Norwegian University of
Science and Technology

System Recovery in Large-Scale Distributed Storage Systems

Svein Aga

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Kjetil Nørvåg, IDI
Co-supervisor: Cyril Banino-Rokkones, Yahoo! Technologies Norway AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The research topic of this master thesis deals with large-scale distributed storage systems and how they handle and recover from a change in the system configuration. Such change occurs when one or more storage nodes are added or removed from the system. The primary goal is that clients should always expect a minimum Quality of Service (QoS) when utilizing the storage system even if the storage system is recovering data.

Although a performance drop-down is generally tolerated during a system recovery, the system should not perform worse than an agreed QoS. More precisely, a storage node will have two different queues of requests. The first queue contains production requests whereas the second queue contains maintenance requests. Both queues have to share available system resources. The pace of processing requests from either queues, can affect several factors in a large-scale storage system, and careful planning is required to avoid hot-spots or saturated nodes.

This project consists of the following: First, a literature survey of existing storage systems will be conducted, especially on how they recover from system changes. Second, designing a model of the system and creating a formal description of the problem. Third, a design of a new methodology to recover from a system change, and finally, simulation and comparison of the new methodology versus recovery methodology in other systems

Assignment given: 15. January 2008
Supervisor: Kjetil Nørvåg, IDI

Abstract

This report aims to describe and improve a system recovery process in large-scale storage systems. Inevitable, a recovery process results in the system being loaded with internal replication of data, and will extensively utilize several storage nodes. Such internal load can be categorized and generalized into a maintenance workload class.

Obviously, a storage system will have external clients which also introduce load into the system. This can be users altering their data, uploading new content, *etc.* Load generated by clients can be generalized into a production workload class.

When both workload classes are actively present in a system, *i.e.* the system is recovering while users are simultaneously accessing their data, there will be a competition of system resources between the different workload classes. The storage must ensure Quality of Service (QoS) for each workload class so that both are guaranteed system resources.

We have created Dynamic Tree with Observed Metrics (DTOM), an algorithm designed to gracefully throttle resources between multiple different workload classes. DTOM can be used to enforce and ensure QoS for the variety of workloads in a system. Experimental results demonstrate that DTOM outperforms another well-known scheduling algorithm.

In addition, we have designed a recovery model which aims to improve handling of critical maintenance workload. Although the model is intentionally intended for system recovery, it can also be applied to many other contexts.

Preface

This thesis was written by Svein Aga as part of a Master degree at the *Department of Computer and Information Science (IDI)*. IDI is part of the *Norwegian University of Science and Technology (NTNU)* in Trondheim, Norway. The work was carried out under supervision of Professor Kjetil Nørvåg and in collaboration with Yahoo! Technologies Norway (YTN).

I would first like to thank Professor Kjetil Nørvåg for our cooperation during the final year of my master education. His extensive knowledge and continuous support have been of great value.

I would also like to thank my supervisor at YTN, Cyril Banino-Rokkones, for his great help and invaluable knowledge during all phases of my thesis. I appreciate his guidance. Besides to my supervisor, I would also thank all employees at YTN for letting me be "part of the team" in both technological and social events.

Finally I would to thank all my friends and family for their moral support during all my years as a student.

Trondheim, 10. June 2008

Svein Aga

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goals and Scope . . . 1
1.3 Contributions . . . 2
1.4 Outline . . . 2

2 Problem Elaboration of System Recovery **3**
2.1 Overview . . . 3
2.2 Workload Classes . . . 6
2.3 Local and Global Level Quality of Service . . . 6
2.4 Sharing Surplus Bandwidth. . . . 10
2.5 Replication Requests - Maintenance or Production? . . . 10
2.6 Requirements . . . 12

3 Related work **14**
3.1 System Recovery in Distributed Storage Systems . . . 14
3.2 Aqua QoS Framework . . . 17
3.3 Bourbon QoS Framework . . . 18
3.4 Round Robin . . . 19
3.5 Weighted Round Robin . . . 20
3.6 Deficit Round Robin . . . 20
3.7 Hierarchical Token Bucket . . . 21

4 Local Level Quality of Service **22**
4.1 Generalized QoS . . . 22
4.2 Dynamic Tree with Observed Metrics . . . 24
4.3 Lower Bound on DTOM QoS . . . 36

5 Experimental Results **37**
5.1 Methodology . . . 37
5.2 Scenario 1: Throttling Disk I/O with uniform workload . . . 38
5.3 Scenario 2: Throttling Disk I/O with non-uniform workload . . . 41
5.4 Scenario 3: Addition and Removal of Workload Classes . . . 43
5.5 Requirement fulfillment . . . 44

6 Conclusion **45**
6.1 Contributions . . . 45
6.2 Future work . . . 46

A Vespa Document Storage **47**

A.1 Documents in VDS 47
A.2 VDS Architecture 48
A.3 Buckets 50
A.4 Recovery and cluster update 51

List of Figures

1 Storage nodes s_2, s_3 and s_4 initially holds a copy of a document (or object). If s_3 fails (dotted line), the object must be replicated to s_k (dashed line) from either s_2 or s_4 in order to stabilize the system. . . 4
2 A (simplified) storage node which enforces local Quality of Service between different classes of workload (queues). 7
3 The difference where QoS is applied on a local approach versus a global approach. 9
4 Different replication strategies [1]. 11
5 The throttling model used in AQuA [2]. 17
6 Queue structure in EBOFS [3]. 18
7 Enhanced queuing structure in Q-EBOFS [4]. 19
8 Deficit Round Robin schedule requests according to the workload class weight. For each round, all workload classes receives new quantum which is added to the deficient counter. 21
9 Our proposed model for handling maintenance requests. There are queues for each replication-degree in the system. 23
10 Three different workload class streams at a storage node. From time 0 to t_a, there is only one workload class present at the node (class 1). At time t_a, workload class 2 starts receiving requests, and workload class 3 at time t_b. Each stream is tagged with DTOM notation. . . 25
11 The DTOM tree structure when throttling between workload class 1 and 2, respectively shown as x_1^0 and x_2^0. 26
12 The DTOM tree structure when throttling between workload class 1, 2 and 3, respectively shown as x_1^0, x_2^0 and x_3^1. 27
13 A DTOM tree which currently has a level k and class $j-1$ is currently the newest added workload class. 29
14 Different workload class streams. At time t_a, workload class j is presented at the storage node (queue no longer empty). Each stream is tagged with DTOM notation to show where it relates in the DTOM tree. 29
15 A DTOM tree where the new class j is added at level k (x_j^k). The new root node is now X^{k+1}. 30
16 Removing workload class j (leaf node x_j^k) from a DTOM tree. History stored in X^{k+1} and X^k can be added together. 31
17 Different workload class streams. At time t_a, workload class j is no longer presented in the system (empty queue). Each stream is marked with DTOM notation to show where it relates in the DTOM tree. 32

18 The resulting DTOM tree after leaf node x_j^k (not shown) has been removed. . . . 33
19 A DTOM tree where node x_i^0 and x_{i+1}^0 are the bottom leaf nodes. Class i is no longer present in the system (x_i^0), and is going to be removed. . . . 34
20 Different workload class streams. At time t_a, workload class i is no longer presented in the system. Each stream is marked with DTOM notation to show where it relates in the DTOM tree data structure. 34
21 A DTOM tree where node x_i^0 (not shown) was removed. x_{i+1}^0 and x_{i+2}^0 are now the new bottom nodes. . . . 35
22 Two queues filled with requests. Queue A is filled with requests of (min) size i and queue B is filled with requests of (max) size k. . . . 36
23 Three different workload class streams. At time t_a, workload class 2 starts sending requests, and workload class 3 at time t_b. Each stream is tagged with DTOM notation. . . . 39
24 Observed utilization for different workload classes when using WRR and request sizes from 5 KB to 10 KB. . . . 40
25 Observed utilization for different workload classes when using DTOM and request sizes from 5 KB to 10 KB. . . . 40
26 Observed utilization for different workload classes when using WRR and request sizes from 5 MB to 10 MB. . . . 42
27 Observed utilization for different workload classes when using DTOM and request sizes from 5 MB to 10 MB. . . . 42
28 Each stream is tagged with DTOM notation. . . . 43
29 Observed throughput when client 3 sends requests in interval. . . . 43
30 Example of fields and data types for documents. . . . 47
31 Overview of Vespa Document Storage architecture. . . . 48
32 The relationship between buckets, slotfiles and documents in VDS. . 51
33 The message chain in a VDS reorganization. . . . 52

List of Tables

1 Notation used in DTOM. 24
2 Percentual division of workload classes according to formula described in section 2.4. 27

1 Introduction

1.1 Motivation

Large-scale distributed storage systems are becoming larger and more complex. They must satisfy many different needs from the variety of contexts they are used in. Thus a storage system is exposed to many different types of workloads, each categorized into classes with different characteristics, priorities and constraints. The storage system must deliver satisfactory performance for each of them under any circumstances.

Inevitably, such large systems, composed of inexpensive commodity hardware, will frequently experience failures. While some errors are short-term and a particular storage node may shortly return to a healthy condition, other errors results in the need for a replacement node. Hence, storage systems must be able to handle node failures in a graceful way. Besides handling node failures, a storage system must also tackle expansion, *e.g.* in order to keep up with the data growth pace in the system, and conversely, removing old and outdated storage nodes.

Both scenarios cause the system to maintain physically stored data. Some of the data becomes more vulnerable than others, thus more critical. However, workload in both scenarios can be generalized into a maintenance workload class.

Besides to maintenance workload, a storage system obviously has production workload as well. Production workload is work generated by external clients. Therefore, when both production and maintenance workload are actively present in a system, a throttling mechanism must ensure both classes receive system resources without blocking each other. In addition, depending on the context, there will be different priorities between production and maintenance workload, and hence each workload will demand Quality of Service.

This master thesis was conducted in tight collaboration with Yahoo! Technologies Norway (YTN). YTN is developing Vespa Document Store (VDS), a large-scale distributed storage system. Research regarding system recovery and Quality of Service in distributed systems are of great interest for YTN.

1.2 Goals and Scope

One of the goals of this thesis is to investigate how QoS can be used to improve a system recovery process. Initially, there are several areas in a storage system

where QoS can be applied. Although we will briefly describe global level QoS, our main focus is local level QoS where also our contribution applies.

Our experimental results are obtained with a simulator adjusted to reflect the behavior of VDS. Simulations are used to illustrate the effectiveness of our algorithm. Implementation into a real system is not within the scope of this thesis, however the conceptual idea is still the same.

1.3 Contributions

This thesis contains more than one contribution to the field of distributed storage systems. We give a brief overview.

First, we give a comprehensive explanation of system recovery in large-scale distributed systems, and more precisely where the different techniques may apply. We also define requirements for a successful system recovery process.

Second, we have created a system recovery model. The model aims to gracefully handle different priorities of critical workload.

Lastly, we have designed and implemented an algorithm, Dynamic Tree with Observed Metrics (DTOM). DTOM is designed to throttle hard-disk bandwidth between multiple workload classes. A lower bound proof and simulations prove its effectiveness.

1.4 Outline

- **Chapter 2** presents system recovery, workload classes and QoS, and focus on narrowing down to where our solution applies.
- **Chapter 3** describes related work concerning system recovery and QoS in distributed storage systems.
- **Chapter 4** presents our contributions, a generalized QoS recovery model and our throttling mechanism, Dynamic Tree with Observed Metrics. Lastly in chapter 4 we show a lower bound proof regarding performance of our algorithm.
- **Chapter 5** shows experimental methodology, as well as the experimental results for our algorithm.
- **Chapter 6** concludes our work and present future work.

2 Problem Elaboration of System Recovery

2.1 Overview

Large scale distributed storage systems are build upon a few to many thousands of storage nodes. Each storage node normally uses inexpensive commodity hardware. From time to time, storage nodes experience failures and inevitability become unavailable [5, 6]. Although many failures can be short-termed and the storage node become available after a restart, other errors cause the storage node to be taken off-line for further maintenance. Thus, data on the failed storage node needs to be recovered and replicated to other functional storage nodes.

On the other hand, a storage system may also increase in the number of nodes, *i.e.* additional disk space is added. Such expansion normally means to obey new system requirements. The newly added nodes need to be populated with data in order to ease off existing nodes. Thus data must be moved from existing nodes to new nodes.

However, both scenarios should be performed without interrupting the normal operation of a storage system. In addition, such recovery operations should require minimal human intervention. The latter is essentiality important since human errors have been identified as a significant source to system failures [7].

A storage system is *stable* when all objects or documents in the storage system reside on the correct node, and respectively *unstable* when data is on the wrong node. In order for an unstable system to become stable, the storage system must *recover*. In short, a data placement algorithm [6, 8, 9, 10, 11, 12, 13, 14, 15, 16] utilizes the number of available storage nodes when determining the placement of an object. If this number changes, already placed objects must be moved so that the placement algorithm can deterministically calculate new whereabouts for the object, given the new number of storage nodes. It is necessary to move objects according to the placement algorithm since the same algorithm is used for data retrieval of objects as well. The process of moving (replicating + deleting) objects so that the system becomes stable is called a *recovery process.*

To better understand the domain of this report, we will give some examples which will clarify and point out the scope. First and foremost, a recovery process strongly involves a data placement algorithm in order to determine whereabouts for data objects. However, in this report, we assume a "perfect" placement algorithm, *i.e.* we will not dive into details regarding mathematical aspects, resulting distribution on nodes, *etc.* Our "placement algorithm" calculates object whereabouts pseudo-randomly, but evenly, so that each object has an equal probability to be placed on

any available storage node.

As an example, assume a given storage node experiences an unexpected hardware failure and can not be safely removed from the storage system. Assume further that data on the node's physical disk can not be recovered and is considered lost. Given the nature of large-scale distributed storage systems [5, 17, 18], all objects are one way or another present on several nodes according to a fault tolerance scheme. Two common fault tolerance schemes are primary-copy and erasure coding replication [19, 20, 21, 22]. In this context, we assume an R-way primary-copy replication scheme.

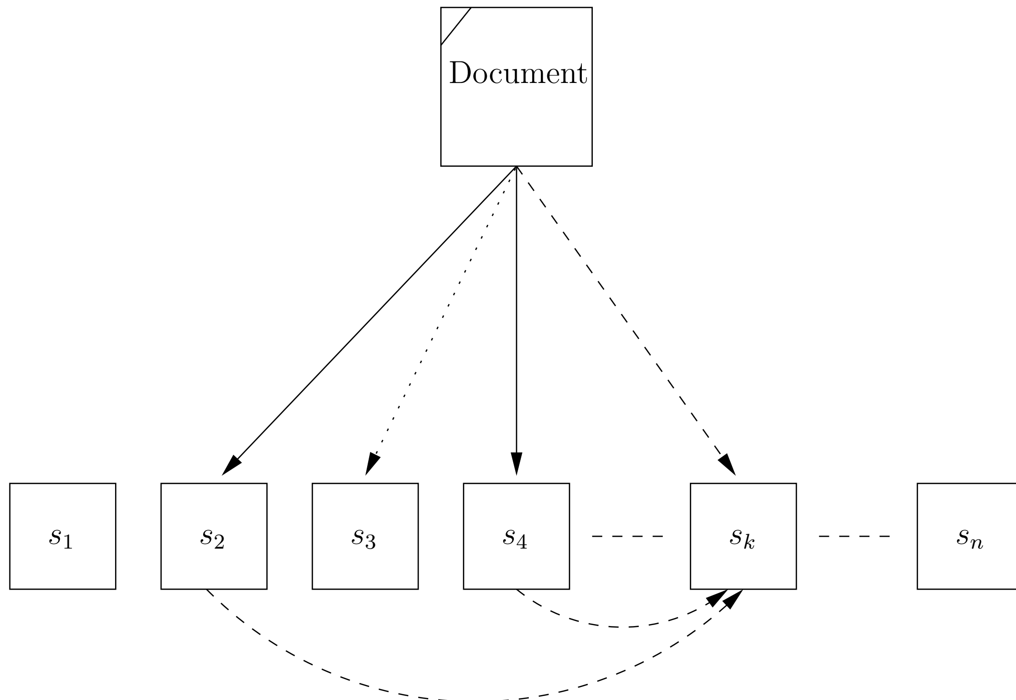

Figure 1: Storage nodes s_2, s_3 and s_4 initially holds a copy of a document (or object). If s_3 fails (dotted line), the object must be replicated to s_k (dashed line) from either s_2 or s_4 in order to stabilize the system.

