Using TCP/IP traffic shaping to achieve iSCSI service predictability

J. Bjørgeengen
IT operations dept
University of Oslo
0373 Oslo
jarle.bjorgeengen@usit.uio.no

H. Haugerud
Faculty of Engineering
Oslo University College
0130 Oslo
Harek.Haugerud@iu.hio.no

Abstract

This paper addresses the unpredictable service availability of large centralized storage solutions. Fibre Channel is a common connection type for storage area networks (SANs) in enterprise storage and currently there are no standard mechanisms for prioritizing workloads using this technology. However, the increasing use of TCP/IP based network communication in SANs has introduced the possibility of employing well known techniques and tools for prioritizing IP-traffic. A method for throttling traffic to an iSCSI target server is devised: the packet delay throttle, using common TCP/IP traffic shaping techniques. It enables close-to-linear rate reduction for both read and write operations. All throttling is achieved without triggering TCP retransmit timeout and subsequent slow start caused by packet loss. A control mechanism for dynamically adapting throttling values to rapidly changing workloads is implemented using a modified proportional integral derivative (PID) controller. An example prototype of an autonomic resource prioritization framework is designed. The framework identifies and maintains information about resources, their consumers, response time for active consumers and their set of throttleable consumers. The framework is exposed to extreme workload changes and demonstrates high ability to keep read response time below a predefined threshold. It exhibits low overhead and resource consumption, promising suitability for large scale operation in production environments.

1 Introduction

Large scale consolidation of storage has been an increasing trend over the last years. There are two main reasons for this: rapid growth in the need for data-storage and economy of scale savings. Also, centralized storage solutions are essential to realize most of the cluster and server virtualization products existing today. In the last few years the storage market has shifted its focus from expensive fibre channel (FC) technology towards common-off-the shelf TCP/IP based technology. Storage networking is converging into familiar TCP/IP networking as performance of TCP/IP equipment increasingly gets more competitive with respect to performance. The times when dedicated storage administrators took care of storage area networks (SANs) are about to disappear as the underlying technology used to build SANs is shifting towards less specialized technology. iSCSI is an example of a technology enabling TCP/IP networks to connect hosts to their virtual disks in the theirs SANs. The growth in networked storage and the complexity in conjunction with large scale virtualization increase the demand for system administrators to understand and master complex infrastructures of which storage devices are a central part. Understanding the effects of performance and resource utilization in TCP/IP based SANs is vital in order to make keepable promises about storage performance. Predictable storage performance is a vital requirement for promising performance of the applications utilizing it, and it is the system administrator's job to ensure that storage performance meets the requirements of the applications. Figure 1 gives a simple overview of how several hosts share resources in an iSCSI storage appliance. Physical resource pools are colored, and virtual disks from those pools share the available I/O resources in the pool.

The advantages of storage consolidation/centralization are duly recognized. However, there is a major difference between performance attributes of a virtual disk in a centralized pool of storage and a dedicated local storage unit: sharing of the underlying hardware resources. A local disk may exhibit low total performance compared to SAN devices with a pool of many striped disks, but the performance of the local drive is predictable. The virtual disk in the storage pool usually has a much higher performance depending on the available capacity of the underlying hardware resources. The key point is the de-

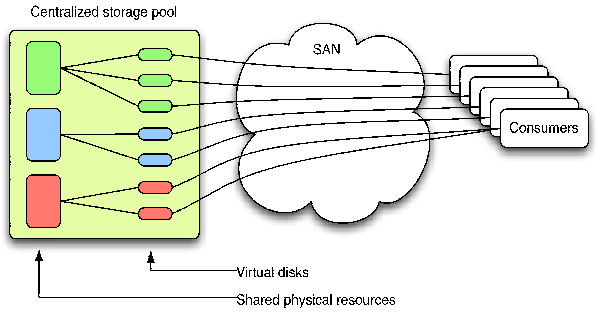

Figure 1: Concept of centralized storage pools

pendence on available capacity, and that available capacity is dependent on the activity towards other virtual disks sharing the same resource pool. A host may saturate the underlying resources of a storage pool causing poor performance of all hosts utilizing virtual disks from that pool. A host utilizing a virtual disk in a shared storage pool has no means to predict the behavior of other hosts utilizing other virtual disks sharing the same pool of resources. Hence, the performance experienced by any host utilizing a virtual disk served by a shared pool is unpredictable by the nature of resource sharing.

Addressing this issue requires a mechanism to prioritize workloads (Quality of Service,QoS) based on some kind of policy defining important and less important workload types. Most storage solutions are able to virtualize the amount of storage presented to the host in a flexible way, but the same storage devices seldom have QoS features. Storage service level agreements (SLAs) presupposes predictability in service delivery, but predictability is not present because of the nature of resource sharing and the absence of prioritization (QoS) mechanisms in storage devices. Application SLAs depend on individual components providing the application with sufficient resources, thus, contributing to the applications' SLA. The disk system is the component saturated first in any computer infrastructure, because it is the slowest one. This condition makes it hard, or impossible, to make keepable promises about performance, and ultimately increases the risk for application SLA violations. Clearly this is sub-optimal situation for system administrators whose mission is to keep applications and their infrastructure running.

The work presented in this paper is motivated by one of the author's experiences with unpredictable service availability of SAN devices at the University of Oslo.

2 System model and design

The goal of this work was to design a working prioritization framework containing throttling, measurements and decision making. The main idea was to utilize common tools in novel ways in order to obtain more predictable service availability of storage devices. The objective was to demonstrate the ability to mend adverse effects of interference between loads using a throttling mechanism for reducing resource contention, thereby improving service availability for important consumers. iSCSI utilizes TCP for transportation and Linux Traffic Control (`tc`) has advanced features for network traffic shaping, hence, the decision to use `tc` for the purpose of throttling was easy.

The amount of consumers that need to be throttled could become large. Also, workloads may rapidly change. Thus, a method to rapidly adapt throttling schemes is a necessary requirement. Traditionally, TCP traffic shaping with Linux Traffic Control is used with static rules targeted only at the network itself, for instance by limiting the network bandwith of traffic to specific IPs. This work utilizes feedback from resources outside of the network layer in order to adapt traffic throttling rules inside the networking layer in a dynamic manner.

In order to have sufficient control of the consumers' resource utilization, both read and write requests must be throttled. It is straightforward to shape outgoing TCP traffic from a server since the rate of transmissions is directly controlled. To the iSCSI server outgoing data translates to delivery of the answer to initiator read requests. Hence, controlling read requests are trivial but controlling write requests is a challenge. iSCSI write requests translates to inbound TCP traffic. Different approaches for dealing with the shaping of inbound traffic are known. The easiest method to achieve this is ingress policing. The concept of ingress policing is to drop packets from the sender when a certain bandwidth threshold is crossed. The congestion control mechanisms of TCP will then adjust the sender rate to a level that can be maintained without packet drops. There are clearly disadvantages to this approach:

- Packet loss which leads to inefficient network link utilization due to packet retransmits.
- The time it takes for the sender to adapt when the receiver decides to change the allowed bandwidth.

