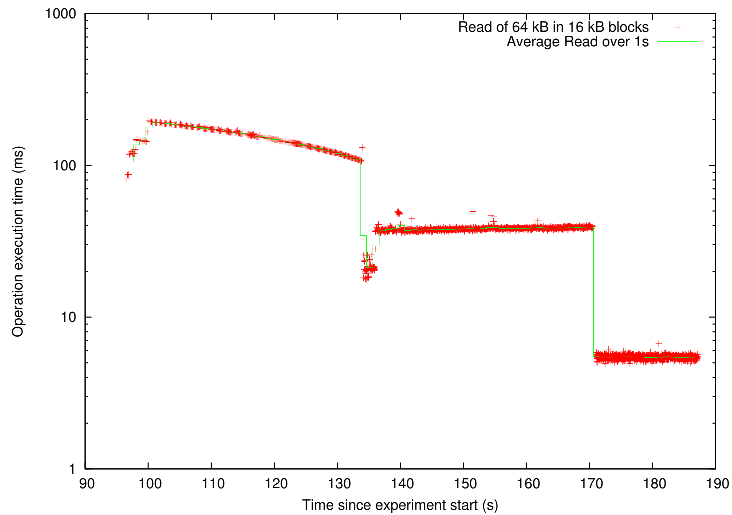


Figure 6.29: Sequential Read after BulkClone with background instantiation of widest range

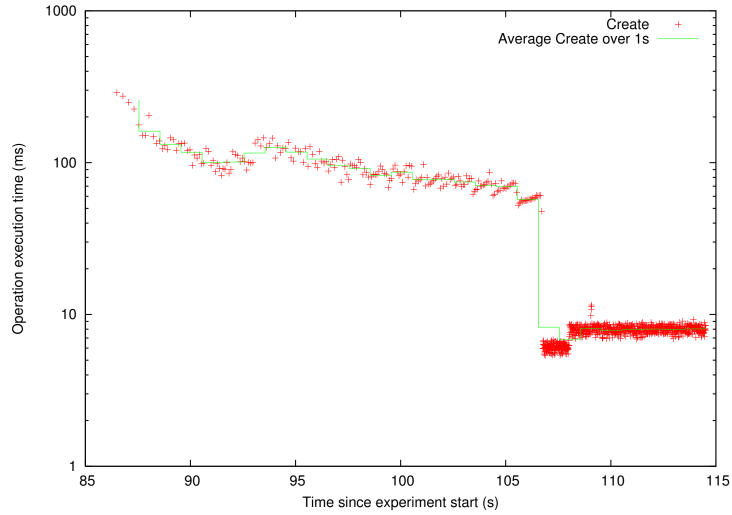


Figure 6.30: Random Create after BulkDelete with background instantiation of widest range

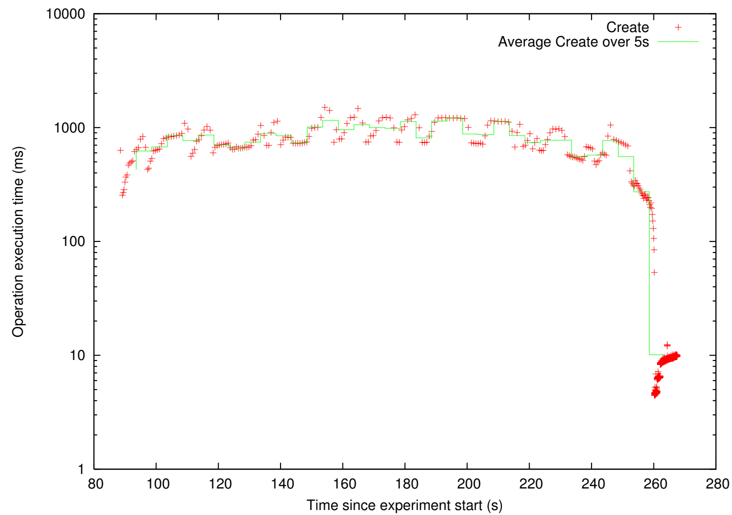


Figure 6.31: Sequential Create after BulkDelete with background instantiation of widest range

Summary

We see results for "widest" that are very similar to those for FIFO and LIFO. This can be attributed to the random workloads' unpredictability and the sequential workloads' competition for instantiation of the same objects and access to the same bulk operation table entry.

6.5.7 Thinnest span of objects bulk operation background instantiation

The fifth background instantiation selector algorithm is called "thinnest". It processes bulk operation table entries starting with the table entry that spans the thinnest range of objects (i.e., the range with the smallest difference in object identifiers). In the case of a tie, it prefers the table entry with the lowest object identifier for BulkClone, and highest object identifier for BulkDelete. For BulkClone and for BulkDelete, the lowest object identifier in the source object range is instantiated.

The results for the two experiments performing Read after BulkClone are very similar to the results as shown for the "FIFO," "LIFO" and "thinnest" background instantiation selector algorithms. With random object Read, there is a period of interference while clones are instantiated, then progressively fewer client accesses need to re-acquire capabilities, and progressively more random client accesses are using re-acquired, valid, cached capabilities. The results for random object Read are shown in Figure 6.32 on page 132. With sequential object Read, the same pattern of interference, capability re-acquisition, and cached capability use shows in the data. The data for random object read are shown in Figure 6.33 on page 132.

As random foreground operations fragment what was once a single bulk operation table entry, the background instantiation processing proceeds in an orderly fashion performing operations on the thinnest range of objects. As sequential foreground operations work, they compete with the background instantiation for database access to the same single bulk operation table entry. The "thinnest" processing is working on the same objects (lowest

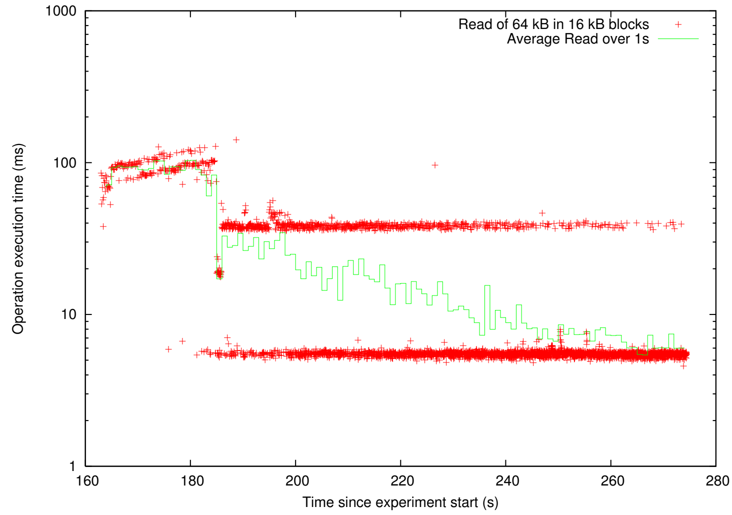


Figure 6.32: Random Read after BulkClone with thinnest range background instantiation

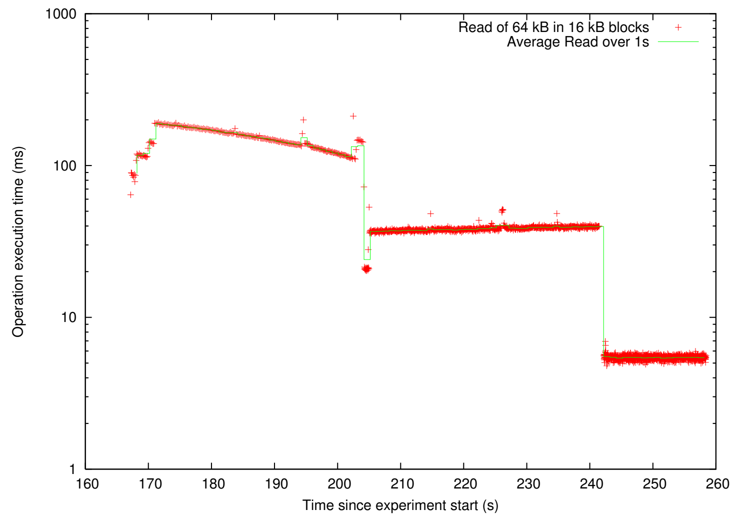


Figure 6.33: Sequential Read after BulkClone with thinnest range background instantiation

object identifier) as the sequential foreground workload and at the same end of the bulk operation table entry.

For Create after BulkDelete, the results for "thinnest" are very similar to those for FIFO, LIFO and "widest." With random Create after BulkDelete, once the background instantiation workload has exhausted all instantiation possibilities, the foreground workload proceeds quickly. Results for random Create are shown in Figure 6.34 on page 134. The sequential Create after BulkDelete continues to compete for access to the same bulk operation table entry and thereby extends the time for experiment completion. Results for sequential Create are shown in Figure 6.35 on page 134. For both, at least 75% of the Create operations executed in less than 10 ms.

Summary

We see results for "thinnest" that are very similar to those for FIFO, LIFO and "widest". This can be attributed to the random workloads' unpredictability and the sequential workloads' competition for instantiation of the same objects and access to the same bulk operation table entry.

Ultimately, these experiments show that it is important to match your background instantiation selector algorithm to your workload. The random instantiation was easy to implement and matches well with random foreground workloads, and with results for sequential workloads. The other selector algorithms exhibited virtually identical performance due to the contention for access to the bulk operation table.

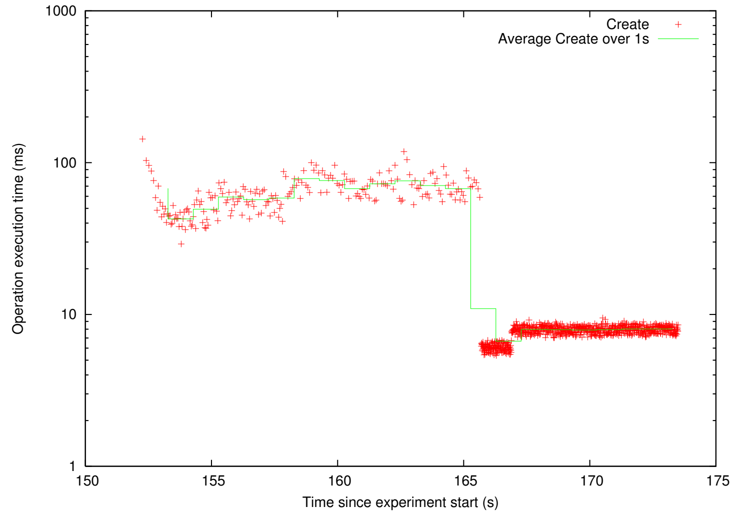


Figure 6.34: Random Create after BulkDelete with background instantiation of thinnest range

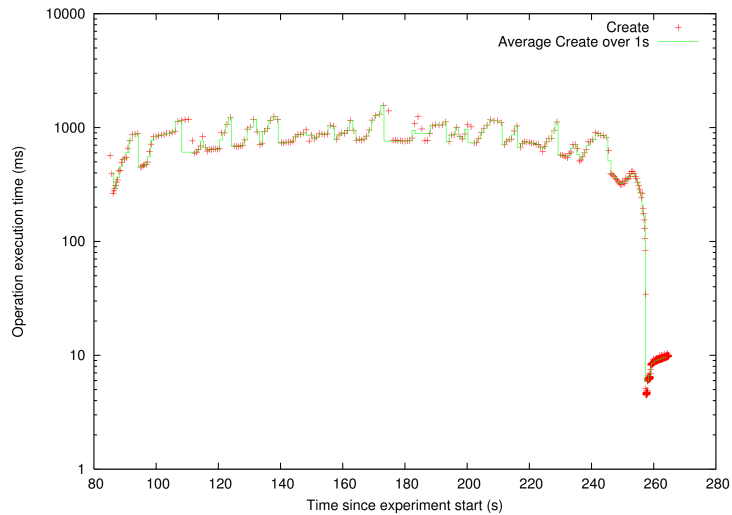


Figure 6.35: Sequential Create after BulkDelete with background instantiation of thinnest range

6.6 NFS server

The experiments in this section use the NFS server described in Section 4.5 on page 42. We first experiment with creating, cloning, and deleting files. Various methods of "cloning" are used to gauge the effects of that bulk operation. Then we use the PostMark benchmark [45] with cloning and background instantiation as we search for the most advantageous clone strategy in a simulated real-world situation. The NFS server is configured with 128 MB of cache memory.