Continuing our example, assume an object is stored on three nodes, s_2, s_3 and s_4 with a 3-way primary-copy replication scheme as shown in Figure 1. s_2 holds the primary copy, while s_3 and s_4 contain replicas. The storage system detects that s_3 is experiencing a failure (dotted line) *e.g.* by a heart-beat monitoring service. The monitoring service notifies all nodes in the system with the loss of s_3 (not shown in the figure). This is also called a system state change, *i.e.* a change in one or more node's local system state. At this time, the document in question is

only replicated twice (s_2 and s_4). The system has become unstable and thus needs recovering.

In order to recover, all nodes will have to iterate over all locally stored objects and find out if one or more objects are affected by the loss of s_3. This is performed with help from the data placement algorithm. The data placement algorithm will determine if an object's primary or replica copy was on the failed node s_3. If so, the placement algorithm has to re-calculate new whereabouts for the object, in our case say s_k instead of s_3. In order to stabilize the system, s_k must be populated with a physical copy of the object in question.

Further, we assume that s_2 is the chosen source to populate the destination node s_k. s_2 will then produce and hence send a *maintenance* request over the network which will contain a physical copy of the document. The name maintenance is because the system has to self-maintain in order to recover. More precisely, the maintenance request will obviously require access to the underlying I/O subsystem on s_2 to acquire physical data blocks for the object. Assuming there are other requests to I/O as well on a storage node, *e.g. production* requests, the maintenance request must be put in a queue while waiting for service time.

To further complicate the problem, we assume both s_3 and s_4 are experiencing problems and must be taken down for maintenance. Physical data on the hard-disks are considered lost. At this moment, the particular document is only replicated one time in the system which is on node s_2. It is now crucial to replicate the particular document, since if s_2 become faulty, data can be lost. Therefore, replication requests concerning documents which are only replicated once, needs to be prioritized over requests that concern documents which are replicated twice (or more). In other words, prioritize less replicated documents. We categorize such replication requests as *critical* workload.

On the other hand, production requests are generated by clients or users outside the storage system. For instance if the storage system was an e-mail system, then production requests would be users reading their e-mails (I/O read requests), or if the user is writing and sending an e-mail, this data also have to be stored (I/O write request). The point is, there can be many variants of production workload as well as maintenance workload. *E.g.* in a commercial storage system, there could be different priority between paying and non-paying customers, however both brings production requests to the system.

To summarize, production requests drive the system while maintenance and critical requests "repair" it.

2.2 Workload Classes

Thus so far, each storage node will send and receive production, maintenance and critical requests whereas each request demands access to underlying I/O, either as a read or write operation. This can be generalized into three different *workload classes* [2, 4]. Production workload generated by outside users, and maintenance and critical workload generated by the internal system itself. In other contexts, *e.g.* scientific or multimedia, other classifications of workload could be more suitable. The point is we want to differentiate access to I/O between the various workload classes. However, note that requests of same workload class have to race against each other for I/O access.

The need for differentiation between workload classes becomes evident when we want to pick the order of which type of request shall gain access to the underlying I/O. Assume a system is experiencing a system state change as described in chapter 2.1. The storage system needs to recover and data must be replicated in order to stabilize. Further assume that one of the storage nodes needs to perform many hundreds or even thousands of maintenance requests. It then becomes obvious that the outside users can't hold back their requests until all maintenance requests are completed. The outside users must expect some kind of *Quality of Service* (QoS) for their production requests.

However, it is also necessary to let maintenance requests access I/O as well. Even though a storage system would normally favor production requests, maintenance requests are still important and must be processed in order for the system to be fully operational, especially critical requests. During system recovery, a temporarily performance drop-down is generally tolerated [21], but it is of course suitable to keep it to a minimum.

Thus, maintenance (and critical) requests can't be kept back and only processed when the system is idle, *i.e.* when there are no pending production requests on a node.

2.3 Local and Global Level Quality of Service

Large-scale storage systems may experience numerous different types of requests, both internal and external. A coarse classification can be *e.g.* production, maintenance and critical requests as described in chapter 2.2.

As shown in Figure 2, we can enforce QoS when picking requests from the different workload classes. All requests want access to the underlying I/O subsystem. Therefore, each workload class gets its own workload class queue. In our scenario,

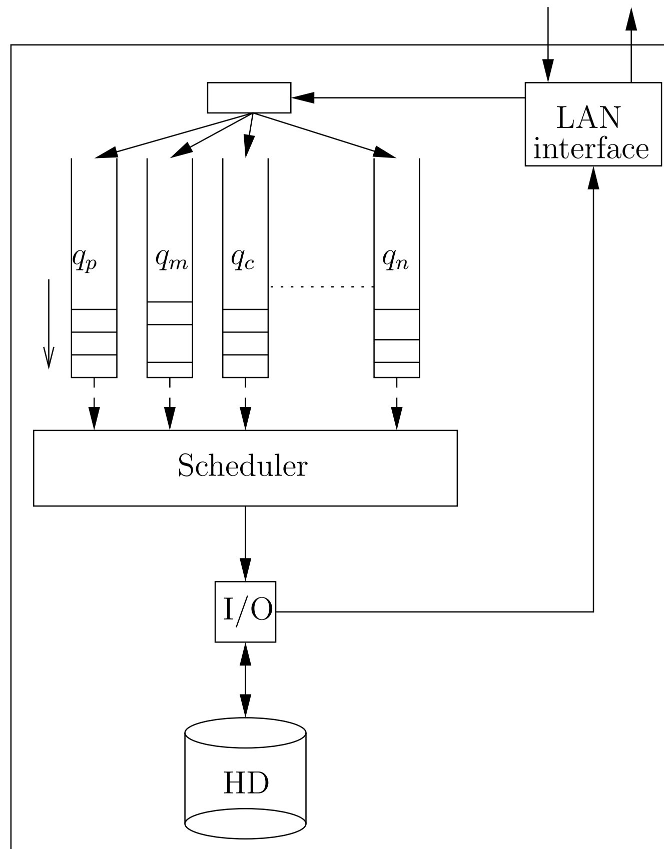

Figure 2: A (simplified) storage node which enforces local Quality of Service between different classes of workload (queues).

we will have one I/O queue for production requests, one I/O queue for maintenance requests and one for critical requests (respectively as q_p, q_m and q_c in Figure 2). A *local throttling mechanism* (described further in chapter 4) will determine the order of picking requests from the different queues (only shown as "Scheduler").

In our context, a weighting scheme of workload classes is suitable in order to better differentiate and prioritize requests between workload classes. Each workload class i will be given a predefined static weight w_i. Higher weight means that this class shall receive more access/bandwidth to I/O as opposed to workload classes with a lower weight.

The local throttling mechanism will have to follow a set of predefined workload sharing specifications, *i.e.* the amount of weight/I/O bandwidth each workload class shall receive. All storage nodes have to employ the same weighting scheme.

To summarize, all incoming requests to a storage node are either received by a

LAN interface, or generated by the internal system before they are placed in their respective workload class queue. A (local) throttling algorithm will pick requests from all queues according to a globally known weighting scheme. Once a request is picked from one of the workload queues, the request can access I/O to perform its intended task.

Although the scheme is globally known, this is still not a global level QoS approach. In [2] they state that when enforcing QoS at every storage node in a system as described above, we will also achieve a global QoS.

There are situations where the above approach does not give any good performance nor QoS. A local QoS mechanism does not have any information of its fellow storage nodes. Thus the node cannot know if other nodes have much pending work, or if all workload queues are empty.

Assume a state change occurs and new storage nodes were added to the system. Recall that we also assumed primary-copy replication scheme, whereas each object is replicated three times. When we want to populate the newly added nodes with objects/documents, each object has potentially three possible sources (one will be surplus and eventually be deleted). If all sources of the object had knowledge about each others pending load, the node with lowest metric/value could be chosen to populate the newly added node.

Gathering and maintaining such information is not easy. First, all nodes have to regularly send queue status information to each other, or to a dedicated server. This will inevitable increase the amount of meta-data floating in the system resulting in significantly more overall system load. In [23], they state that meta-data operations can make as much as half of system workloads. Second, keeping a dedicated centralized server for such information can introduce potential bottleneck that will reduce system scalability [4].

The main point of global level QoS is to redirect maintenance requests off a node which already has much pending (production) requests. In other words, utilize (and maintain) globally known information which could improve and guarantee bandwidth allocation.

To better understand the domain of the different QoS approaches, see Figure 3. The local level QoS approach, depicted in Figure 3a, enforces differentiation between workload classes on a per node basis, *i.e.* as an extra QoS layer on top of each node. As explained above, the local approach does not utilize any information from other nodes regarding system load nor pending requests at workload queues. However, it will always throttle disk I/O according to the predetermined specification. Every storage node in Figure 3a will mainly follow the same simplified node

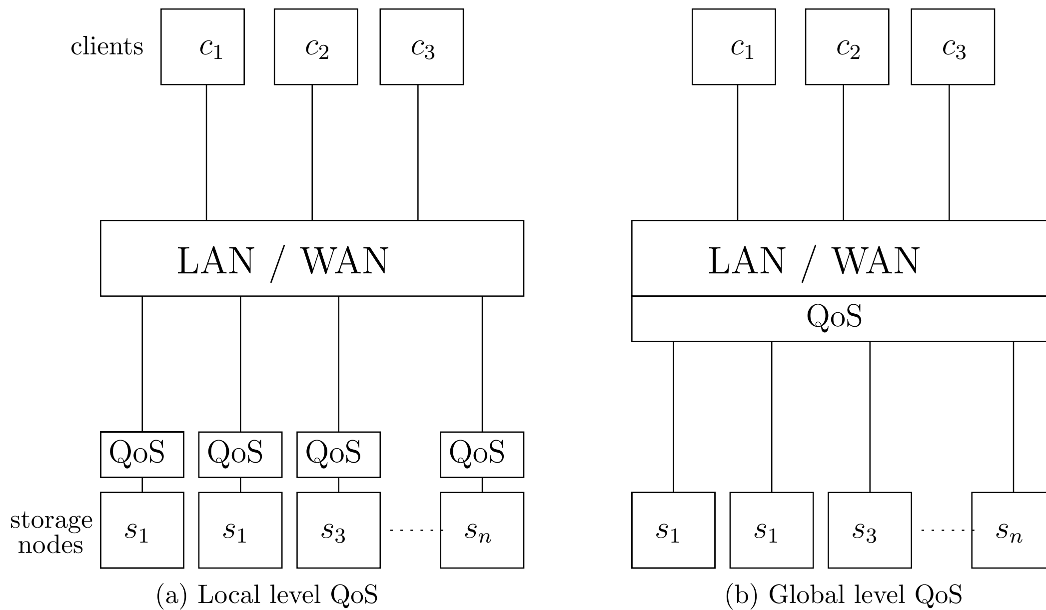

Figure 3: The difference where QoS is applied on a local approach versus a global approach.

structure as shown in Figure 2.

Since it is a local approach, all decisions are taken solely on own recorded metrics (or any other mechanism to support the throttling). There is no need to gather any outside information. This also simplifies implementation significantly. Such an approach is also called a decentralized solution.

The global approach is depicted in Figure 3b. Instead of acting on per node basis, a global level approach will make a QoS layer on top of all storage nodes, and hence providing QoS. A global level approach will consider pending request and/or current load of all nodes. Thus, the global QoS can redirect requests to less loaded nodes to better balance the overall throughput, or easier fulfill the predetermined QoS specification regarding the different workload classes. However, as mentioned above, a global approach might use a dedicated server which makes it a centralized solution. Although a centralized solution has some advantages, there are also disadvantages to consider. Centralized solutions is undeniable a possible bottleneck in addition to be vulnerable for a single point of failure.

2.4 Sharing Surplus Bandwidth.

In our context, there are situations where a storage system will have “surplus” bandwidth available for re-allocation. Assume we have a QoS specification which grants 50/30/20 % bandwidth sharing for respectively workload classes A, B and C. Each of the three different workload classes has their own characteristics, *e.g.* different priority demands *etc.*

Assume that one of the workload classes, class C, does not generate any requests for a while, while the two others have an almost continuous arrival rate of new requests. Practically this means to throttle bandwidth between workload classes that only are actively present in the system, *i.e.* class A and B. How do we split the “unused” allocation of class C?

In our example, class C should receive 20 % of the bandwidth as specified. One approach could be to split the workload class evenly between others, *i.e.* class A and B get additional 10 % resulting in a 60 / 40 sharing. The drawback of this approach is that the sharing is not proportional to the intended sharing specification. Class B would receive more bandwidth while class A gets less. A proportional sharing in our example gives a sharing of 62.5 % for class A and 37.5 % for class B. In a generalized sense, the proportional sharing calculation uses the following equation:

$$\sum_{i=1,n} (w_i \times p_i) = 1$$

w_i is the predefined weight for workload class i and $p_i \in \{0, 1\}$ where 0 when class not present, otherwise 1. Note that this equation can also be used when more than one class is not present in a multi-class workload sharing scheme.

2.5 Replication Requests - Maintenance or Production?

As described earlier in this report, the system generates maintenance requests, normally in order to recover from an unstable system state to a stable state. Another aspect to consider whether or not as maintenance or production workload, is when an outside user is putting objects into the system (I/O write). Obviously, this is production workload. However, a put request usually involves more than one storage node.

There are a few different strategies on how to populate replicas of an object on to storage nodes. Figure 4 shows three different replication strategies which is described in the RADOS [1] paper.

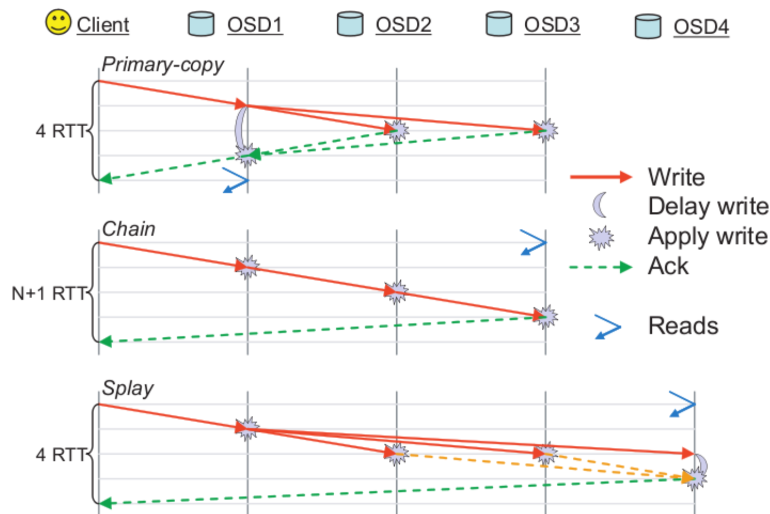

Figure 4: Different replication strategies [1].

The first strategy is called primary-copy replication [19]. After a request has been received on the storage node which will hold the primary copy (OSD1 in the figure), the request is forwarded to all replicas in parallel. The primary will wait for all replicas to acknowledge the change before the request is committed on to the primary.

The second strategy is called chain replication [24]. It differs from primary copy in that updates are executed in sequence instead of parallel.

The last replication strategy described in RADOS, is called *splay*. Splay is a hybrid between the two described above. As shown in Figure 4, replication is initiated by the primary copy. The primary will send update requests to all other replicas so that they can execute the put request in parallel. It is the last node (OSD4 in figure) who will report back to the client when all replicas have been committed.

An obvious question to ask is, should replication requests as just described, be regarded as production or maintenance workload? Both views have advantages and disadvantages.

If we consider replication requests as production workload, then the system will inevitable increase the overall production workload with a factor of $R-1$ (in a R-way replication). Or the opposite, if we consider the initial put request as production, and rest of the replication as maintenance, the outside user will experience a lower latency than the first approach since one normally would favor production requests over maintenance requests, however this is depending on the context. Since it is

appropriate to acknowledge the user when *e.g.* a put request has been committed to a physical hard-disk, the question arises, when shall the system send such an acknowledgement back to the client? Should the whole request be treated as a synchronous request, *i.e.* do not report back to the client before all replicas have been successfully committed, or treat replication to the primary as synchronous and report back to the user while rest of the replicas are committed asynchronous?

