Communicating Quality of Service Requirements to an Object-Based Storage Device

Kevin KleinOsowski
Department of Electrical and Computer Engineering
Univ. of MN, Minneapolis
kevinko@ece.umn.edu

Tom Ruwart
Digital Technology Center
Univ. of MN, Minneapolis
tmruwart@dtc.umn.edu

David J. Lilja
Department of Electrical and Computer Engineering
Univ. of MN, Minneapolis
lilja@ece.umn.edu

Abstract

Obtaining consistent bandwidth with predictable latency from disk-based storage systems has proven difficult due to the storage system's inability to understand Quality of Service (QoS) requirements. In this paper, we present a feasibility study of QoS with the Object-based Storage Device (OSD) specification. We look at OSD's ability to provide QoS guarantees for consistent bandwidth with predictable latency. Included in this paper is a description of QoS requirements of a sample application and how these requirements are translated into parameters that are then communicated to, and interpreted by, the OSD. Implementation problems lead to the failure of a hard real-time QoS model, but this failure is not due to the OSD protocol. The paper concludes with a description of how well the Revision 9 OSD standard (OSDR9) is able to accommodate QoS. We provide suggestions for improving the OSD specification and its ability to communicate QoS requirements.

1. Introduction

The Object-based Storage Device (OSD) protocol is an extension of the Small Computer System Interface (SCSI) command protocol. The OSD protocol is intended for storing data in variable length objects rather than fixed length blocks. Furthermore, objects may have arbitrary attributes associated with them, whereas traditional block-based storage does not have any attributes. Object-based storage devices represent parts of files as data objects with attributes. These attributes help describe the object (objects and files may not correspond one-to-one). Because data is stored as an object, more details about the data, such as QoS requirements, may be stored with it. Information may be communicated to the OSD explicitly or implicitly and this information can be used to provide QoS information for I/O operations. This type of functionality goes beyond the limits of block-based disk storage systems where only the bytes of file data are stored and a richer attribute mechanism to describe the data is missing. Since current storage systems store files only in this simple form, there exists no ability to understand QoS attributes. As disk-based storage systems become larger and more highly interconnected with a multi-user/multi-application environment (e.g., Storage Area Networks), the best effort response mechanism used by traditional block-based disk systems will no longer be sufficient for applications that require some level of QoS. This shift toward storage area networks makes OSDs more attractive since OSDs may provide predictable I/O to applications. In this work, we take the existing OSD reference implementation created by Intel Research and add to it the mandatory commands defined in the OSDR9 [8]. We further extend its capabilities by adding the OSD attribute mechanism that was defined in the OSDR9. We then use this enhanced reference implementation to demonstrate how the attribute mechanism may be used to communicate QoS attributes to an OSD target. Once these QoS attributes are communicated from initiator to OSD target, the target may then interpret the attributes to provide a soft real-time QoS.

2. Background and Motivation

In May 2004 the Storage Networking Industry Association (SNIA) released the OSD protocol specification Revision 9 [8]. This protocol defines the communication between SCSI initiators and SCSI OSD targets to promote interoperability. With this specification, Intel Research developed a reference implementation [5] to foster experimentation and testing of Internet Small Computer Systems Interface (iSCSI) and OSD. The reference implementa-

Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005)
0-7695-2318-8/05 $20.00 © 2005 **IEEE**

tion contains basic OSD functionality along with the iSCSI communication mechanism.

With the OSD protocol to guide us, we look at the details needed for QoS. QoS may be summarized as the ability to offer and guarantee an individual requirement or attribute. Techniques for QoS in network communications [7] can be applied to OSD. Either Integrated Services (IntServ) or Differentiated Services (DiffServ) may be used to provide a framework for QoS. The major difference between these two frameworks for QoS lie in how each performs quality guarantees. IntServ performs quality guarantees with reservations from end-to-end, thus points along the communication channel understand the IntServ protocol. DiffServ aggregates service into classes which receive a provision of the resource rather than a reserved amount and only endpoints determine how to handle the classes. Applying networking communications QoS techniques to OSD is possible since both networking communications and disk storage possess requirements that may need to be guaranteed. These requirements may be distilled into attributes specific to the application involved. In audio-video applications this may lead to at least nine different attributes [3], such as frame size, frame rate or image clarity.

