On Application-level Approaches to Avoiding TCP Throughput Collapse in Cluster-based Storage Systems

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

ABSTRACT

TCP *Incast* plagues scalable cluster-based storage built atop standard TCP/IP-over-Ethernet, often resulting in much lower client read bandwidth than can be provided by the available network links. This paper reviews the *Incast* problem and discusses potential application-level approaches to avoiding it.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Design, Performance

Keywords

Cluster-based Storage, TCP, *Incast*

1. INTRODUCTION

Cluster-based storage systems are becoming an increasingly important target for both research and industry [1, 15, 10, 13, 9, 6]. 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. In addition, 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. This paper discusses approaches to addressing an important barrier to high-performance storage over TCP/IP: the *Incast* problem [14, 13].

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. 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 [5]. 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.

Our recent study of the root network-level causes of *Incast* determined that some solutions exist to delay its catastrophic effects, such as increasing the amount of buffer space in network switches [14]. However, as the number of source nodes involved in a parallel data transfer are increased, at some point any particular switch configuration will experience throughput collapse. Our previous analysis of TCP-level solutions showed some improvement when moving from Reno to NewReno TCP variants, but none of the additional improvements helped significantly. Other techniques to mask the *Incast* problem, such as drastically reducing TCP's retransmission timeout timer, can further improve performance but not without additional drawbacks.

Without a clear network or transport-level solution to prevent *Incast* with an arbitrarily large number of participating storage servers, the remaining option is to avoid *Incast* by architecting the system to perform well under a limited scale. Example avoidance techniques involve limiting the number of servers responsible for a block request and throttling the send rate of servers. However, as we move to larger and more highly parallelized petascale environments, restrictions of scale become much more constraining.

This paper summarizes existing approaches for reducing the effects of TCP *Incast* and discusses potential application-level solutions. The "application" in this case is the distributed storage system software. It has context about the parallel transfer, such as the number of servers over which the required data is striped, that is not available at the network layer and below. It also has control over when data transfers are initiated. Such knowledge and control could be used to avoid *Incast* by limiting the number of servers accessed at a time, staggering data transfers from those servers, and/or explicitly scheduling data transfers. These

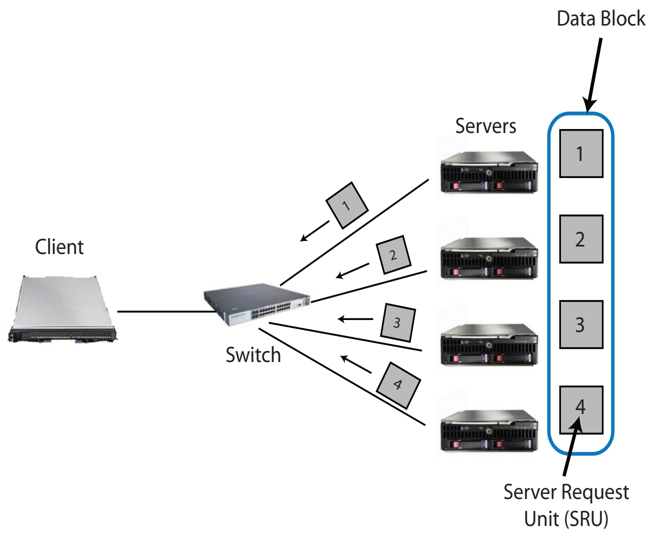

Figure 1: Terminology for a synchronized reads environment, where a client requests data from multiple servers.

application-level approaches may avoid *Incast* altogether, allowing the scaling of cluster-based storage toward petascale levels atop commodity TCP/IP and Ethernet networks.

2. THE *INCAST* PROBLEM

This section describes the *Incast* problem and discusses TCP and Ethernet-level approaches to reducing its impact.

2.1 Throughput Collapse of Synchronized Reads

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)* (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*.

Most networks are provisioned such that the client's bandwidth to the switch should be the throughput bottleneck of any parallel data transfer [11, 12]. 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 a catastrophic performance drop in a cluster-based storage network environment when a client requests data from four or more servers.

TCP timeouts are the primary cause of this goodput collapse [14], where we define goodput as the data throughput observed by the application as contrasted with the amount of data passed over the network (which includes retransmissions). When goodput degrades, many servers still send their *SRU* quickly, but one or more other servers experience

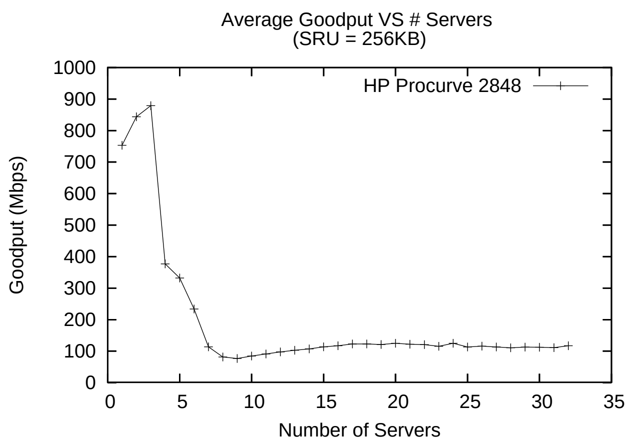

Figure 2: TCP throughput collapse for synchronized reads performed on a storage cluster.

a timeout due to packet losses. During this timeout period, servers that have completed their data transfers must wait for the client to receive the complete data block before the next block is requested, resulting in an underutilized link.