Another concern is safety of user data. An object replicated twice in a storage system has higher survivability in comparison to an object only replicated once. Sending an acknowledgement too early, the user can not know for sure if all replicas were successfully updated. Waiting for all replicas to report a successful commit (synchronous replication), the user will necessarily experience more latency. An approach could be to *e.g.* send the user an acknowledge when $R/2$ of the replication request have successfully committed (synchronous), and hence treat them as production workload. The rest of the $R/2$ requests can be treated as maintenance workload (asynchronous).

All of the above become of importance when deciding proper weights for production and maintenance workload classes. Note that this is more implementation specific, and none of the approaches are necessarily the "best" or preferred alternative, however we think it is important to consider when deciding proper weights for the different workload classes, especially regarding the weight/prioritizing of maintenance requests.

2.6 Requirements

In order to have a successful recovery process of data in a storage system, it is necessary to have defined requirements to a proposed recovery solution. This chapter describes our goals and views for a recovery process to be successful.

I - Availability of data during a recovery process.

When a storage system is recovering from unstable to a stable state, all user data must still be available during the entire recovery process. However, note that an overall performance drop down is expected and accepted.

The performance drop down must be adjustable according to a predetermined percentage (or similar). This means that outside users should at least expect a minimum throughput or allocation of a node's capabilities. In other words, users should expect a QoS.

II - Class-based workload sharing.

In a large-scale storage system, there will most likely be many different types of workloads, which further can be categorized into different types of classes. A workload class is defined as workload with similar characteristics. As described in section 2.2, we have so far defined three different types of workload, respectively production, maintenance and critical workload.

In some systems there might be desirable to split up the production workload class even further. Assume we have three clients whereas each of them has different demands. It then becomes evident that the different clients shall receive different amounts of bandwidth (system resources). Even though our intended problem domain is system recovery in storage systems, our work on class-based workload sharing should also be applicable in a more general approach to obtain QoS in large-scale storage systems.

Critical workload must also be highly prioritized, since we can not tolerate any loss of user data.

III - Task scheduling to avoid network bandwidth conflicts.

Scheduling of requests, *e.g.* maintenance requests, is necessary to avoid that some nodes become saturated while other nodes are close to idle. Heavily loaded or saturated nodes should not receive additional load since this will reduce performance of the system.

Directing workload to under-utilized or idle nodes is not a straight-forward task. First, client access pattern can be hard to predict since any object can be placed on any storage node. Some solutions try to distribute workload among replicas of a particular (user) object, based on observed metrics [25, 26]. They record the time spent at disk which is normalized with the size of the request. The recorded metrics are stored in a centralized fashion which is thereafter used to distribute the workload. A similar approach could be adapted, however we wish to avoid possible bottlenecks with a centralized server since the system may have to scale to the extreme.

Second, if we assume data objects to be spread uniformly across all storage nodes, then the workload will also share the same characteristics [8, 9, 10]. In our context, production workload can be hard to predict in order to spread workload. However, maintenance workload are generated and triggered by the system itself and thus, it can to some extent be controlled. Scheduling maintenance requests so that under-utilized or less-loaded nodes handle more maintenance workload will improve the overall recovery performance without affecting client performance, rather the contrary and improving the performance.

IV - Barrier functionality.

A barrier function [4] is a way to ensure that tasks in many queues are completed before executing new tasks. If a state change occurs, *e.g.* one or more nodes are added or removed, data has to be recovered in order for the system to recover to stable. However, when the change occurred, there might be requests that can still complete successfully and one wish to process the requests before applying the change.

On the other hand, if there are pending write requests, and a state change resulted in some objects are only present with 1 copy in the system, one wish to spread copies in order to maintain the fault tolerance scheme for such objects before applying write updates.

The main reason for having the barrier function is to ensure data consistency. If the queuing discipline is different than FIFO, *e.g.* a priority based queue or similar, the order in which requests arrive is important. A task marked with a barrier flag means that this task has to complete before any other tasks can be completed, or the opposite, a task marked with barrier flag means that all pending tasks have to get service time before this particular task gets service time.

V - Performance metrics.

In order to decide if a recovery process is successful or not, we must also be able to measure its performance and compare it to other solutions. It must be possible to isolate throughput or bandwidth utilization for each workload class in order to validate if the solution throttles requests according to the predefined static weighting scheme.

3 Related work

3.1 System Recovery in Distributed Storage Systems

VDS

Vespa Document Storage (VDS) (see appendix A) is a large-scale distributed storage system developed by Yahoo! Technologies Norway (YTN). VDS is designed to provide high-availability, survivability, high-performance and scalability to data-intensive applications running on inexpensive commodity hardware.

VDS consists of multiple clients, distributors and storage nodes, but only one fleet controller. Distributors are responsible for maintaining *buckets* among storage nodes. A bucket is a form of data unit which is replicated on to storage nodes.

The fleet controller controls and monitors the global system state, and propagates system changes to all nodes.

System recovery is initiated by distributors. When the fleet controller has broadcasted information regarding the loss of a particular storage node, each distributor will traverse their internal mapping of buckets and find out if the loss is affected by any of their buckets. If some of the buckets are affected, then the data placement algorithm used in VDS will be used to recalculate whereabouts for the bucket.

Thereafter, the distributor initiates a replication request at the primary copy of the bucket. The primary will communicate with other replica holders in a chain to ensure that all replicas will contain identical data.

Ceph

Ceph [1, 3, 17, 27] is a distributed file system that focuses on performance, reliability and extreme scalability. It is continuously being developed by the Storage Systems Research Center (SSRC) at the University of California, Santa Cruz. It is released under Lesser General Public License (LGPL) and the source code is freely available to download from SourceForge [28].

The architecture of Ceph consists of three main components; the client, a metadata server (MDS) cluster and a cluster of object storage devices (OSD). Each node in Ceph has two dimensions regarding local system state. A faulty node can either be *down*, but still *in*, *e.g.* when carrying out reboot. If the state is down and *out*, it is considered lost and thus data replication is initiated.

Data recovery in Ceph is based on Fast Recovery Mechanism [29] (FARM). All data objects in Ceph are placed in Placement Groups (PG) [1], and PGs are replicated across n OSDs (n-way replication scheme) using the CRUSH [6] data placement algorithm.

System recovery is initiated when a PG's membership has changed. *I.e.* when a *up* and *in* storage node receives a new global system state, the node will iterate over all locally stored PGs and re-calculate the mappings with CRUSH in order to determine which PG it is responsible for, either as primary or replica.

If the node is a replica member of a PG, it must *peer* with other members of the PG, *i.e.* retrieve the latest PG version number. If it is primary replica of a PG, the node has to collect current and former PG version numbers from all PG members. Next, the primary will send an incremental log update or if necessary, a complete content summary. The goal is to let all members of a PG agree on the current content in the PG.

[1] In FARM [29], PGs was called Redundancy Groups

When the above goal is obtained, each member of the PG is independently responsible for retrieving outdated or missing user data from other PG members.

Since recovery is driven entirely by individual storage nodes, each PG affected by a change, either removal (failure) or addition of new nodes, will recover in parallel. Such a change will usually involve multiple nodes, and since they recover in parallel, it will decrease recovery time and improve the overall data safety.

The Google File System

The Google File System [5] (GFS) is a distributed file system designed to handle performance, scalability, reliability and availability. Its design is strongly influenced by key observations in both their application workloads and technological environment.

The GFS architecture consists of a single GFS master node, multiple GFS chunckservers (contains user data) and multiple GFS clients. It is the master node who handles and initiates all operations in a GFS cluster. This makes GFS a centralized distributed storage system.

System recovery in GFS is also handled by the master. The master will periodically send out requests to the chuckservers. If one of the chunckservers does not respond to the heartbeat signal, the master will mark the respectively chunkserver as down in its internal chunkserver-map, and further initiate chunk replication in order to maintain the fault tolerance scheme.

Sorrento

The Sorrento [18] is a self-organizing, high-performance and data-intensive distributed storage system. The system will self-organize storage resources when (1) storage nodes are added or removed from the system, or (2) when one or more nodes are detected to be faulty.

The architectural components in Sorrento are mainly clients, namespace servers and storage providers. In addition, there is a membership manager which all nodes send heart-beat signals to. Each storage provider has a location table which contains information about locally stored user data, or segment ID's.

If a storage provider fails to send its heart-beat signal to the membership manager, the manager will inform all other nodes with the loss, and their location table is updated. When a node detects that one of their objects aren't fully replicated (according to the replication degree), it will lazily propagate data to replicas of an object.

3.2 Aqua QoS Framework

AQuA [2] is an adaptive Quality of Service-aware framework proposed to initially support Quality of Service in Ceph. In addition, AQuA is also a general QoS framework for any distributed storage systems. The goal of the QoS mechanism in AQuA is to throttle and allocate disk bandwidth between different workload classes. QoS assurance is provided individually for each storage node, *i.e.* a local level QoS. The general framework model is shown in Figure 5.

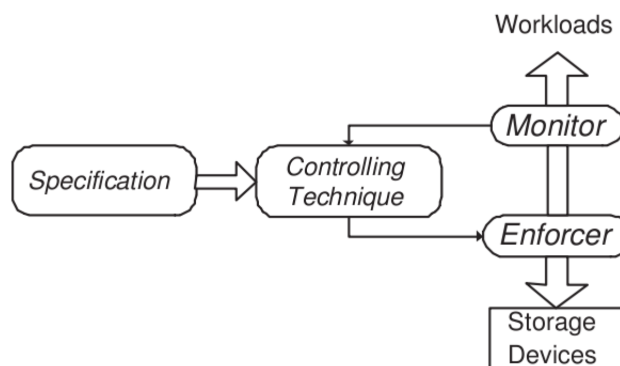

Figure 5: The throttling model used in AQuA [2].

The Specification is a set of declarations which describes the desired quality level for a particular entity. An entity can be a client, group of clients, class of applications, *etc.* The throttling mechanism proposed in AQuA, Hierarchical Token Bucket (see 3.7) uses hard-disk throughput. Other mechanisms, such as SLED [30], utilize average response time, or Facade [31] which specify read and write latency as a function of request rate.

The Monitor is the component that monitors the quality level for the different identities. This can be *e.g.* observed throughput, response time or latency as described in the former paragraph.

Based on what the Monitor component has observed, it is the Controlling Technique component that utilizes the data, and checks the data against the Specification. Hence it is the Enforcer that performs adjustments from the Controlling Technique.

3.3 Bourbon QoS Framework

The currently QoS framework used in Ceph is called Bourbon [4]. Bourbon is enabled by Q-EBOFS, which is a QoS-aware object-based file system. The main objective of Q-EBOFS is to split workload into different classes and make class-based guaranties of disk bandwidth. Different classes can have different priorities, however different workload within the same class will still compete for resources.

In the Bourbon framework they use a weighting scheme for specifying the target share for each class. *E.g.* in a two-class scenario, class A should be guarantied 80 % of the total bandwidth, and class B 20 %. However, keep in mind that a client (or another OSD) has to tag the workload with the respective workload class in order to fulfill the guaranties. Un-tagged requests are placed in a "best-effort" workload class queue. It is up to the administrator to decide a proper weight for un-tagged requests.

In order to better understand the queuing structure in Q-EBOFS, we will briefly explain the original idea in EBOFS [3]. First, modern disk schedulers must support the barrier function. A barrier request guarantees that all requests prior to the barrier request will be completed before requests arriving after the barrier. This is useful to ensure the order in which data is committed to disk. EBOFS supports the barrier functionality (see Figure 6). Whenever the root queue in EBOFS receives a barrier request, a new elevator queue is made, and new requests are added to the new queue.

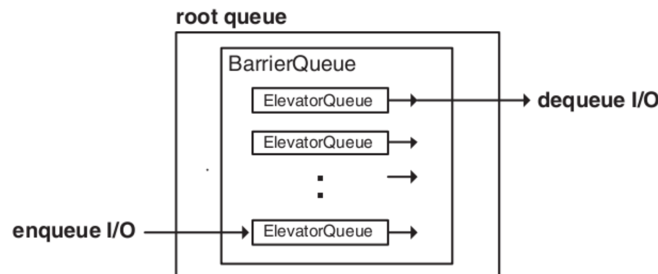

Figure 6: Queue structure in EBOFS [3].

Q-EBOFS does also support the barrier functionality in the same way as in EBOFS. Each elevator queue consists of multiple FIFO-queues. Each elevator queue and its FIFO-queues are encapsulated within an abstraction barrier queue. The root queue is then composed of a list of barrier queues as shown in Figure 7.

The different FIFO-queues represent each workload class. In a two class scenario we have one queue for each class, and in addition, a default queue for un-tagged

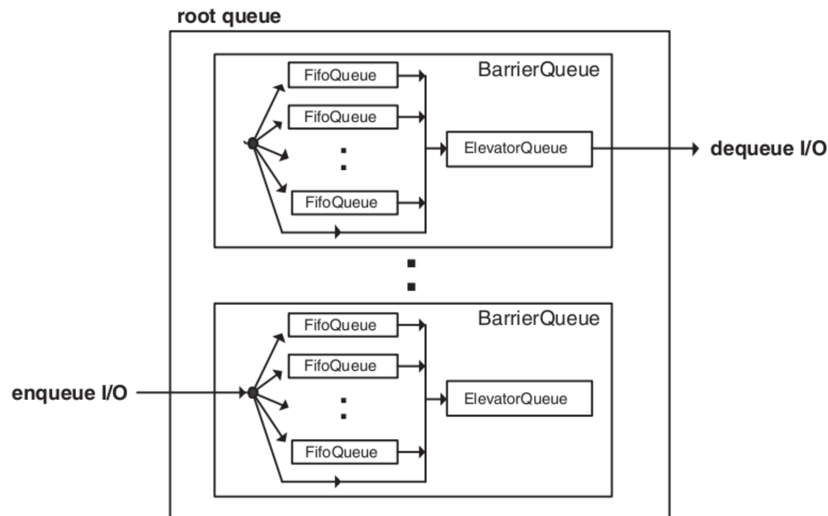

Figure 7: Enhanced queuing structure in Q-EBOFS [4].

workload requests.

The bandwidth sharing between the class-queues is enforced with how the elevator queue picks requests from each FIFO-queue. The current implementation of Q-EBOFS in Ceph uses a Weighted Round Robin (WRR) (see chapter 3.5) for proportional sharing.

3.4 Round Robin

There are many variants on how a Round Robin (RR) [32] approach could be applied in the world of class-based bandwidth throttling. As mentioned earlier, it all narrows down to select the order in which to process requests from different queues, whereas all requests want access to the underlying I/O.

In the simplest sense, a RR approach could pick one request from each queue regardless of size or any other means. When all workload classes have obtained access to I/O, then simply redo the process all over again. If a queue is empty, *i.e.* no pending workload for the particular class, then simply skip to next queue/class.

On average, this approach would lead to all workload classes getting an equal amount of bandwidth, *i.e.* each class would get (*max_bandwidth* / *num_classes*) percentage.

The obvious drawback with this approach is no differentiation or prioritizing between different workload classes, which also is a requirement described in sec-

tion 2.6.

3.5 Weighted Round Robin

Weighted Round Robin (WRR) [33] meets the requirement about workload class differentiation as described in section 2.6. With a WRR approach each workload class is assigned a predetermined static weight. A WRR implementation could *e.g.* process one maintenance request for every fourth production request. This results in a weighting scheme of 80/20 % sharing. Again, this implies that all requests have same size in order to share bandwidth between classes as intended.

Another drawback of using WRR and throttling on requests is when request sizes tend to vary significantly. Having small production requests, each of only a few KB, then a large maintenance request of 10 Mb will dramatically degrade QoS. Short-term peaks, however, will only affect workload classes with lesser weight than superior ones.

Instead of throttling on requests, a WRR implementation could throttle on request size. *E.g.* for every 5 Mb of maintenance requests, then process 20 Mb of production requests. Obviously, this would lead to requests are half processed, because the workload class share has been consumed. In addition, the storage architecture must support requests that are half processed (preemptive) being placed on a buffer (or some other logic), so when the particular workload class receives access to I/O again, the request can continue exactly where it was paused.