Classifying disk requests on disk-based storage systems to guarantee QoS has been studied by Wijayaratne and Reddy [9]. The desired outcome is consistent bandwidth with predictable latency. Historically, block-based storage systems do not perform well in providing consistent bandwidth with predictable latency due to the inability to communicate QoS requirements. This inability to communicate the QoS requirements to the storage system has led to many different approaches to scheduling disk requests [1, 2, 3, 4, 6]. These scheduling mechanisms are done in front of the disk system since scheduling attributes are not understood by the disk system. Unfortunately, it is increasingly challenging to perform external scheduling because internally the disk drives themselves may reorder block read requests thwarting attempts to schedule these requests. A coarse-grained approach to shaping best-effort requests that have no quality guarantees may work around the disk's internal reordering of I/O requests [10].

This coarse-grained approach shows promise in a mixed workload environment where best effort requests are throttled in favor of soft real-time requests for disk I/O operations. However, a mechanism with more precise control is needed to smooth out startup spikes in data rates. The token bucket filter used in the coarse-grained approach can be applied to QoS in OSDs. Admission control with applications that may function on reduced resources [6] proves to be effective in offering QoS.

Over-provisioning a storage system presents a simple solution for providing QoS on OSDs. However, this naive solution underutilizes the storage system and increases hardware cost. Even though it is a simple solution, over-provisioning will not guarantee real-time QoS and my not guarantee soft real-time QoS either.

Table 1. Mandatory OSD Protocol Revision 9 Commands

Command Name	Added/Updated
APPEND	Added
CREATE	Updated
CREATE WRITE	Added
CREATE PARTITION	Updated
FLUSH OBJECT	Added
FORMAT OSD	Added
GET ATTRIBUTES	Updated
LIST	Added
PERFORM SCSI COMMAND	Added
PERFORM TASK MGMT FUNCTION	Added
READ	Updated
REMOVE	Updated
REMOVE PARTITION	Updated
SET ATTRIBUTES	Updated
SET KEY	Added
SET MASTER KEY	Added
WRITE	Updated

IntServ requires QoS attributes to be communicated end-to-end. This then sets up a reservation for required attributes. In an OSD, this reservation is accomplished with the attribute mechanism defined by the OSDR9. Admission control needed for IntServ can be determined by the OSD target given that it knows the experimentally demonstrated disk system and communication limits.

DiffServ QoS requires 1) classification, 2) traffic shaping, and 3) monitoring of requests. Within the OSD specification, a *class of request* can be communicated using an attribute for bandwidth requirement. Traffic shaping and monitoring can be conducted inside the OSD target. Traffic shaping and monitoring affect the response times of requests and are not part of the OSD protocol.

As we show in Section 4, once specific requirements are communicated to the OSD using OSD attributes, the necessary data rates and deadlines can be met in a soft real-time setting. Our implementation of the OSD protocol specification with a target and an initiator demonstrates how OSD attributes can be used in communicating QoS, thus allowing the OSD target to provision the available resources to match the incoming I/O requests. This implementation communicates the attributes needed to provide QoS for a single initiator or multiple initiators connected to the same OSD target.

3. Methodology

In order to demonstrate how OSDR9 can be used to communicate QoS requirements, we first extend the capabilities of the v20 reference implementation developed by Intel Research [5]. With the new functionality added to the reference implementation, we have a toolkit to assist in developing an application framework for communicating, interpreting, and enforcing QoS attributes. We then define the attributes we use to describe QoS. We investigate the OSD target and what it needs to enforce QoS. Finally, we use the Linux operating system, kernel 2.4.20, for testing the negotiated aspects of QoS.