Ingress policing might be sufficient for a small number of senders and seldom changes in the receivers' accepted bandwidth. However, the ability to change bandwidth limitations fast is needed for rapid adaption to workload changes. When the number of consumers and bandwidth

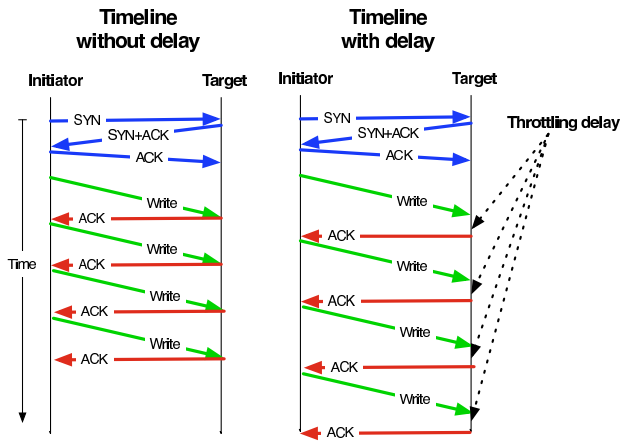

Figure 2: Principle of throttling by delaying packets

limits changes rapidly, this method does not scale, and adapts slowly and inefficiently.

2.1 Dynamic throttling

This paper suggests a novel method of throttling designed to address the limitations just described. The method implies introducing a variable additional delay to packets sent back to initiators, the clients in SCSI terminology. Read requests are simply throttled by delaying all outbound packets containing payload. Outbound ACK packets containing no payload are delayed in order to throttle write request without dropping packets. This method is illustrated in Figure 2. The actual delay is obtained using the `netem` module of Linux Traffic Control, and packets are assigned different delays based on `Iptables` marks.

In section 5 we propose an array agnostic version of this throttle by implementing it in a standalone bridge. The method of delaying packets makes this an attractive idea because of the delay method. Using packet delay rate instant rate reduction is achieved without dropping packets.

As previously argued, the need for a dynamic selection method for throttling packets is needed. `Iptables` provides this dynamic behavior with its many available criteria for matching packets. Combined with the `mark` target, which can be detected by the use of `tc's fw` filters, it is possible to set up a predefined set of delays that covers the needed throttling range with sufficient granularity.

The entities that consume resources in this context are the iSCSI initiators. The entity that provides the resources of interest to the initiators is the iSCSI target. Both initiators and targets have IP addresses. IP addresses can be used for throttling selections. The IP address of the iSCSI initiator will be chosen as the entity to which throttling will apply. Differing priorities for consumers will translate into different throttling schemes of those consumers' IP addresses. The underlying idea is to apply throttling to less important requests in order for important requests to have enough resources available to meet their requirements.

Packet delay throttling makes it possible to influence rates in both directions on a per initiator basis. In production environments the amount of initiators to keep track of quickly becomes overwhelming if throttling is based on individual consumer basis. Moreover, it is likely that the same throttling decisions should be applied to large groups of initiator IP addresses. Applying the same rules, over and over again, on lists of IP addresses is inefficient. To avoid this inefficiency the `Ipset` tool is needed [1]. It is a patch to the Linux kernel that enables creation of sets, and a companion patch to `Iptables` that makes `Iptables` able to match against those sets. This is a fast and efficient method of matching large groups of IP addresses in a single `Iptables` rule: the set of throttleable initiator IP addresses.

2.2 Throttling decision

As pointed out by previous research, remaining capacity is not constant, it is dependent on both rate, direction and pattern of the workloads. Hence, an exact measure of remaining capacity is hard to maintain. However, it is possible to indirectly relate how close the resource is to saturation by measuring individual consumer response times without any knowledge about the cause. In previous research virtual disk response time has successfully been utilized as a saturation level measure [2, 3, 4]. This work uses a similar approach. An Exponentially Weighted Moving Average (EWMA) of the response time is applied before it is used as the input signal to the control mechanism. EWMA is widely adopted as a successful method in the process control field for smoothing sensor input signals. In the process control field, this filter is commonly named a time constant low pass filter. The standard moving average is susceptible for spikes in the data. It is not desirable to trigger large throttling impacts caused by transient spikes in the average wait time, throttling should only occur as a result of persistent problems. The utilization of EWMA enables this behavior.

Interference from less important read or write jobs may lead the measured consumer response time to exceed the desired threshold. The framework should then respond by adding a throttling delay to the packets of the interfering loads, but it is difficult to determine the

exact size of this delay. The idea of using a Proportional Integral Derivative (PID) controller as decision engine emerged from the observations of the relation between interfering workloads, the interference by other consumers and the efficient operation of the packet delay throttle. This behavior is similar to the control organs used to control industrial processes operated by PID controllers in the field of control engineering where they are widely used in order to keep process variables close to their set points and ensure stability for complete industry plants [5].

The purpose of our PID controller is to control throttling such that the consumer wait time of important requests stays below or equal to a preset value even when the load interference changes rapidly. The given value of maximum wait time for storage resources is likely to be constant, and close to the saturation point of the underlying resource. However there is nothing that prevents implementation of dynamically adjustable thresholds. The main purpose of the controller in this work is to keep response time of important requests from violating a given threshold in spite of rapidly changing amounts of interference from less important requests. The appendix contains a more detailed description of the PID controller.

2.3 Automated PID control approach

The ultimate goal of this work was the design of a fully automated per-resource read-response-time-controller as an example technique to utilize the throttle and the controller in order to ensure maximum read response times. Other prioritization schemes are equally possible. This section describes experimental results where the automated framework is exposed to the same loads as in the previous section. However, the selection of throttleable consumers are automatically inferred by the framework by the use of simple workload profiling: write activity of a certain amount.

Most I/O schedulers, and those parts of an entity responsible for servicing application I/O requests, generally have a preference for satisfaction of read requests over write requests. This is because waiting for read requests is blocking applications from continuing their work. Thus, read-over-write prioritization demonstrated here comprises a relevant use case for the throttle and the controller.

Usually, write requests are written to cache, at several levels in the I/O path, for later de-staging to permanent storage without blocking the application from further operation. Hence, throttling write requests can be done to a certain limit without affecting application performance. Nevertheless, it has been demonstrated through earlier experimental results that write requests are able to adversely influence the more important read requests. The design goal of the final prototype is the utilization of earlier results to automatically prevent write requests from adversely impacting read requests, thus contributing to improved application service predictability without the need for user input.

In the previous section the saturation level indicator and the set of throttleable consumers where predefined in order to influence the wait time of the important consumers. This section will describe the design of a prototype that completely automates the detection of saturation level and the identification of throttleable consumers, on a per resource basis. Instead of the prototype of the previous section's reliance on user determined list of important consumers, this prototype uses the read-over-write prioritization to automatically find out what to monitor and which consumers are eligible for write throttling.

In most storage devices, the disk group from which virtual disks are allocated, is bound to become the resource first saturated. This is the reason that LVM was chosen to reproduce a similar environment in the lab setup. In the lab setup, volume groups represent the shared resource that logical volumes are striped across. The objective of the prototype is to control the saturation level caused by write activity on a per-resource basis, thereby indirectly controlling the read response time of the resource. This translates to per volume group in the lab setup. In order to achieve this in the lab prototype, the following requirements will be met:

- An entity that maintains sets of IP addresses that are known to be doing write activity at a certain level: eligible throttlers.
  - Each set should have name of the resource of which its members are consumers.
  - Each set should be immediately throttleable by using its name.
- An entity that maintains a value representing the saturation level on a per-resource basis.
- An entity that spawns a PID controller for each resource and:
  - Uses the resource' saturation level as input.
  - Throttles the set of throttleable consumers for that particular resource so that the saturation level is kept below a set threshold.

