Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems

Amar Phanishayee, Elie Krevat, Vijay Vasudevan,
David G. Andersen, Gregory R. Ganger, Garth A. Gibson, Srinivasan Seshan
Carnegie Mellon University

CMU-PDL-07-105
September 2007

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Cluster-based and iSCSI-based storage systems rely on standard TCP/IP-over-Ethernet for client access to data. Unfortunately, when data is striped over multiple networked storage nodes, a client can experience a TCP throughput collapse that results in much lower read bandwidth than should be provided by the available network links. Conceptually, this problem arises because the client simultaneously reads fragments of a data block from multiple sources that together send enough data to overload the switch buffers on the client's link. This paper analyzes this Incast *problem, explores its sensitivity to various system parameters, and examines the effectiveness of alternative TCP- and Ethernet-level strategies in mitigating the TCP throughput collapse.*

Acknowledgements: We would like to thank Jeff Butler, Abbie Matthews, and Brian Mueller at Panasas Inc. for allowing us and helping us to conduct experiments on their systems. We thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Network Appliance, Oracle, Seagate, and Symantec) for their interest, insights, feedback, and support. Finally, we'd like to thank Michael Stroucken for his help managing the PDL cluster, and Michael Abd-el-Malek for feedback on our work. This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0546551, #CNS-0326453 and #CCF-0621499, by the Army Research Office under agreement number DAAD19–02–1–0389, by the Department of Energy under Award Number #DE-FC02-06ER25767, and by DARPA under grant #HR00110710025.

Keywords: Cluster-based storage systems, TCP, performance measurement and analysis

1 Introduction

Cluster-based storage systems are becoming an increasingly important target for both research and industry [3, 30, 15, 24, 14, 10]. These storage systems consist of a networked set of smaller storage servers, with data spread across these servers to increase performance and reliability. Building these systems using commodity TCP/IP and Ethernet networks is attractive because of their low cost and ease-of-use, and because of the desire to share the bandwidth of a storage cluster over multiple compute clusters, visualization systems, and personal machines. Non-IP storage networking lacks some of the mature capabilities and breadth of services available in IP networks. However, building storage systems on TCP/IP and Ethernet poses several challenges. In this paper, we analyze one important barrier to high-performance storage over TCP/IP: the *Incast* problem [24].

TCP *Incast* is a catastrophic throughput collapse that occurs as the number of storage servers sending data to a client increases past the ability of an Ethernet switch to buffer packets. As we explore further in Section 2, the problem arises from a subtle interaction between relatively small Ethernet switch buffer sizes, the communication patterns common in cluster-based storage systems, and TCP's loss recovery mechanisms. Briefly put, data striping couples the behavior of multiple storage servers, so the system is limited by the request completion time of the *slowest* storage node [9]. Small Ethernet buffers get exhausted by a concurrent flood of traffic from many servers, which results in packet loss and one or more TCP timeouts. These timeouts impose a delay of hundreds of milliseconds—orders of magnitude greater than typical data fetch times—significantly degrading overall throughput.

This paper provides three contributions. First, we explore in detail the root causes of the *Incast* problem, characterizing its behavior under a variety of conditions (buffer space, varying numbers of servers, etc.). We find that *Incast* is a general barrier to increasing the number of source nodes in a cluster-based storage system. While increasing the amount of buffer space available can delay the onset of *Incast*, any particular switch configuration *will* have some maximum number of servers that can send simultaneously before throughput collapse occurs.

Second, we examine the effectiveness of existing TCP variants (e.g., Reno [5], NewReno [13], SACK [22], and limited transmit [4]) designed to improve the robustness of TCP's loss recovery. While we do find that the move from Reno to NewReno substantially improves performance, none of the additional improvements help. As we show in Section 5, this is because in the remaining cases, TCP loses *all* packets in its window or loses retransmssions; as a result, *no* clever loss recovery algorithms can help.

Finally, we examine a set of techniques that are moderately effective in masking the *Incast* problem, such as drastically reducing TCP's retransmission timeout timer (Section 5.2). With some of these solutions, building a high-performance cluster filesystem atop TCP/IP and Ethernet can be practical. Unfortunately, while these techniques can be effective, none of them is without drawbacks. Our final conclusion is that no existing solutions are entirely sufficient, and further research is clearly indicated to devise a principled solution for the *Incast* problem.

2 Background

In cluster-based storage systems, data is stored across many storage servers to improve both reliability and performance. Typically, their networks have high bandwidth (1-10 Gbps) and low latency (round-trip-times of 10s to 100s of μseconds) with clients separated from storage servers by one or more switches.

In this environment, data blocks are striped over a number of servers, such that each server stores a fragment of a data block, denoted as a *Server Request Unit (SRU)*, as shown in Figure 1. A client requesting a data block sends request packets to all of the storage servers containing data for that particular block; the client requests the next block only after it has received all the data for the current block. We refer to such reads as *synchronized reads*.

This simple environment abstracts away many details of real storage systems, such as multiple stripes per data block, multiple outstanding block requests from a client, and multiple clients on a single switch making requests across a shared subset of servers. However, this is the most basic representative setting

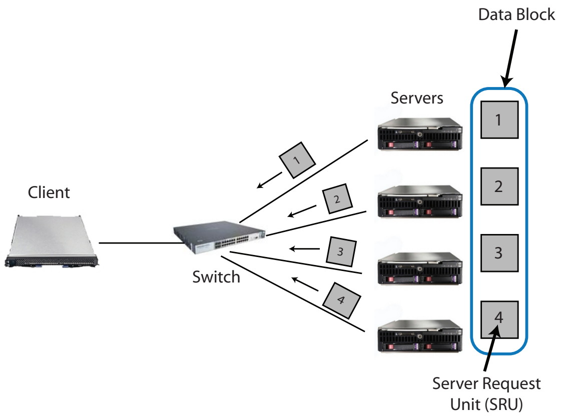


Figure 1: Terminology for a synchronized reads environment, where one client requests data from multiple servers.

in which *Incast* can occur and simplifies our analysis.