6.6.1 Baseline behavior

To establish the baseline behavior of the NFS server, we use simple command-line programs to create, delete, write, and read files. Timing is accomplished through the use of the `date` command with microsecond resolution. Versions of the experiments were conducted with and without BulkClone being used so that the different execution times can be compared.

In these baseline experiments four computers are involved. There is a single storage node holding data. There is a single metadata server, running with bulk operations enabled. A single NFS server is running and using a data distribution of $1-of-1$ with 16 kB blocks. No files of appreciable size are created directly by the benchmark, so only internal operations of the NFS server (i.e., manipulations of metadata and directories) generate accesses storage. A single NFS client accesses the NFS server.

Our baseline experiment involves creating and deleting sets of NFS files. For each of various numbers of files in a set, the following operations are executed and bracketed by calls to `date`. First, the entire set of files is created in an empty directory, one file at a time, using the `touch` command. Second, in three-out-of-four cases, the file set is copied by using either a BulkClone operation through the NFS server or by using the `cp` command. Third, and finally, the entire set of files is removed, one file at a time, by using the `rm` command. A single instance of the NFS server is used for an experimental run, and it remains mounted on the NFS client throughout operations on the different file set sizes.

Table 6.5: NFS File Create/Clone/Delete benchmark summary – The summary information from experiments that created a number of files in one directory on the NFS server, cloned them all, and then deleted the original files. The raw data can be found in Section A.5 starting on page 221.

Baseline experiment average results, times in H:M:s.ms			
Experiment	Size of file set		
	100	1000	10000
Clone phase			
Baseline	N/A	N/A	N/A
Clone via copy	11.789	1:45.991	22:41.289
Prolific clone	0.704	1.384	3.020
Chain-of-clones clone	0.924	1.606	16.762
Delete phase			
Baseline	5.364	56.810	10:08.828
Clone via copy	5.861	58.643	10:06.536
Prolific clone	11.148	1:52.000	43:38.226
Chain-of-clones clone	14.021	2:32.604	4:59:07.283

The average execution times of the phases of the various experiments are shown in Table 6.5 on page 136. For each of the three file set sizes, there are data points for each of the three experiment phases with different clone variations in the second phase. Comparisons of results are valid column-wise (across different clone strategies) and row-wise (across different file set sizes).

During the second phase of the experiment, we expect that cloning a file system with more objects should take longer as the file set increases in size. Only a single "clone" operation is issued per experiment, in contrast to the multiple create or remove operations that are executed once for each file in the file sets. The lowest value in a column indicates the quickest manner of cloning determined by our experiments. The baseline experiment did not involve a copy or clone of the files, so there are no entries for its row. Making a "clone" by copying the file set into another directory is painfully slow. Both cloning

situations outperform it (although prolific cloning takes much longer with 10,000 files), with prolific cloning presenting the lowest average time. The clone time takes longer with larger file sets, because the NFS server needs to flush its caches to capture a consistent file system state before performing a BulkClone.

The final phase of the experiment deletes the created objects one-by-one with the `rm` command. The "baseline" and "clone via copy" experiments yield similar results as no BulkClone is involved and therefore no clone instantiation takes place. Both situations simply remove files from the file system one-by-one. Prolific cloning again outperforms chain-of-clones cloning in this phase. The difference is dramatic in the case of 10,000 files: prolific cloning takes only 4 times longer than the baseline case, but it takes one-sixth the time of chain-of-clones (which is almost 30 times slower than the baseline case). The performance difference between the two clone strategies is expected, given the observances made in Section 6.4.1 on page 106.

6.6.2 PostMark

The PostMark benchmark [45] is a C program that performs a number of transactions upon files. Each transaction randomly performs a read or append (write) of a random file followed by a create or delete of a random file. The configuration used for PostMark runs is shown in Figure 6.36 on page 138.

Our experiments first measure PostMark unmodified. Then, we modify three copies of the program to create clones (one to use the `cp` command, one to use prolific clones, and one to use chains-of-clones) every ten-percent of the way through the execution of the benchmark's transaction phase. Experiments are run with and without various background instantiation selector algorithms active.

The experiments involve four computers: the storage node, the metadata server, the NFS server, and the NFS client where the PostMark benchmark program is run.

The results from individual experiment runs are available in various tables in the Appendix, Section A.6 starting on page 229. A summary of the results is presented here in Table 6.6 on page 139.

```
PostMark v1.5 :  3/27/01
Reading configuration from file '/tmp/postmark_script_1.pmrc'
Current configuration is:
The base number of files is 500
Transactions:  500
Files range between 500 bytes and 9.77 kilobytes in size
Working directory:  /ux1/ss/ss224
Block sizes are:  read=512 bytes, write=512 bytes
Biases are:  read/append=5, create/delete=5
Using Unix buffered file I/O
Random number generator seed is 42
Report format is verbose.
```

Figure 6.36: PostMark configuration for NFS experiments.
The NFS server is mounted at `/ux1/ss/ss224`. Bias values of 5 indicate a 50% chance.

Running the default configuration of PostMark serves as an "optimal" value for the hardware and software setup used in the experiments. The difference between the information gathered here and that from experiments with clones being made helps us to learn about the costs induced with using bulk operations to clone a file system.

The first experiment with a modified PostMark binary uses the `cp` command to copy all of the files in the benchmark to a separate directory on the NFS server every 10% of the way through the benchmark's transaction phase (i.e., at 0%, 10% ... 90%). This effort of copying the files accomplishes nearly the same results of creating a file system snapshot as using the BulkClone operation (except for atomicity). All of the file contents are captured, but the timestamp metadata of the copy acting as the snapshot is different. When using copy to simulate cloning, the total time is 4.5 times longer than the plain PostMark run. The transaction time is over 11 times longer, and the read and write bandwidth are 5 times slower than the default case.

The second modified PostMark executable uses the BulkClone operation to clone the entire file system upon which the benchmark is running every 10% of the way through the transaction phase in a chain-of-clones fashion. Without background instantiation, the total

Table 6.6: Summary of PostMark results – Average results from multiple experimental runs of various configurations of the PostMark benchmark. Most experiments involved periodic BulkClone operations issued through the NFS server every 10% through the transaction phase (i.e., at 0%, 10% . . . 90%). BulkClones were made in a chain-of-clones manner or a prolific clones manner. For most experiments, background instantiations, with one of three algorithms (random, LIFO, or thinnest range), were also active. Original data for the summary presented here can be found in Section A.6 starting on page 229.

Postmark experiment average results				
Experiment	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
No cloning	156.07	57.67	9.32	30.38
Clone via copy	718.80	634.00	1.96	6.40
Chain	292.62	181.29	4.83	15.73
Chain, Random	401.70	285.60	3.50	11.39
Chain, LIFO	409.60	301.10	3.42	11.14
Chain, Thinnest	380.10	289.90	3.70	12.06
Prolific	315.38	149.81	4.52	14.72
Prolific, Random	348.80	250.80	4.03	13.13
Prolific, LIFO	333.80	234.50	4.22	13.76
Prolific, Thinnest	318.20	222.90	4.42	14.40

execution time is twice that of the base case, the transaction time takes 3 times as long, and the read and write bandwidth is halved. Compared to the use of copy instead of clone, the total execution time is over 2 times faster, transaction time is over 3 times faster, and the read and write bandwidths are 2.5 times higher. Adding background instantiation of bulk operations further slows the execution of the benchmark. However, the decrease in performance is still a great improvement over the use of copy: total time and transaction time are twice as fast and bandwidths are double.

The third modified PostMark executable uses the BulkClone operation in a prolific clone fashion. Without background instantiation, the total execution time is twice that of the base case, the transaction time takes just less than 3 times as long and, the read and

write bandwidth is halved. Compared to the use of copy instead of clone, the total execution time is over 2 times faster, transaction time is over 4 times faster, and the read and write bandwidths are twice as high. Adding background instantiation of bulk operations further slows the execution of the benchmark. However, the decrease in performance is still a great improvement over the use of copy: total time and transaction time are more than twice as fast and bandwidths are more than doubled.

Summary

Overall, using BulkClone to create snapshots during the runs of PostMark showed better performance than naïve copying of files. Both prolific and chains-of-clones exhibited about half of the performance of the default PostMark run, with plain chains-of-clones doing the best (and belying its poor performance in the create-clone-delete benchmark).

6.7 Summary

Early in this chapter, our baseline results showed the performance of the implemented system on basic operations. Building the metadata server around the PostgreSQL database allowed for easier implementation of the system. However, it induced massive overheads compared with what one might expect of custom C/C++ code. We noted that our bulk operation algorithms contributed additional overhead after comparing operations with, and without, bulk operation code active: bulk operations are not free. As an experimental platform for bulk operations, however, it is still valid for demonstrating the concept of delayed instantiation bulk operations.

Client accesses suffer after bulk operations. This is due, in part, to the necessary re-acquisition of capabilities that were invalidated as a part of the bulk operation execution process. On-demand instantiation also increases these operation times.

Background instantiation is capable of eliminating the instantiation step of subsequent accesses to objects affected by bulk operations. When the metadata server is performing background instantiation concurrent with a foreground client workload, performance

suffers until instantiation completes. Random selection of the object to instantiate in the background avoids pathological behaviors encountered in other selection algorithms.

Prolific cloning is a more sustainable cloning strategy, since operation time for chains-of-clones increased with each operation. With some consideration for NFS file server metadata structures (e.g., directory entries), the BulkClone operation can be used to provide a file service with snapshots and forks atop distributed, object-based storage.

Chapter 7

Conclusions and future work

This dissertation presents and analyzes a solution whereby a distributed, object-based storage system can perform atomic clone and delete operations on ranges of objects and use them to perform storage management tasks. The method uses existing client-side algorithms, simple single-object operations at storage nodes, and changes to the metadata server. Details of the changes to the metadata server algorithms are presented in detail. The protocols are designed to accommodate $m-of-n$ data encodings and multiple-storage-node data distributions. Using the opportunities available within the semantics of the BulkClone operation, two strategies for using that operation are analyzed — chains-of-clones and prolific clones — with prolific cloning found to provide much more robust performance. Schemes for mitigating the costs of delayed instantiation of bulk operations through background instantiation are investigated and found to be an important component: they address one of the major costs associated with delayed instantiation bulk operations. An NFS server using the storage system's BulkClone operation provides a capstone demonstration of the ability to generate file system snapshots in distributed, object-based storage.

Overall, this dissertation introduces and explores a method of atomically cloning and deleting large portions of the object identifier namespace to distributed, object-based storage systems. The method allows for centralized coordination of bulk operations across multiple storage nodes. The method introduces access costs after bulk operations that are associated with capability revocation and on-demand instantiation. The on-demand in-

stantiation cost can be eliminated through background instantiation during idle time, and random choice of which objects to background instantiate avoids pathological competition with foreground workloads.

Future work

This section outlines three areas for additional study of delayed-instantiation bulk operations: evaluation in a non-database metadata server, capability system enhancements to overcome revocation costs, better background instantiation idle-time detection and selector algorithms.

First, our use of the PostgreSQL database system for the base of the metadata server had its advantages and disadvantages. It allowed us to relatively quickly implement and experiment with algorithms. It provided a high-level programming language and natively supported transactions. The debugging features eased the implementation task. However, performance was poor due to integration issues with the overall storage system (a metadata server front-end and helper were necessary) and poor optimization of queries (even with available indices). A custom-built metadata server should be an order of magnitude faster, because it would not have the overheads associated with database generalities of access and can use low-level, hand-optimized routines. Experiments with a more performant metadata server should confirm the effectiveness of the approach and the relative merits of the approach studied.

Second, after a bulk operation, capability re-acquisition saps performance by introducing significant latency. Three possibilities could be explored. First, clients could be smarter. They could erase cached capabilities for affected objects when they issue a bulk operation. This would only help the issuing system, however. Also, clients could be informed when capabilities are revoked, just as storage nodes are contacted now. Then, they would know when to purge capabilities from their cache (uncontacted clients would proceed to encounter “invalid capability” messages). Second, alternatives to capability revocation could be investigated. Storage nodes could suspend access to objects affected by a bulk operation as they go through a two-phase commit to enact a bulk operation. This would be akin to placing bulk operation responsibility at the storage nodes and therefore

would work best in error-free cases. Third, range-based capabilities, good for access to a range of objects, could be used to involve more objects in capability granting protocol steps (and revocation steps). Then, when a client requests a new capability after a revocation, it does not need to re-acquire capabilities for each and every object.