The requirements are fulfilled by three perl programs working together with `Iptables, Ipset` and `Traffic Control`, utilizing shared memory for information exchange and perl threads for spawning parallel PID controllers. Figure 2.3 illustrates the concept of the framework implemented by the three scripts.

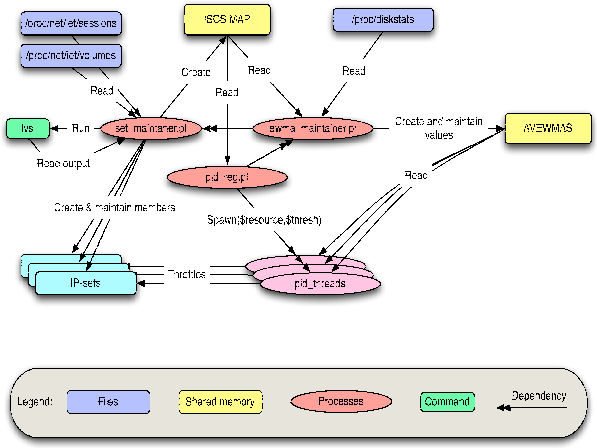

Figure 3: Automated controller framework overview

2.3.1 Automatic population of throttling sets

The Perl program `set_maintainer.pl` reads information about active iSCSI connections from `/proc/net/iet/*`, where information about each iSCSI target id is found: the connected consumer IP and servicing device. For all active iSCSI sessions, the device-mapper (`dm`) name and consumer IP address is recorded. The `lvs` command is used to record the logical volume name and volume group membership of each device-mapper device detected to participate in an active iSCSI session. The information found for each of the device-mapper device is recorded in a data structure and mapped into a shared memory segment with the key *ISCSIMAP*. For each of the volume groups involved in active iSCSI sessions, an empty IP-set is created with the same name as the volume group. When iSCSI device maps are exported to shared memory and the necessary IP-sets are created, the program enters maintenance mode. This is a loop that continuously monitors exponentially weighted averages (EWMAs) of the write sector rates of all `dm` devices involved in active iSCSI sessions. For each of the previously created IP-sets, it then determines the set of consumers that have a write sector rate exceeding a preset configurable threshold. The generated set is compared with the in-kernel IP-set for that resource, and any differences are converged to match the current set of eligible consumes for throttling that were detected. The IP-sets are converged once every second, yielding continuously updated per resource IP-sets known to contain consumers exhibiting write activity at certain level. These sets are immediately throttleable by `Iptables` matching against them.

2.3.2 Automatic determination of saturation monitors

The `ewma_maintainer.pl` program reads the shared memory information exported by the `set_maintainer.pl` program. For each resource, it continuously calculates an exponentially moving average of the read response time using information obtained from `/proc/diskstats`. Only consumers having read activity are included in the calculation. The data structure containing the resources' read response time EWMAs is tied to a shared memory segment with key *AVEWMAS* and updated every $100ms$. The read response time EWMAs serve as per resource saturation indicators which will be used as input values to the subsequently described PID controller threads.

2.3.3 Per resource PID control

The `pid_control.pl` program attaches to the shared memory segment with the key *AVEWMAS*, and reads the saturation indicators maintained by the `ewma_maintainer.pl` program. For each of the resources (volume groups) found in the *AVEWMAS* shared memory segment, a PID controller thread is created with the resource name and its accepted read response time threshold as parameters. Each PID control thread monitors the saturation level of its designated resource and directly controls the delay throttle of the set containing current consumers exhibiting write activity towards that resource. The `pid_control.pl` then detaches from the worker threads and enters an infinite sleep loop, letting the workers control resource saturation levels in parallel until a `SIGINT` signal is received.

3 Results

Experiments are executed using a Linux based iSCSI appliance using striped logical volume manager (LVM) volumes as virtual disks. Each of four striped logical volumes are presented to the blade servers using iSCSI enterprise daemon [6, 7]. The blade servers act as iSCSI initiators and are physically connected to the external iSCSI target server using a gigabit internal blade center switch. Figure 4 shows the architecture of the lab setup.

3.1 Without throttling

When there is no throttling mechanism in place, there is free competition for available resources. Figure 5 shows how four equal read loads, run on each of the equally powerful blade servers, share the total bandwidth of the disk resources, serving each of the logical volumes to

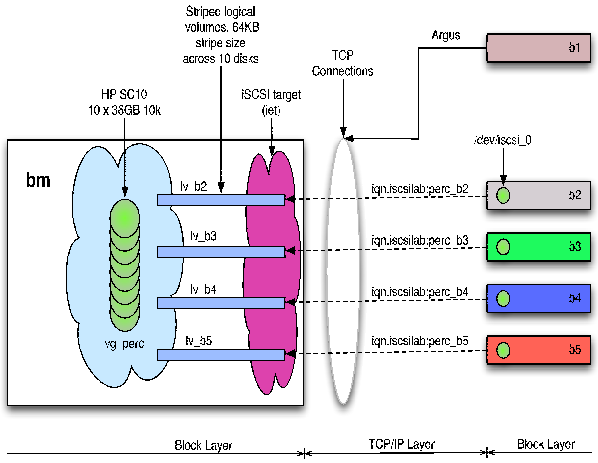

Figure 4: Concept sketch of the lab setup

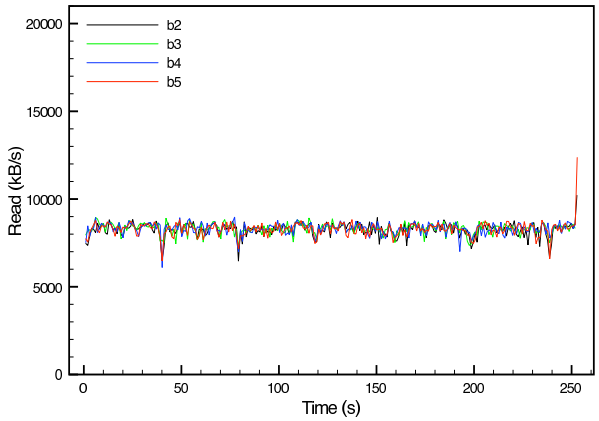

Figure 5: Equal sequential read load from four identically equipped blade servers without throttling

which the blade servers' iSCSI block devices are attached. The plotted read rates show what each of the consuming blade servers achieve individually.

3.2 Throttling by packet delay

Throttling of workloads has been utilized as a means to influence remaining capacity by many previous works, and it is normally carried out by some kind of rate limitation applied to the workloads. Utilization of the iSCSI protocol comes with the additional benefit of utilizing TCP traffic shaping tools to enforce rate limitation. In order to examine the effects on consumers by throttling taking place in the TCP layer, a number of experiments were executed. The first throttling approach involved bandwidth limitations by using hierarchical token bucket filters (HTB). The expected effect of throttling individual consumers was achieved, but the pure bandwidth throttler had a few practical limitations: the need for constantly calculating the bandwidth to be applied and, more important, the inefficient way of controlling write requests. Controlling write rates was not possible without packet loss, resulting in slow and inefficient convergence towards bandwidth target.

The shortcomings of the bandwidth shaping method, especially with respect to writing, inspired the idea of using packet delay for throttling. The `netem` module of Linux Traffic control was used to add delay to packets in a dynamic way in conjunction with `Iptables` packet marks. The concept is to add a small wait time to outgoing ACK packets containing no payload, thus slowing down the packet rate of the sender: the iSCSI writer. The main outcome of the design and subsequent experiments is an efficient way of throttling individual iSCSI consumers' traffic in both directions, with close-to-linear rate reduction and without packet loss. The experiments show that it is possible to throttle write and read activity using the same set of delay queueing disciplines (qdiscs) in Linux Traffic Control (`tc`). For writes, the outgoing ACK packets containing no payload are delayed, and for reads all other packets are delayed.