The need for a high performance environment that supports parallel operations such as synchronized reads is particularly important because of such recent projects as *pNFS*. *pNFS* is a component of NFSv4.1 that supports parallel data transfer [31, 26, 18]. Many pNFS deployments stripe data across NFS file servers in an environment similar to ours.

Most networks are provisioned such that the client's bandwidth to the switch should be the throughput bottleneck of any parallel data transfer [16, 21]. Unfortunately, when performing synchronized reads for data blocks across an increasing number of servers, a client may observe a TCP throughput drop of one or two orders of magnitude below its link capacity. Figure 2 illustrates this performance drop in a cluster-based storage network environment when a client requests data from just seven servers.

Early parallel network storage projects, such as the NASD project [15], observed TCP throughput collapse in cluster-based storage systems during synchronous data transfers. This was documented as part of a larger paper by Nagle et al. [24], who termed the problem *Incast* and attributed it to multiple senders overwhelming a fixed-size switch buffer. However, while Nagle demonstrated the problem and suggested that an alternative TCP implementation shows a modest improvement, a full analysis and measurement of the problem and possible solutions was not performed.

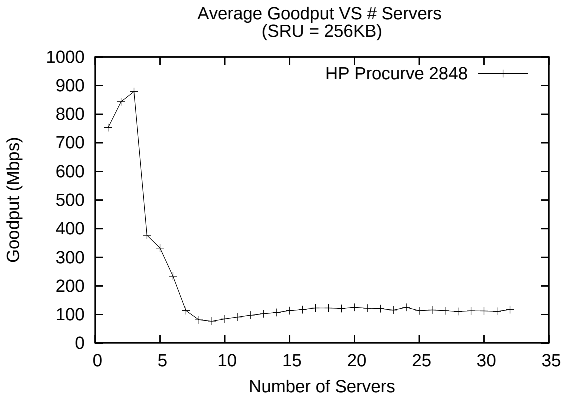


Figure 2: TCP throughput collapse for a synchronized reads application performed on a storage cluster.

To our knowledge, *Incast* has never been thoroughly studied. Current systems attempt to avoid TCP throughput collapse by limiting the number of servers involved in any one block transfer, or by artificially limiting the rate at which they transfer data. These solutions, however, are typically specific to one configuration (e.g. number of servers, data block sizes, line capacities, etc.), and thus are not robust to changes in the storage network environment.

3 Experimental Setup

In this section, we describe the simulation and real system environments where we measure the effects of *Incast* and the corresponding workloads that we use in both settings.

3.1 Simulation Environment

All of our simulations use *ns-2* [2], an event-based network simulator that models networked applications at the packet granularity. Our default simulation configuration consists of a client and multiple servers all connected to the same switch as shown in Figure 1.

Table 1 shows the parameters, and their corresponding default values, that we vary in simulation. We choose a 256KB default *SRU* size to model a production storage system [1]. From our simulations, we obtain global and per-flow TCP statistics such as retransmission events, timeout events, TCP window

Parameter	**Default**
Number of servers	—
SRU Size	256KB
Link Bandwidth	1 Gbps
Round Trip Time (RTT)	100μ
Per-port switch output buffer size	—
TCP Implementation: Reno, NewReno, SACK	NewReno
Limited Transmit	disabled
Duplicate-ACK threshold (da_{thresh})	3
Slow Start	enabled
RTO_{min}	200ms

Table 1: Simulation parameters with default settings.

sizes, and other TCP parameters to aid in our analysis of *Incast*.

Our test application performs synchronized reads over TCP in *ns-2* to model a typical striped file system data transfer operation. The client requests a data block from *n* servers by sending a request packet to each server for one *SRU* worth of data. When a client receives the entire data block of $n{\cdot}SRU$ total bytes, it immediately sends request packets for the next block. Each measurement runs for 20 seconds of simulated time, providing enough data transfer to accurately calculate throughput.

3.2 Cluster-based Storage Environment Details

Our experiments use a networked group of storage servers as configured in production storage systems. Our application performs the same synchronized reads protocol as in simulation and measures the achieved throughput. All systems have 1 Gbps links and a client-to-server Round Trip Time (RTT) of approximately 100μs. We evaluated three different storage clusters:

- **Procurve:** An HP Procurve 2848 switch connects a client to up to 64 servers, running Linux 2.6.18 SMP, through one or more HP Procurve 2848 ethernet switches configured in a tree hierarchy.[1]
- **S50:** A Force10 S50 switch connects 48 Redhat4 Linux 2.6.9-22 machines (1 client, 47 servers) on one switch.
- **E1200:** A Force10 E1200 switch with 672 ports with at least 1MB output buffer per port. This switch connects a client to 87 servers all running Redhat4 Linux 2.6.9-22 (1 client, 87 servers).

For our workload and analysis, we keep the *SRU* size fixed while we scale the number of servers, implicitly increasing the data block size with the number of servers.[2]

4 Reproducing Incast

In this section, we first demonstrate *Incast* occurring in several real-world cluster-based storage environments. Using simulation, we then show that *Incast* is a generic problem and identify the causes of *Incast*. We find that simulation results validate the results obtained from our experimental setup. Finally, we show that attempts to mitigate *Incast* by varying parameters such as switch buffer size and *SRU* size are incomplete solutions that either scale poorly or introduce system inefficiencies when interacting with a filesystem.