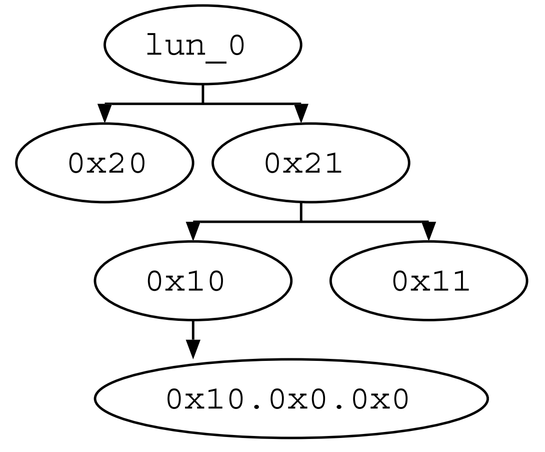
3.1. Additions to the Reference Implementation

When using the Linux operating system and its file system, the reference implementation provides the simulation environment to test an OSD. With the reference implementation we develop an OSD target that portrays objects with files using Linux file system routines. Therefore, when an object is created, a file or files on a Linux file system are created. Attributes that relate to the object are simulated by additional files that follow a similar naming scheme which helps organize the attributes for lookups. Therefore, the front side of the OSD target speaks the OSD protocol at a basic level. The back side translates these basic object routines into standard Linux file system function calls such as *open()*, *read()* and *write()*.

The Intel reference implementation was chosen due to its progress on the OSD protocol framework. The basic implementation of iSCSI OSD functionality in the reference implementation was designed for optimal performance. There are few buffer copies and macros are used to eliminate the need for some function calls. Most significantly, the code is organized well enough to easily add new commands to current functionality.

Table 1 displays the OSD commands added to this reference implementation [8]. This table is a full list of mandatory OSD commands dictated by the OSDR9. Because these OSD commands are designated mandatory by the OSDR9 protocol specification, they are implemented. These commands are useful for testing the functionality of the OSD. In the OSD specification, objects and partitions are represented with a corresponding numerical Object ID and Partition ID respectively. Objects and attributes are simulated as files associated with a similar name. Partitions are simulated with a directory given the name of the Partition ID. The Logical Unit Number (LUN) makes up the topmost directory name for the OSD. This format is illustrated in Figure 1.

LUN	0
Partition ID	0x21
Object ID	0x10



data filename: /lun_0/0x21/0x10

attribute filename: /lun_0/0x21/0x10.0x0.0x0

Figure 1. Example OSD Back-end Files

Storing object attributes, specifically vendor attributes used to communicate QoS, is new to the reference implementation. OSDR9 provides details about storing, processing, and accessing these OSD attributes. All attributes have a Page number and Attribute number associated with them. Some attributes stored with the object are defined by OSD with specific functionality. Other attributes defined by OSD are not given specific functional use. Rather, these attributes, which are known as *Vendor* attributes, are stored on disk or in memory, or are used in other ways. One other way to use the Vendor attributes is to describe the different communication parameters required by the initiator for QoS.

Illustrated in Figure 1 is attribute 0 on page 0 for the object with Object ID 0x10. This is the *Page Identification* attribute which is used internally by the OSD. The OSD assigns this information to the object's In Page Identification. This attribute is a statically stored value that may be read, but not modified by, an initiator. This functionality is required by OSDR9.

Attributes that are used by the OSD running environment are valuable and will be used to communicate QoS attributes. These attributes have little use when stored since their meaning is specific to the current session only. This attribute mechanism is missing from block-based disk storage system and proves to be critical for making communication of QoS-specific information possible.