Figure 6 shows the effect of packet delay based throttling on the same workload as in Figure 5, and Figure 7 shows the effect when writing the same load that was previously read.

The shaping is done in using `Iptables'` packet marking abilities to place packets from individual consumers in different predefined delay qdiscs at different points in time. In this experiment, a shaping script on the target server is throttling down blade servers b2, b3 and b4 at predefined time offsets from the start time of the experiment and releasing them at later points in time. Throttling of blade server b2 frees up resources to the remaining consumers. Next, throttling of b3 and b4 gives increased resources to the remaining consumers. When b2 is freed, b5 is already done with its job, and most resources are available to b2 which increases its throughput dramatically. When b3 is freed, b2 and b3 share the resources again and stabilize at approximately 14 MB/s each. Finally b4 is freed, and b2, b3 and b4 share the resources, each having a throughput of ca. 10 MB/s. When b4 finishes its job, there are two machines left to share the resources, and when b3 finishes, only b2 is left to consume all resources.

Figures 6 and 7 shows a drop in throughput for un-

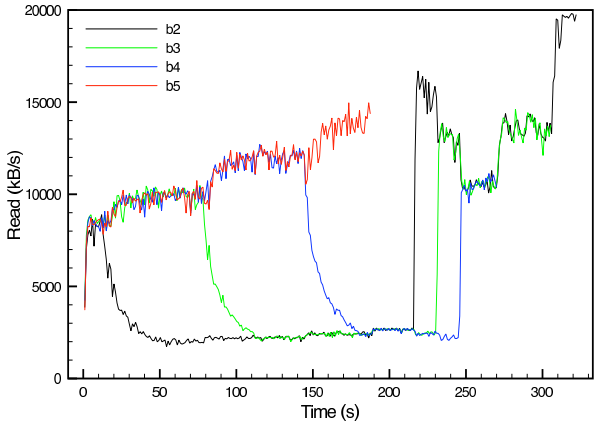

Figure 6: Throttling of initiator's sequential read activity using delayed ACK packets in tc(1).

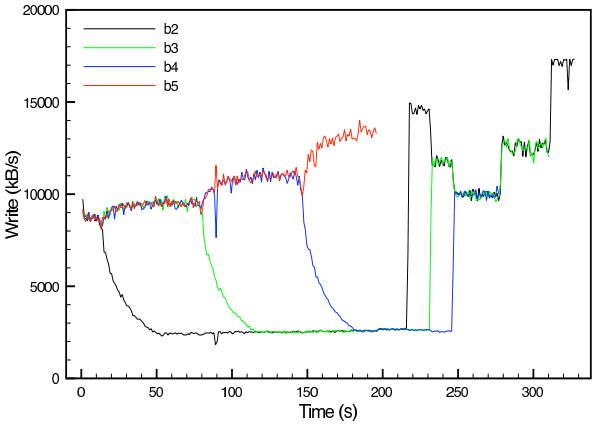

Figure 7: Throttling of initiator's sequential write activity using delayed ACK packets in tc(1).

throttled consumers when throttling starts. No plausible explanation was found for this, and additional research is necessary to identify the cause of this.

3.3 Introduced delay vs throughput

Previous results suggest that the method of introducing artificial delay to outgoing packets could be an efficient way of throttling iSCSI initiators in order to decrease the pressure on shared resources like disk groups. To find out the predictability of throttling as an effect of artificial delay, 200 MB of data was repeatedly read and written from the iSCSI initiator device of one blade server, measuring the time it took to complete each job. Each job were repeated 20 times for each value of artificial delay. Figures 9 and 8 show the results with error indicators, representing the standard error, on top of the bars. The precision of the means is so high that it is hard to see the error indicators at all.

The plots show that variation of artificial delay between 0 and 9.6 ms is consistently able to throttle reads between 22 MB/s and 5 MB/s and writes between 15 MB/s and 2.5 MB/s. There is no absolute relationship between artificial delay and throughput. Rather, the introduced delay has an immediate rate reducing effect regardless of what the throughput was when throttling started. Figures 9 and 8 suggests that there is a close-to-linear functional relationship between introduced delay, the start rate and the resulting rate after throttling.

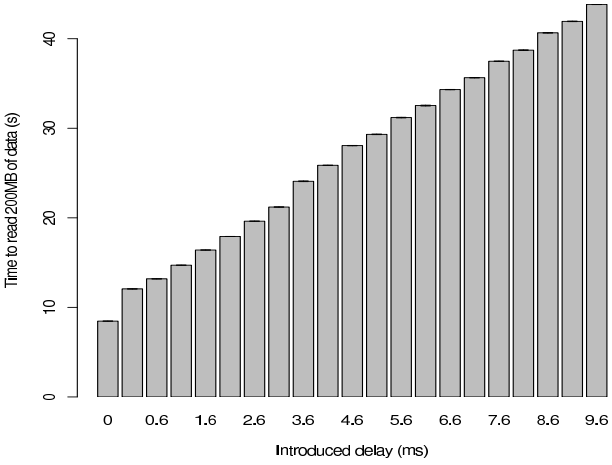

Figure 8: Repeated measurements of the time used to read 200 MB with stepwise increase in artificial delay of outgoing packets from target server.

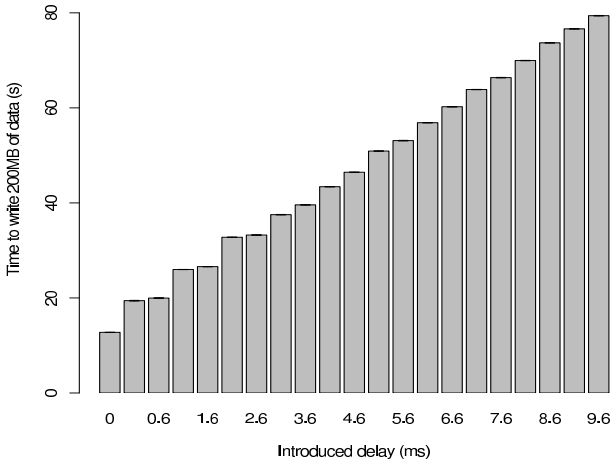

Figure 9: Repeated measurements of the time used to write 200 MB with stepwise increase in artificial delay of outgoing packets (ACK packets) from target server.

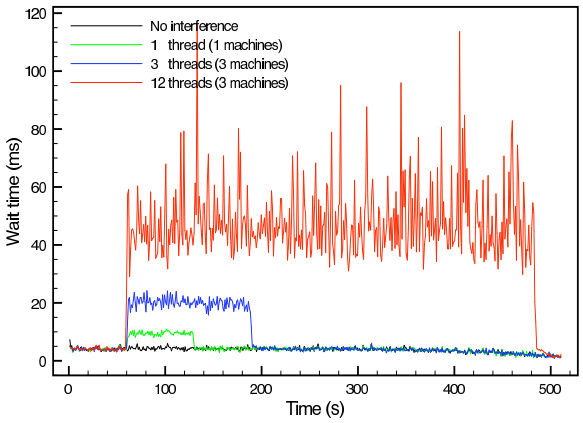

Figure 10: The effect on average wait time for a rate limited (256kB/s) random read job running on one server during interfering write activity from 1 and 3 other machines respectively. The interference is started one minute into the timeline.