Third, management of background instantiation could be improved. The presented algorithms performed poorly with our aggressive idle-time detection system. More accurate and generous idle-detection could be pursued, and the suite of selection algorithms could be expanded to be more intelligent. Much work has been done on idle time detection in other aspects of storage, and these schemes should apply here as well.

Index

ApproveWrite, 39, 92

background instantiation, 29, 34, 40, 109
block encoding algorithm, 15
block size, 14
bulk operation, 3, 4
  at client, 24
  at metadata server, 24
  at storage nodes, 22
  correctness, 75
  delayed instantiation, 5
  responsibility, 21
    considerations, 21
BulkClone, 26, 38, 56, 60, 106
  chain-of-clones, 26
  instantiation, 28
  prolific clone, 26
  uses, 27
BulkDelete, 28, 38, 55, 62, 102
  uses, 28
byte stream, 12

capability, 13, 85
  authorization, 13
  delayed instantiation, 31
  generation, 85
  invalid, 13
  related work, 9
  revocation, 13, 32, 36, 37, 86, 94
chain-of-clones, 26, 61, 106
client
  API, 17
  capability, 13
  description, 16
  implementation, 35
clone, 10
  snapshot, 10
copy-on-write, 31
Create, 46, 74, 87, 103

data distribution, 14, 59
delayed instantiation, 5, 29, 61
  benefits, 30
  capability, 31
  example, 5, 30
Delete, 51
distributed, object-based storage, 1, 2
  client, 2
  metadata server, 2
  namespace, 2
  namespace server, 2

- related work, 7
- storage node, 2

Divorce, 71

Enumerate, 47, 73
extent, 14

FinishWrite, 39, 92
fork, 10
fragments, 15

GetMDOID, 64

immediate execution, 5
immediate instantiation, 29
InstantiateHole, 68, 104
InstantiatePassThroughLimits, 66
instantiation, 29, 33, 61
- background, 29
- delayed, 29
- example, 30
- immediate, 29
- on-demand, 29

lazy evaluation, 31
Linux, 81
list of storage nodes, 16
Lookup, 47, 74, 89, 95

m-of-n encoding, 16, 33
metadata server
- API, 20
- BulkClone table, 59
- BulkDelete table, 58
- capability, 13
- description, 19
- Divorce, 71
- ExtentMetadata table, 59
- GetMDOID, 64
- implementation, 37
- InstantiateHole, 68
- InstantiatePassThroughLimits, 66
- object table, 58
- PendingWrite table, 59

multi-tenant, 1

namespace, 11
NFS server, 42, 135
- snapshot, 44

object, 11
- attributes, 12
- grouping, 25

object identifier, 11, 43
- namespace, 11

object model, 7
object-based storage, 8
- characteristics, 8
- standardization, 10

OID, *see* object identifier
on-demand instantiation, 29, 31

pass-through clone, 62, 63
PostgreSQL, 81, 84
PostMark, 135

prolific clone, 26, 60, 106

range, 25, 26
    restriction, 26
range splitting, 31
    with timestamp tracking, 32
Read, 54, 98
    SSIO_Read, 54, 98, 107
revocation set, 33
root-object, 42

snapshot, 10, 44, 138
    fork, 10
    related work, 9
storage blow-up, 16
storage management, 3
    bulk operation, 4
storage node
    API, 18
    capability, 13
    description, 17
    implementation, 36
super-block, 42
super-object, 27

threshold based encoding, 32
threshold encoding parameters, 15
timestamp tracking, 31, 32
    with range splitting, 32

workload scripting, 82
Write, 52, 89
    SSIO_Write, 52, 95

Bibliography

Numbers at the end of an entry refer to the pages on which that entry is cited.

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 59–72. USENIX Association, 2005. 1, 8, 9, 33, 81

[2] M. E. Adiba and B. G. Lindsay. Database snapshots. *International Conference on Very Large Databases* (Montreal, Canada, 01–03 October 1980), pages 86–91. IEEE, 1980. 9

[3] M. K. Aguilera, S. Spence, and A. Veitch. Olive: distributed point-in-time branching storage for real systems. *Symposium on Networked Systems Design and Implementation* (San Jose, CA, 08–10 May 2006), pages 367–380. USENIX Association, 2006. 9

[4] K. A.-H. M. Ali. *Object-oriented storage management and garbage collection in distributed processing systems*. PhD thesis. Royal Institute of Technology, Department of Computer Science, Stockholm, Sweden, December 1984. 8

[5] G. Almes and G. Robertson. An extensible file system for HYDRA. *Third International Conference on Software Engineering (ICSE).* (Atlanta, GA). IEEE Computer Society, May 1978. 8, 9

[6] Amazon Web Services LLC. Amazon Simple Storage Service (Amazon S3), June 2009. http://aws.amazon.com/s3/. 1, 8

[7] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. *Deductive and Object-Oriented Databases* (Kyoto, Japan, 4–6 December 1989), pages 223–240, 1989. 8

[8] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object store. *IEEE Symposium on Mass Storage Systems* (San Diego, CA, 07–10 April 2003), pages 165–176. IEEE, 2003. 1, 8

[9] E. Bertino and L. Martino. Object-oriented database management systems: concepts and issues. *IEEE Computer*, **24**(4):33–47. IEEE, April 1991. 8

[10] A. D. Birrell and R. M. Needham. A universal file server. *IEEE Transactions of Software Engineering*, **SE–6**(5):450–453. IEEE, September 1980. 8, 9

[11] J. Brandenberg. OSD snapshot proposal v3.14. Storage Networking Industry Association (SNIA), 21 November 2007. SNIA OSD Working Group, Subgroup Snapshot proposal document from www.snia.org. 10, 22

[12] M. R. Brown, K. N. Kolling, and E. A. Taft. The Alpine file system. *ACM Transactions on Computer Systems*, **3**(4):261–293, November 1985. 9

[13] J. Brzezinski and D. Wawrzyniak. Consistency requirements of distributed shared memory for Dijkstra's mutual exclusion algorithm. *International Conference on Distributed Computing Systems* (Taipei, Taiwan, 10–13 April 2000), pages 618–625. IEEE, 2000. 10

[14] B. Callaghan, B. Pawlowski, and P. Staubach. *RFC 1813 - NFS version 3 protocol specification*. RFC–1813. Network Working Group, June 1995. 9, 42

[15] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, **3**(1):63–75, February 1985. 9

[16] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. *Mime: high performance parallel storage device with strong recovery guarantees*. Technical report HPL-92-9. Hewlett-Packard Laboratories, Concurrent Systems Project, 18 March 1992. 8

[17] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: a survey of backup techniques. *Joint NASA and IEEE Mass Storage Conference* (March 1998), 1998.

[18] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. *USENIX Annual Technical Conference* (San Francisco, CA), pages 43–60, 1992.

[19] Cluster File Systems Inc. *Lustre: a scalable, high-performance file system*. White Paper. Cluster File Systems, Inc., November 2002. 1, 8

[20] P. F. Corbett and D. G. Feitelson. Design and implementation of the Vesta parallel file system. *Scalable High-Performance Computing Conference* (Knoxville, Tennessee), pages 63–70, 23–25 May 1994. 9

[21] R. A. Coyne and H. Hulen. An introduction to the Mass Storage System Reference Model, version 5. *IEEE Symposium on Mass Storage Systems* (Monterey, CA), pages 47–53. IEEE Computer Society, 26–29 April 1993. 8

[22] S. C. Crawley. *The entity system: an object-based filing system*. PhD thesis, published as 86. University of Cambridge Computer Laboratory, April 1986. 8

[23] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. *AFIPS Fall Joint Computer Conference*, pages 212–230, 1965. 8

[24] J. Dion. The Cambridge file server. *Operating Systems Review*, **14**(4):26–35, October 1980. 8

[25] EMC Corp. ATMOS: A global offering for information storage and distribution, June 2009. http://www.emc.com/products/detail/software/atmos.htm. 1, 8

[26] EMC Corp., August 2009. http://www.emc.com. 42

[27] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: enterprise storage systems on a shoestring. *Hot Topics in Operating Systems* (Lihue, HI, 18–21 May 2003), pages 133–138. USENIX Association, 2003. 9

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *ACM Symposium on Operating System Principles* (Lake George, NY, 10–22 October 2003), pages 29–43. ACM, 2003. 1, 8, 9

[29] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998. 1, 7

[30] G. A. Gibson, D. F. Nagle, K. Amiri, F. A. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. *Filesystems for network-attached secure disks*.

CMU–CS–97–118. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, July 1997. 1, 8, 9

[31] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, and D. Rochberg. *A case for network-attached secure disks*. Technical Report CMU–CS–96–142. School of Computer Science, Carnegie Mellon University, September 1996. 7

[32] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Seattle, WA, 15–18 June 1997). Published as *Performance Evaluation Review*, **25**(1):272–284. ACM, June 1997. 7

[33] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, **31**(3):288–298, March 1988. 9

[34] H. Gobioff. *Security for a high performance commodity storage subsystem*. PhD thesis, published as Technical Report CMU–CS–99–160. School of Computer Science, Carnegie Mellon University, July 1999. 8, 9

[35] H. Gobioff, G. Gibson, and D. Tygar. *Security for network attached storage devices*. CMU-CS-97-185. School of Computer Science, Carnegie Mellon University, October 1997. 8, 9

[36] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993. 9

[37] R. Grimm, W. C. Hsieh, W. de Jonge, and M. F. Kaashoek. Atomic recovery units: failure atomicity for logical disks. *International Conference on Distributed Computing Systems* (Hong Kong, 27–30 May 1996), pages 26–36. IEEE, 1996. 9

[38] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, **13**(1):124–149. ACM Press, 1991. 10

[39] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994. 10

[40] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system.

ACM Transactions on Computer Systems (TOCS), **6**(1):51–81. ACM, February 1988. 3, 10

[41] A. R. Hurson, S. H. Pakzad, and J.-B. Cheng. Object-oriented database management systems: evolution and performance issues. *Computer*, **26**(2):48–60. IEEE, February 1993. 8

[42] *Information technology - Object Based Storage Devices Command Set (OSD)*. INCITS 400 – 2004 [R2008]. InterNational Committee for Information Technology Standards (INCITS), February 2009. 10

[43] S. Iren, D. Nagle, J. Satran, D. Naor, M. Factor, J. Muth, T. Lanzatella, J. Breher, and M. Chanalapaka. *OSDv2 collections*. Draft version 0.6. Storage Networking Industry Association (SNIA), June 2006. 1, 8, 10

[44] M. Ji. *Instant snapshots in a federated array of bricks*. Technical report HPL-2005-15. Hewlett-Packard, January 2005.

[45] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997. 135, 137

[46] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S.-T. Tu, and E. R. Zayas. DEcorum file system architectural overview. *Summer USENIX Technical Conference* (Anaheim, California), pages 151–163, 11–15 June 1990.