3.6 Deficit Round Robin

Another variant of RR is Deficit Round Robin (DRR) [34], or Deficit Weighted Round Robin (DWRR). In addition to handle differentiation between workload classes, DRR also handles variation in request sizes. For each round, all workload classes receive a numeric share called a “quantum”, each in proportional to the predefined static weight. *E.g.* if the initial share for all workload classes is 1000, then a class demanding 60 % will get 1600 of quantum. For each round, the quantum is added to a deficient counter (DC) value for the particular class.

It is the DC who controls the scheduling. For each round, DRR will process requests as long as the size of the request is lesser than DC, and hence reduce DC according to the request size. As shown in Figure 8, queue A for workload class A has two pending requests. During this round, the request with size 1200 can

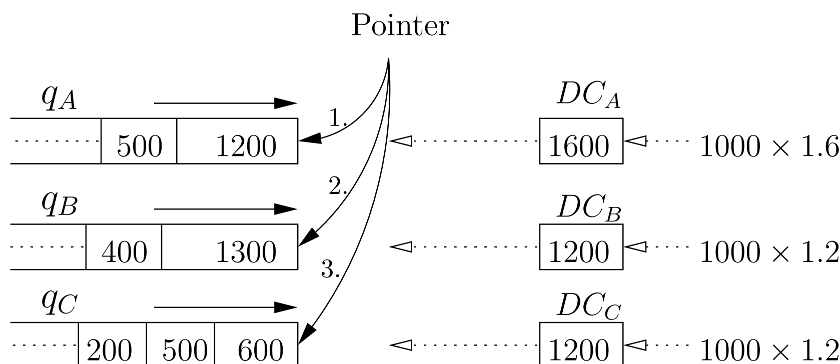

Figure 8: Deficit Round Robin schedule requests according to the workload class weight. For each round, all workload classes receives new quantum which is added to the deficient counter.

be processed, but not the next one because DC will only be 400. However, next round will DC for class A will be 2000 (1600+400).

On average, all workload classes will get disk bandwidth as intended. But given the nature of workload in our scenario, a request can be either read or write request. Throttling on request size regardless if it is a read or a write, will give an unbalanced workload sharing. It is possible to add an extra weight whether if the request is a read or a write, but again the inexactness factor must be tolerated.

DRR also assumes that the request size is known in advance, and further checks if the request can be processed within the current DC. However, knowing the size of a read request can be hard, since data sizes aren't usually stored separately.

3.7 Hierarchical Token Bucket

Hierarchical Token Bucket filter (HTB) is the throttling algorithm used in AQuA [2] to share disk I/O between different workload classes. The principle of HTB is to share and differentiate I/O access into a hierarchical structure so that each workload class receives the guaranteed bandwidth. In addition, if not all known workload classes are present at the time, surplus bandwidth are distributed proportionally according to their priority (or weight).

The root node at the top of the tree will contain a predetermined number of tokens *e.g.* according to hard-disk factory specifications. If the disk has an average throughput of 50 MB per second, then the root token rate will be 50K tokens per second (1 token is 1KB of data).

A child node in HTB represents a workload class (FIFO) queue. Each child node has its own bucket of tokens. In order to process a request from its respective queue, the child node must at least contain the number of tokens necessary to process the request. *I.e.* if the request is x KB large, then x tokens will be removed from the bucket in the child node. In addition, x tokens are also removed from the root node.

Unused tokens at the root node are freely available for other classes to take which means extra bandwidth. Even if a workload class node has used all its tokens, a request can still be processed since the root node may have surplus tokens.

Even though HTB has the advantage of sharing bandwidth between different workload classes, it also has some drawbacks. First, setting the max throughput at the root node can be essentially hard since throughput of the disk depends on many other factors (other hardware in the computer). Second, throughput is also quite different whether the request is a read or a write. Having only 1 value to decide the throttling can result in a semi-optimal sharing.

4 Local Level Quality of Service

4.1 Generalized QoS

So far in this report we have mainly looked at two different workload classes, respectively production and maintenance. We have also considered a critical workload class and the properties it brings. Our context is system recovery in large-scale storage systems, and thus we have categorized expected workload into three classes. Recall that in our scenario, a workload class is a generalization of workload requests with common properties.

Despite the fact that our domain is system recovery, our solution also applies in more a general QoS framework. Instead of having one workload class which generalizes all client communication into *production*, it might be more appropriate to further divide the production workload class into one or more subclasses. Although we still specify that production workload class shall receive a given percentage of the I/O bandwidth, further differentiation between sub-classes can improve and enhance the QoS experienced by *e.g.* the outside user.

Another example of useful division of the production workload class could be to give different priority to different IP addresses. Users located geographically far away from the storage system could get their own workload class, so that they are at least guaranteed a decent throughput-rate.

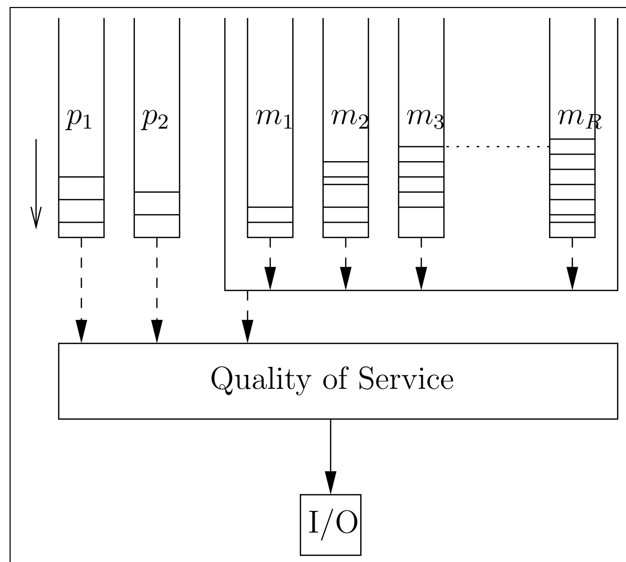

Figure 9: Our proposed model for handling maintenance requests. There are queues for each replication-degree in the system.

The maintenance workload class can also be divided into sub-classes. Our proposed maintenance model, shown in Figure 9, uses a logical maintenance queue. The logical queue consists of multiple queues. Each queue represents how many replicas there are left of an object. *E.g.* a maintenance request regarding an object only replicated once in the system (critical) is placed in workload queue m_1. A request regarding an object replicated twice in the system will be placed in queue m_2, and so on. Therefore, in an R-way primary copy replication scheme, we will have R maintenance queues.

Despite having multiple maintenance queues, we treat it as one single logical queue. When throttling workload against the I/O subsystem, and the next request shall be a maintenance request, we simply check if there are pending requests in the most critical queue, respectively queue m_1. If m_1 is empty, then check if queue m_2 has any pending requests and so forth with queues $m_3, \ldots, m_R$. In other words, the most critical requests will always be processed before non-critical ones.

Our model will simplify implementation regarding how to solve prioritizing of critical requests. Another maintenance approach, but not as equally elegant as our model, could be to have multiple single queues as described earlier in this report. The major drawback with such an approach is that non-critical requests get access while still critical tasks are pending and waiting service time. Even though if we further increase the weight for critical workload, sooner or later other non-critical requests could still be processed while there are still pending critical

requests.

Another important aspect of critical maintenance requests is, when shall a request be treated as critical? This question is fairly easy to answer in a R-way replication scheme. Requests regarding objects replicated twice or more in the system are "normal" maintenance requests, and requests regarding one replica in the system is inevitable considered as critical. There can also be maintenance requests regarding fully replicated objects as well, *e.g.* when the storage system is expanded in the number of nodes. No objects are missing, but the system has become unstable and thus needs recovering.

Our maintenance model solves the decision problem fairly elegant. We don't have to give individual weights to sub-classes of maintenance, but instead assign a weight for maintenance workload as a whole. Since we have one queue for each replication degree of an object, we obtain a good degree of what is considered critical workload.

4.2 Dynamic Tree with Observed Metrics

The Dynamic Tree with Observed Metrics (DTOM) algorithm is our approach to differentiate and throttle requests of different workload classes. The DTOM algorithm is designed to be flexible and gracefully adapt to changes in the workload stream. DTOM stores all observed metrics in a dynamic tree structure which is further utilized in the throttling decision.

Even though this report has its main focus on system recovery and respectively workload classes such as production, maintenance and critical, DTOM can handle numerous different workload classes. DTOM can be used to provide QoS in a multi-workload class system.

Notation	Description
w_i	Weight for class i
x_j^i	A leaf node with level index i and class j
X^i	An inner node with level index i
n	The number of workload classes present at a storage node
X_{n+1}	All observed metrics at a given level
L_{n+1}	Computations based on the left leaf node
R_{n+1}	Computations based on the right inner node

Table 1: Notation used in DTOM.

This section is organized as follows. First we will describe and give concrete examples on how the DTOM tree grows in size. There will first be examples on how DTOM adds workload class 1, 2 and 3 to the tree. Thereafter, we will describe the general case of adding workload class j. Next, we will show how a workload class is removed from the DTOM tree. The notation used in this section is shown in Table 1.

Adding workload class 1, 2 and 3

Assume we have an idle storage node where all workload class queues are empty. The node is waiting for incoming requests. At time 0, workload class queue q_1 receives requests. When there is only one type of workload present at the storage node, the whole disk bandwidth shall obviously be given to the particular workload class (100 %), and thus no throttling is necessary.

At time t_a workload class queue q_2 also receives requests, *i.e.* there are now two queues who got pending requests. When there are at least two workload classes present at the storage node, the disk bandwidth shall be divided between them according to the weighting scheme. In our case, class 1 got weight $w_1 = 1$ and class 2 got $w_2 = 3$. Therefore, workload class 1 shall receive 25 % while class 2 shall receive 75 % of the disk bandwidth. Note that adding up the utilization (in percentage) for all workload classes at a given time, the sum shall be 100 %, and hence we know that all disk bandwidth is utilized.

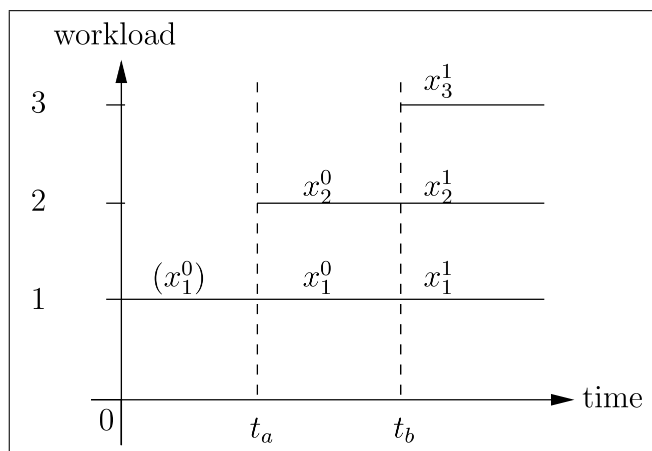

Figure 10: Three different workload class streams at a storage node. From time 0 to t_a, there is only one workload class present at the node (class 1). At time t_a, workload class 2 starts receiving requests, and workload class 3 at time t_b. Each stream is tagged with DTOM notation.

Figure 10 shows the different workload streams, each tagged with DTOM notation.

We see from time 0 to time t_a only workload class 1 is present, and from time t_a to t_b workload classes 1 and 2 are present. This is denoted with x_1^0 and x_2^0. The respective DTOM tree data structure is shown in Figure 11.

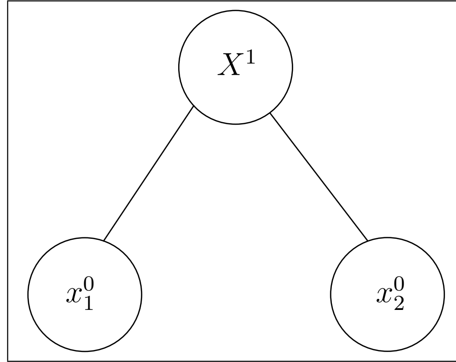

Figure 11: The DTOM tree structure when throttling between workload class 1 and 2, respectively shown as x_1^0 and x_2^0.

x_1^0 and x_2^0 will store observed metrics for the classes. When a request is picked from either of the workload class queues, in this case queue 1 or 2, we record the start time, *e.g.* get time in millisecond (or nanosecond if supported and/or necessary). When the storage node is done with processing the request, regardless if the request was I/O read or write, we record the completion time in the same way as the start time. The time spent, *i.e.* completion time minus start time, is added to either x_1^0 or x_2^0. Therefore, when throttling between class 1 and 2, we have to check the observed metrics, calculated with the weight (formula will be described later), for respectively class 1 and 2.

As shown in Figure 10, workload class queue q_3 receives requests from time t_b. We have now three different workload classes present at the storage node. Class 3 got weight $w_3 = 4$. The disk bandwidth shall now be divided between three classes. With our weighting scheme, this results in a 12.5/37.5/50 % utilization respectively for workload class 1, 2 and 3. If there only were class 1 and 3, the utilization would be 20/80 respectively. Table 2 shows the percentual division for each possible utilization scenario with our weighting scheme.

Figure 12 shows the DTOM tree structure when there are three classes present at the storage node. We have now a new leaf node, x_3^1, which shall store observed metrics for workload class 3. The upper index 1 in x_3^1 means this class is in level 1. X^2 is our new root node.

After the third workload class became present at the storage node, new observed metrics for workload class 1 and 2 will now be stored at the inner node X^1 (inner

Workload	Percentage of class 1 - 2 - 3
1	100 - 0 - 0
2	0 - 100 - 0
3	0 - 0 - 100
1,2	25 - 75 - 0
1,3	20 - 0 - 80
2,3	0 - 57.1 - 42.9
1,2,3	12.5 - 37.5 - 50

Table 2: Percentual division of workload classes according to formula described in section 2.4.

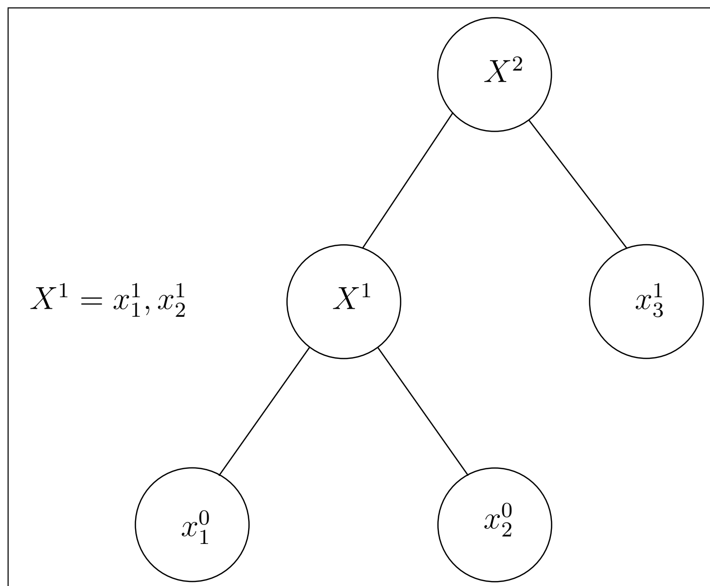

Figure 12: The DTOM tree structure when throttling between workload class 1, 2 and 3, respectively shown as x_1^0, x_2^0 and x_3^1.

child node of root node), respectively as x_1^1 and x_2^1 (bottom leaf nodes). x_1^0 and x_2^0 holds observed metrics when there were two classes at the storage node, but they will still be important in the throttling decision. The complete history for workload class 1 will be $x_1^0 + x_1^1$, and $x_2^0 + x_2^1$ for workload class 2.