With an attribute mechanism in place, ordering operations that read and modify the attributes is critical. The getting and setting of attributes can accompany other commands, such as a CREATE object command, or FORMAT OSD command. The GET and SET of attributes may also be done as a command by itself, rather than an additional function of a command. The order of processing these “dual” commands is important as a GET of an attribute before the CREATE command would return an error because the object was not yet created. This example demonstrates the possibility of a CREATE command having an attached GET attribute functionality. The SET operation associated with a READ operation is defined to behave differently than we need. We discuss a potential change in this processing in Section 4.

The GET and a SET operations may be performed at the same time. This is done with a GET ATTRIBUTE command and a SET attribute option. The correct functionality is to GET the requested attributes before the associated SET command is completed. This behavior is different compared to most commands in the specification that have all SET operations completed first, and the GET request performed last.

OSDR9 protocol specification covers security mechanisms for OSD commands and describes checksums for the Command Descriptor Block (CDB). The CDB is the structure used to communicate command information between the initiator and the OSD target. Neither the security mechanism nor the checksum scheme are implemented for this work. Neither are deemed necessary since, in the simple OSD client server model, there are only point-to-point links. These links are secure enough to disregard the security mechanism. The checksum mechanism is likewise skipped since TCP network communications provides data correction mechanisms at lower levels.

With the enhanced reference implementation, we create a test application called *1cmd* is determined by the initiator. This attribute is fed back to the OSD target before the next read command. With the round-trip attribute, the OSD target is informed of the time it took to perform the operation from the initiator's view. The OSD target has calculated the desired response time, and may compare the desired response time to this actual round-trip attribute to then adjust its next actions accordingly.

The following example clarifies the attribute communication. We desire a 22 megabyte per second data link for our application, or more precisely 22,000,000 bytes per second. One second is a long time, and we will not read all this data in one buffer, so it must be broken up into multiple reads, with a buffer size of 22,000 bytes. That would imply we need to produce one thousand read requests from the initiator to the target in a one second time interval, or request a read every one thousandth of a second. For this example the buffer size and bandwidth requirement are arbitrary. However applications have different requirements and memory limits, so these attributes are important and must be communicated to the OSD target before any data is transfered.

In the example given, both the initiator and the OSD target become aware of the bandwidth required and they would both know how often to expect a read request to occur. The OSD target can use this information to prevent cheating from the initiator if it requested a response more often. The OSD target would be ready to service a request that came in late by caching the response. It is optional to acknowledge this late request and skip over it. This option may be communicated as another attribute, but we have not yet modeled it. We instead work with the premise that late is better than not at all. The clock for the next read is adjusted to compensate for this late read. All following reads will be late with respect to the original time since the request-response model of OSD restricts any sooner delivery to the initiator.

3.3. QoS in the OSD Target

With test applications ready to test the initiator side of the experiment, we now develop an OSD target application. The original reference implementation target application is used and enhanced with OSD attribute capabilities. This enhanced initiator is implemented as a Linux kernel module. As a kernel module, the enhanced OSD target runs with full privileges under Linux. This full privilege level gives the best response time and access to operating system resources. Each new initiator connects to a new thread in the OSD target. This is done to sidestep bottlenecks within the code of the OSD target.

To provide a simple DiffServ QoS, we use a token bucket filter. The token bucket filter provides a means to limit the best effort requests to the OSD target thereby providing timely responses for the QoS requests. Best effort requests are distinguished from QoS requests by their lack of QoS attributes. Best effort requests are not guaranteed any type of service, but are guaranteed not to starve with the token bucket filter. Our token bucket filter represented in Figure 3 operates similar to that proposed by Wu and Brandt [10]. The missed deadline notification that is used to tune the filter described by Wu and Brandt, is accomplished in our model by using the round-tip time delay and comparing it with the expected response time. If it turns out to be longer, the deadline is missed. If there is a delayed response on one operation, it must signal an improved response on the following operation with all other variables remaining the same. The token bucket filter provides at best a soft real-time QoS.