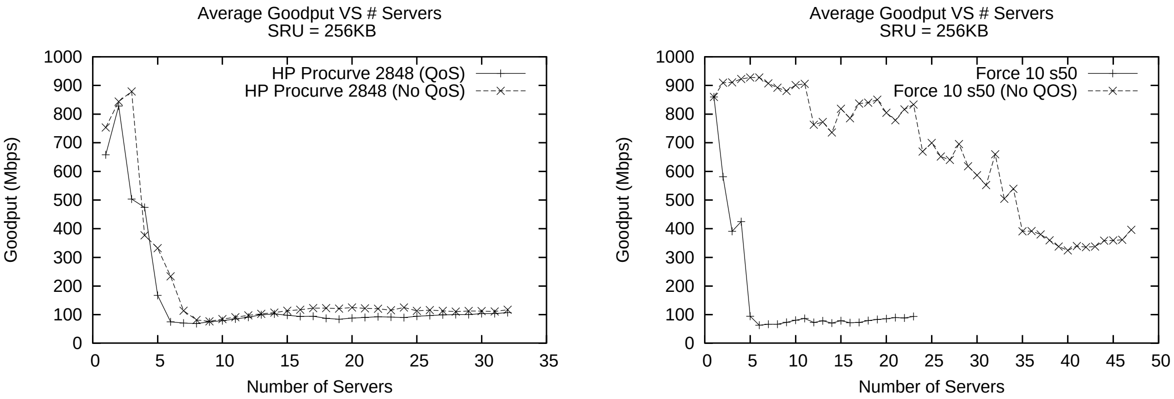
4.1 *Incast* in real systems

To ensure that the throughput collapse shown in Figure 2 is not an isolated instance, we study *Incast* on the three storage clusters described in Section 3.2. Figure 3 indicates that both the Procurve and S50 environments experience up to an order of magnitude drop in *goodput* (throughput as observed by the application). The E1200, however, exhibited no drop at least up to the 87 servers available, which we attribute to the large amount of buffer space available on the switch.

In our analysis, we use estimates of the output buffer sizes gathered from network administrators and switch specifications. Unfortunately, we are unable to determine the exact per-port buffer sizes on

[1] Although this topology does not exactly match our simulation topology, we find that multiple switches do not prevent *Incast*.

[2] Some storage systems might instead scale by keeping the block sized fixed and increasing the number of servers used to stripe data over, thus decreasing the effective SRU size when spreading a block over more servers. We explore independently the effects of changing the SRU size and increasing the number of servers in Sections 4 and 5, so the effects of *Incast* can also be predicted under this alternative scaling model.



(a) HP Procurve 2848

(b) Force10 S50

Figure 3: *Incast* observed on different switch configurations. *Procurve 2848* (a) depicts TCP throughput collapse at seven or more servers with or without QoS support enabled. *Force10 S50* (b) with QoS support disabled significantly delays the onset of *Incast*.

these switches. First, most switches dynamically allocate from a shared memory pool for each link's output buffer. Second, it is unclear how much memory is allocated to QoS queues when enabled. However, our estimates are corroborated by simulation results.

Many switches provide QoS support to enable prioritization of different kinds of traffic. A common implementation technique for providing QoS is to partition the output queue for each class of service. As a result, disabling QoS increases the effective size of the output queues, though the amount of this increase varies by switch. As shown in Figure 3(a), disabling QoS support on the Procurve environment does not significantly change throughput – it still collapses above 8 servers. This suggests that the switch does not allocate much additional buffer space when disabling QoS support. In contrast, Figure 3(b) shows that disabling QoS support on the Force10 S50 *significantly* delays the onset of *Incast*, though *Incast* does eventually manifest. However, the E1200 environment, which has a large amount of per-port buffer space on the switch, does not exhibit *Incast* with as many as 87 servers. This result strongly suggests that switch buffer sizes play an important role in mitigating *Incast*. We evaluate the effect of buffer sizes on throughput collapse in Section 4.3.

4.2 Validation and Analysis in Simulation

To determine how general a problem *Incast* is for cluster-based storage over TCP/IP/Ethernet, we also reproduce *Incast* in the *ns-2* network simulator. Figure 4 shows *Incast* in simulation with an order of magnitude collapse at 8 servers and beyond, and these results closely match those from the Procurve environment. The differences between the results, including the difference in behavior below 3 servers, have a few possible causes. First, simulated source nodes serve data as rapidly as the network can handle, while real systems are often slightly slower. We attribute the worse performance of the real system between 1-3 servers to these differences. Also, simulation does not model Ethernet switching behavior, which may introduce small timing and performance differences.

Despite these differences, the simulation validates our real world measurements, showing that *Incast* occurs in approximately the same manner for both real world system measurements and simulation.

An analysis of the TCP traces obtained from simulation reveals that TCP retransmission timeouts are the primary cause of *Incast* (Figure 5).[3] When goodput degrades, most servers still send their *SRU*

[3] TCP goodput could also be degraded by a large number of packet retransmissions that waste network capacity. We find, however, that retransmitted packets make up only about 2% of all transmissions. This overhead is not significant when compared to the penalty of a retransmission timeout.

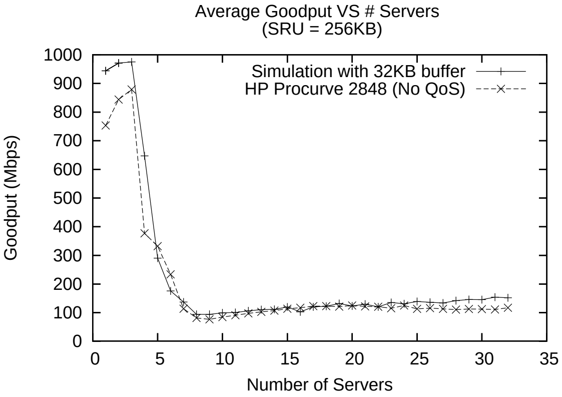


Figure 4: Comparison of *Incast* in simulation and in real world cluster-based settings.

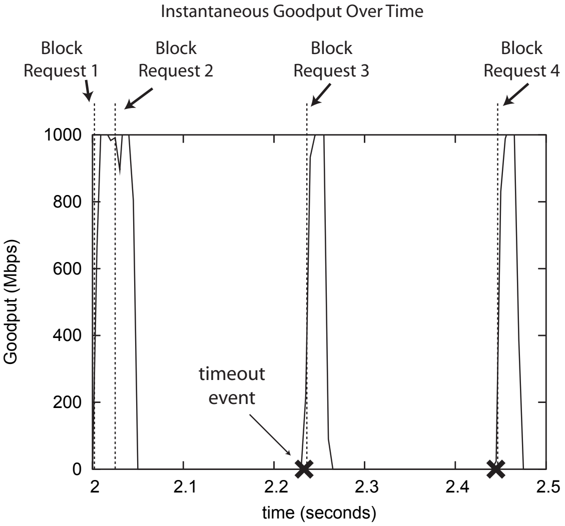


Figure 5: Instantaneous goodput averaged over 5ms intervals. Timeouts are the primary cause of *Incast* and one stalled flow during a block transfer results in an idle link duration of 200ms. Timeout events indicate when a flow begins recovery.

quickly, but one or more other servers experience a timeout due to packet losses. The servers that finish their transfer do not receive the next request from the client until the client receives the complete data block, resulting in an underutilized link.