The throttling decision will now have to utilize historical information from all levels of the tree. When deciding which workload class is next to obtain disk access, we always start calculations from the root node, in our case X^2. We have to find out if the next class is workload class 3 (right side of X^2) or if the next class is either workload class 1 or 2 (left side of X^2). The following formulas are used in the

decision:

$$X_{n+1} \;=\; \sum_{i=1,n}\; x_i^n \quad (1)$$

$$R_{n+1} \;= \frac{x_n^n}{X_{n+1} \times w_n} \quad (2)$$

$$L_{n+1} \;= \sum_{i=1,n-1} \Big(\frac{x_i^n}{X_{n+1} \times \sum_{j=1,n-1}\; w_j}\Big) \quad (3)$$

First we need to summarize all observed metrics at the level in question using formula (1). This is because we need to know the total disk time when there are 3 classes present. In our case this will be:

$$X_2 \;=\; \sum_{i=1,3}\; x_i^1 \;\; = x_1^1 + x_2^1 + x_3^1$$

We first calculate the right side of the root node using formula (2):

$$R_2 = \frac{x_1^3}{X_2 \times w_3}$$

Calculations for the left side uses formula (3). In our case, this will be:

$$L_2 = \sum_{i=1,2} \Big(\frac{x_i^1}{X_1 \times \sum_{j=1,2} w_j}\Big)$$

$$= \Big(\frac{x_1^1}{X_2 \times (w_1 + w_2)}\Big) + \Big(\frac{x_2^1}{X_2 \times (w_1 + w_2)}\Big)$$

Having both L_2 and R_2, we can check the values against each other. The question to answer is, shall we process a request from workload class 3 ($R_2 < L_2$), or shall we process a request from either workload class 1 or 2 ($L_2 < R_2$). If $R_2 < L_2$, *i.e.* process a request of class 3, then the throttle decision is done. If not, we have to go down one level in the tree (to X^1) and perform both L_1 and R_1.

In a DTOM tree with multiple classes, the algorithm is still the same. We start from the root node and compute L_{n+1} and R_{n+1} for every level until we have found a leaf node ($R_{n+1} < L_{n+1}$). Thus, the algorithm will recursively work its way downwards in the tree.

Adding a workload class j

We have already shown how workload class 1, 2 and 3 were added to the DTOM tree. In this paragraph we will describe when we add workload class j. By adding a new workload class, we mean that the class queue is no longer empty and have

pending requests. The current level of the tree is k. Workload class $j-1$ is the most recent added class. The currently top three levels of the tree is shown in Figure 13 and the different workload streams are shown in Figure 14.

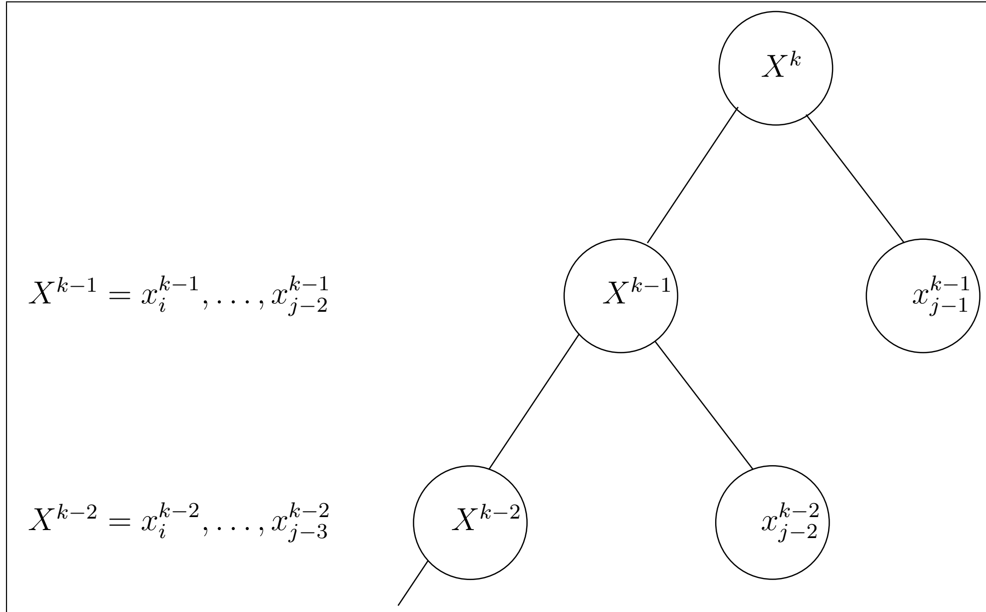

Figure 13: A DTOM tree which currently has a level k and class $j-1$ is currently the newest added workload class.

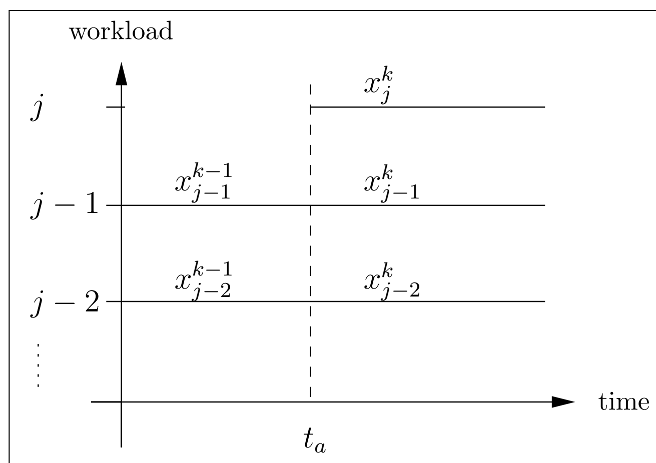

Figure 14: Different workload class streams. At time t_a, workload class j is presented at the storage node (queue no longer empty). Each stream is tagged with DTOM notation to show where it relates in the DTOM tree.

The algorithm for adding a new workload class j is as follows:

1. Make a new root node X^{k+1}.
2. Make a new leaf node x_j^k and connect it to X^{k+1}. New observed metrics for class j will be stored in leaf node x_j^k.
3. Connect the old root X^k to the new root node X^{k+1}.
4. Initialize an array in X^k. New observed metrics for all classes except j will be stored in this array.

X^k is currently the root node. The inner node X^{k-1} contains an array of registered I/O access to all underlying workload classes, *i.e.* in this case $X^{k-1} = \{x_i^{k-1}, \ldots, x_{j-2}^{k-1}\}$. This means that X^{k-1} will store $k-1$ recorded metrics for workload class i to $j-2$. Metrics for workload class $j-1$ is stored at the leaf node x_{j-1}^{k-1}.

As shown in Figure 14, workload class j starts to receive requests at time t_a and shall therefore be included in the tree. Before time t_a, we already have recorded metrics for workload class i to $j-1$ (i to $j-3$ is not shown in the figure) in respectively all levels from 0 to $k-1$. In other words, this is recorded history for all present classes. As explained earlier, recorded history will support the decision when deciding the next workload class to obtain I/O.

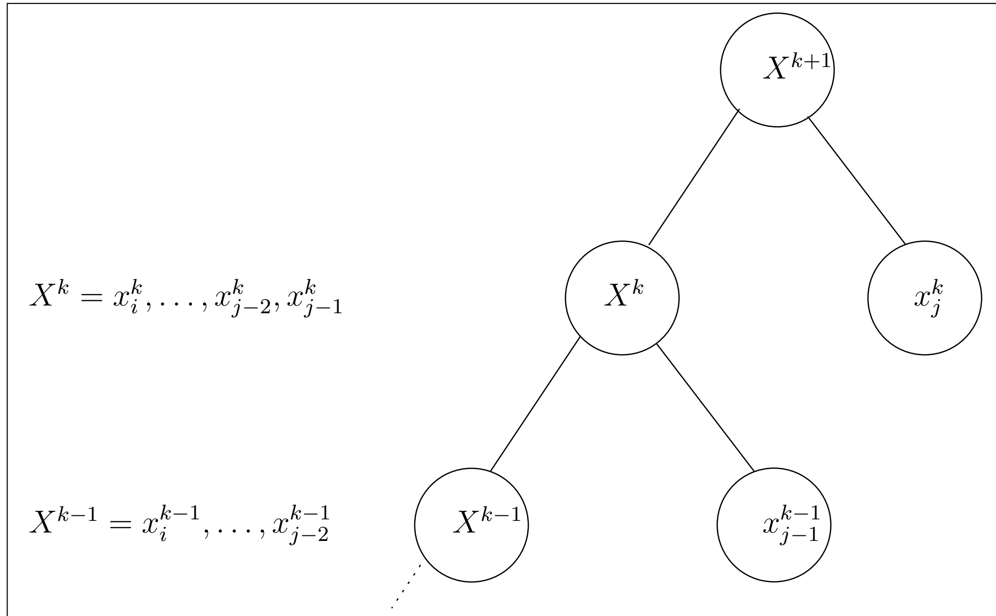

Figure 15: A DTOM tree where the new class j is added at level k (x_j^k). The new root node is now X^{k+1}.

We start by making a new root node X^{k+1} and a leaf node x_j^k which we attach to

X^{k+1}. New observed metrics for workload class j (when there are k levels) will be recorded in leaf node x_j^k. We then attach the old root node X^k to X^{k+1}. X^k is now an inner node while X^{k+1} is the new root node for the tree. New observed metrics for all classes except j will be recorded in X^k. Lastly, we will initialize the array in X^k which will store recorded metrics for $X^k = \{x_i^k, \ldots, x_{j-1}^k\}$. The new tree is shown in Figure 15.

Removing workload class j; the general case

There are two different cases when removing a workload class from the DTOM tree. The first case, and probably the most used one, describes when removing a workload class either in the middle or on the top of the tree. The other case (described later) regards when removing a workload class at the bottom of the tree. In general, removing a workload class means that the respective workload class queue has become empty and should no longer be considered when throttling. Figure 16 shows our initial DTOM tree and Figure 17 shows the different workload streams.

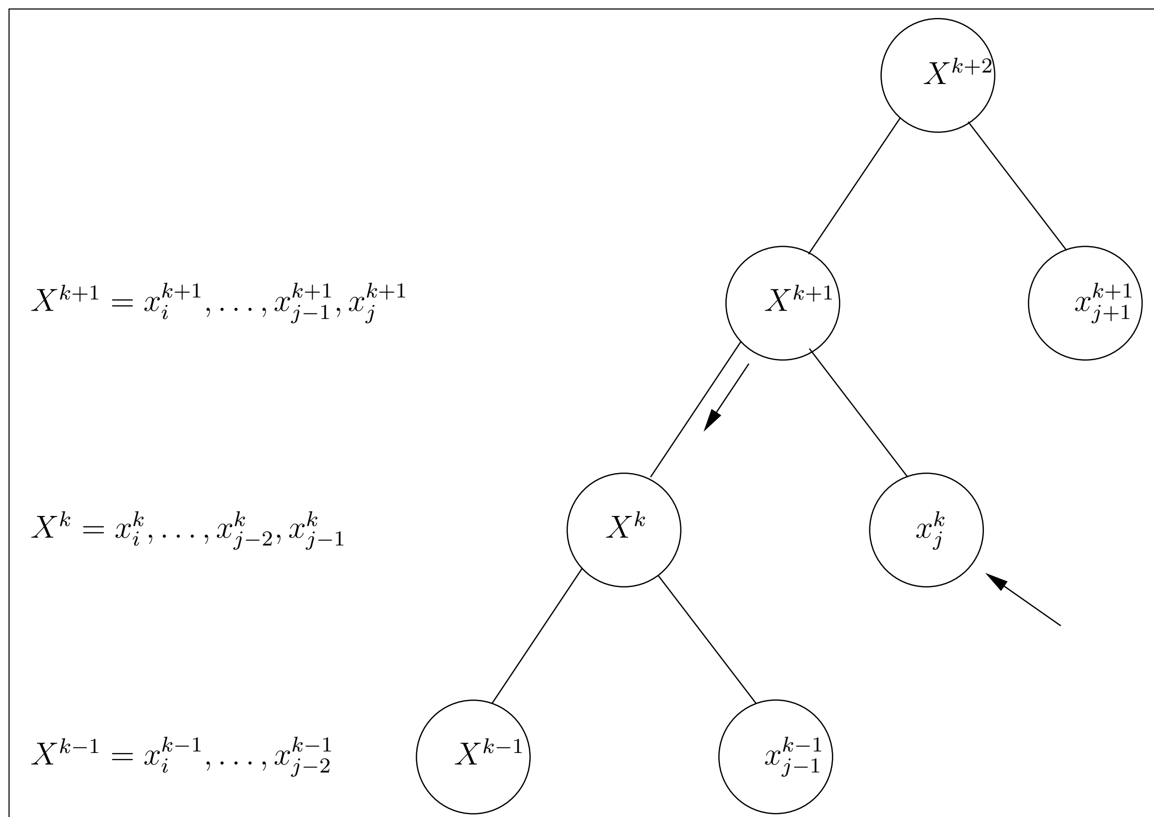

Figure 16: Removing workload class j (leaf node x_j^k) from a DTOM tree. History stored in X^{k+1} and X^k can be added together.

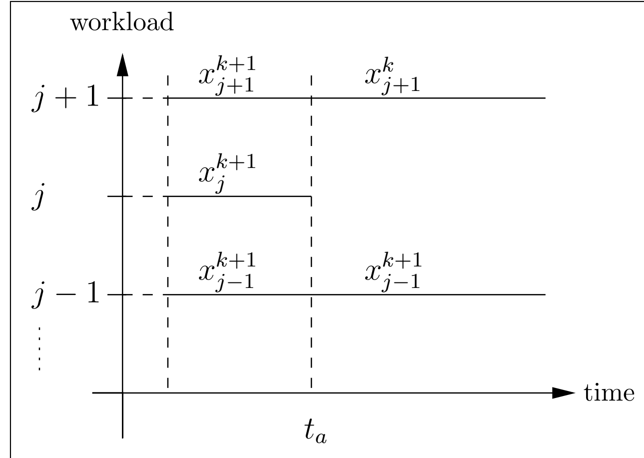

Figure 17: Different workload class streams. At time t_a, workload class j is no longer presented in the system (empty queue). Each stream is marked with DTOM notation to show where it relates in the DTOM tree.

We summarize the algorithm to remove leaf node x_j^k as follows:

1. Remove leaf node x_j^k from the tree.
2. Remove all x_j^R where $R \in \{k+1, \ldots, n\}$. This means remove all history upwards in the tree.
3. Add x_A^{k+1} with x_A^k where $A \in \{i, \ldots, j-1\}$. Since we removed left leaf node of X^{k+1}, we can add up historical values together.
4. Remove inner node X^{k+1} and connect X^k with X^{k+2}.
5. Transform $k+2$ to $k+1$ for all X^{k+2} to X^n.

At time t_a, there are no more pending requests in workload class queue j (not shown in the figure), and therefore, it is no longer necessary to have x_j^k present in the tree. In addition to removing x_j^k, we also remove all occurrences (recorded metrics) of class j upwards in the tree, *i.e.* history of class j stored at inner node X^{k+1} to X^n. We can remove the information because all history regarding class j is no longer necessary for future throttling.

Further, history stored at X^{k+1} can now be added up with history stored at X^k. The reason for this is when deciding which class is next to receive I/O, and the algorithm has worked its way from the root node and downwards to X^{k+1}, it will only summarize values in X^{k+1} and X^k, since x_i^k is removed.

When values in X^k and X^{k+1} are added together, we can safely remove X^{k+1}

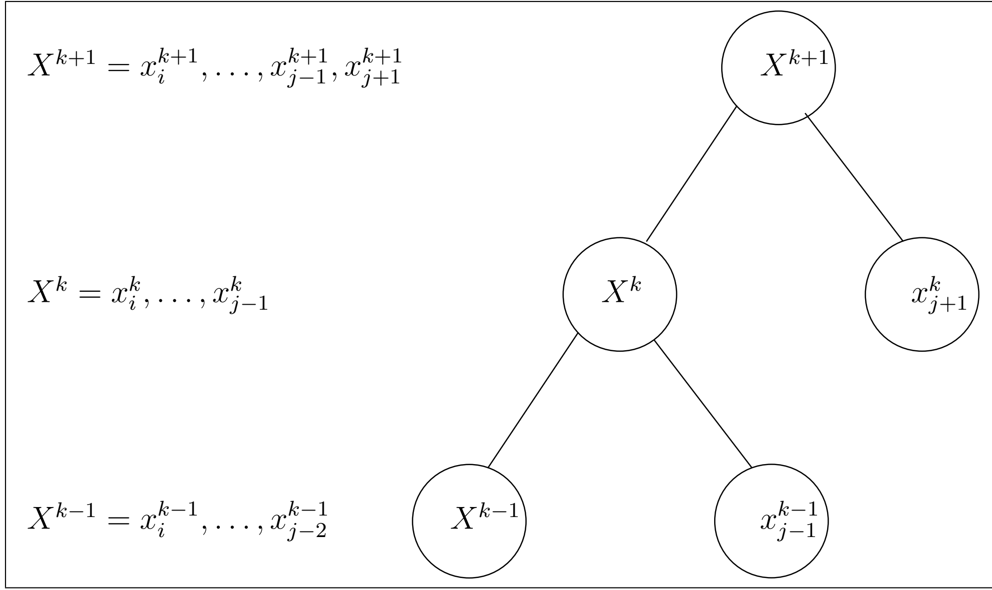

Figure 18: The resulting DTOM tree after leaf node x_j^k (not shown) has been removed.

without loosing any important history regarding other classes. Removing X^{k+1} results in the tree being inconsistent regarding registered levels on all nodes above X^k, both inner and leaf nodes (next node upwards after X^k is at the moment X^{k+2}). Thus, we need to transform $k+2$ to $k+1$ on all nodes starting from X^{k+2}. The final tree structure is shown in Figure 18.