3.4 Interference between loads

Figure 10 demonstrates that a small random read job, that causes negligible I/O load by itself, has its response time increased with the amount of load caused by threads running on other hosts. The graphs in the figure is from 4 different runs of the sram random read job, but with different degree of interference in the form of write activity to other logical volumes residing on the same striped volume group. This picture comprises the essence of load interference. The consumer executing the small random read job is unable to get predictable response times from its virtual disk because of activity from other storage consumers.

3.5 Effect of throttling on wait time

Figure 11 shows the effect on a small read job's average wait time when throttling the 12 interfering sequential writers. Packet delay throttling is done in the periods $100s - 190s$ and $280s - 370s$, using $4.6ms$ and $9.6ms$ packet delay respectively. Clearly the throttling of interference contributes to wait time improvement. The magnitude of improvement is higher if the wait time is high before throttling (i.e. level of saturation is high). It means that the throttling cost for improving response time from terrible to acceptable can be very low, but the cost of throttling increases as the response time improves (decreases).

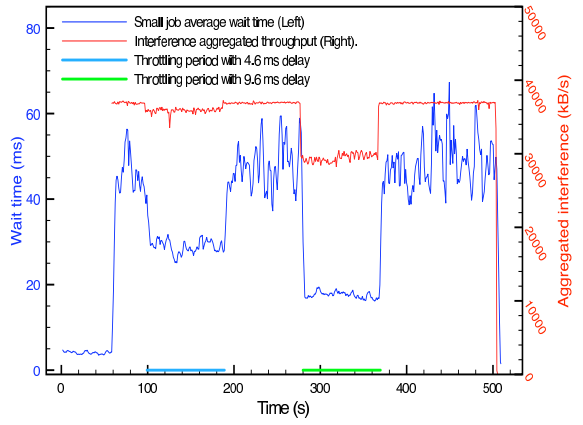

Figure 11: The effect on a small read job's wait time when throttling interfering loads with delays of 4.6 ms and 9.6 ms respectively.

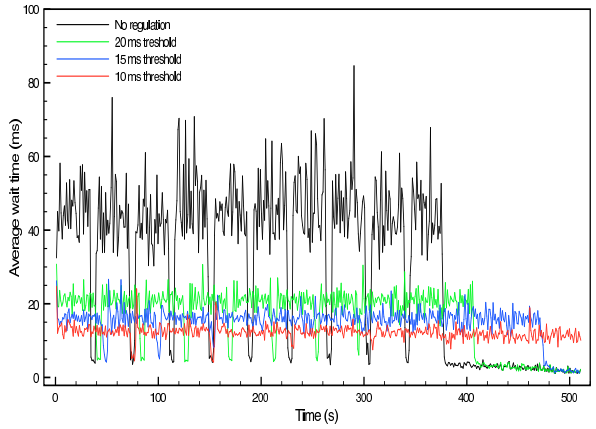

Figure 12: The average wait time of a rate limited (256kB/s) random read job with 12 interfering write threads started simultaneously and repeated with 5 seconds pause in between. The black plot shows the effect with free resource competition. The colored plots show how the PID regulator keeps different latency thresholds by regulating interfering workloads.

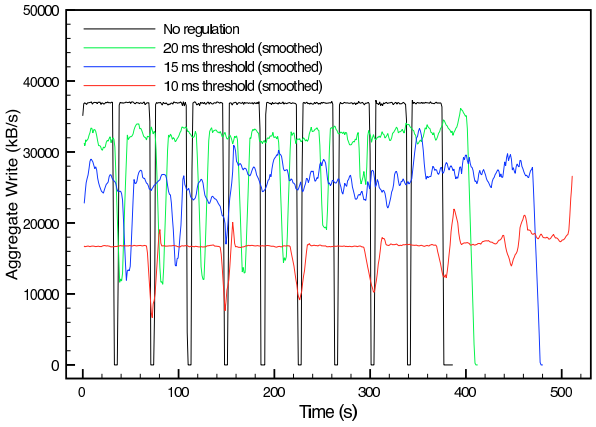

Figure 13: The aggregated throughput caused by throttling to keep latencies at the set thresholds in Figure 12.

3.6 PID control of response time

Figure 12 demonstrates the PID controller's ability to keep the actual wait time below or equal to the desired threshold. The black plot shows how latency is affected by various changing interfering workloads when no throttling is enabled. The colored plots show the effect of the same interfering workloads, but now with the PID regulator enabled having thresholds set to 20,15 and 10 ms respectively. Figure 13 shows the throttling effect on the corresponding interfering workloads (aggregated throughput). Notable is the relatively higher latency improvement for the random read job by throttling aggregate write throughput from its maximum of 39 MB/s down to 33 MB/s, yielding an improvement of 25 ms lower latency. Taking the latency down another five milliseconds costs another seven MB/s of throttling to achieve. Clearly the throttling cost for each step of improved latency increases as latency improves.

3.6.1 Automated PID results

Figure 14 shows that the results with per resource saturation level auto-detection, and dynamically maintained throttleable consumer sets, is close to the results in the previous section where throttleable consumers and response time monitors where defined manually. Figure 15 shows the resulting aggregated write rates as a consequence of the automated throttling carried out to keep read response time below the set thresholds in Figure 14. Again, the black plot depicts response-time/write-rate without regulation, and the colored ones depicts the same but with regulation at different threshold values.

The results shows that the automated per resource PID control framework is able to closely reproduce the previous results where throttleable consumer sets and resource saturation indicators were manually given as parameters to the PID regulators.

There is a slight delay in the throttle response compared to the previous section, giving a slightly larger magnitude and duration of the overshoot created by the simultaneous starting of 12 interfering threads. It is reasonable to speculate that this is caused by the additional time required to populate the sets of throttleable consumers.

During experiment execution, the OUTPUT chain of the Netfilter mangle table was monitored with the command `watch iptables -L OUTPUT -t mangle`. As expected, the rule that marks the outbound ACK packets of all consumers in the set of throttleable consumers appeared as soon as the response time threshold was violated. Further observation revealed rapid increase of the `mark` value as the write interference increased in magnitude, thus directly inhibiting write activity to a level that does not cause write-threshold violation. The command `watch ipset -L` was used to observe that an empty set with the same name as the active resources (the vg_aic volumgroup) were created upon startup of the `set_maintainer.pl` program. Furthermore, the set was populated with the correct IP

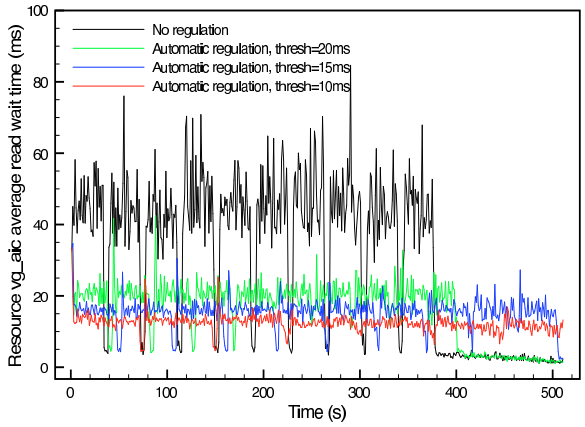

Figure 14: The average wait time of a rate limited (256kB/s) random read job interfered by 12 write threads started simultaneously and repeated with 5 seconds pause in between. The black plot shows the effect with free resource competition. The colored plots show how the PID regulator keeps different response time thresholds by regulating interfering workloads. In this plot, the resource saturation indicator and the set of throttleable host are maintained automatically.