Why do timeouts occur? Reading blocks of data results in simultaneous transmission of packets from servers. Because the buffer space associated with the output port of the switch is limited, these simultaneous transmissions can overload the buffer resulting in losses. TCP recovers from losses by retransmitting packets that it has detected as being lost. This loss detection is either data-driven or is based on a timeout for a packet at the sender.

A TCP sender assigns sequence numbers to transmitted packets and expects TCP acknowledgements (ACKs) for individual packets from the receiver. The TCP receiver acknowledges the last packet it received in-order. Out-of-order packets generate duplicate ACKs for the last packet received in-order. Receiving multiple duplicate ACKs for a packet is an indication of a loss – this is data-driven loss detection. Timeouts are used as a fallback option in the absence of enough feedback, and are typically an indication of severe congestion.

In Figure 3(a), we see an initial drop from 900Mbps to 500Mbps between 3-5 servers on the Procurve. Analysis of TCP logs reveal that this drop in throughput is caused by the delayed ACK mechanism [5]. In the delayed ACK specification, an acknowledgement should be generated for at least every second packet and must be generated within 200ms of the arrival of the first unacknowledged packet. Most TCP implementations wait only 40ms before generating this ACK. This 40ms delay causes a “mini-timeout”, leading to underutilized link capacity similar to a normal timeout. However, normal timeouts are responsible for the order of magnitude collapse seen beyond 5 servers in *Incast*. We explore TCP-level solutions to avoid timeouts and to reduce the penalty of timeouts in detail in Section 5.

4.3 Reducing Losses: Larger Switch Buffers

Since timeouts are the primary cause of *Incast*, we try to prevent the root cause of timeouts – packet losses – to mitigate *Incast* by increasing the buffer space allocated per port on switches. Section 4.1 hinted at the fact that a larger buffer size on switches delays the onset of *Incast*. Figure 6 shows that doubling the size of the switch’s output port buffer in simulation doubles the number of servers that can transmit before the system experiences *Incast*.

With a large enough buffer space, *Incast* can be avoided for a certain number of servers, as shown by the 1024KB buffer line in Figure 6. This is corroborated by the fact that we were unable to observe *Incast* with 87 servers on the Force10 E1200 switch, which has very large buffers. But Figure 6 shows that for a 1024KB buffer, 64 servers only utilize about 65% of the client's link bandwidth, and *doubling* the number of servers only improves goodput to 800Mbps.

Unfortunately, switches with larger buffers tend to cost more (the E1200 switch costs over $500,000 USD), forcing system designers to choose between overprovisioning, future scalability, and hardware budgets. This suggests that a more cost-effective solution is needed to address the problem of *Incast*-caused timeouts.

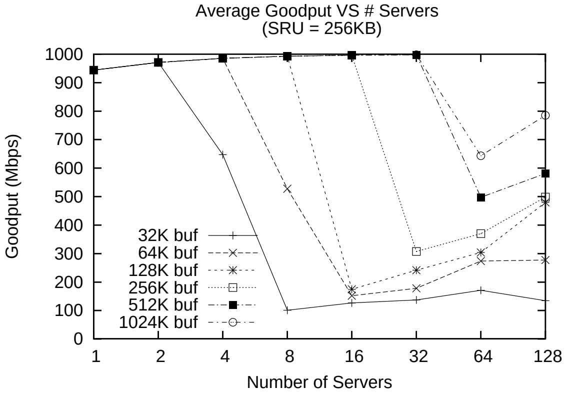


Figure 6: Effect of varying switch buffer size: doubling the size of the switch's output port buffer doubles the number of servers that can be supported before the system experiences *Incast*.

4.4 Reducing Idle Link Time by Increasing *SRU* Size

Figure 7 illustrates that increasing the *SRU* size improves the overall goodput. With 64 servers, the 1000KB *SRU* size run is two orders of magnitude faster than the 10KB *SRU* size run. Figure 8 shows that real switches, in this case the Force10 S50, behave similarly.

TCP performs well in settings without sychronized reads, which can be modeled by an infinite *SRU* size. The simple TCP throughput tests in *netperf* do not exhibit *Incast* [24]. With larger *SRU* sizes, servers will use the spare link capacity made available by any stalled flow waiting for a timeout event; this effectively reduces the ratio of timeout

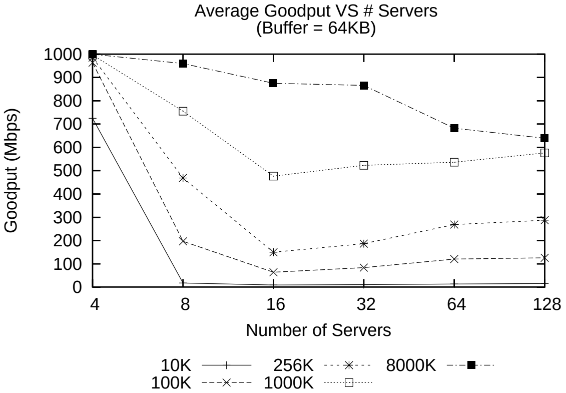


Figure 7: Effect of varying *SRU* size: for a given number of servers, a larger *SRU* improves goodput.

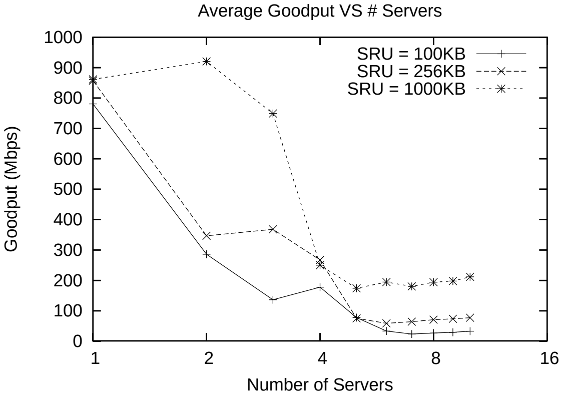


Figure 8: Effect of varying *SRU* size for Force10 S50 with QoS support enabled.