[47] S. N. Khoshafian and G. P. Copeland. Object identity. *Object-Oriented Programming: Systems, Languages, and Applications* (Portland, OR, September 29–October 2 1986), pages 406–416. ACM Press, 1986. 9

[48] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7):558–565, 1978. 10

[49] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982. 10

[50] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):84–92, 1996. 8

[51] P. W. Madany. *An object-oriented framework for file systems*. PhD thesis, published as UIUCDCS–R–92–1751. Department of Computer Science, University of Illinois at Urbana-Champaign, June 1992. 8

[52] E. T. Mueller. *Implementation of nested transactions in a distributed system*. Technical report. University of California, Los Angeles, 1983. 9

[53] E. T. Mueller, J. D. Moore, and G. J. Popek. A nested transaction mechanism for LOCUS. *ACM Symposium on Operating System Principles* (Bretton Woods, New Hampshire). Published as *Operating Systems Review*, **17**(5):71–89, October 1983. 9

[54] Network Appliance, Inc., August 2009. http://www.netapp.com. 42

[55] M. A. Olson. The design and implementation of the Inversion file system. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 205–217, January 1993. 9

[56] Panasas, Inc. Panasas ActiveScale Storage Cluster, October 2006. http://www.panasas.com/products_overview.html. 1, 8

[57] Parascale, Inc. ParaScale Cloud Storage Software – Build your own storage cloud, June 2009. http://www.parascale.com/. 1, 8

[58] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: design and implementation. *Summer USENIX Technical Conference* (Boston, MA, 06–10 June 1994), pages 1–16. USENIX Association, 1994. 9

[59] F. J. Pollack, K. C. Kahn, and R. M. Wilkinson. The iMAX–432 object filing system. *ACM Symposium on Operating System Principles* (Asilomar, Ca). Published as *Operating Systems Review*, **15**(5):137–147, December 1981. 8, 9

[60] K. W. Preslan, S. R. Soltis, C. J. Sabol, M. T. O'Keefe, G. Houlder, and J. Coomes. Device locks: mutual exclusion for storage area networks. *IEEE Symposium on Mass Storage Systems* (San Diego, CA, 15–18 March 1999), pages 262–274. IEEE, 1999. 10

[61] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 89–101. USENIX Association, 2002. 8, 10

[62] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, **1**(1):3–23. ACM Press, February 1983. 9

[63] D. P. Reed and L. Svobodova. SWALLOW: a distributed data storage system for a local network. *International Workshop on Local Networks* (Zurich, Switzerland), August 1980. 9

[64] E. Riedel and J. Satran. OSD Technical Work Group, October 2006. http://www.snia.org/tech_activities/workgroups/osd/. 8, 9

[65] O. Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, **3**(4):1–27. ACM, 2008. 8

[66] O. Rodeh and A. Teperman. zFS - a scalable distributed file system using object disks. *IEEE Symposium on Mass Storage Systems* (San Diego, CA, 07–10 April 2003), pages 207–218. IEEE, 2003. 1, 8

[67] S. Z. Roger W. Cox, Pushan Rinnen. Magic Quadrant for Midrange Enterprise Disk Arrays, June 2009. http://mediaproducts.gartner.com/reprints/hds/article11/article11.html. 10

[68] J. H. Saltzer. Naming and binding of objects. In , volume 60, pages 99–208. Springer-Verlag, Berlin, 1978. 9

[69] B. A. Sanders. The information structure of distributed mutual exclusion algorithms. *ACM Transactions on Computer Systems*, **5**(3):284–299, August 1987. 10

[70] Secretariat, Computer and Business Equipment Manufacturers Association. *Draft proposed American National Standard for information systems – Small Computer System Interface-2 (SCSI-2)*, Draft ANSI standard X3T9.2/86-109., 2 February 1991 (revision 10d). 7

[71] M. Spezialetti and P. Kearns. Efficient distributed snapshots. *International Conference on Distributed Computing Systems* (Cambridge, MA), pages 382–388. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986. 9

[72] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, **34**(10):78–92. ACM, October 1991. 9

[73] Sun Microsystems, Inc. *NFS: network file system protocol specification*. RFC–1094. Network Working Group, March 1989. 9

[74] L. Svobodova. A reliable object-oriented data repository for a distributed computer system. *ACM Symposium on Operating System Principles* (Asilomar, Ca). Published as *Operating Systems Review*, **15**(5):47–58, December 1981. 8

[75] L. Svobodova. File servers for network-based distributed systems. *ACM Computing Surveys*, **16**(4):353–398, December 1984. 9

[76] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):224–237. ACM, 1997. 8

[77] B. Walker, G. Popek, R. English, C. Kline, and G. Theil. The LOCUS distributed operating system. *ACM Symposium on Operating System Principles* (Bretton Woods, New Hampshire). Published as *Operating Systems Review*, **17**(5):49–70, October 1983. 9

[78] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. OBFS: a file system for object-based storage devices. *NASA Goddard/IEEE Conference on Mass Storage System and Technologies* (Adelphi, MD, 13–16 April 2004). IEEE, 2004. 1, 8

[79] R. O. Weber. SCSI object-based storage device commands-2 (OSD-2). Storage Networking Industry Association (SNIA), 16 January 2009. Draft r05 of SNIA OSD Working Group proposal. 10

[80] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. Long. Ceph: A scalable, high-performance distributed file system. *Symposium on Operating Systems Design and Implementation* (December). USENIX Association, 2006. 1, 8

[81] J. J. Wylie. *A read/write protocol family for versatile storage infrastructures*. PhD thesis, published as Technical Report CMU-PDL-05-108. Carnegie Mellon University, October 2005. 9, 33

[82] A. K. Yeo, A. L. Ananda, and E. K. Koh. A taxonomy of issues in name systems design and implementation. *Operating Systems Review*, **27**(3):4–18. ACM Press, 1993. 9

Appendix A

Experimental results

This chapter holds the statistical tables summarizing the results of various experiments from the Evaluation chapter, starting on page 80.

A.1 Baseline behavior results

A.1.1 Database access experiment results

This table describes data mentioned in Section 6.2.1 starting on page 84.

Table A.1: Ping metadata server back-end database – This data shows the basic cost of interacting with the metadata server back-end database through the use of a “ping” function.

Ping of Metadata Server database backend	
Data points	1000
Minimum value (ms)	4.737
Maximum value (ms)	8.996
Mean (ms)	5.099
Standard deviation (ms)	0.305
1st Percentile	4.816
5th Percentile	4.866
10th Percentile	4.878
25th Percentile	4.990
50th Percentile	5.011
75th Percentile	5.120
90th Percentile	5.240
95th Percentile	5.420
99th Percentile	6.420

A.1.2 Capability experiment results

These tables describe data mentioned in Section 6.2.2 starting on page 85.

Table A.2: Acquiring storage node addressing information – Experimental timings of enumeration of storage node addressing information with two storage nodes in the system.

Filtering Directory Service for storage node endpoints	
Data points	1000
Minimum value (ms)	0.975
Maximum value (ms)	2.130
Mean (ms)	1.352
Standard deviation (ms)	0.430
1st Percentile	0.990
5th Percentile	0.992
10th Percentile	0.995
25th Percentile	0.998
50th Percentile	1.122
75th Percentile	1.956
90th Percentile	1.989
95th Percentile	2.008
99th Percentile	2.057

Table A.3: Capability revocation – Results for operation timing of capability revocation at two storage nodes.

Capability revocation at 2 storage nodes	
Data points	1000
Minimum value (ms)	1.045
Maximum value (ms)	2.007
Mean (ms)	1.239
Standard deviation (ms)	0.091
1st Percentile	1.091
5th Percentile	1.118
10th Percentile	1.146
25th Percentile	1.173
50th Percentile	1.224
75th Percentile	1.282
90th Percentile	1.349
95th Percentile	1.394
99th Percentile	1.526

A.1.3 Create experiment results

These tables describe data mentioned in Section 6.2.3 starting on page 87.

Table A.4: Create(), sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the Create() function with no bulk operation code activated and no background instantiations being performed. Create() is called sequentially to the target object set.

Sequential Create()	
Data points	10000
Minimum value (ms)	9.237
Maximum value (ms)	89.732
Mean (ms)	10.020
Standard deviation (ms)	1.165
1st Percentile	9.430
5th Percentile	9.549
10th Percentile	9.609
25th Percentile	9.716
50th Percentile	9.859
75th Percentile	10.034
90th Percentile	10.287
95th Percentile	10.958
99th Percentile	13.164

Table A.5: Create(), random, no bulk operation code, no background instantiation – Statistics for the timing of calls to the Create() function with no bulk operation code activated and no background instantiations being performed. Create() is called randomly to the target object set.

Random Create()	
Data points	10000
Minimum value (ms)	1.675
Maximum value (ms)	82.697
Mean (ms)	10.313
Standard deviation (ms)	1.478
1st Percentile	9.568
5th Percentile	9.706
10th Percentile	9.776
25th Percentile	9.904
50th Percentile	10.063
75th Percentile	10.278
90th Percentile	10.911
95th Percentile	12.487
99th Percentile	13.804

Table A.6: Create(), sequential, bulk operation code, no background instantiation – Statistics for the timing of calls to the Create() function with bulk operation code activated and no background instantiations being performed. Create() is called sequentially to the target object set.

Sequential Create()	
Data points	10000
Minimum value (ms)	2.233
Maximum value (ms)	174.643
Mean (ms)	15.610
Standard deviation (ms)	2.984
1st Percentile	14.758
5th Percentile	15.086
10th Percentile	15.184
25th Percentile	15.329
50th Percentile	15.487
75th Percentile	15.671
90th Percentile	15.914
95th Percentile	16.245
99th Percentile	19.091

Table A.7: Create(), random, bulk operation code, no background instantiation – Statistics for the timing of calls to the Create() function with bulk operation code activated and no background instantiations being performed. Create() is called randomly to the target object set.

Random Create()	
Data points	10000
Minimum value (ms)	14.829
Maximum value (ms)	46.842
Mean (ms)	16.084
Standard deviation (ms)	1.337
1st Percentile	15.173
5th Percentile	15.338
10th Percentile	15.413
25th Percentile	15.547
50th Percentile	15.714
75th Percentile	15.953
90th Percentile	16.622
95th Percentile	19.768
99th Percentile	21.016

A.1.4 Write experiment results

These tables describe data mentioned in Section 6.2.4 starting on page 89.

Table A.8: Lookup(), sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the Lookup() function with no active bulk operation code and no background instantiations being performed. Lookup() is called (twice for each Write()) as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks.

Sequential Lookup() of 64 kB with 16 kB blocks	
Data points	2000
Minimum value (ms)	7.190
Maximum value (ms)	47.954
Mean (ms)	7.635
Standard deviation (ms)	1.204
1st Percentile	7.255
5th Percentile	7.338
10th Percentile	7.382
25th Percentile	7.460
50th Percentile	7.563
75th Percentile	7.662
90th Percentile	7.766
95th Percentile	7.853
99th Percentile	8.969

Table A.9: ApproveWrite(), sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the ApproveWrite() function with no active bulk operation code and no background instantiations being performed. ApproveWrite() is called as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks.

Sequential ApproveWrite() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	10.535
Maximum value (ms)	65.921
Mean (ms)	11.052
Standard deviation (ms)	1.917
1st Percentile	10.597
5th Percentile	10.676
10th Percentile	10.712
25th Percentile	10.797
50th Percentile	10.903
75th Percentile	11.022
90th Percentile	11.204
95th Percentile	11.342
99th Percentile	13.569

Table A.10: SSIO_Write(), sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the low-level SSIO_Write() function with no active bulk operation code and no background instantiations being performed. SSIO_Write() is called as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks. It is the function that actually sends data to the storage node.

Sequential SSIO_Write() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	5.619
Maximum value (ms)	41.584
Mean (ms)	6.308
Standard deviation (ms)	1.146
1st Percentile	5.761
5th Percentile	5.895
10th Percentile	5.986
25th Percentile	6.143
50th Percentile	6.277
75th Percentile	6.374
90th Percentile	6.473
95th Percentile	6.578
99th Percentile	7.400

Table A.11: Finish_Write(), sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the Finish_Write() function with no active bulk operation code and no background instantiations being performed. Finish_Write() is called as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks.

Sequential FinishWrite() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	14.053
Maximum value (ms)	101.446
Mean (ms)	14.662
Standard deviation (ms)	3.850
1st Percentile	14.125
5th Percentile	14.208
10th Percentile	14.235
25th Percentile	14.304
50th Percentile	14.391
75th Percentile	14.511
90th Percentile	14.691
95th Percentile	14.937
99th Percentile	19.096

Table A.12: Write(), sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the client library Write() function with no active bulk operation code and no background instantiations being performed. Write() is called to the target object set and sequential access is done to 64 kB in 16 kB blocks.

Sequential Write() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	45.577
Maximum value (ms)	235.872
Mean (ms)	47.377
Standard deviation (ms)	7.674
1st Percentile	45.997
5th Percentile	46.245
10th Percentile	46.361
25th Percentile	46.563
50th Percentile	46.792
75th Percentile	47.089
90th Percentile	47.479
95th Percentile	47.878
99th Percentile	55.888

Table A.13: Lookup(), sequential, active bulk operation code, no background instantiation – Statistics for the timing of calls to the Lookup() function with active bulk operation code and no background instantiations being performed. Lookup() is called (twice for each Write()) as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks.

Sequential Lookup() of 64 kB with 16 kB blocks	
Data points	2000
Minimum value (ms)	8.629
Maximum value (ms)	48.409
Mean (ms)	9.184
Standard deviation (ms)	1.253
1st Percentile	8.736
5th Percentile	8.835
10th Percentile	8.868
25th Percentile	8.951
50th Percentile	9.045
75th Percentile	9.284
90th Percentile	9.463
95th Percentile	9.535
99th Percentile	10.834

Table A.14: ApproveWrite(), sequential, bulk operation code, no background instantiation – Statistics for the timing of calls to the ApproveWrite() function with active bulk operation code and no background instantiations being performed. ApproveWrite() is called as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks.