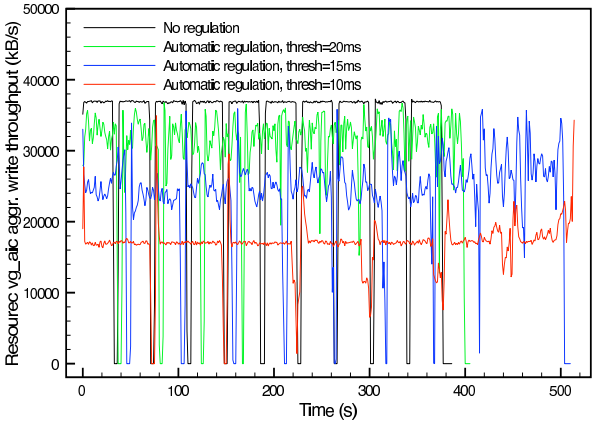

Figure 15: The aggregated throughput caused by throttling to keep latencies at the set thresholds in Figure 14

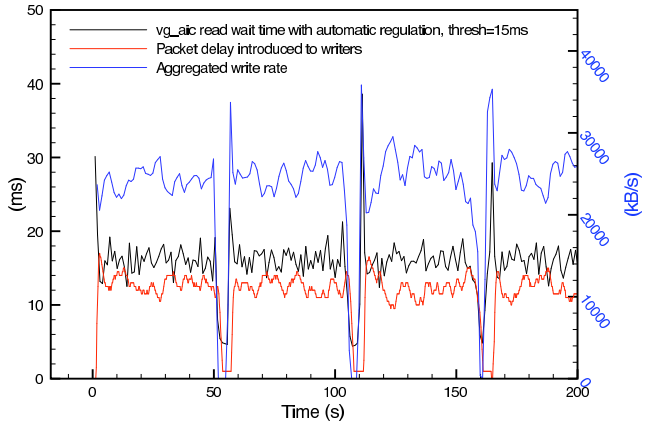

Figure 16: The resource average wait time, the throttling delay and the aggregated write rate with a set resource-wait-time-threshold of 15ms

addresses as the write activity of consumers violated the set threshold, and the IP addresses were removed from the set when consumers ceased/reduced write activity.

Before creating the workload used in this experiment, various smaller workloads were tested while plotting average wait time in realtime during experiments. By applying various increasing and decreasing write interference, the PID controller's behavior was observed in real time. The controller exhibited remarkable stability when gradually increasing interference. Hence, it was decided to produce the most extreme workload variation possible in the lab for the plotted results by turning on and off 12 writer threads (powered by three machines) simultaneously.

It is interesting to examine how the throttle-produced packet delay changes as the the PID controller decides throttle values. Thus, the experiments were run again, capturing the packet delay applied to the set of throttleable hosts along the duration of the experiment. Figure 16 shows the monitored resource's (vg_aic) actual wait time, the throttle value (packet delay) produced by the PID controller and the actual resource's aggregated write rate. The 12 writer threads want as much I/O bandwidth as they can get (37 MB/s without regulation), however, they get throttled by introducing the packet delay seen in the red plot. The decreased write rate caused by packet delay prevents resource saturation, which again prevents read response time of the resource from exceeding the set threshold of 15 ms.

3.7 Measuring overhead

This work introduces new features to prioritize workloads sharing a common resource. It is timely to ask if this new feature comes with an added overhead. When no throttling occurs overhead is unwanted. Since no `Iptables` rules are active when no throttling occurs, there is no overhead introduced by `Iptables`. The only possible source of overhead in this situation is the static `tc` queueing disciplines (qdiscs) and/or the static filters attached to the root qdisc. All outgoing packets are checked for marks by the static filters and there is a risk that this checking introduce overhead. To investigate if the existence of static delay queues and their filters add overhead, the difference in throughput was measured with static qdiscs present and absent.

Throttling only occurs when response time of a resource violates the preset threshold. When no throttling occurs, there is a negligible worst case overhead of 0.4% for reads and 1.7% for writes caused by the static traffic control filters which are always present and ready to detect packet marks. After the experiments where finalized we discovered that `Iptables` is able to classify packets directly to `tc` qdiscs making the check for `Iptables` marks superfluous and there will be no overhead at all when the treshold is not violated. This was confirmed by experiment.

4 Background and previous work

The challenges regarding storage QoS are well recognized and there has been numerous approaches to design of such systems like Stonehenge [8, 9, 10], Cello [11], Façade [12], Triage [13], Argon [14], Chameleon [15] and Aqua [16, 17].

Despite all the research done in the field, specifications regarding QoS functionality are seldom found in the specification sheets of storage devices.

The ability to specify service level objectives (response time and bandwidth), among other data management features, has been the subject of a decade long research at HP Labs Storage Systems department. Looking back in retrospect, Wilkes [18] points out the challenges of incorporating the research results into real production implementations. The challenge is to persuade users to trust the systems to do the right thing. This is a human challenge, one perhaps rooted in general healthy skepticism to new technology and bad experiences from earlier implementations that turned out to not fully take all real life parameters into account. Wilkes points out the need to remember that systems are built to serve people, and the success of technical accomplishments is dictated by how comfortable people ultimately are with them [18].

iSCSI based storage devices are the major competitor to FC based storage devices at the moment. With its lower cost, easier configuration and maintenance and increasingly competitive performance, iSCSI seems to be the enabler of large scale adoption of IP based SAN devices. The introduction of IP as a transportation layer introduces an additional, well known and well trusted toolbox for enforcing policy and fairness amongst storage consumers. Tools for traffic shaping in the TCP/IP layer have been around for many years. The combination of well known and trustworthy throttling mechanisms and an extended knowledge about storage system internals makes an appealing, pragmatic and non-intrusive approach to the problem of QoS in storage systems. Instead of introducing the need to build trust towards new interposed scheduling algorithms, bound to add uncertainty and overhead, this work suggests utilization of previously known and trusted tools to obtain workload prioritization in case of resource saturation. Lumb and coworkers point out the lack of a traffic shaper in storage systems [12] (presumably FC based storage systems). However, when utilizing TCP/IP as transport mechanisms, traffic shapers are available.

The work described in this paper takes a different approach to the problem by utilizing well known tools, with a high level of trust from other fields, and applying them to the storage QoS problem for iSCSI storage devices. The market for iSCSI based storage devices is growing rapidly, making it an interesting target for QoS research. The need for a throttling mechanism, as a means to control storage consumers, has been recognized by previous works [12, 15, 13, 2, 4, 3, 16, 17], and they interpose their own throttlers/schedulers in the critical data path. However, since iSCSI uses TCP for transportation, it is possible to use well known network traffic shaping tools for the purpose of this throttling. With the growing amount of virtual appliances utilizing iSCSI targets as their disk storage, our approach enables global storage QoS directly contributing to application SLAs using well known tools with established trust in the networking field.

5 Future work

This work opens several interesting paths for further research and applications. By using the fundamental ideas explored, it should be possible to create QoS modules to be used as external bridges in front of iSCSI appliances or integrated into Linux based iSCSI appliances similar to the lab prototype. By utilizing the ideas from this work, system administrators and vendors can offer QoS for iSCSI storage. Hence, they can offer differentiated SLAs to storage consumers with a confidence previously very difficult to achieve and contribute their share to overall application SLAs.