Finding	Location
Incast is caused by too-small switch output buffers: increasing buffer size can alleviate the situation.	§4.3
TCP NewReno and SACK improve goodput considerably over TCP Reno, but do not prevent *Incast*.	§5.1.1
Improvements to TCP loss recovery using Limited Transmit or reducing the Duplicate ACK threshold do not help.	§5.1.2
Reducing the penalty of a timeout by lowering the minimum retransmission value can help significantly, but poses questions of safety and generality.	§5.2
Enabling Ethernet flow control is effective only in the very simplest setting, but not for more common multi-switched systems.	§6

Table 2: Summary of Major Results.

time to transfer time.

Unfortunately, an *SRU* size of 8 megabytes is quite impractical: most applications ask for data in small chunks, corresponding to an *SRU* size range of 1-64KB. For example, when requesting an 8MB chunk from the storage system, one would like to stripe this chunk across as many servers as needed to saturate the link. In addition, the larger the *SRU* size, the more prefetching the storage system has to commit to, which allocates pinned space in the client kernel memory, thus increasing memory pressure, a prime source of client kernel failures in a fast file system implementation [1].

5 TCP-level Solutions

Because TCP timeouts are the primary reason that *Incast* hurts throughput, we analyze TCP-level solutions designed to reduce both the number and penalty of timeouts. We perform this analysis using *ns-2* simulations.

5.1 Avoiding Timeouts

In this section, we analyze three different approaches to avoiding timeouts by:

- Improving TCP's resilience to common loss patterns by using alternate TCP implementations;
- Addressing the lack of sufficient data-driven feedback;
- Reducing the traffic injection rate of exponentially growing TCP windows during Slow Start [5].

Analysis Method - Performance and Timeout Categorization: For each approach, we ask two questions: 1) how much does the approach improve goodput and 2) if timeouts still occur, why? To answer the second question, we look at the number of Duplicate ACKs Received at the point when a flow experiences a Timeout (the *DART* count). The purpose of this analysis is to categorize the situations under which timeouts occur to understand whether the timeout could have been avoided.

There are three types of timeouts that cannot be avoided by most TCP implementations. The first occurs when an entire window of data is lost and there is *no* feedback available for TCP to use in recovery, leading to a *DART* value of zero. We categorize this kind of timeout as a *Full Window Loss*.

The second type occurs when the last packet of an *SRU* is dropped and there is no further data available in this block request for data-driven recovery. We categorize this type of timeout as a *Last Packet Loss* case. We find, however, that there are relatively few *Last Packet Loss* cases.

The last unavoidable timeout situation occurs when a retransmitted packet triggered by TCP's loss recovery mechanism is *also* dropped. Since there is no way for the sender to know whether this retransmitted packet is dropped, the sender experiences a timeout before retransmitting the packet again. We categorize this unavoidable timeout as a *Lost Retransmit*. The *DART* count does not help in categorizing *Lost Retransmit* cases; we examine the TCP trace files to identify these situations.

5.1.1 Alternative TCP Implementations – Reno, NewReno, SACK

Many TCP variants help reduce expensive timeouts by using acknowledgements to more precisely identify packet losses [19, 5, 13, 22]. A well-documented problem with the classic TCP Reno algorithm is that it recovers poorly from multiple losses in a window, leaving it susceptible to patterns of loss that cause a timeout [13]. For example, with a window size of six, Reno will *always* experience a timeout when the first two packets of the window are lost.

The most popular solutions to this problem are the improved retransmission algorithms in TCP NewReno [13] and the selective acknowledgements scheme in TCP SACK [22]. TCP NewReno, unlike Reno, does not exit fast recovery and fast retransmit when it receives a partial ACK (an indication of another loss in the original window), but instead immediately transmits the next packet indicated by the partial ACK. TCP SACK uses a selective acknowledgment scheme to indicate the specific packets in a window that need to be resent [12].

Figure 9 shows that both TCP NewReno and TCP SACK outperform TCP Reno.[4] The patterns of packet loss from which TCP Reno suffers are indeed observed in this scenario. Note that TCP NewReno offers up to an order of magnitude better performance compared to TCP Reno in this example. Unfortunately, none of the TCP implementations can eliminate the large penalty to goodput caused by *Incast*.

Figure 11(a) and (b) shows the *DART* distribution for TCP Reno and NewReno, while Table 3 shows the categorization of timeout events. The total number of timeouts per data block is much lower for NewReno, partially explaining the goodput improvement over Reno. While most timeouts can be categorized as *Full Window Loss* cases or *Lost Retransmit* cases, there are still 78 timeouts that do not fall into these cases: they occur when the flows obtain *some*, but not enough feedback to trigger data-driven loss recovery. We next examine two schemes designed to improve these remaining cases.

5.1.2 Addressing the Lack of Sufficient Feedback – Limited Transmit and Reduced Duplicate ACK Threshold

When a flow has a small window or when a sufficiently large number of packets in a large window are lost, Limited Transmit [4] attempts to ensure that

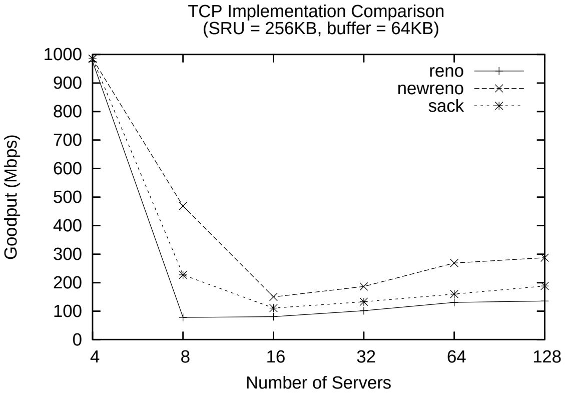


Figure 9: NewReno outperforms Reno, SACK, though *Incast* is still observed.