Sequential ApproveWrite() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	13.104
Maximum value (ms)	84.657
Mean (ms)	13.652
Standard deviation (ms)	2.541
1st Percentile	13.212
5th Percentile	13.289
10th Percentile	13.336
25th Percentile	13.400
50th Percentile	13.487
75th Percentile	13.594
90th Percentile	13.710
95th Percentile	13.861
99th Percentile	15.731

Table A.15: SSIO_Write(), sequential, active bulk operation code, no background instantiation – Statistics for the timing of calls to the low-level SSIO_Write() function with active bulk operation code and no background instantiations being performed. SSIO_Write() is called as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks. It is the function that actually sends data to the storage node.

Sequential SSIO_Write() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	5.440
Maximum value (ms)	16.743
Mean (ms)	6.413
Standard deviation (ms)	0.562
1st Percentile	5.731
5th Percentile	5.945
10th Percentile	6.033
25th Percentile	6.174
50th Percentile	6.304
75th Percentile	6.460
90th Percentile	6.868
95th Percentile	7.534
99th Percentile	7.868

Table A.16: Finish_Write(), sequential, bulk operation code, no background instantiation – Statistics for the timing of calls to the Finish_Write() function with active bulk operation code and no background instantiations being performed. Finish_Write() is called as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks.

Sequential FinishWrite() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	15.120
Maximum value (ms)	99.237
Mean (ms)	15.802
Standard deviation (ms)	3.772
1st Percentile	15.224
5th Percentile	15.299
10th Percentile	15.342
25th Percentile	15.432
50th Percentile	15.538
75th Percentile	15.661
90th Percentile	15.865
95th Percentile	16.054
99th Percentile	18.707

Table A.17: Write(), sequential, bulk operation code, no background instantiation – Statistics for the timing of calls to the Write() function with active bulk operation code and no background instantiations being performed. Write() is called to the target object set and sequential access is done to 64 kB in 16 kB blocks.

Sequential Write() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	52.329
Maximum value (ms)	253.959
Mean (ms)	54.317
Standard deviation (ms)	8.274
1st Percentile	52.756
5th Percentile	52.964
10th Percentile	53.117
25th Percentile	53.402
50th Percentile	53.711
75th Percentile	54.067
90th Percentile	54.631
95th Percentile	55.159
99th Percentile	61.710

Table A.18: Re-Write(), sequential, no bulk operation code, no background instantiation – When over-writing data on recently written objects, these are the statistics for the timing of calls to the Write() function with no active bulk operation code and no background instantiations being performed. Write() is called to the target object set and sequential access is done to 64 kB in 16 kB blocks. The client uses valid cached capabilities during the operation.

Sequential Re-Write() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	5.179
Maximum value (ms)	6.890
Mean (ms)	5.817
Standard deviation (ms)	0.211
1st Percentile	5.348
5th Percentile	5.499
10th Percentile	5.560
25th Percentile	5.673
50th Percentile	5.818
75th Percentile	5.945
90th Percentile	6.041
95th Percentile	6.123
99th Percentile	6.558

Table A.19: Re-Write(), revoked caps, sequential, no bulk ops, no background instantiation – When over-writing data on recently written objects with revoked capabilities, these are the statistics for the timing of calls to the Write() function with no active bulk operation code and no background instantiations being performed. Write() is called to the target object set after metadata and capabilities have been cached, and then capabilities are revoked and finally sequential access is done to 64 kB in 16 kB blocks.

Post-revoke sequential re-Write() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	29.525
Maximum value (ms)	140.515
Mean (ms)	31.316
Standard deviation (ms)	6.269
1st Percentile	29.795
5th Percentile	30.082
10th Percentile	30.238
25th Percentile	30.464
50th Percentile	30.698
75th Percentile	31.004
90th Percentile	31.344
95th Percentile	31.965
99th Percentile	37.502

Table A.20: Cache-hit re-Lookup(), seq, no bulk ops, no background instantiation – When over-writing data on recently written objects with revoked capabilities, these are the statistics for the timing of calls to the Lookup() function with no active bulk operation code and no background instantiations being performed. Lookup() is called as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks. This Lookup() operation finds cached metadata and capabilities for the over-write, but the capabilities have been revoked and another Lookup() will need to be performed.

Sequential fast re-Lookup() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	0.072
Maximum value (ms)	0.732
Mean (ms)	0.094
Standard deviation (ms)	0.027
1st Percentile	0.076
5th Percentile	0.080
10th Percentile	0.080
25th Percentile	0.084
50th Percentile	0.088
75th Percentile	0.093
90th Percentile	0.121
95th Percentile	0.129
99th Percentile	0.169

Table A.21: Re-Lookup() to MDS, seq, no bulk ops, no background instantiation – When over-writing data on recently written objects with revoked capabilities, these are the statistics for the timing of calls to the Lookup() function with no active bulk operation code and no background instantiations being performed. Lookup() is called as a part of the higher-level Write() operation to the target object set and sequential access is done to 64 kB in 16 kB blocks. This Lookup() operation goes over the network to the MDS and re-acquires capabilities and metadata for the over-write.

Sequential slow re-Lookup() of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	17.884
Maximum value (ms)	128.777
Mean (ms)	19.385
Standard deviation (ms)	6.243
1st Percentile	18.181
5th Percentile	18.365
10th Percentile	18.445
25th Percentile	18.619
50th Percentile	18.807
75th Percentile	19.008
90th Percentile	19.263
95th Percentile	19.537
99th Percentile	25.505

Table A.22: SSIO_Write() after revoke, seq, no bulk ops, no background instantiation – There are two calls to the low-level SSIO_Write() function when re-writing data with revoked capabilities. The first call will fail with an error informing the caller that the capabilities presented are invalid. The second call will succeed after the client library logic has re-acquired capabilities. These are the statistics for the calls to SSIO_Write() with sequential writes to 64 kB of an object in 16 kB blocks.

Sequential SSIO_Write() of 64 kB with 16 kB blocks after revoke	
Data points	2000
Minimum value (ms)	5.007
Maximum value (ms)	7.841
Mean (ms)	5.793
Standard deviation (ms)	0.273
1st Percentile	5.233
5th Percentile	5.376
10th Percentile	5.463
25th Percentile	5.615
50th Percentile	5.780
75th Percentile	5.978
90th Percentile	6.107
95th Percentile	6.178
99th Percentile	6.476

A.1.5 Read experiment results

These tables describe data mentioned in Section 6.2.5 starting on page 98.

Table A.23: Read(), sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the (sslib)Read() function with no active bulk operation code and no background instantiations being performed. Read() is called with valid capabilities and sequential access is done to 64 kB in 16 kB blocks.

Initial sequential Sslib Read with caps of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	5.194
Maximum value (ms)	8.393
Mean (ms)	5.901
Standard deviation (ms)	0.335
1st Percentile	5.450
5th Percentile	5.548
10th Percentile	5.609
25th Percentile	5.718
50th Percentile	5.848
75th Percentile	5.996
90th Percentile	6.158
95th Percentile	6.326
99th Percentile	7.514

Table A.24: Read(), sequential, no bulk operation code, no background instantiation, invalid caps – Statistics for the timing of calls to the Read() function with no active bulk operation code, no background instantiations being performed, and invalid capabilities (that are re-acquired). Read() is called with invalid capabilities and sequential access is done to 64 kB in 16 kB blocks.

Sequential Sslib Re-Read with caps of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	26.207
Maximum value (ms)	136.606
Mean (ms)	27.873
Standard deviation (ms)	5.918
1st Percentile	26.494
5th Percentile	26.713
10th Percentile	26.831
25th Percentile	27.015
50th Percentile	27.274
75th Percentile	27.616
90th Percentile	28.013
95th Percentile	28.301
99th Percentile	33.612

Table A.25: Lookup(), fast, bad caps, sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the Lookup() function with no active bulk operation code and no background instantiations being performed. Lookup() is called and finds locally cached capabilities and metadata to use for reading objects.

Sequential fast re-Lookup() without caps of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	0.069
Maximum value (ms)	0.484
Mean (ms)	0.092
Standard deviation (ms)	0.021
1st Percentile	0.076
5th Percentile	0.076
10th Percentile	0.080
25th Percentile	0.084
50th Percentile	0.087
75th Percentile	0.091
90th Percentile	0.124
95th Percentile	0.129
99th Percentile	0.144

Table A.26: SSIO_Read(), fast, bad caps, sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the SSIO_Read() function with no active bulk operation code and no background instantiations being performed. SSIO_Read() is called with invalid capabilities and sequential access is attempted to 64 kB in 16 kB blocks. These are the stats for the quicker operations during re-read. This read attempt is quicker because no data is transferred.

Sequential fast Read() without caps of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	2.785
Maximum value (ms)	3.700
Mean (ms)	3.129
Standard deviation (ms)	0.128
1st Percentile	2.888
5th Percentile	2.953
10th Percentile	2.992
25th Percentile	3.039
50th Percentile	3.108
75th Percentile	3.198
90th Percentile	3.311
95th Percentile	3.372
99th Percentile	3.467

Table A.27: Lookup(), slow, bad caps, sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the Lookup() function with no active bulk operation code and no background instantiations being performed. Lookup() is called and re-acquires capabilities from the metadata service after a previous operation returned with an "invalid capability" error.

Sequential slow re-Lookup() without caps of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	18.131
Maximum value (ms)	127.994
Mean (ms)	19.482
Standard deviation (ms)	5.893
1st Percentile	18.273
5th Percentile	18.422
10th Percentile	18.516
25th Percentile	18.686
50th Percentile	18.884
75th Percentile	19.159
90th Percentile	19.587
95th Percentile	19.880
99th Percentile	25.220

Table A.28: SSIO_Read(), slow, bad caps, sequential, no bulk operation code, no background instantiation – Statistics for the timing of calls to the SSIO_Read() function with no active bulk operation code and no background instantiations being performed. SSIO_Read() is called with re-acquired capabilities and sequential access is done to 64 kB in 16 kB blocks. These are the stats for the slower operations during re-read. This read attempt takes longer because data is actually transferred.

Sequential slow Read() without caps of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	4.392
Maximum value (ms)	5.305
Mean (ms)	4.941
Standard deviation (ms)	0.141
1st Percentile	4.551
5th Percentile	4.691
10th Percentile	4.751
25th Percentile	4.862
50th Percentile	4.950
75th Percentile	5.033
90th Percentile	5.114
95th Percentile	5.163
99th Percentile	5.253

Table A.29: Read(), sequential, bulk operation code, no background instantiation, invalid caps – Statistics for the timing of calls to the Read() function with active bulk operation code, no background instantiations being performed, and invalid capabilities (that are re-acquired). Read() is called with invalid capabilities and sequential access is done to 64 kB in 16 kB blocks.

Sequential Re-Read with bulk ops and caps of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	29.819
Maximum value (ms)	150.190
Mean (ms)	32.009
Standard deviation (ms)	6.338
1st Percentile	30.132
5th Percentile	30.415
10th Percentile	30.543
25th Percentile	30.804
50th Percentile	31.252
75th Percentile	31.869
90th Percentile	32.660
95th Percentile	33.240
99th Percentile	41.302

Table A.30: Lookup(), slow, bad caps, sequential, with bulk operation code, no background instantiation – Statistics for the timing of calls to the Lookup() function with active bulk operation code and no background instantiations being performed. Lookup() is called and re-acquires capabilities from the metadata service after a previous operation returned with an "invalid capability" error.

Sequential slow Lookup() without caps, with bulk ops, of 64 kB with 16 kB blocks	
Data points	1000
Minimum value (ms)	21.567
Maximum value (ms)	141.371
Mean (ms)	23.548
Standard deviation (ms)	6.316
1st Percentile	21.852
5th Percentile	22.042
10th Percentile	22.160
25th Percentile	22.412
50th Percentile	22.774
75th Percentile	23.354
90th Percentile	24.080
95th Percentile	24.580
99th Percentile	32.811

A.2 BulkDelete experiment results

These tables describe data mentioned in Section 6.3 starting on page 102.

Table A.31: BulkDelete of single objects, sequential, no background instantiation – Statistics for the timing of calls to the BulkDelete function with bulk operation code active and no background instantiations being performed.

Sequential BulkDelete of single objects	
Data points	1000
Minimum value (ms)	6.889
Maximum value (ms)	21.419
Mean (ms)	7.946
Standard deviation (ms)	1.277
1st Percentile	7.006
5th Percentile	7.085
10th Percentile	7.180
25th Percentile	7.360
50th Percentile	7.738
75th Percentile	8.134
90th Percentile	8.430
95th Percentile	8.719
99th Percentile	15.338

Table A.32: Create after BulkDelete of single objects, sequential, no background instantiation – Statistics for the timing of calls to the Create command on objects recently deleted with bulk operation code active and no background instantiations being performed.