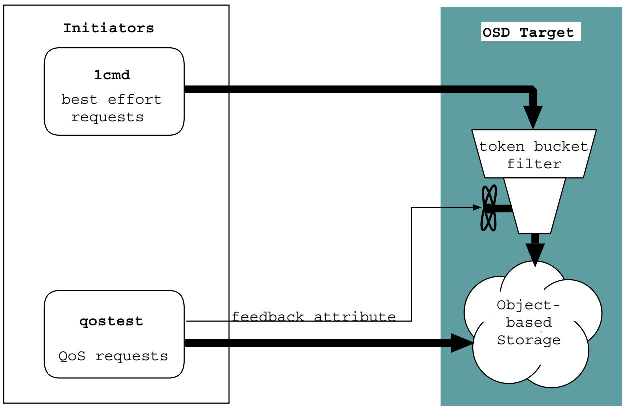


Figure 3. Simulation Environment

In order to provide a hard real-time QoS, a close watch on time must be done. Responses to requests have deadlines associated with them. Failure to meet these deadlines may either result in an acknowledgement of the request with no data, or a delayed response. This type of failure recovery may be signaled with another attribute, but this functionality is not modeled with our implementation. IntServ QoS model is a good choice for hard real-time QoS since the reservation of resources ahead of time helps determine if requests can be serviced with the associated deadline.

Hard real-time QoS on an OSD target is implemented with a Linux kernel module. The Linux operating system has a timer mechanism that allows the scheduling and execution of an event at an exact time in the future. Kernel timers in Linux run in interrupt time which, among other things, means they are polled about 100 times per second. This granularity is enough to test a lightly loaded OSD providing QoS.

To prepare the OSD target for a hard real-time QoS session, the initiator sets QoS attributes for bandwidth and buffer size. The response time is defined as:

$$respond = \frac{buffersize}{bandwidth}$$

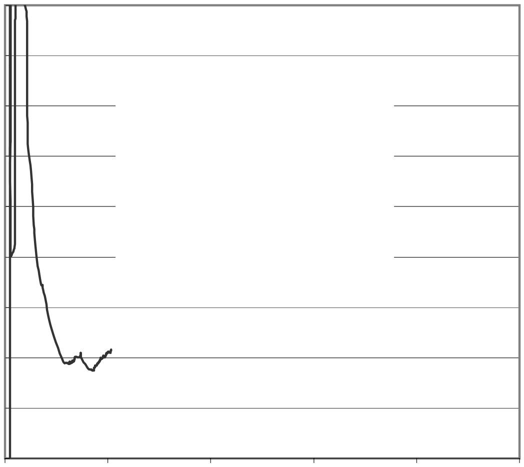
A timer within the OSD target is set to unblock the OSD target and respond at the future time. The OSD target then blocks. When the timer expires, the response is finished. Unfortunately this approach can fail due to limitations inherent in the Linux kernel. The unblocking of a request does not occur predictably. Deadlines are missed because of the unpredictable late periods. The attribute mechanism for communicating QoS is not the point-of-failure. Rather, it is the hard real-time implementation in Linux.

3.4. Simulation Environment

The experimental environment consists of a single computer with multiple initiators communicating with a single OSD target. The single computer model helps reduce or eliminate the affects of using networking QoS protocols. We are not focusing on the limits of the hardware in the testing environment. Those limitations may be looked at as infinite for our tests. The focus is directed on OSDR9 and its ability to communicate QoS attributes.

In addition to using a single computer for the testing environment, large files, over 300,000,000 bytes, are used with I/O operations. The reasoning is that smaller files would exploit the available buffering and so would yield suspiciously fast responses due to buffering in the Linux I/O subsystem. With larger files the buffering effect is amortized through the cycle of reading or writing the object. On our test system the average maximum attainable throughput from the disk system, obtained empirically, is 32,850,000 bytes per second for sequential reads.