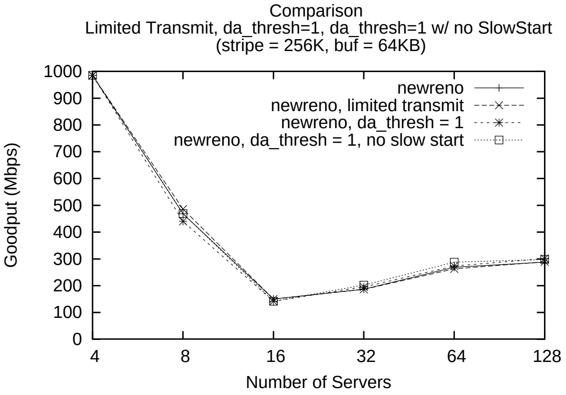
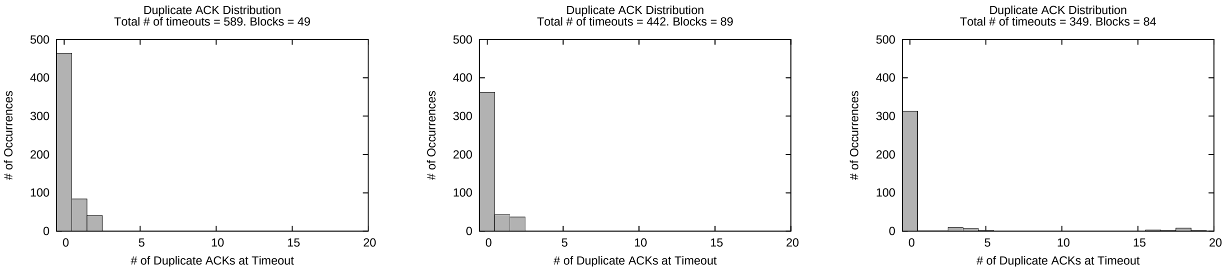


Figure 10: NewReno variants designed to improve loss recovery provide no benefit.

enough packets are sent to trigger the 3 duplicate acks necessary to enter fast retransmit and fast recovery. Alternatively we can reduce the duplicate ACK threshold (da_{thresh}) from 3 to 1 to automatically trigger fast retransmit and fast recovery upon receiving any duplicate acknowledgement.

Figure 10 illustrates that neither of these mechanisms provide *any* throughput benefit over TCP NewReno. We plot the *DART* distribution for setting da_{thresh}=1 in Figure 11(c). The reduced retransmit variant successfully eliminates timeouts when only 1 or 2 duplicate ACKs were received. Unfortunately, this improvement does not increase goodput because each data block transfer still experiences at least one timeout. These remaining timeouts are mostly due to full window losses or lost retransmissions, which none of the TCP variants we study can eliminate.

[4] We are currently investigating why NewReno outperforms SACK.



(a) Reno

(b) NewReno

(c) NewReno with $da_{thresh} = 1$.

Figure 11: Distribution of Duplicate Acknowledgements Received at a Timeout (*DART*) recorded for a 20s run with 16 servers, 64 packet switch buffer, 256KBytes SRU size.

	Reno (Fig. 11(a))	NewReno (Fig. 11(b))	NewReno + dup1 (Fig. 11(c))
# of timeout events	589	442	349
# of full window losses	464	362	313
# lost retransmits	61	2	41
# lost retransmits when $DART >= da_{thresh}$	0	0	34
# lost retransmits when $DART < da_{thresh}$	61	2	7
# last packets dropped	2	5	2
# of data blocks	49	84	89

Table 3: Table categorizing timeout events under different TCP scenarios (corresponding to Figure 11)

5.1.3 Disabling TCP Slow Start

Finally, we disable TCP Slow Start to prevent network congestion produced by flows exponentially increasing window sizes to discover link capacity following a timeout (or the beginning of a TCP transfer). Figure 10 shows that forcing TCP flows to discover link capacity using only additive increase does not alleviate the situation. We leave an analysis of even more conservative congestion control algorithms for future work.

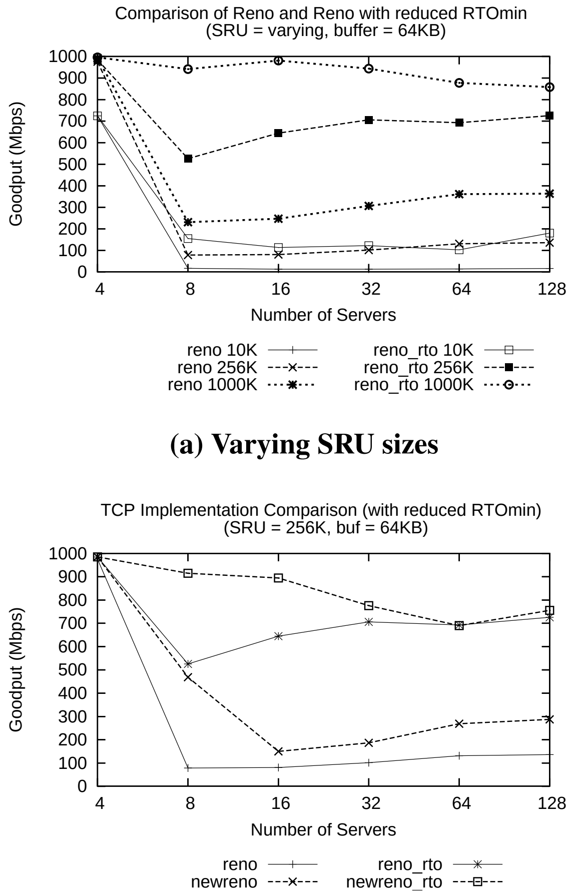
5.2 Reducing the Penalty of Timeouts

Because many of the TCP timeouts seem unavoidable (e.g. *Full Window Loss*, *Lost Retransmit*), here we examine instead reducing the time spent waiting for a timeout. While this approach can significantly improve goodput, this solution should be viewed with caution because it also increases the risk of premature timeouts, particularly in the wide-area [6]. We discuss the consequences of this effect below.