Removing workload class i; a special case

The above paragraph describes how DTOM removes workload classes from the tree, either if the workload class was connected in the middle or at the top of the DTOM tree. This paragraph, however, describes when removing a workload class from the bottom of the tree. Although the different cases are similar, there are still some changes which require a slightly different approach. Figure 19 shows our initial DTOM tree and Figure 20 shows the different workload streams.

We summarize the algorithm to remove a bottom leaf node x_i^0 as follows:

1. Remove leaf node x_i^0 from the DTOM tree.
2. Remove all x_i^R from X^2 to X^n where $R \in \{2, \ldots, n\}$. This means to remove all history of class i upwards the tree.
3. Reset and reconnect leaf node x_{i+1}^0 from inner node X^1 to X^2.
4. Remove inner node X^1 from the tree.

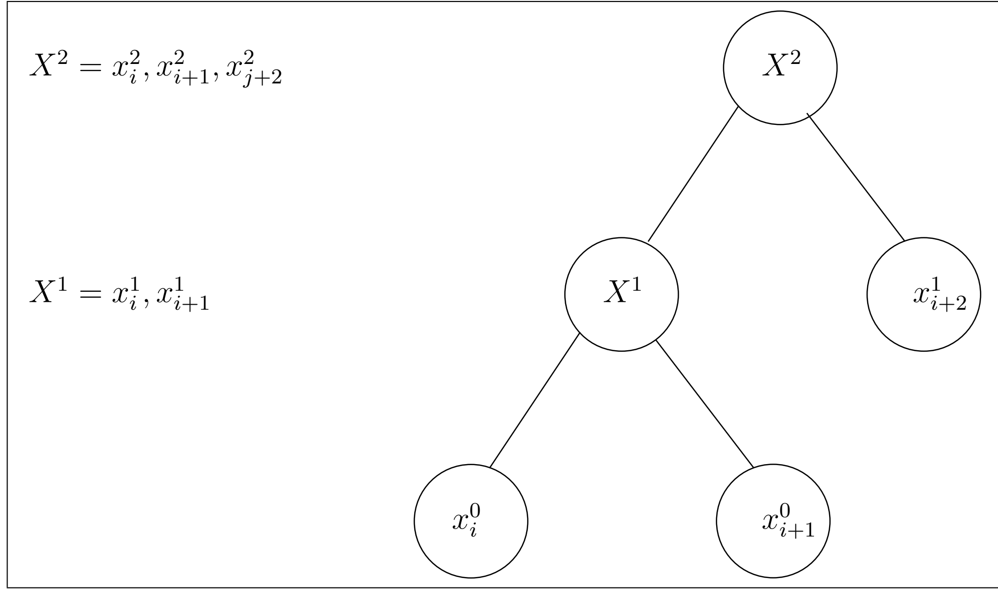

Figure 19: A DTOM tree where node x_i^0 and x_{i+1}^0 are the bottom leaf nodes. Class i is no longer present in the system (x_i^0), and is going to be removed.

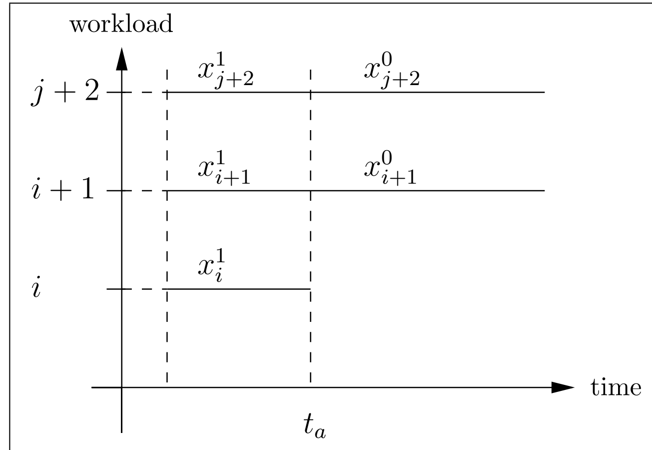

Figure 20: Different workload class streams. At time t_a, workload class i is no longer presented in the system. Each stream is marked with DTOM notation to show where it relates in the DTOM tree data structure.

5. Transform all nodes (except x_{i+1}^0) from k to $k-1$.

If either class i or $i+1$ (respectively x_i^0 or x_{i+1}^0 at the bottom of the tree) does not have any more pending requests in their respective workload class queues, they must be removed from the DTOM tree. In this case, assume there are no more

pending requests of class i.

We first disconnect and remove leaf node x_i^0 from the tree since this historical data is no longer necessary. In addition, history stored upwards in the tree for workload class i must also be removed. This means remove all history for the class in question at all inner nodes in the tree (same as with removing a workload class in the middle of the DTOM tree).

The remaining class at the bottom, x_{i+1}^0, must reset its recorded metrics based on what is stored at the inner node X^1, and then connect to node X^2 instead of X^1. This is because history at bottom level is no longer necessary for either of the classes, however metrics stored on X^1 is important when throttling disk I/O between class $i+1$ and $i+2$. They have a history together.

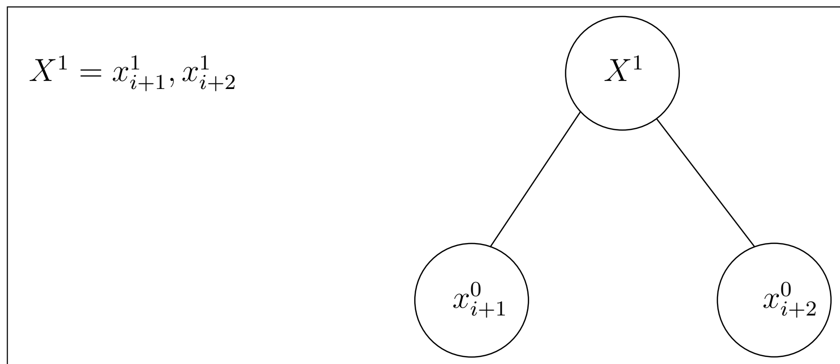

Figure 21: A DTOM tree where node x_i^0 (not shown) was removed. x_{i+1}^0 and x_{i+2}^0 are now the new bottom nodes.

When leaf node x_{i+1}^0 is connected to inner node X^2, we can safely remove inner node X^1 since useful information is already stored. At the moment, our tree will have an inconsistency regarding level indexes, and thus we need to transform indexes in most of the nodes. This means to first transform leaf node x_{i+2}^1 to x_{i+2}^0. Class $i+1$ and $i+2$ are now the bottom leaf nodes in our tree. Lastly we need to transform all inner nodes and their connected leaf nodes from X^k to X^{k-1}. Our final DTOM tree structure is shown in Figure 21.

Summary

To summarize, we store all disk occupancy times for all workload classes which is fully organized in the DTOM data structure. We know how much I/O each class has obtained at any time, regardless of how many classes there are in the structure, and we can always know which class is next to receive I/O access. DTOM handle both addition and removal of workload classes gracefully without risking any loss of data.

Note that DTOM will not give each class the *exact* amount of I/O. By this we mean that on average, DTOM will throttle requests to disk I/O according to the predefined weight of the class. However, a possible drawback with DTOM is when a request is significantly larger than the average request. Since the observed metrics is updated after processing a request, it does not take any consideration to how large next request is. This means if the throttling specification demands that *e.g.* production request shall never occupy less than a given percentage of the bandwidth, then this approach can not make this guarantee.

Chapter 4.3 shows a lower bound proof regarding DTOM and chapter 5 shows experimental results of DTOM compared with WRR.

4.3 Lower Bound on DTOM QoS

In this section, we want to prove the effectiveness of DTOM, or more importantly, the worst case. We state that the worst DTOM can perform, *i.e.* the variance in actual throttling, is the max size of a request.

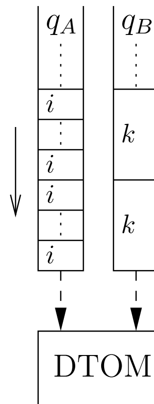

Figure 22: Two queues filled with requests. Queue A is filled with requests of (min) size i and queue B is filled with requests of (max) size k.

Assume the minimum request size is i, and the maximum size is k. For the proof, consider two FIFO workload class queues where the first queue, q_A, only contains requests of size i and the other queue, q_B, contains requests of size k. The workload queues filled with requests are shown in Figure 22.

Translating our example to DTOM, we have the same tree as shown in Figure 11 (note the difference in workload class indexes). Since DTOM checks already received I/O before deciding the next request, the worst case will thus be the max

size. As an example, if DTOM has processed a request from queue q_B (with size k), the algorithm will further process requests of queue q_A until class A has received its share. Hence, the worst case is the max size of a request.

5 Experimental Results

This section shows experimental results for the DTOM algorithm introduced in section 4.2. DTOM is implemented in a discrete-event simulator [35]. A discrete-event simulator is controlled by an event scheduler, which is the heart of the simulator. It decides what to do next and when to do it.

To support the event scheduler, a global time variable is used. Each event will be triggered by the time, *e.g.* at time 0 the system is populated with client requests.

Yahoo! Technologies Norway (YTN) is developing a simulator based on the SimGrid toolkit [36]. SimGrid is a library that allows us to simulate distributed applications in a heterogeneous distributed environment. The simulator is adjusted and configured to simulate a VDS [2] system. It is entirely written in C. Experimental results in this report are produced with the use of YTN's simulator.

This chapter is organized as follows. In section 5.1 we present our testing methodology. Section 5.2 to 5.4 show experimental results for our three test scenarios. Lastly, in section 5.5 we validate our DTOM algorithm against the requirements described in chapter 2.6.

5.1 Methodology

Our experimental test system consists of three clients, one distributor and one storage node using a VDS context. Having only one distributor and one storage node is sufficient, since we want to isolate observed throughput for different workload classes on a local level. Because of having only one storage node, we use a primary-copy replication strategy with only one replica. Thus each request/document generated by the clients will only be replicated one time in the system. Note that such a configuration is not adequate in real implementations, but in our test case it will give us isolated test data regarding our throttling algorithm.

Each of the three clients will generate and send put (I/O write) requests to the storage node (through the distributor). Each request is tagged with the client

[2] See appendix A for more information about VDS

ID, *i.e.* we will have three different workload classes in the system (each client represents a workload class). When the storage node has committed a request to disk, an acknowledgment which describes disk occupancy is sent back to the client. With the acknowledgment we can compute the client throughput at the storage node, and more importantly, utilization in percentage of the disk for each client/workload class.

The three client workloads have different weights at the storage node. Our weighting scheme is 1/3/4 respectively for client 1, 2 and 3. This means that client 3 shall receive more disk bandwidth (when present) compared to the other classes.

If we transform our weighting scheme to percentual division according to methodology described in chapter 2.4, we will have several cases. Recall Table 2 (in chapter 4.2) to see the differences in percentage.

There are three different scenarios we want to test our DTOM algorithm. The next subsections show results for the different test scenarios.

5.2 Scenario 1: Throttling Disk I/O with uniform workload

Our first scenario consists of testing DTOM against the well-known and widely used WRR when three clients are generating and sending uniform requests to the storage node. The request sizes will vary between 5 KB to 10 KB.

First, we want to see how DTOM and WRR utilize disk bandwidth. At time t_0 client 1 starts sending requests to the storage node. Obviously, client 1 shall occupy the whole disk bandwidth. At time $t_a = 25$ client 2 starts to send requests and bandwidth are divided between them. At time $t_b = 50$ the third client starts sending requests, and thus the algorithms have to throttle disk bandwidth according to the weighting scheme.

In addition to see how the algorithms utilize the disk bandwidth between workload classes, we also want to see the variation of the utilized bandwidth, *i.e.* how close will the occupied bandwidth be according to the values shown in Table 2. The workload streams are shown in Figure 23.

As shown in Figure 24 and 25, both WRR and DTOM utilize the whole disk bandwidth which is as intended. At time $t_a = 25$ client 2 starts to generate requests. We can see from the figures that the utilization for class 2 stabilizes at approximately 75 %, and at 25 % for class 1.

Lastly, at time $t_b = 50$ the algorithms have to throttle between three workload

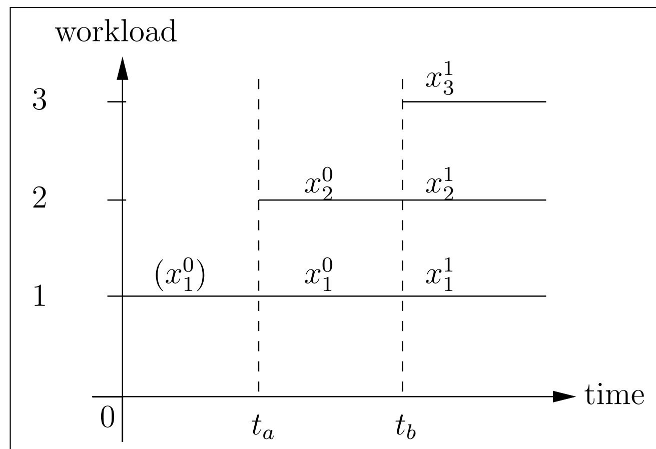

Figure 23: Three different workload class streams. At time t_a, workload class 2 starts sending requests, and workload class 3 at time t_b. Each stream is tagged with DTOM notation.

classes. Workload class 3 ($w_1 = 4$) occupy approximately 48-52 % of the disk bandwidth, class 2 ($w_2 = 3$) occupy approximately 36-40 %, and class 1 ($w_1 = 1$) occupies the remaining 12-13 % bandwidth. Both algorithms throttle the disk bandwidth gracefully when new classes are added. The percentual divisions of the classes are in accordance to Table 2.

In Figure 24 (WRR) we see that there is a small variation in the graphs indicating disk bandwidth (intended sharing plus/minus 3) in contrast with results shown in Figure 25 (DTOM) which is much more stable (intended sharing plus/minus 1). This means that DTOM does throttle disk bandwidth slightly better than WRR, even when the request sizes are respectively small.

Next section will show experimental results when request sizes is increased from 5-10 KB to 5-10 MB.

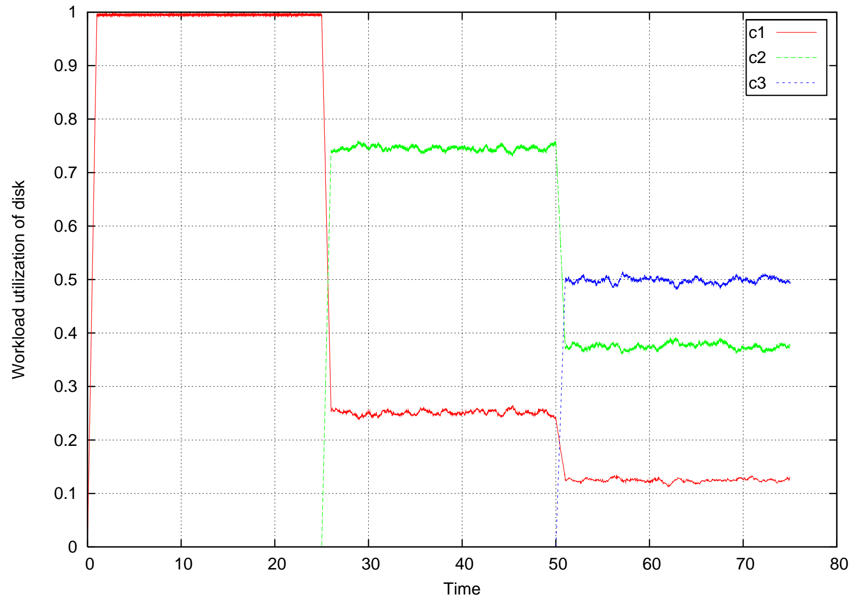

Figure 24: Observed utilization for different workload classes when using WRR and request sizes from 5 KB to 10 KB.