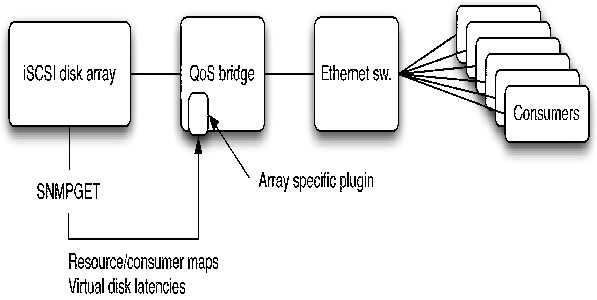

Figure 17: Illustration of how the framework could be utilized as an independent black box with limited array knowledge.

Figure 17 illustrates an approach for moving the controller to an external bridge. Information about consumer/resource mapping and virtual disk read latencies would be necessary in order to directly utilize the techniques demonstrated here. In the figure, usage of SNMP GET requests towards the array is suggested as an easy method for this purpose. However, the ultimate black box approach would be to infer this information from packet inspection. If achievable, this approach could serve as a self contained, non-intrusive, iSCSI QoS machine applicable to all iSCSI solutions regardless of their make and the feedback loop to the storage device would not be necessary. But it is unlikely that the actual consumer/resource mapping can be detected by packet inspection since this is internal storage device knowledge. However, it could be indirectly inferred by using a predefined initiator naming convention that contain resource membership.

Even with high sampling rate, and convergence rate of throttleable consumer sets, the PID controller framework consumes little resources. Small resource consumption and overhead are important attributes to enable high scalability. The small resource consumption and overhead seen in the lab prototype makes it reasonable to project high scalability in a production environment with large amounts of resources and consumers per resource. Combined with the suggested PID controller tuning and rearrangement of `tc` filters an even smaller footprint can be achieved.

The measuring point where virtual disk response time is measured must be moved in order to detect bottlenecks that occur before the local disks of the target server. An approach using agents on iSCSI initiators would be the best way of considering all bottlenecks along the data path by providing the initiator-experienced wait time to the throttling bridge. The advantage of this approach is its simplicity, and how efficiently it will capture all bottlenecks along the iSCSI data path. The disadvantage is its reliance on initiator host modifications. A viable approach could be to use the attribute *has_agent_installed* to infer relative higher importance to the set of initiator that has agents, and automatically use the set of consumers not having agents as a first attempt of throttling before resorting to prioritization between initiators with agents installed. Using this approach, the action of installing an agent serves both the purpose of making performance metrics available to the controller and telling about the membership in the set of hosts with the least importance.

Previously developed algorithms other than the PID algorithm can be combined with the throttling techniques from this work to create even more efficient and/or general purpose QoS mechanisms for iSCSI or even other IP/Ethernet based storage technologies. Furthermore, the PID control algorithm could be evaluated as a means to create stability and predictability in other infrastructure components than just iSCSI devices. It is likely that the problem of controlling iSCSI consumers is not the only one where a PID controller can contribute.

There is always a persistent and large interest in workload classification/modeling techniques in various research areas, not only in the storage field. Together with the ever-evolving efforts to model storage devices, this research can be combined with the ideas and results in this paper in order to add improved and even more generalized frameworks. For example, these techniques could be used to elect candidates for the different sets of throttleable consumers in more sophisticated ways. Also, more advanced algorithms could be combined with response time measurements in order to more accurately detect and/or predict if there is a real problem about to occur.

6 Conclusion

Resource sharing is widely used in storage devices for the purpose of flexibility and maximum utilization of the underlying hardware. Sharing resources like this introduces a considerable risk of violating application service level agreements caused by the unpredictable amount of I/O capacity available to individual storage consumers. The difficulties experienced by system administrators in making keepable promise about storage performance and the amount of previous research in the storage QoS field clearly emphasizes the need for practical and real-world-usable QoS mechanisms for storage systems.

iSCSI based storage solutions are capturing increased market share from FC based storage solutions due to increased performance and low cost. Thus, iSCSI is an interesting target technology for devolpment of QoS mech-

anisms for wide industry and system administrator adoption. The fact that iSCSI utilizes TCP for transportation makes it possible, and very interesting, to adapt well known network traffic shaping tools for the purpose of QoS in iSCSI environments.

This work reproduces and demonstrates the nature of resource sharing, the effect of resource saturation on throughput and consumer response time, and the resulting interference caused by load interaction. Using a Linux based iSCSI storage appliance, experiments reproduce the varying performance of individual consumers caused by other consumers' activity. The lab environment, verified to exhibit similar properties to problematic real-world storage solutions, is then used to design methods to solve some relevant aspects of load interference. The methods involve using a network packet delay method, available in the `netem` module of Linux Traffic Control, in novel ways and a modified proportional integral derivative (PID) controller. By combining the features of the `netem` module with Iptables' ability to dynamically mark packets, an efficient bidirectional mechanism for throttling individual iSCSI initiators consumers is created. The created packet delay throttle is utilized by a modified PID controller implemented in software. The PID controller utilizes the packet delay throttle as a means to influence its input value: the average wait time of the resource being controlled. The resource being controlled in the lab setup is LVM volume groups, but the methods are generally adaptable to any kind of resource exhibiting similar attributes.

The effect of packet delay throttling and the PID controllers' suitability as decision engine is thoroughly examined through experimental results. Finally, all previously designed and tested elements used in single aspect experiments are tied together in a prototype for a autonomous resource control framework that is able to keep resource read response time below a configurable threshold by throttling write activity to the resource automatically. In spite of rapidly varying write workloads, the framework is able to keep a resource read response time below the set threshold. The set of throttleable write consumers is automatically maintained and ready to be used by the PID controller monitoring read response time. The framework spawns a PID controller per resource, using per resource sets of throttleable consumers and per resource response time measurements. The sets of throttleable consumers are automatically populated using simple workload profiling.

This work opens several interesting paths for further research and applications. By using the fundamental ideas explored, it is possible to create QoS modules to be used as an external bridge in front of iSCSI appliances or integrated into Linux based iSCSI appliances similar to the lab environment. Previously developed algorithms can be combined with the throttling techniques from this paper to create even more efficient and/or general purpose QoS mechanisms for iSCSI or even other IP/Ethernet based storage technologies. Furthermore, the PID control algorithm could be evaluated as a means to create stability and predictability in other infrastructure components than just iSCSI devices.

By using the basic building blocks of this work it is possible to create a vast amount of prioritization schemes. The few examples given serves as a demonstration of the inherent opportunities. With the modular design of the different programs it should be trivial to reimplement the framework in similar setups with minor adjustments only.

With the small resource consumption footprint of the prototype, and room for further improvement of it, this concept should scale to enterprise level production environments with large amounts of resources and storage consumers.

By utilizing the ideas from this work, system administrators and vendors can offer QoS for iSCSI storage, thereby making it possible to offer differentiated SLAs to storage consumers supporting application SLAs with a confidence previously very difficult to achieve.

References

[1] Home page of ipset. URL `http://ipset.netfilter.org/`.

[2] A. Gulati and I. Ahmad. Towards distributed storage resource management using flow control. *ACM SIGOPS Operating Systems Review*, 42(6):10–16, 2008.

[3] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.