Sequential Create after BulkDelete of single objects	
Data points	1000
Minimum value (ms)	15.227
Maximum value (ms)	46.111
Mean (ms)	18.234
Standard deviation (ms)	3.218
1st Percentile	15.444
5th Percentile	15.693
10th Percentile	15.912
25th Percentile	16.644
50th Percentile	17.950
75th Percentile	19.268
90th Percentile	19.947
95th Percentile	20.178
99th Percentile	44.496

A.3 BulkClone experiment results

These tables describe data mentioned in Section 6.4 starting on page 106.

Table A.33: Repeated BulkClone of 1000 objects, prolific – Statistics for calling BulkClone of 1000 objects 2000 times. Cloning was done in a prolific clones manner.

Prolific BulkClone of 1000 objects	
Data points	2000
Minimum value (ms)	27.936
Maximum value (ms)	412.371
Mean (ms)	251.640
Standard deviation (ms)	42.028
1st Percentile	34.917
5th Percentile	247.883
10th Percentile	250.672
25th Percentile	253.924
50th Percentile	258.199
75th Percentile	263.401
90th Percentile	266.540
95th Percentile	268.926
99th Percentile	300.088

Table A.34: Repeated BulkClone of 1000 objects, chain-of-clones – Statistics for calling BulkClone of 1000 objects 25 times. Cloning was done in a chain-of-clones manner.

Chain-of-clones BulkClone of 1000 objects	
Data points	25
Minimum value (ms)	309.041
Maximum value (ms)	3414.768
Mean (ms)	1778.515
Standard deviation (ms)	970.079
1st Percentile	0.000
5th Percentile	329.781
10th Percentile	452.211
25th Percentile	1001.541
50th Percentile	1723.062
75th Percentile	2564.575
90th Percentile	3138.611
95th Percentile	3279.318
99th Percentile	3414.768

Table A.35: Read of BulkClone source objects – Statistics for the timing of Read of the source objects after a BulkClone.

Read of BulkClone source objects	
Data points	1000
Minimum value (ms)	44.305
Maximum value (ms)	124.206
Mean (ms)	74.331
Standard deviation (ms)	15.652
1st Percentile	45.028
5th Percentile	58.919
10th Percentile	60.138
25th Percentile	64.727
50th Percentile	71.319
75th Percentile	78.948
90th Percentile	83.907
95th Percentile	120.155
99th Percentile	122.616

Table A.36: Read of BulkClone destination objects – Statistics for the timing of Read of the destination objects after a BulkClone.

Read of BulkClone destination objects	
Data points	1000
Minimum value (ms)	29.387
Maximum value (ms)	467.728
Mean (ms)	72.956
Standard deviation (ms)	20.762
1st Percentile	42.799
5th Percentile	63.467
10th Percentile	64.471
25th Percentile	67.224
50th Percentile	71.886
75th Percentile	77.353
90th Percentile	81.033
95th Percentile	82.305
99th Percentile	89.686

A.4 Background instantiation experiment results

These tables describe data mentioned in Section 6.5 starting on page 109.

Table A.37: Create after BulkDelete and completed background instantiation – Statistical summary of Create operations executed after a BulkDelete has been completely instantiated by background instantiation.

Create after BulkDelete and completed background instantiation	
Data points	1000
Minimum value (ms)	15.200
Maximum value (ms)	22.626
Mean (ms)	16.012
Standard deviation (ms)	0.725
1st Percentile	15.374
5th Percentile	15.501
10th Percentile	15.573
25th Percentile	15.701
50th Percentile	15.874
75th Percentile	16.045
90th Percentile	16.265
95th Percentile	16.851
99th Percentile	19.344

Table A.38: Read after BulkClone and completed background instantiation – Statistical summary of Read operations executed after a BulkClone has been completely instantiated by background instantiation. The read is of 64 kB in 16 kB blocks.

Read after BulkClone and completed background instantiation	
Data points	1000
Minimum value (ms)	11.080
Maximum value (ms)	85.953
Mean (ms)	36.390
Standard deviation (ms)	6.819
1st Percentile	12.496
5th Percentile	18.574
10th Percentile	35.682
25th Percentile	36.482
50th Percentile	37.414
75th Percentile	38.393
90th Percentile	39.442
95th Percentile	41.052
99th Percentile	53.050

Table A.39: Create after BulkDelete 1:1 with Sleep – Statistical summary of Create operations after BulkDelete randomly performed in a one-to-one ratio with Sleep operations of 18 ms.

Create after BulkDelete, 1 Create per 1 Sleep	
Data points	955
Minimum value (ms)	5.712
Maximum value (ms)	774.735
Mean (ms)	33.516
Standard deviation (ms)	47.941
1st Percentile	5.906
5th Percentile	7.904
10th Percentile	17.223
25th Percentile	18.060
50th Percentile	18.837
75th Percentile	29.937
90th Percentile	62.008
95th Percentile	87.598
99th Percentile	270.094

Table A.40: Create after BulkDelete 1:3 with Sleep – Statistical summary of Create operations after BulkDelete randomly performed in a one-to-three ratio with Sleep operations of 18 ms.

Create after BulkDelete, 1 Create per 3 Sleep	
Data points	971
Minimum value (ms)	4.246
Maximum value (ms)	323.903
Mean (ms)	23.771
Standard deviation (ms)	28.760
1st Percentile	6.048
5th Percentile	13.421
10th Percentile	15.325
25th Percentile	15.742
50th Percentile	16.042
75th Percentile	17.160
90th Percentile	39.195
95th Percentile	68.159
99th Percentile	162.186

Table A.41: Create after BulkDelete 1:7 with Sleep – Statistical summary of Create operations after BulkDelete randomly performed in a one-to-seven ratio with Sleep operations of 18 ms.

Create after BulkDelete, 1 Create per 7 Sleep	
Data points	953
Minimum value (ms)	5.700
Maximum value (ms)	197.064
Mean (ms)	18.737
Standard deviation (ms)	13.733
1st Percentile	7.622
5th Percentile	15.309
10th Percentile	15.563
25th Percentile	15.845
50th Percentile	16.168
75th Percentile	16.596
90th Percentile	19.127
95th Percentile	30.665
99th Percentile	89.657

Table A.42: Read after BulkClone 1:1 with Sleep – Statistical summary of Read operations after BulkClone randomly performed in a one-to-one ratio with Sleep operations of 32 ms. The read is of 64 kB in 16 kB blocks.

Read after BulkClone, 1 Read per 1 Sleep	
Data points	999
Minimum value (ms)	4.961
Maximum value (ms)	274.599
Mean (ms)	56.076
Standard deviation (ms)	34.423
1st Percentile	21.978
5th Percentile	37.057
10th Percentile	37.435
25th Percentile	38.197
50th Percentile	39.175
75th Percentile	57.400
90th Percentile	111.969
95th Percentile	131.383
99th Percentile	179.563

Table A.43: Read after BulkClone 1:3 with Sleep – Statistical summary of Read operations after BulkClone randomly performed in a one-to-three ratio with Sleep operations of 32 ms. The read is of 64 kB in 16 kB blocks.

Read after BulkClone, 1 Read per 3 Sleep	
Data points	985
Minimum value (ms)	21.596
Maximum value (ms)	214.281
Mean (ms)	49.871
Standard deviation (ms)	27.042
1st Percentile	35.496
5th Percentile	36.985
10th Percentile	37.386
25th Percentile	38.039
50th Percentile	39.101
75th Percentile	40.382
90th Percentile	90.656
95th Percentile	115.628
99th Percentile	153.855

Table A.44: Read after BulkClone 1:7 with Sleep – Statistical summary of Read operations after BulkClone randomly performed in a one-to-seven ratio with Sleep operations of 32 ms. The read is of 64 kB in 16 kB blocks.

Read after BulkClone, 1 Read per 7 Sleep	
Data points	956
Minimum value (ms)	18.093
Maximum value (ms)	204.853
Mean (ms)	43.714
Standard deviation (ms)	18.583
1st Percentile	35.941
5th Percentile	36.659
10th Percentile	37.180
25th Percentile	38.024
50th Percentile	38.851
75th Percentile	39.813
90th Percentile	43.296
95th Percentile	83.861
99th Percentile	132.867

Table A.45: Random Read after BulkClone with Random background instantiation – Statistics for the timing of random Read of the source objects after a BulkClone while random bulk operations are being instantiated in the background.

Data points	2000
Minimum value (ms)	4.945
Maximum value (ms)	140.341
Mean (ms)	25.603
Standard deviation (ms)	28.109
1st Percentile	5.101
5th Percentile	5.256
10th Percentile	5.353
25th Percentile	5.465
50th Percentile	5.659
75th Percentile	38.830
90th Percentile	65.777
95th Percentile	86.577
99th Percentile	116.845

Table A.46: Sequential Read after BulkClone with Random background instantiation – Statistics for the timing of sequential Read of the source objects after a BulkClone while random bulk operations are being instantiated in the background.

Data points	1000
Minimum value (ms)	16.827
Maximum value (ms)	147.663
Mean (ms)	49.968
Standard deviation (ms)	22.834
1st Percentile	18.390
5th Percentile	22.226
10th Percentile	36.923
25th Percentile	38.455
50th Percentile	39.494
75th Percentile	59.124
90th Percentile	85.985
95th Percentile	103.306
99th Percentile	122.014

Table A.47: Random Create after BulkDelete with Random background instantiation – Statistics for the timing of random Create of objects after a BulkDelete while random bulk operations are being instantiated in the background.

Data points	1000
Minimum value (ms)	7.096
Maximum value (ms)	496.595
Mean (ms)	34.979
Standard deviation (ms)	44.277
1st Percentile	7.146
5th Percentile	7.281
10th Percentile	7.390
25th Percentile	7.732
50th Percentile	8.395
75th Percentile	61.645
90th Percentile	95.977
95th Percentile	115.394
99th Percentile	153.367

Table A.48: Sequential Create after BulkDelete with Random background instantiation – Statistics for the timing of sequential Create of objects after a BulkDelete while random bulk operations are being instantiated in the background.

Data points	1000
Minimum value (ms)	4.265
Maximum value (ms)	641.768
Mean (ms)	26.051
Standard deviation (ms)	44.690
1st Percentile	5.864
5th Percentile	5.991
10th Percentile	6.069
25th Percentile	8.095
50th Percentile	8.548
75th Percentile	32.992
90th Percentile	60.286
95th Percentile	82.454
99th Percentile	147.391

Table A.49: Random Read after BulkClone with FIFO background instantiation – Statistics for the timing of random Read of the destination objects after a BulkClone while bulk operations are being instantiated in the background via FIFO ordering.

Data points	4000
Minimum value (ms)	4.734
Maximum value (ms)	153.142
Mean (ms)	16.333
Standard deviation (ms)	22.678
1st Percentile	5.029
5th Percentile	5.190
10th Percentile	5.277
25th Percentile	5.408
50th Percentile	5.537
75th Percentile	6.429
90th Percentile	39.038
95th Percentile	56.111
99th Percentile	108.182

Table A.50: Sequential Read after BulkClone with FIFO background instantiation – Statistics for the timing of sequential Read of the destination objects after a BulkClone while bulk operations are being instantiated in the background via FIFO ordering.

Data points	2000
Minimum value (ms)	4.858
Maximum value (ms)	205.745
Mean (ms)	34.389
Standard deviation (ms)	44.894
1st Percentile	5.066
5th Percentile	5.222
10th Percentile	5.311
25th Percentile	5.440
50th Percentile	6.574
75th Percentile	38.622
90th Percentile	119.703
95th Percentile	150.726
99th Percentile	184.284

Table A.51: Random Create after BulkDelete with background instantiation via FIFO processing – Statistics for the timing of random Create of objects after a BulkDelete while bulk operations are being instantiated in the background via FIFO processing.

Data points	1000
Minimum value (ms)	5.345
Maximum value (ms)	225.328
Mean (ms)	22.491
Standard deviation (ms)	31.096
1st Percentile	5.592
5th Percentile	6.207
10th Percentile	7.150
25th Percentile	7.416
50th Percentile	7.917
75th Percentile	8.435
90th Percentile	74.788
95th Percentile	88.756
99th Percentile	124.476

Table A.52: Sequential Create after BulkDelete with background instantiation via FIFO processing – Statistics for the timing of sequential Create of objects after a BulkDelete while bulk operations are being instantiated in the background via FIFO processing.