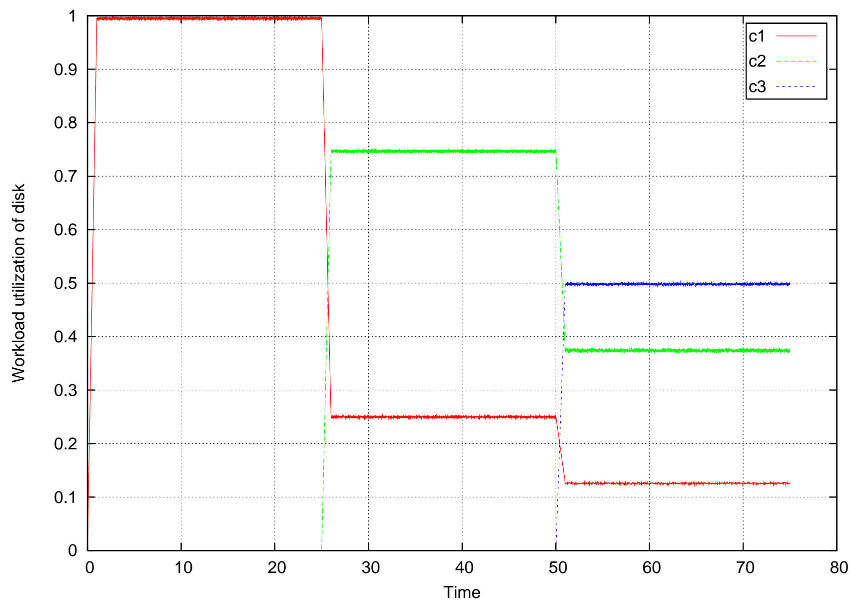

Figure 25: Observed utilization for different workload classes when using DTOM and request sizes from 5 KB to 10 KB.

5.3 Scenario 2: Throttling Disk I/O with non-uniform workload

Scenario 2 is almost identical with scenario 1, however the request sizes will now be between 5 MB to 10 MB. Non-uniform request sizes can affect the throttling since we can expect higher short-term load peaks. The workload streams from client 1, 2 and 3 are shown in Figure 23, but now with $t_a = 2500$ and $t_b = 5000$. We can not have the same time frame for scenario 1 and 2 since they have different request sizes. Having time values $t_a = 25$ and $t_b = 50$ in scenario 2 would lead to an inaccurate test result, as the same with having $t_a = 2500$ and $t_b = 5000$ for scenario 1. We are not testing DTOM with 5-10 KB against DTOM with 5-10 MB, but in particular, testing DTOM against WRR with the different request sizes.

As with scenario 1, we see that both algorithms (WRR and DTOM shown in respectively Figure 26 and 27) does throttle roughly close to the intended sharing shown in Table 2.

While scenario 1 had small variations in the actual throttling (plus/minus 3 for WRR and plus/minus 1 for DTOM), variations in this test scenario is much more noticeable. In Figure 26 (WRR) we can see, for all workloads, that the variation is widely fluctuating. Although the average can be close to intended values in Table 2, however disk bandwidth guarantees can not be made, or at least one have to carefully consider which guarantees one may or may not want.

Figure 27 shows how DTOM throttle disk bandwidth when request sizes are varying from 5 MB to 10 MB. DTOM handle the larger request much more gracefully compared to WRR, and it is much easier to make certain disk bandwidth guarantees. In other words, DTOM provides better QoS guarantees than WRR.

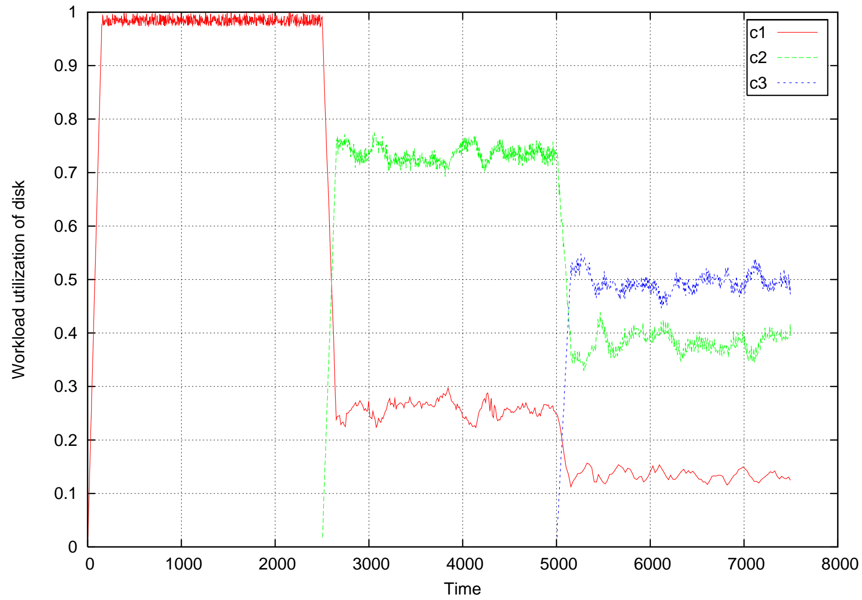

Figure 26: Observed utilization for different workload classes when using WRR and request sizes from 5 MB to 10 MB.

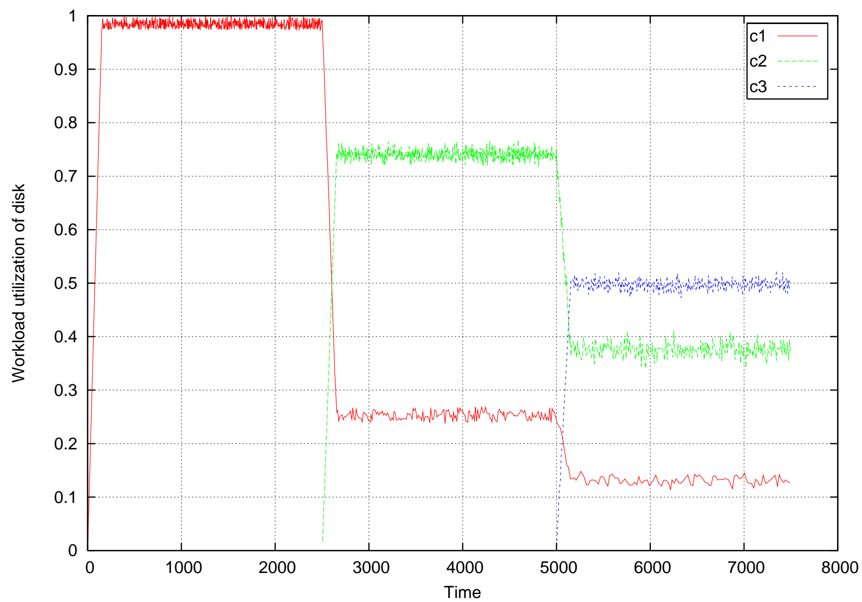

Figure 27: Observed utilization for different workload classes when using DTOM and request sizes from 5 MB to 10 MB.

5.4 Scenario 3: Addition and Removal of Workload Classes

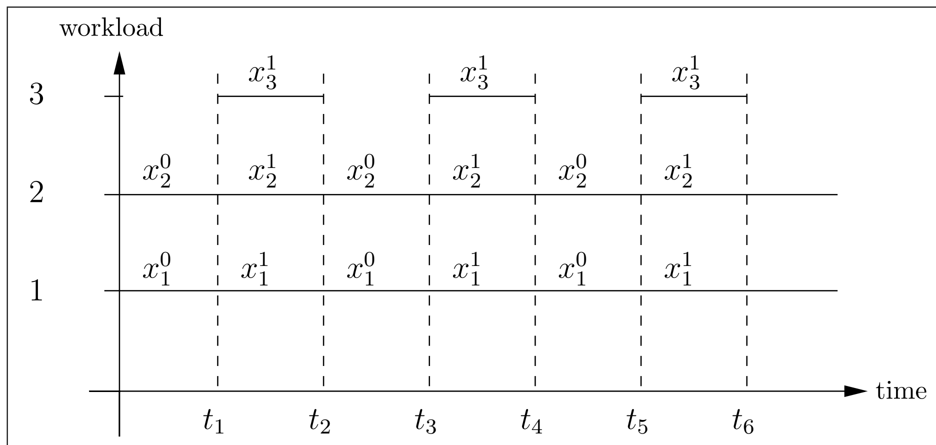

Figure 28: Each stream is tagged with DTOM notation.

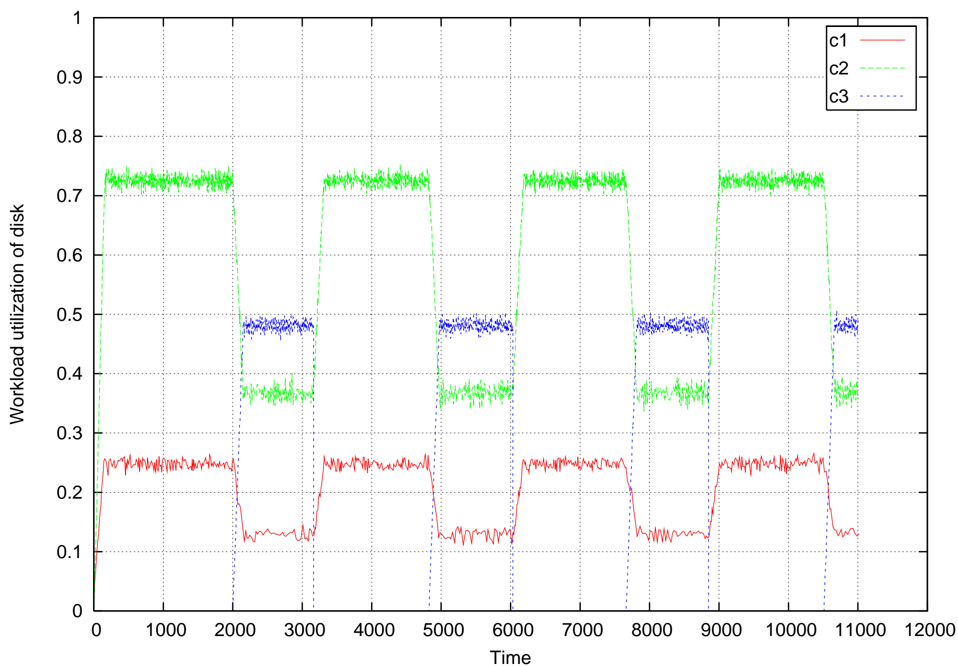

Figure 29: Observed throughput when client 3 sends requests in interval.

Scenario 3 consists of having the third client generating workload in intervals. We want to see how DTOM handle frequent addition and removal of workload classes, and further observe how it will share the disk bandwidth. Request sizes

are unimportant in this scenario, however it is set to vary between 5 MB to 10 MB. Figure 28 shows the different workload streams in scenario 3.

As shown in Figure 29, from time $t = 0$ to $t = 2000$ there are only two workload classes present in the system, respectively client 1 and 2. At time $t = 2000$ the third client starts to generate and send 500 requests to the storage node. After the 500 requests have been sent, client 3 sleeps additional 2000 time units, before the process starts over again.

As long as only client 1 and 2 are present in the system, DTOM give workload generated by client 1 approximately 25 % of the disk bandwidth while client 2 receives roughly 75 %. When the workload generated from client 3 is present, the workload class with highest weight, it occupies approximately 50 %, while the two others occupy close to 37.5 % and 12.5 %. This is in accordance to intended values in Table 2.

5.5 Requirement fulfillment

In chapter 2.6 we defined requirements in order to have a successful recovery of a storage system. With the use of DTOM, some of the requirements are fulfilled, while others are not fulfilled.

Requirement I (Availability of data during a recovery process) and II (Class-based workload sharing) can be fulfilled with the support of DTOM. Experimental results showed that we can isolate and give individual bandwidth guarantees to individual workload classes. Although our experimental results showed a more generic workload scenario, it still applies in a recovery scenario as well. As long as production and maintenance workloads are tagged (and weighted) accordingly, DTOM will make sure each workload class gets the intended disk bandwidth guarantee.

In addition, employing our maintenance model described in section 4.1, we also solve the problem of critical requests. The model will make sure the most critical requests gets prioritized over not so critical requests. Using both the maintenance model and DTOM in a recovery scenario, we know that critical and regular maintenance requests will not block production requests, however an administrator have to carefully choose proper weights for the workload classes.

The third requirement (Task scheduling to avoid network bandwidth conflicts) is not fulfilled, since our proposed solution provides only QoS on a local level, and does not redirect requests to other nodes.

Requirement IV (Barrier functionality) can be fulfilled with our proposed solution. We can have the system to generate a barrier request whenever a barrier

functionality is needed. When DTOM picks a request which is a barrier request, the queue will be "blocked" until all other queues are either empty or blocked. When there are no more requests to process before the barrier, all blocked queues gets unblocked.

The last requirement (Performance metrics) is also fulfilled. Since we can have clients to generate different types of workload, we can measure their throughput against the particular storage node. Combining throughput for all clients, we can calculate the utilization for each workload class of the disk bandwidth. Thus, we can validate that our proposed solution divides bandwidth in accordance to the weighting scheme.

6 Conclusion

6.1 Contributions

In this thesis we have conducted a thorough research on both how and where to utilize QoS regarding a system recovery process in a large-scale distributed storage system. We have identified several different types of workloads that storage systems are exposed to, and hence we categorize those with similar properties and priorities into individual workload classes. Our coarse classification consists of production, maintenance and critical workload.

We have carefully explained the difference between local and global level QoS approaches, however the focus in thesis have extensively been on the local level. A local level QoS approach does not require any centralized stored information nor any particular communication between nodes. Instead, it will throttle system resources in order to provide local QoS guarantees to different workload classes. We also identified requirements for a successful system recovery.

In order to elegantly handle the different degrees of critical workload, we designed a system recovery model. The model aims to improve handling and selection of the most critical workload over less important workload. In our context, the model will always select critical requests over regular maintenance requests. We also gave a description of how to classify critical workload.

Our main contribution in this thesis is the design and implementation of the DTOM algorithm. DTOM provides local level QoS by throttling system resources between different workload classes. Each workload class may have different priorities (weights). Since DTOM store and utilize historical information (observed metrics) in the throttling decision, it will always select the correct workload class

which is next to obtain system resources. We provided a detailed explanation of the algorithm and we also proved its effectiveness by a theoretical proof.

Experimental results were obtained with a simulator. We compared DTOM against another well-known and widely used throttling algorithm, WRR. DTOM did undoubtedly outperform WRR. We also did experiments on how DTOM handles multiple additions and removals of active workload classes. Again, DTOM was proven to be successful. Therefore, we can certainly conclude that DTOM can be employed to guarantee class-based QoS at the local level. Albeit that our context is system recovery, DTOM can still be used in a any general class-based QoS approach.

6.2 Future work

Throughout this thesis, we have mainly considered write requests, and hence treated all requests accordingly. Extensive experimenting with multiple read and write requests can further verify the effectiveness of our algorithm DTOM. In addition, significantly expanding and scaling up the experimental test system will give the possibility to aggregate data from numerous storage nodes. With aggregated data we could verify and study how a local QoS approach can be used to attain global QoS.

We have assumed that all our queue data structures employ a FIFO queuing discipline. There is no doubt that FIFO is well fitted for queuing structures regarding production workload, *i.e.* workload generated by the outside environment. This is because we usually have difficulties predicting user access patterns, and FIFO would be most fair.