Within our token bucket filter, best effort requests are not given any QoS. Their performance will suffer at the sake of QoS guaranteed requests. There is a built-in starvation prevention mechanism that ensures that at least some best effort requests will obtain service. Figure 3 displays a representation of the token bucket filter. There exists within the token bucket filter an ability for the best effort requests to regain additional service once the QoS requests have finished to ensure full utilization when no QoS requests are present. When an initiator sets the QoS attributes, all operations are now considered higher priority than best effort requests. However, between QoS requests there is no differentiation. Therefore the QoS requests are competing against themselves. The round robin scheduling effect from the process scheduling is the equalizing force between them.



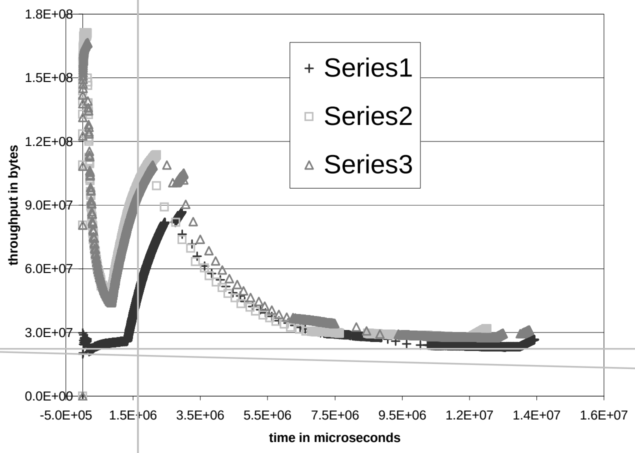
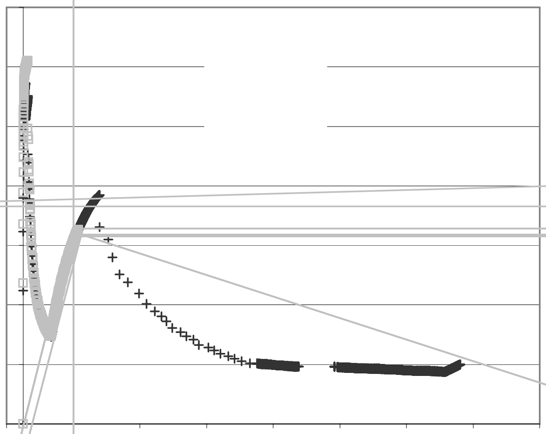


Figure 5. Throughput of best effort alone



the Digital Technology Center's Intelligent Storage Consortium (DISC).

References

[1] R. Abbott and H. Garcia-Molina. Scheduling I/O Requests with Deadlines: a Performance Evaluation. In *11th Real-Time Systems Symposium*, pages 113–124, 1990.

[2] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, June 1999.

[3] S. Chatterjee and M. Brown. Adaptive QoS Resource Management in Dynamic Environments. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 997–998, June 1999.

[4] M. de Miguel, J. Ruiz, and M. Garcia. QoS-Aware Component Frameworks. In *Tenth IEEE International Workshop on Quality of Service*, pages 161–169, May 2002.

[5] R. Intel Corporation. Intel iSCSI Reference Implementation. Details available at `http://www.intel.com/technology/computing/storage/iscsi/index.htm`.

[6] W. Lee and B. Sabata. Admission Control and QoS Negotiations for Soft-Real Time Applications. In *IEEE International Conference on Multimedia Computing and Systems*, volume 1, pages 147–152, June 1999.

[7] Z. Wang. *Internet QoS: Architectures and Mechanisms for Quality of Service*. Morgan Kaufmann Publishers, 2001.

[8] R. O. Webster. Information Technology - SCSI Object-Based Storage Device Commands (OSD). February 2004. Rev. 9.

[9] R. Wijayaratne and A. Reddy. Integrated QoS management for disk I/O. In *IEEE International Conference on Multimedia Computing and Systems*, volume 1, pages 487–492, June 1999.

[10] J. C. Wu and S. A. Brandt. Storage Access Support for Soft Real-Time Applications. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04), Toronto, Canada*, May 2004.

0-7695-2318-8/05 $20.00