Data points	1000
Minimum value (ms)	4.190
Maximum value (ms)	1111.061
Mean (ms)	123.614
Standard deviation (ms)	234.727
1st Percentile	4.293
5th Percentile	4.361
10th Percentile	5.708
25th Percentile	7.619
50th Percentile	8.038
75th Percentile	8.514
90th Percentile	540.930
95th Percentile	649.026
99th Percentile	857.375

Table A.53: Random Read after BulkClone with LIFO background instantiation – Statistics for the timing of random Read of the destination objects after a BulkClone while bulk operations are being instantiated in the background via LIFO processing.

Data points	4000
Minimum value (ms)	4.794
Maximum value (ms)	124.237
Mean (ms)	16.408
Standard deviation (ms)	22.404
1st Percentile	5.068
5th Percentile	5.192
10th Percentile	5.273
25th Percentile	5.396
50th Percentile	5.532
75th Percentile	6.484
90th Percentile	40.001
95th Percentile	56.054
99th Percentile	104.829

Table A.54: Sequential Read after BulkClone with LIFO background instantiation – Statistics for the timing of sequential Read of the destination objects after a BulkClone while bulk operations are being instantiated in the background via LIFO processing.

Data points	2000
Minimum value (ms)	4.757
Maximum value (ms)	255.401
Mean (ms)	36.646
Standard deviation (ms)	51.484
1st Percentile	5.065
5th Percentile	5.224
10th Percentile	5.311
25th Percentile	5.451
50th Percentile	6.231
75th Percentile	41.744
90th Percentile	52.677
95th Percentile	181.387
99th Percentile	226.983

Table A.55: Random Create after BulkDelete with background instantiation via LIFO processing – Statistics for the timing of random Create of objects after a BulkDelete while bulk operations are being instantiated in the background via LIFO processing.

Data points	1000
Minimum value (ms)	5.582
Maximum value (ms)	181.531
Mean (ms)	20.865
Standard deviation (ms)	26.851
1st Percentile	5.715
5th Percentile	6.005
10th Percentile	6.430
25th Percentile	7.607
50th Percentile	8.186
75th Percentile	8.809
90th Percentile	66.626
95th Percentile	76.761
99th Percentile	102.103

Table A.56: Sequential Create after BulkDelete with background instantiation via LIFO processing – Statistics for the timing of sequential Create of objects after a BulkDelete while bulk operations are being instantiated in the background via LIFO processing.

Data points	1000
Minimum value (ms)	4.460
Maximum value (ms)	1908.945
Mean (ms)	178.780
Standard deviation (ms)	385.890
1st Percentile	4.655
5th Percentile	4.706
10th Percentile	4.761
25th Percentile	7.382
50th Percentile	8.878
75th Percentile	9.344
90th Percentile	881.167
95th Percentile	1120.506
99th Percentile	1467.446

Table A.57: Random Read after BulkClone with background instantiation of widest range – Statistics for the timing of random Read of the destination objects after a BulkClone while bulk operations are being instantiated in the background by processing the widest range.

Data points	4000
Minimum value (ms)	4.620
Maximum value (ms)	151.933
Mean (ms)	16.643
Standard deviation (ms)	23.735
1st Percentile	5.045
5th Percentile	5.205
10th Percentile	5.288
25th Percentile	5.408
50th Percentile	5.539
75th Percentile	6.177
90th Percentile	38.678
95th Percentile	83.965
99th Percentile	109.100

Table A.58: Sequential Read after BulkClone with background instantiation of widest range – Statistics for the timing of sequential Read of the destination objects after a BulkClone while bulk operations are being instantiated in the background by processing the widest range.

Data points	2000
Minimum value (ms)	4.948
Maximum value (ms)	196.323
Mean (ms)	34.322
Standard deviation (ms)	45.163
1st Percentile	5.102
5th Percentile	5.244
10th Percentile	5.341
25th Percentile	5.449
50th Percentile	6.679
75th Percentile	38.484
90th Percentile	119.235
95th Percentile	151.997
99th Percentile	185.686

Table A.59: Random Create after BulkDelete with background instantiation via widest range processing – Statistics for the timing of random Create of objects after a BulkDelete while bulk operations are being instantiated in the background via widest range processing.

Data points	1000
Minimum value (ms)	5.379
Maximum value (ms)	289.039
Mean (ms)	25.922
Standard deviation (ms)	38.819
1st Percentile	5.536
5th Percentile	5.937
10th Percentile	6.227
25th Percentile	7.509
50th Percentile	8.039
75th Percentile	8.597
90th Percentile	87.893
95th Percentile	107.305
99th Percentile	145.506

Table A.60: Sequential Create after BulkDelete with background instantiation via widest range processing – Statistics for the timing of sequential Create of objects after a BulkDelete while bulk operations are being instantiated in the background via widest range processing.

Data points	1000
Minimum value (ms)	4.454
Maximum value (ms)	1506.501
Mean (ms)	177.551
Standard deviation (ms)	336.435
1st Percentile	4.532
5th Percentile	4.650
10th Percentile	4.755
25th Percentile	6.501
50th Percentile	9.309
75th Percentile	9.996
90th Percentile	747.383
95th Percentile	952.015
99th Percentile	1218.787

Table A.61: Random Read after BulkClone with thinnest range background instantiation – Statistics for the timing of random Read of the destination objects after a BulkClone while bulk operations are being instantiated in the background by processing the thinnest range.

Data points	4000
Minimum value (ms)	4.574
Maximum value (ms)	141.372
Mean (ms)	16.401
Standard deviation (ms)	22.455
1st Percentile	5.059
5th Percentile	5.212
10th Percentile	5.290
25th Percentile	5.415
50th Percentile	5.547
75th Percentile	6.632
90th Percentile	39.233
95th Percentile	68.891
99th Percentile	101.813

Table A.62: Sequential Read after BulkClone with thinnest range background instantiation – Statistics for the timing of sequential Read of the destination objects after a BulkClone while bulk operations are being instantiated in the background by processing the thinnest range.

Data points	2000
Minimum value (ms)	4.792
Maximum value (ms)	211.549
Mean (ms)	34.431
Standard deviation (ms)	44.507
1st Percentile	5.048
5th Percentile	5.220
10th Percentile	5.302
25th Percentile	5.442
50th Percentile	6.947
75th Percentile	38.739
90th Percentile	117.097
95th Percentile	150.453
99th Percentile	182.570

Table A.63: Random Create after BulkDelete with background instantiation via thinnest range processing – Statistics for the timing of random Create of objects after a BulkDelete while bulk operations are being instantiated in the background via thinnest range processing.

Data points	1000
Minimum value (ms)	5.412
Maximum value (ms)	143.182
Mean (ms)	19.109
Standard deviation (ms)	23.778
1st Percentile	5.565
5th Percentile	5.897
10th Percentile	6.199
25th Percentile	7.465
50th Percentile	8.011
75th Percentile	8.609
90th Percentile	62.230
95th Percentile	72.959
99th Percentile	89.146

Table A.64: Sequential Create after BulkDelete with background instantiation via thinnest range processing – Statistics for the timing of sequential Create of objects after a BulkDelete while bulk operations are being instantiated in the background via thinnest range processing.

Data points	1000
Minimum value (ms)	4.494
Maximum value (ms)	1567.636
Mean (ms)	178.192
Standard deviation (ms)	333.035
1st Percentile	4.533
5th Percentile	4.634
10th Percentile	6.100
25th Percentile	8.359
50th Percentile	9.265
75th Percentile	9.970
90th Percentile	780.187
95th Percentile	901.647
99th Percentile	1171.904

A.5 NFS server baseline results

These tables describe data mentioned in Section 6.6.1 starting on page 135.

These tables constitute the raw data for the NFS server baseline experiments summary results reported in Table 6.5 on page 136. The experiment consisted of creating a number of files, cloning them, and then deleting them. A total of four experiments are run, no cloning, cloning via copying files, cloning in a prolific clone manner, and cloning in a chain-of-clones manner.

Table A.65: NFS File Remove – This table shows the NFS file remove benchmark in a single directory. The specified number of just-created and empty files are removed with the `rm` command on a newly started NFS server instance for each experiment run.

NFS file remove via `rm`, times in H:M:s.ms			
Run	File set size		
	100	1000	10000
1	5.267	57.488	10:12.546
2	4.635	57.390	10:10.695
3	6.316	57.460	10:08.381
4	6.399	57.383	10:08.453
5	4.643	50.590	10:09.152
6	4.519	57.345	10:07.810
7	4.985	57.567	10:07.736
8	3.014	57.131	10:09.138
9	6.540	57.408	10:12.602
10	4.636	57.357	10:09.284
11	5.376	57.548	10:04.189
12	6.425	57.486	10:09.115
13	6.423	57.509	10:08.503
14	4.512	51.834	10:08.723
15	6.417	57.433	10:08.376
16	6.437	57.444	10:05.905
17	4.955	57.473	10:12.404
18	4.658	57.424	10:10.081
19	4.601	57.453	10:10.653
20	6.522	57.483	10:02.818
Total time	1:47.290	18:56.217	3:22:56.573
Average time	5.364	56.810	10:08.828
Single op time	0.053	0.056	0.060

Table A.66: NFS File clone via copy – This table shows the timing of clone creation via copying of files from a single directory to another.

NFS file Clone via copy with `cp`, times in H:M:s.ms			
Run	File set size		
	100	1000	10000
1	14.459	1:40.725	22:03.407
2	7.159	2:30.278	23:22.949
3	7.156	1:36.427	24:18.250
4	14.549	1:46.592	23:12.357
5	7.175	2:16.344	22:17.244
6	4.402	1:34.835	23:46.964
7	14.563	1:27.480	23:19.519
8	14.463	2:30.581	22:08.980
9	14.488	1:44.167	21:30.190
10	14.550	1:24.037	24:40.933
11	7.119	1:26.997	23:18.059
12	14.427	2:12.145	22:49.132
13	14.555	1:20.834	22:02.024
14	14.446	1:39.865	21:56.554
15	14.413	1:37.295	22:06.873
16	7.114	1:37.060	21:35.044
17	14.587	2:06.247	20:08.015
18	7.143	1:26.306	22:22.279
19	14.522	1:44.376	23:52.604
20	14.498	1:37.239	22:54.412
Total time	3:55.798	35:19.839	7:33:45.798
Average time	11.789	1:45.991	22:41.289
Single op time	0.117	0.105	0.136

Table A.67: NFS File Remove after clone via copy – This table shows the NFS file remove benchmark in a single directory. The specified number of just-created and empty files are removed with the `rm` command on a newly started NFS server instance for each experiment run. Before this stage, the file system is cloned via the `cp` command. The data is comparable to that shown in Table A.65 on page 222.

NFS file remove via `rm` after Clone by `cp`, times in H:M:s.ms			
Run	File set size		
	100	1000	10000
1	6.452	58.849	10:06.449
2	4.560	59.110	10:11.030
3	5.291	58.365	10:13.007
4	6.461	58.978	10:08.706
5	5.049	58.643	10:08.715
6	4.637	58.315	10:06.707
7	6.440	58.531	9:43.370
8	6.488	58.906	10:08.193
9	6.449	58.715	10:09.362
10	6.452	1:00.892	10:07.490
11	4.370	58.562	10:04.996
12	6.453	57.739	10:06.777
13	6.426	58.528	10:06.550
14	6.478	58.828	10:06.187
15	6.463	58.002	10:04.541
16	4.390	58.212	10:06.669
17	6.441	57.961	10:09.219
18	4.952	58.821	10:08.488
19	6.425	58.642	10:10.568
20	6.537	58.252	10:03.699
Total time	1:57.223	19:32.860	3:22:10.734
Average time	5.861	58.643	10:06.536
Single op time	0.058	0.058	0.060

Table A.68: NFS prolific BulkClone – This table shows experimental NFS prolific BulkClone times.

NFS server prolific BulkClone between Create and Delete, times in H:M:s.ms			
Run	File set size		
	100	1000	10000
1	0.449	1.486	2.766
2	0.457	1.064	2.923
3	0.447	1.546	3.428
4	0.948	0.425	2.942
5	0.990	1.549	2.820
6	0.449	1.589	2.917
7	0.955	1.556	2.719
8	0.951	1.253	3.403
9	0.451	1.557	3.013
10	0.953	1.550	3.082
11	0.955	1.563	3.353
12	0.449	1.076	2.760
13	0.451	1.249	3.437
14	0.955	1.561	2.691
15	0.449	1.539	3.263
16	0.956	1.574	2.800
Total time	11.273	22.145	48.325
Average time	0.704	1.384	3.020

Table A.69: NFS File Remove after prolific BulkClone – This table shows the NFS file remove benchmark in a single directory. The specified number of just-created and empty files are removed with the `rm` command on a newly started NFS server instance for each experiment run. Before this stage, the file system is cloned. The data is comparable to that shown in Table A.65 on page 222.