The penalty of a timeout, or the amount of time a flow waits before retransmitting a lost packet without the “fast retransmit” mechanism provided by three duplicate acks, is the retransmission timeout (*RTO*). Estimating the *RTO* value trades timely response to losses for premature timeouts. A premature timeout has two negative effects: 1) it leads to a spurious retransmission; and 2) with every timeout, TCP reduces its slow start threshold (*ssthresh*) value by half and enters Slow Start even though no packets were lost. Since there is no congestion, TCP thus would underestimate the link capacity and throughput would suffer. TCP has a conservative minimum RTO (RTO_{min}) value to guard against spurious retransmissions [27, 19].

Popular TCP implementations use an RTO_{min} value of 200ms [29]. Unfortunately, this value is orders of magnitude greater than the round-trip times in *SAN* settings, which are typically around 100μseconds for existing 1Gbps Ethernet SANs, and 10μseconds for Infiniband and 10Gbps Ethernet. This large RTO_{min} imposes a huge throughput penalty because the transfer time for each data block is significantly smaller than RTO_{min}.

Figure 12 shows that reducing RTO_{min} from 200ms to 200μs improves goodput by an order of magnitude for between 8 to 32 servers. In general,



(a) Varying SRU sizes

(b) Different TCP implementations

Figure 12: A lower RTO value ($RTO_{min} = 200\mu$s) in simulation improves goodput by an order of magnitude for both Reno and NewReno. *rto* represents runs with a modified RTO_{min} value.

for any given *SRU* size, reducing RTO_{min} results in an order of magnitude improvement in goodput using TCP Reno (Figure 12(a)). Figure 12(b) shows that even with an aggressive RTO_{min} value of 200μs, TCP NewReno still observes a 30% decrease in goodput for 64 servers.

Unfortunately, setting RTO_{min} to such a small value poses significant implementation challenges and raises questions of safety and generality.

Implementation Problems: Reducing RTO_{min} to 200μs requires a TCP clock granularity of 100μs, according the standard RTO estimation algorithm [27, 19]. BSD TCP and Linux TCP implementations are currently unable to provide this fine-grained timer. BSD implementations expect the OS to provide two coarse-grained "heartbeat" software interrupts every 200ms and 500ms, which are used to handle internal per-connection timers [7]; Linux TCP uses a TCP clock granularity of 10ms. A TCP timer in microseconds needs either hardware support that does not exist or efficient software timers [8] that are not available on most operating systems.

Safety and Generality: Even if sufficiently fine-grained TCP timers were supported, reducing the RTO_{min} value can be harmful, especially in situations where the servers communicate with clients in the wide-area. Allman et. al. [6] note that RTO_{min} can be used for trading "timely response with premature timeouts" but there is no optimal balance between the two in current TCP implementations; a very low RTO_{min} value increases premature timeouts. Earlier studies of RTO estimation in similar high-bandwidth, low-latency ATM networks also show that very low RTO_{min} values result in spurious retransmissions [28] because variation in the round-trip-times in the wide-area clash with the standard RTO estimator's short RTT memory.

6 Ethernet Flow Control

Some Ethernet switches provide a per-hop mechanism for flow control that operates independently of TCP's flow control algorithm. When a switch that supports Ethernet Flow Control (EFC) is overloaded with data, it may send a "pause" frame to the port sending data to the congested buffer, informing all devices connected to that port to stop sending or forwarding data for a designated period of time. During this period, the overloaded port can reduce the pressure on its queues.

We find that EFC is effective in the simplest configuration (i.e. all clients and servers connected to one switch), but does not work well with more than one switch, has adverse effects on other flows in all configurations, and is inconsistently implemented across different switches.

We measured the effect of enabling Ethernet flow control on a single HP Procurve 2848 switch, where one client and multiple servers were directly connected to the switch. Figure 13 shows that Ethernet flow control can significantly improve performance.

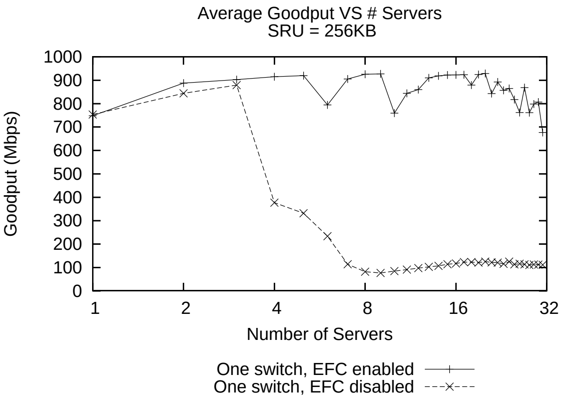


Figure 13: Enabling Ethernet flow control can mitigate *Incast*.

Unfortunately, TCP goodput remains highly variable and is lower than it would be without *Incast*.

Despite its potential benefits, our simple network topology and workload hide adverse side effects that surface when Ethernet flow control is used on larger multi-switch networks with many more clients and active TCP flows. For many of these reasons, most switch vendors and network operators keep flow control inactive.

The most significant problem is head-of-line blocking, which occurs when a pause frame originating from one congested port stops several other flows from communicating simultaneously. The effects of head-of-line blocking can be particularly severe in heterogeneous bandwidth settings where one slow link can cause other faster links to be underutilized. In other words, pause frames pause *all* traffic entering a port, regardless of whether that traffic is causing congestion.

In large part because of the complexities of head-of-line blocking, many switch vendors disable inter-switch flow control, and the particular interactions of multiple switches is not standardized. For instance, in order to provide link aggregation between two HP Procurve 2848 switches, our system was configured with a virtual interface for the trunk over which the switch could not enable flow control.

7 Related Work

Providing storage via a collection of storage servers networked using commodity TCP/IP/Ethernet components is an increasingly popular approach. The *Incast* problem studied in depth in this paper has been noted by several researchers (e.g., [15, 17, 25, 24]) in developing this approach.

Nagle et al. [25] briefly discussed the switch buffer overruns caused by clients reading striped data in a synchronized many-to-one traffic pattern. Upgrading to better switches with larger buffer sizes was one solution used by these researchers. They also mentioned the possibility of using link-level flow control, but focus on its difficulty handling non-trivial switch topologies effectively without understanding the higher level notion of striping.