However, maintenance workload generated by the system itself is far easier to predict and hence control compared to production workload. We identify this as a concrete scheduling problem. First, if we can redirect requests to less-loaded storage nodes instead of saturated nodes, the overall performance will be increased. Second, a “correct” scheduling of requests will avoid network bandwidth conflicts. Again, this will improve the overall performance. The scheduling problem has received extensive research, and therefore, we could most likely discover principles which can be employed in our context.

Instead of only testing DTOM against WRR, we could test DTOM against other scheduling algorithms as well. More research on other scheduling algorithms and how they can throttle system resources in our context, can further verify the effectiveness of DTOM.

A Vespa Document Storage

Vespa Document Storage (VDS) is a large-scale distributed storage system developed by Yahoo! Technologies Norway (YTN). VDS is designed to provide high-availability, survivability, high-performance, and scalability to data-intensive applications running on inexpensive commodity hardware.

A.1 Documents in VDS

User data in VDS are stored as "documents", where a single document will have predefined fields with given data types. Fields and data types for a document is specified by a *search definition*. See Figure 30 for an example.

```
1.  document music {
2.      field title type string {
3.      }
4.      field artist type string {
5.      }
6.      field year type int {
7.      }
8.  }
```

Figure 30: Example of fields and data types for documents.

On the first line in Figure 30 is the declaration of the document type. In our example the document type is *music*. The music data got three defined fields. The two first types are strings and the last type is int.

In addition to fields and data types, each document has an unique *document ID* which also is a URI for the document. A URI is represented as a string, and must fulfill the properties for a defined URI scheme. Common for all URI schemes are the fields *namespace* and *user specified part*. The namespace is to avoid naming collisions, and therefore, many different VDS users can share the same VDS cluster and still avoiding that user data is overwritten which is caused by identifier collision. The user specified part has a more unrestricted usage compared to namespace. Users are free do specify whatever they want in order to name and identify their documents accordingly.

There are currently three defined URI schemes in VDS:

doc:<namespace>:<user-specified>
The doc scheme is the most generic and common scheme. Using this scheme will map documents to random locations and gives the best distribution across all nodes in the system.

groupdoc:<namespace>:<groupname>:<user-specified>
This scheme introduces group name as part of the URI. Documents with the same groupname in the URI will, in contradiction with doc scheme, be placed close to each other in VDS. This will increase the efficiency of document retrieval within the same group.

userdoc:<namespace>:<userid>:<user-specified>
The userdoc scheme introduces a user identifier (userid). This is a 64 bit number. As same as with group doc, documents with userdoc scheme are stored close to each other for more effectively retrieval. Although the name exists for legacy reasons, a user can use this scheme to group documents using a number.

A.2 VDS Architecture

The VDS architecture consists mainly of four individual parts; clients, distributors, storage nodes and the fleet controller. How the different components interact with each other is shown in Figure 31.

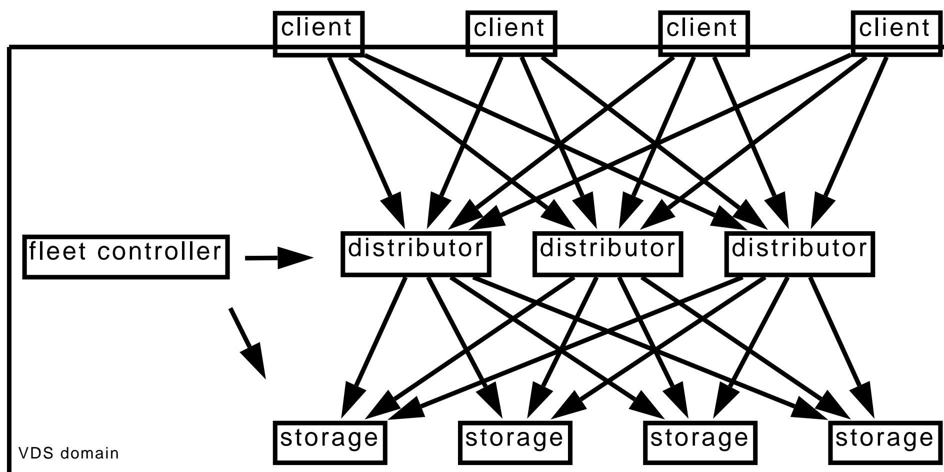

Figure 31: Overview of Vespa Document Storage architecture.

Clients communicate only with distributor nodes and never directly against storage nodes. When clients need to communicate with a VDS cluster, they can use a client

library, *e.g.* the *Document API*. The Document API is currently implemented in C++ and Java.

The client utilize information from the fleet controller, the global cluster state, when deciding on which distributor it shall communicate with. More precisely, *e.g.* with a document put (write) operation, the client will calculate which bucket the document shall be placed in and hence which distributor is responsible for that particular bucket. The cluster state supports this calculation.

The client does not need to periodically query the VDS cluster for an updated cluster state, nor will the fleet controller broadcast updates to the clients. However, when a client query the wrong distributor, the distributor will respond back to the client with his cluster state. The client will utilize the newly received cluster state and re-send the query. This simplifies the client logic and reduces the need for more advanced client configuration.

A VDS cluster also contains at least one *distributor*. The distributor is responsible for buckets in the system. If there are more than one distributor, then each distributor is responsible for a non-overlapping set of buckets. What distributor is responsible for which bucket, is calculated with a deterministic algorithm called ideal state algorithm.

Distributors can be seen as an abstraction layer between clients and storage nodes. Many operations executed by a distributor is completely invisible for the client. The effect is that client logic is further simplified and reduces the amount of client configuration, since they will see the system with a constant number of distributors and buckets in the VDS cluster.

Distributors are responsible for all bucket operations. Such operations can be ensuring replication degree, synchronization and bucket splitting. Ensuring replication degree means to make sure that each bucket is replicated according to the replication scheme, *e.g.* after one or more nodes have been added or removed from the system. Synchronization means to ensure that all bucket replicas contain the same data, *i.e.* consistency of data. Bucket splitting is necessary when a particular bucket has received too much data and become to large.

The *storage* node is the node with least logic. Their main responsibility is to store buckets on physical hard-disks. The storage node is passive and will only perform operations initialized by the distributor.

The last of the main components in a VDS cluster is the *fleet controller*. Since the fleet controller is responsible for maintaining the global system state, there can only one fleet controller for each cluster. This is to avoid inconsistency with the cluster state.

Each distributor and storage node in the VDS cluster will periodically be asked for their system state. Based on gathered information, the fleet controller can broadcast the the global system state to all nodes in the cluster whenever there there is a change that require actions.

If the fleet controller encounters a problem and becomes unavailable, the VDS cluster will continue to work as normal. However, if one of the distributors or storage nodes become faulty, all operations against it will also fail since no cluster state will be broadcasted. Note that the fleet controller does not have any persistent state, hence any other node can easily be configured as a replacement.

A.3 Buckets

Instead for managing and handling documents directly on storage nodes, VDS divides the document space into buckets. A bucket is a form of data unit. Although there can be millions of documents in a storage cluster, there will be a fixed number of buckets. A distributor will never perform any actions directly on a single document stored on a storage node, but instead it will act on a bucket level. Meta-data for each bucket is kept in memory by the distributor.

It is much easier to maintain a fixed number of buckets instead of millions of documents. *E.g.* in a system state change where a storage node is lost due to hardware failure, it is easier to calculate which buckets was on the failed storage node and hence initiate replication, instead of traversing many millions of documents deciding whether or not the document was affected by the loss.

How a document is mapped on a bucket, is determined by the document ID (URI). In some cases, one may wish to store documents in the same bucket (groupdoc-scheme) for faster access. This is specified in the URI. However, if this is not specified, *e.g.* using the doc-scheme, a pure hash value is calculated by the formula $md5(document_id) \bmod num_buckets$ deciding which bucket the document shall be placed in.

On an implementation level, each bucket consists of several files called slotfiles. Each slotfile has a default size of 10MB, however the size is configurable. The size and the slotfile should reflect the average expected size of documents that will be stored in the system. The point is that several documents shall fit into one slotfile. The relationship between documents, buckets and slotfiles is shown in Figure 32.

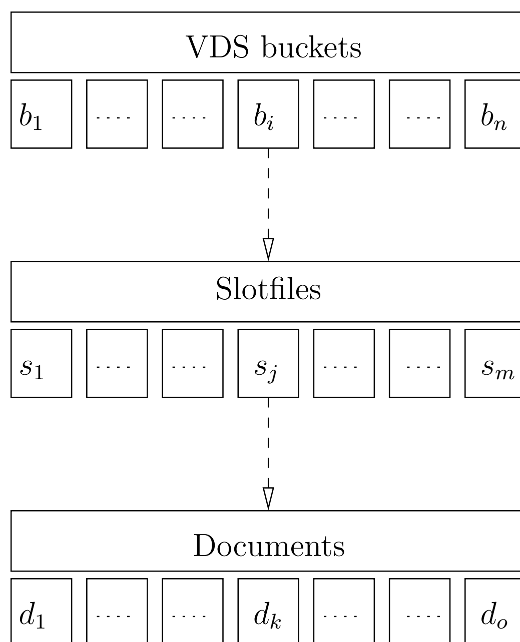

Figure 32: The relationship between buckets, slotfiles and documents in VDS.

A.4 Recovery and cluster update

When a VDS cluster encounters a failure, *e.g.* such as a storage node not responding, it is the fleet controller who is responsible to detect the failure (*e.g.* heart beat monitoring). The fleet controller uses four different states for nodes and their status; up, maintenance, retirement and down. When a node is set to up, it is fully operational. A node with state maintenance means that the node is temporarily down, but is expected to return shortly. If the node is a storage node, the system will not initiate replication of data on it. The retirement state means that the node is expected to be removed from the system, and thus the system will start to replicate data off the node to other (replacement) nodes.

E.g. if a distributor node is set to state down, all buckets being maintained by the node will be transferred to other distributor nodes using the ideal state algorithm. Since all distributor nodes know the algorithm and the global system state, they can determine which buckets they are responsible for (and which they aren't).

If the failing node is a storage node, then each distributor will remove the failed storage node from their internal mapping of all buckets. Further, the distributor will calculate new whereabouts for their buckets, and then populate the particular storage nodes with physical data. The distributor will then calculate the current

ideal state, based on the available number of storage nodes, and hence generate a list of requests that will put the system into ideal state again. Once the list is fully generated, the distributor will immediately start carrying out the tasks.

Let us further say that bucket b_1 shall be placed on storage node s_1, s_2 and s_3 in order to balance the system. The ideal state algorithm calculated s_1 to be the primary copy of the bucket, and we further assume it is also the most recent replica of the bucket. The distributor will then issue a replication command to s_1 to start replicating the bucket to s_2 and s_3. More precisely, s_1 will send all bucket data to s_2 which will migrate the data locally and further forward it to s_3. When s_3 is done with migrating bucket data, it will send an acknowledgment message back to s_2 confirming the migration. s_2 will also send an acknowledgment back to s_1 which eventually will send the acknowledgment back to the distributor that initiated the replication. This chain of messages floating between the particular storage nodes is shown in Figure 33.

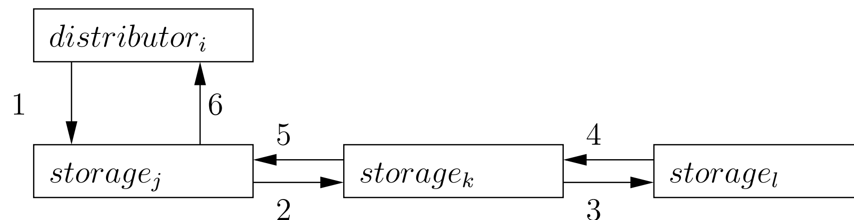

Figure 33: The message chain in a VDS reorganization.

However, if one of the storage nodes in the chain fail to migrate bucket data, the whole replication transaction is aborted, and the error is reported back to the distributor which in turn will take action based on the error message.

References

[1] S. Weil, A. Leung, S. A. Brandt, and C. Maltzahn, “Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters,” in *PDSW ’07: Proceedings of the ACM Petascale Data Storage Workshop*, 2007.

[2] J. Wu and S. A. Brandt, “The design and implementation of AQuA: an adaptive quality of service aware object-based storage device,” in *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2006, pp. 209–218.

[3] S. A. Weil, *Maximizing OSD performance with EBOFS, Technical Report SSRC-04-02*, University of California, Santa Cruz, Mar 2004.

[4] S. A. Brandt and J. Wu, "Providing quality of service support in object-based file system," in *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[6] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: controlled, scalable, decentralized placement of replicated data," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing.* New York, NY, USA: ACM, 2006, p. 122.

[7] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do internet services fail, and what can be done about it," 2003. [Online]. Available: citeseer.ist.psu.edu/oppenheimer03why.html

[8] S. D. Yao, C. Shahabi, and R. Zimmermann, "Broadscale: Efficient scaling of heterogeneous storage systems," *Int. J. on Digital Libraries*, vol. 6, no. 1, pp. 98–111, 2006.

[9] A. Goel, C. Shahabi, S.-Y. D. Yao, and R. Zimmermann, "Scaddar: An efficient randomized technique to reorganize continuous media blocks," *icde*, vol. 00, p. 0473, 2002.

[10] Z. Zeng and B. Veeravalli, "On the design of distributed object placement and load balancing strategies in large-scale networked multimedia storage systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 3, pp. 369–382, 2008.

[11] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings.* IEEE Computer Society, 1980, pp. 212–223.

[12] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH* — a scalable, distributed data structure," *ACM Transactions on Database Systems*, vol. 21, no. 4, pp. 480–525, 1996. [Online]. Available: citeseer.ist.psu.edu/litwin96lh.html

[13] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing-a fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, no. 3, pp. 315–344, 1979.

[14] R. J. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," *ipdps*, vol. 01, p. 96a, 2004.

[15] ——, "A fast algorithm for online placement and reorganization of replicated data," *ipdps*, vol. 00, p. 57b, 2003.

[16] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM Symposium on Theory of Computing*, May 1997, pp. 654–663. [Online]. Available: citeseer.ist.psu.edu/karger97consistent.html

[17] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 22–22.

[18] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu, "A self-organizing storage cluster for parallel data-intensive applications," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 52.

[19] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 562–570.

[20] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran, "Decentralized erasure codes for distributed networked storage," *IEEE/ACM Trans. Netw.*, vol. 14, no. SI, pp. 2809–2816, 2006.

[21] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," *dsn*, vol. 00, pp. 336–345, 2005.

[22] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag, 2002, pp. 328–338.

[23] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *ATEC'00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2000, pp. 4–4.

[24] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation.* Berkeley, CA, USA: USENIX Association, 2004, pp. 7–7.

[25] C. Wu and R. Burns, "Tunable randomization for load management in shared-disk clusters," *Trans. Storage*, vol. 1, no. 1, pp. 108–131, 2005.

[26] ——, "Improving I/O performance of clustered storage systems by adaptive request distribution," *hpdc*, vol. 0, pp. 207–217, 2006.

[27] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing.* Washington, DC, USA: IEEE Computer Society, 2004, p. 4.

[28] "Ceph: pentabyte scale storage." [Online]. Available: http://ceph.sourceforge.net/

[29] Q. Xin, E. L. Miller, and T. J. E. Schwarz, "Evaluation of distributed recovery in large-scale storage systems," in *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, 2004.

[30] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee, "Performance virtualization for large-scale storage systems," *srds*, vol. 00, p. 109, 2003.

[31] C. Lumb, A. Merchant, and G. Alvarez, "Facade: Virtual storage devices with performance guarantees," 2003. [Online]. Available: citeseer.ist.psu.edu/lumb03facade.html

[32] "Round-robin scheduling." [Online]. Available: http://en.wikipedia.org/wiki/Round-robin_scheduling

[33] "Weighted round robin." [Online]. Available: http://en.wikipedia.org/wiki/Weighted_round_robin

[34] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 4, pp. 231–242, 1995.

[35] D. J. Lilja, *Measuring computer performance: a practitioner's guide.* New York, NY, USA: Cambridge University Press, 2000.

[36] H. Casanova, "Simgrid: A toolkit for the simulation of application scheduling," 2001. [Online]. Available: citeseer.ist.psu.edu/casanova01simgrid.html