NFS file remove via `rm` after prolific BulkClone, times in H:M:s.ms			
Run	File set size		
	100	1000	10000
1	7.787	1:50.010	43:08.985
2	15.233	1:52.349	44:02.127
3	7.793	1:59.779	43:59.248
4	15.542	1:50.384	44:17.423
5	15.712	1:52.004	43:21.120
6	7.800	1:52.025	43:11.894
7	7.661	1:52.849	42:45.765
8	15.517	1:51.981	43:46.032
9	7.880	1:49.508	43:27.969
10	15.584	1:54.400	43:22.590
11	15.506	1:49.931	44:27.330
12	7.800	1:50.134	43:29.480
13	7.785	1:52.057	43:14.073
14	15.616	1:51.664	43:46.113
15	7.798	1:52.903	44:14.815
16	7.354	1:50.026	43:36.655
Total time	2:58.373	29:52.010	11:38:11.627
Average time	11.148	1:52.000	43:38.226
Single op time	0.111	0.112	0.261

Table A.70: NFS chain-of-clones BulkClone – This table shows the experimental NFS chain-of-clones BulkClone times.

NFS server prolific BulkClone between Create and Delete, times in H:M:s.ms			
Run	File set size		
	100	1000	10000
1	1.022	1.867	15.581
2	0.994	1.063	16.774
3	0.989	1.747	17.080
4	0.464	1.325	17.316
5	0.994	1.279	16.630
6	0.991	1.426	16.807
7	0.993	2.741	16.596
8	0.991	1.993	16.764
9	0.984	1.062	16.782
10	0.992	1.811	16.962
11	0.992	1.313	16.866
12	1.008	1.419	16.944
13	0.987	1.796	16.966
14	0.921	1.831	16.443
15	0.990	1.436	17.241
16	0.465	1.591	16.439
Total time	14.784	25.706	4:28.198
Average time	0.924	1.606	16.762

Table A.71: NFS File Remove after chain-of-clones BulkClone – This table shows the NFS file remove benchmark in a single directory. The specified number of just-created and empty files are removed with the `rm` command on a newly started NFS server instance for each experiment run. Before this stage, the file system is cloned in a chain-of-clones manner. The data is comparable to that shown in Table A.65 on page 222.

NFS file remove via `rm` after prolific BulkClone, times in H:M:s.ms			
Run	File set size		
	100	1000	10000
1	15.908	2:35.443	4:59:01.326
2	15.635	2:44.199	4:57:14.419
3	15.515	2:27.073	5:00:53.713
4	7.728	2:34.290	5:00:23.927
5	15.511	2:39.637	4:59:29.850
6	15.498	2:27.902	4:59:20.120
7	6.729	2:30.035	4:55:57.209
8	15.498	2:29.133	4:58:25.275
9	15.510	2:36.112	4:59:58.011
10	15.527	2:31.286	4:59:46.801
11	15.528	2:26.736	4:56:54.610
12	15.500	2:24.880	4:59:53.344
13	15.523	2:26.916	5:00:55.247
14	15.493	2:25.490	4:57:48.048
15	15.490	2:46.108	4:59:18.728
16	7.748	2:36.418	5:00:35.902
Total time	3:44.349	40:41.665	3 days 7:45:56.538
Average time	14.021	2:32.604	4:59:07.283
Single op time	0.140	0.152	1.794

A.6 PostMark experimental results

These tables constitute the raw data for the PostMark benchmark results reported in Table 6.6 on page 139

Table A.72: PostMark with no bulk operations – Runs of the PostMark benchmark with default settings and no BulkClone operations.

Postmark, plain				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	121	53	11.55	37.63
2	125	57	11.18	36.42
3	108	38	12.94	42.16
4	145	59	9.64	31.40
5	180	63	7.76	25.29
6	180	63	7.76	25.29
7	125	59	11.18	36.42
8	180	63	7.76	25.29
9	180	63	7.76	25.29
10	180	62	7.76	25.29
11	170	62	8.22	26.78
12	180	62	7.76	25.29
13	180	62	7.76	25.29
14	180	62	7.76	25.29
15	107	37	13.06	42.55
Mean	156.07	57.67	9.32	30.38
Median	180.00	62.00	7.76	25.29
Standard Deviation	30.21	8.65	2.07	6.73

Table A.73: PostMark with Clone via Copy – Runs of the PostMark benchmark with default settings and a clone of active file data made every 10% through the transaction phase (i.e., at 0%, 10% ... 90%) through the use of the `cp` command.

Postmark, Copy as Clone				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	683	620	2.05	6.67
2	783	696	1.78	5.81
3	737	643	1.90	6.18
4	703	605	1.99	6.48
5	631	569	2.21	7.22
6	654	590	2.14	6.96
7	631	566	2.21	7.22
8	889	791	1.57	5.12
9	702	607	1.99	6.49
10	775	653	1.80	5.87
Mean	718.80	634.00	1.96	6.40
Median	702.50	613.50	1.99	6.49
Standard Deviation	80.28	67.89	0.21	0.67

Table A.74: PostMark with prolific clones – Runs of the PostMark benchmark with default settings and a prolific BulkClone made every 10% through the transaction phase (i.e., at 0%, 10% . . . 90%).

Postmark, prolific clones				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	389	191	3.59	11.70
2	273	135	5.12	16.68
3	279	138	5.01	16.32
4	281	138	4.97	16.20
5	274	138	5.10	16.62
6	275	135	5.08	16.56
7	363	165	3.85	12.54
8	390	199	3.58	11.67
9	277	136	5.04	16.44
10	338	140	4.13	13.47
11	386	190	3.62	11.79
12	275	137	5.08	16.56
13	348	146	4.02	13.08
14	285	140	4.90	15.97
15	278	137	5.03	16.38
16	335	132	4.17	13.59
Mean	315.38	149.81	4.52	14.72
Median	283.00	138.00	4.94	16.09
Standard Deviation	47.10	22.89	0.63	2.06

Table A.75: PostMark with chains-of-clones – Runs of the PostMark benchmark with default settings and a chain-of-clones BulkClone made every 10% through the transaction phase (i.e., at 0%, 10% … 90%).

Postmark, chain-of-clones				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	265	178	5.27	17.18
2	266	184	5.25	17.12
3	327	182	4.27	13.92
4	272	184	5.14	16.74
5	266	184	5.25	17.12
6	325	183	4.30	14.01
7	258	173	5.42	17.65
8	262	177	5.33	17.38
9	320	175	4.37	14.23
10	336	193	4.16	13.55
11	327	183	4.27	13.92
12	313	173	4.46	14.55
13	321	176	4.35	14.18
14	271	190	5.16	16.80
15	264	178	5.29	17.25
16	264	178	5.29	17.25
17	325	184	4.30	14.01
18	260	177	5.37	17.51
19	331	188	4.22	13.75
20	355	191	3.94	12.82
21	285	180	4.90	15.97
22	267	179	5.23	17.05
23	261	178	5.35	17.44
24	282	183	4.95	16.14
Mean	292.62	181.29	4.83	15.73
Median	277.00	181.00	5.04	16.44
Standard Deviation	31.91	5.45	0.51	1.65

Table A.76: PostMark with prolific clones and random background instantiation – Runs of the PostMark benchmark with default settings and a prolific BulkClone made every 10% through the transaction phase (i.e., at 0%, 10% ... 90%). Random background instantiation was active during the experiment.

Postmark, prolific clones, random background instantiation				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	337	263	4.15	13.51
2	388	256	3.60	11.73
3	315	241	4.44	14.45
4	385	252	3.63	11.83
5	333	250	4.20	13.67
6	318	244	4.39	14.32
7	332	258	4.21	13.71
8	329	258	4.25	13.84
9	375	243	3.73	12.14
10	376	243	3.72	12.11
Mean	348.80	250.80	4.03	13.13
Median	335.00	251.00	4.18	13.59
Standard Deviation	28.73	7.79	0.33	1.06

Table A.77: PostMark with prolific clones and LIFO background instantiation – Runs of the PostMark benchmark with default settings and a prolific BulkClone made every 10% through the transaction phase (i.e., at 0%, 10% . . . 90%). LIFO background instantiation was active during the experiment.

Postmark, prolific clones, LIFO background instantiation				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	320	247	4.37	14.23
2	296	222	4.72	15.38
3	372	237	3.76	12.24
4	334	259	4.18	13.63
5	300	229	4.66	15.18
6	376	242	3.72	12.11
7	366	236	3.82	12.44
8	297	220	4.70	15.33
9	313	237	4.46	14.55
10	364	216	3.84	12.51
Mean	333.80	234.50	4.22	13.76
Median	327.00	236.50	4.28	13.93
Standard Deviation	32.89	13.19	0.41	1.35

Table A.78: PostMark with prolific clones and thinnest range background instantiation – Runs of the PostMark benchmark with default settings and a prolific BulkClone made every 10% through the transaction phase (i.e., at 0%, 10% . . . 90%). Background instantiation of the thinnest range was active during the experiment.

Postmark, prolific clones, thinnest range background instantiation				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	296	226	4.72	15.38
2	300	226	4.66	15.18
3	328	210	4.26	13.88
4	355	224	3.94	12.82
5	361	228	3.87	12.61
6	300	227	4.66	15.18
7	304	231	4.60	14.98
8	299	227	4.67	15.23
9	349	216	4.00	13.05
10	290	214	4.82	15.70
Mean	318.20	222.90	4.42	14.40
Median	302.00	226.00	4.63	15.08
Standard Deviation	27.39	6.98	0.36	1.19

Table A.79: PostMark with chains-of-clones and random background instantiation – Runs of the PostMark benchmark with default settings and a chain-of-clones BulkClone made every 10% through the transaction phase (i.e., at 0%, 10% … 90%). Random background instantiation was active during the experiment.

Postmark, chain-of-clones, random background instantiation				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	424	279	3.30	10.74
2	395	261	3.54	11.53
3	347	275	4.03	13.12
4	432	299	3.23	10.54
5	377	302	3.71	12.08
6	426	293	3.28	10.69
7	431	299	3.24	10.56
8	382	308	3.66	11.92
9	420	290	3.33	10.84
10	383	250	3.65	11.89
Mean	401.70	285.60	3.50	11.39
Median	407.50	291.50	3.44	11.18
Standard Deviation	29.03	18.95	0.27	0.86

Table A.80: PostMark with chains-of-clones and LIFO background instantiation – Runs of the PostMark benchmark with default settings and a chain-of-clones BulkClone made every 10% through the transaction phase (i.e., at 0%, 10% . . . 90%). LIFO background instantiation was active during the experiment.

Postmark, chain-of-clones, LIFO background instantiation				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	426	294	3.28	10.69
2	401	329	3.48	11.35
3	428	296	3.26	10.64
4	430	298	3.25	10.59
5	379	307	3.69	12.01
6	429	298	3.26	10.61
7	427	294	3.27	10.66
8	394	262	3.55	11.56
9	393	320	3.56	11.58
10	389	313	3.59	11.70
Mean	409.60	301.10	3.42	11.14
Median	413.50	298.00	3.38	11.02
Standard Deviation	20.16	18.22	0.17	0.55

Table A.81: PostMark with chains-of-clones and thinnest range background instantiation – Runs of the PostMark benchmark with default settings and a chain-of-clones BulkClone made every 10% through the transaction phase (i.e., at 0%, 10% . . . 90%). Background instantiation of the thinnest range was active during the experiment.

Postmark, chain-of-clones, thinnest range background instantiation				
Run	Total Time (s)	Transaction Time (s)	Read (kB/s)	Write (kB/s)
1	379	310	3.69	12.01
2	430	297	3.25	10.59
3	419	289	3.33	10.87
4	352	280	3.97	12.93
5	369	295	3.79	12.34
6	352	281	3.97	12.93
7	359	285	3.89	12.68
8	361	285	3.87	12.61
9	348	277	4.02	13.08
10	432	300	3.23	10.54
Mean	380.10	289.90	3.70	12.06
Median	365.00	287.00	3.83	12.47
Standard Deviation	33.73	10.41	0.31	1.01