[4] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Modeling workloads and devices for io load balancing in virtualized environments. *SIGMETRICS Perform. Eval. Rev.*, 37(3):61–66, 2009. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1710115.1710127.

[5] F. Haugen. *PID control*. Tapir Academic Press, 2004.

[6] Home page of lvm. URL `http://sourceware.org/lvm2/`.

[7] iscsi enterprise target project homepage. URL `http://iscsitarget.sourceforge.net/`.

[8] L. Huang, G. Peng, and T. Chiueh. Multi-dimensional storage virtualization. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):14–24, 2004.

[9] Lan Huang. Stonehenge: A high performance virtualized storage cluster with qos guarantees. Technical report, 2003.

[10] G. Peng. *Availability, fairness, and performance optimization in storage virtualization systems*. PhD thesis, Stony Brook University, 2006.

[11] P. Shenoy and H.M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems*. *Real-Time Systems*, 22 (1):9–48, 2002.

[12] C.R. Lumb, A. Merchant, and G.A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, page 144. USENIX Association, 2003.

[13] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage (TOS)*, 1(4):480, 2005.

[14] M. Wachs, M. Abd-El-Malek, E. Thereska, and G.R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 5–5. USENIX Association, 2007.

[15] S. Uttamchandani, L. Yin, G.A. Alvarez, J. Palmer, and G. Agha. CHAMELEON: a self-evolving, fully-adaptive resource arbitrator for storage systems. URL `https://www.usenix.org/events/usenix05/tech/general/full_papers/uttamchandani/uttamchandani_html/paper.html`.

[16] J.C. 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*, pages 209–218. Citeseer.

[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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[18] W. John. Traveling To Rome: A Retrospective On The Journey. *Operating systems review*, 43(1):10–15, 2009.

[19] F. Haugen. *Anvendt reguleringsteknikk*. Tapir, 1992.

A The PID controller

The problem investigated in this paper is similar to a process control problem solved by PID controllers. Figure 11 demonstrates the instant rate reducing throttling effect freeing capacity, which again influences read response time. Section 3.3 describes a stepwise close-to-linear relationship similar to what a PID controller needs in order to work. Figure 18 shows the concept of a PID controller [1].

PID controllers can be implemented in software using a numerical approximation method. This work uses a numerical implementation of the PID controller with virtual disk wait-time as input signal and packet delay as output signal.

The packet delay throttle is implemented as a range of integers representing a stepwise proportional throttling mechanism. Each integer step represents an increased packet delay, thus, a decreased rate. Figures 8 and 9 suggest that steps of $0.5ms$ is a suitable granularity. At $0.5ms$ granularity, the amount of steps is determined from maximum allowed artificial packet delay: i.e. zero

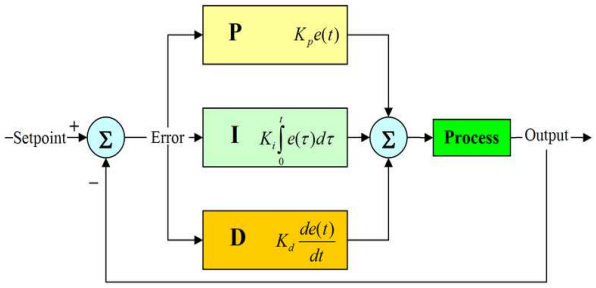

Figure 18: Block diagram of a PID controller. Licensed under the terms of Creative Commons Attribution 2.5 Generic .

rate reduction plus 21 increasing steps of rate reduction with a maximum delay of $20ms$.

$$u(t) = \underbrace{K_p e(t)}_{\text{Proportional}} + \underbrace{\frac{K_p}{T_i}\int_0^t e(\tau)d\tau}_{\text{Integral}} + \underbrace{K_p T_d e'(t)}_{\text{Derivative}} \quad (1)$$

Equation 1 represents the continuous function for outputting throttling amount as a function of the set-point error $e(t)$, the difference between the set value (threshold) and real value. Hence, the PID controller is an error driven controller. The proportional part is the first part of the function and is parameterized by the proportional gain K_p. The second part is the integral part. It is proportional to both the error and the duration of it and is parameterized by the integral time T_i. The purpose of the integrating part is to eliminate the residual steady state error that occurs with a proportional-only controller. The third part of the equation is the differential part. It is parameterized by the derivative gain tuning parameter T_d. The purpose of the derivative part is to slow down the rate of change of the controller output, thereby reducing the magnitude of overshoot created by the integral part.

When computer based controllers replaced older analogue PID controllers, the PID function was discretized using Euler's backward method and became the basic discrete function shown in equation 2 yielding the so-called discrete PID algorithm on incremental form. The function is used as the basis for most discrete PID controllers in the industry [5, 19]. This paper implements a variation of equation 2 that takes the distance above preset response time threshold as input error signal and computes an output throttling value. The modified algorithm is named a single sided PID controller because it only throttles when the error is positive, that is, when the real value is higher than the set threshold.

The PID control algorithm is a direct implementation of equation 2 below with two exceptions: the negative

throttling value is capped to the maximum throttling step corresponding to the integer value of the packet delay class with the highest delay, and the positive throttling value capped to zero. This is done to prevent integral windup: the integral part accumulating too high values that takes a long time to wind down again, and to disable throttling completely when the error is negative: real value is below the threshold. The output value of the PID controller is rounded up to the next integer value, and that integer becomes the `Iptables` mark to apply to all outgoing ACK packets matching destination addresses of the iSCSI initiator IP addresses in the set of throttleable consumers.

$$u_k = \underbrace{K_p e_k}_{Proportional} + \underbrace{u_{i_{k-1}} + \frac{K_p T}{Ti} e_k}_{Integral} + \underbrace{\frac{K_p T_d}{T}(e_k - e_{k-1})}_{Derivative} \quad (2)$$

The PID controller must be tuned for optimal control of the process. In control engineering, the best operation of the controller is when the actual value always is stable and equal to the set point no matter how fast the set point changes or environmental forces influence the actual value. This ideal behavior is never achievable in real world applications of the PID controller: there are always physical limitations that makes the ideal case a theoretical utopia. The tuning process' concern is finding the parameters to the controller that makes it behave as close to the theoretical ideal as possible. There are several known methods to tune PID controllers. The Ziegler-Nichols method, the improved Åstrøm-Hägglund method and the Skogestad method are some widely used methods in control engineering [5]. These methods have not been considered during this paper since a few iterative experiments and according parameter adjustments yielded stable and good controller performance in short time. Thus, the process in this paper is easy to tune compared to many industrial processes. However, thorough tuning efforts is likely to produce similar controller efficiency with less less resource usage of the controller loop.

In addition to the PID parameters, the sample interval influences loop stability and tuning. Generally, the discrete PID controller approaches the behavior of a continuous PID controller when the sample interval goes to zero. The reason to keep sample interval short is increased stability and the reason for increasing the sample interval is minimizing resources utilized by the controller. The sample interval used in this paper was found by experimenting with values and observing CPU usage. A sample interval of $100ms$ yielded very stable controller behavior and CPU utilization of approximately 1%.

However, lowering the sample frequency more may be possible without sacrificing stability. Another benefit of lowering the sampling frequency is calmer operation of the throttle. It may not be necessary to move the throttled IP addresses around as agilely as in the experiments, but it must be agile enough to capture rapid workload interference changes. The interval of $100ms$ seems to be a fair balance between controller resource consumption and agility.

Notes

[1]Created by the Silverstar user @ Wikipedia, as required by CC licensing terms.