In later work, Nagle et al. [24] again report on the effects of *Incast* on scalable cluster-based file storage performance. Specifically they report on experiments with a real product-quality system where a single client reads a file sequentially using an 8MB synchronization block size striped across multiple storage servers. As the number of storage servers is increased, keeping all other variables of the network constant, the authors observe a linear scaling of storage bandwidth for up to 7 storage servers, a steady plateau until around 14 servers, and then a rapid dropoff. The primary cause of this performance collapse is attributed to multiple senders overwhelming the buffer size of the network switch. This prior work also observed that the *Incast* problem does not appear when a streaming network benchmark like `netperf` is run. The main reason for the performance collapse is therefore identified to be the synchronized and coordinated reads in the SAN environment. Nagle et al. also discuss modest performance gains introduced by using SACK and reducing the TCP retransmission timeouts. Although this last point is not quantified, the paper observes that problems of degraded performance still persist with these changes.

Our work builds upon these papers by analyzing and explaining *why Incast* causes the problems observed and quantifying the effects of various TCP- and Ethernet-level modifcations.

The *Incast* problem studied here represents a specific form of network congestion. Early work on congestion control in the wide-area by Van Jacobson addressed the TCP congestion collapse of the Internet around 1985 [19]. Adopted as the basis of TCP congestion control, the idea was to provide a method for a connection to discover and dynamically adjust to the available end-to-end bandwidth and send at that rate. Chiu and Jain [11] describe why the window mechanism of "additive increase / multiplicative decrease" achieves fairness and stability.

Unfortunately, TCP's congestion control and avoidance algorithms are not directly applicable to all settings. For example, they are known to struggle with wireless settings, where packet losses may not be due to congestion, and with high-latency, high-bandwidth network settings [20]. The incast problem is another example, as explained in this paper.

The performance and fairness of TCP when many flows share the same bottleneck was studied by Morris [23]. As the number of TCP flows through a bottleneck increases to the point when there are more flows than packets in the bandwidth-delay product, there is an increasingly high loss rate and variation of unfair bandwidth allocation across flows. This paper applies some of Morris's methods and analysis techniques to the synchronized read scenario that causes *Incast*.

8 Conclusion

TCP *Incast* occurs when a client simultaneously receives a short burst of data from multiple sources, overloading the switch buffers associated with its network link such that all original packets from some sources are dropped. When this occurs, the client receives no data packets from those sources and so sends no acknowledgement packets, requiring the sources to timeout and then retransmit. Often, the result is an order of magnitude decrease in goodput.

Unfortunately, this traffic pattern is very common for the growing class of cluster-based storage systems. When data is striped across multiple storage nodes, each client read creates this pattern and large sequential reads create it repeatedly (once for each full stripe).

Whether or not TCP *Incast* will cause goodput collapse in a system depends on details of the TCP implementation, network switch (esp. buffer sizes), and system configuration (e.g., the number of servers over which data is striped). Unfortunately, avoiding collapse often requires limiting striping to a small number of servers. Techniques such as very short timeouts and link-level flow control can mitigate the effects of TCP *Incast* in some circumstances, but have their own drawbacks. No existing solution is entirely satisfactory, and additional research is needed to find new solutions by building on the understanding provided by this paper.

References

[1] Private communication with Jeff Butler, Panasas Inc.

[2] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/, 2006.

[3] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: Versatile cluster-based storage. In *FAST*, 2005.

[4] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042 (Proposed Standard), January 2001.

[5] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[6] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. *SIGCOMM Comput. Commun. Rev.*, 31(2 supplement):124–151, 2001.

[7] Mohit Aron and Peter Druschel. TCP implementation enhancements for improving webserver performance. Technical Report TR99-335, 6, 1999.

[8] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst.*, 18(3):197–228, 2000.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *HotOS*, pages 33–38, 2001.

[10] Peter J. Braam. File systems for clusters from a protocol perspective.

[11] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, 1989.

[12] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[13] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582 (Experimental), April 1999. Obsoleted by RFC 3782.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[15] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 92–103, New York, NY, USA, 1998. ACM Press.

[16] G. Grider, H.B. Chen, J. Junez., S. Poole, R. Wacha, P. Fields, R. Martinez, S. Khalsa, A. Matthews, and G. Gibson. PaScal - A New Parallel and Scalable Server IO Networking Infrastructure for Supporting Global Storage/File Systems in Large-size Linux Clusters. In *Proceedings of the 25th IEEE International Performance Computing and Communications Conference, Phoenix, AZ*, April 2006.

[17] Roger Haskin. High performance NFS. Panel: High Performance NFS: Facts & Fictions, SC'06.

[18] Dean Hildebrand, Peter Honeyman, and Wm. A. Adamson. pnfs and linux: Working towards a heterogeneous future. In *8th LCI International Conference on High-Performance Cluster Computing*, Lake Tahoe, CA.

[19] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM Press.

[20] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, New York, NY, USA, 2002. ACM Press.

[21] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, 1985.

[22] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[23] R. Morris. TCP behavior with many flows. In *ICNP '97: Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, page 205, Washington, DC, USA, 1997. IEEE Computer Society.

[24] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[25] David F. Nagle, Gregory R. Ganger, Jeff Butler, Garth Goodson, and Chris Sabol. Network support for network-attached storage. In *Hot Interconnects*, Stanford, CA, 1999.

[26] Brian Pawlowski and Spencer Shepler. Network file system version 4 (nfsv4) charter page.

[27] Vern Paxson and Mark Allman. Computing TCP's Retransmission Timer. RFC 2988.

[28] Allyn Romanow and Sally Floyd. Dynamics of TCP traffic over ATM networks. *SIGCOMM Comput. Commun. Rev.*, 24(4):79–88, 1994.

[29] Pasi Sarolahti and Alexey Kuznetsov. Congestion control in Linux TCP. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 49–62, Berkeley, CA, USA, 2002. USENIX Association.

[30] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[31] S. Shepler, M. Eisler, and D. Noveck. NFSv4 minor version 1 – draft standard.