2.2 TCP- and Ethernet-level Improvements

Incast poses a significant barrier to improving TCP throughput in petascale environments, where the number of servers communicating simultaneously with a client will need to be increased to effectively use free network bandwidth. In this section, we summarize the results of a recent study showing the effectiveness of several TCP and Ethernet level approaches to mitigating *Incast* [14]. None of these existing solutions are sufficient to prevent *Incast*.

Switches with larger output buffer sizes can mitigate *Incast*. With larger buffers, fewer packets are dropped, resulting in fewer timeouts and higher TCP throughput. Measurements indicate that a two-fold increase in buffer space can support data striping over twice as many servers. Unfortunately, switches with larger buffers tend to cost more, forcing system designers to maintain a difficult balance between over-provisioning, future scalability, and hardware budget constraints.

Recent TCP implementations show improvements by more effectively avoiding timeout situations, but they do not prevent *Incast*. Choosing TCP NewReno [4] or TCP SACK [8] over TCP Reno shows a modest improvement but still suffers from throughput collapse given enough servers. Further improvements to TCP loss recovery, such as Limited Transmit [2], show little to no improvement beyond TCP NewReno.

Reducing the penalty of timeouts by lowering TCP's minimum retransmission timeout value can help significantly, but cannot avoid *Incast* at a larger scale of servers. Furthermore, reducing the timeout value to sufficiently low levels is difficult, requiring a TCP timer granularity in microseconds, and is questionable in terms of safety and generality [3].

Ethernet flow control can be effective when all communicating machines are on one switch, but when scaled in more common multi-switched systems (as is expected for petascale environments), it has adverse effects on other flows in all configurations and is inconsistently implemented across different switches.

There are a number of current Ethernet initiatives to add congestion management with rate-limiting behavior and a granular per-channel link level flow control [16]. These initiatives aim to achieve "no-drop" Ethernet responses to congestion. However, it will take a number of years before these new standards are added to switches.

These TCP- and Ethernet-level solutions represent a number of ways to reduce the negative effects of *Incast* for a fixed number of servers engaging in a synchronized data transfer. But, at some number of servers, any particular switch configuration will experience throughput collapse. Since TCP timeouts are nearly unavoidable when TCP loses either packet retransmissions or all the packets in its window, and the length of each timeout cannot be further reduced due to implementation problems and spurious retransmissions, we believe that improved TCP implementations have little hope in preventing *Incast*.

3. APPLICATION-LEVEL SOLUTIONS

Since a solution to *Incast* does not appear to exist in the TCP or Ethernet layer, we focus here on potential application-level solutions that might more effectively manage the available bandwidth in a storage system.

3.1 Increasing Request Sizes

A common approach in storage systems is to amortize the cost of disk seek time by requesting more data from each disk. In a synchronized reads scenario, larger request sizes provide an additional benefit at the network layer. Increasing the SRU size at each server reduces the amount of idle link time, and when some flows have stalled other flows continue to send more useful data [14]. Whenever possible, to avoid *Incast* clients should request more data from a smaller number of servers. Even when striping across fewer servers is not appropriate, a client can reduce the impact of *Incast* by condensing many small requests for data into larger requests. However, larger SRU sizes also require the storage system to allocate pinned space in the client kernel memory, thus increasing memory pressure, which is a prime source of client kernel failures in a fast file system implementation [7].

3.2 Limiting the Number of Synchronously Communicating Servers

A common theme of application-level approaches to prevent the onset of *Incast* is to restrict the number of participating servers in a synchronized data transfer. For example, one might experiment with the number of servers that are involved in a synchronized data transfer, identify at what scale servers begin to experience degraded throughput, and limit the parallelism of all subsequent data transfers to be within that acceptable range.

From our discussions with industry specialists, we know that, among other things, Panasas uses a variant of the above idea by limiting the number of servers participating in any one data transfer. They divide a larger pool of servers into smaller RAID groups of a restricted range and limit any communication to one RAID group at a time. A single block still cannot be spread over more servers than are available in any one RAID group, but different blocks from a file may be distributed over different RAID groups, spreading the load of requests over a larger number of servers.

3.3 Throttling Data Transfers

A client can throttle the servers' send rates by advertising a smaller TCP receive buffer. This throttling technique will artificially limit the TCP window size of any request, improving scalability as more requests are made in parallel and to a larger number of servers. Unfortunately, a static throttling rate may have the adverse effect of underutilizing the client's link capacity if TCP windows are not allowed to increase to the proper size. A client that discovers a well-performing throttle rate for one request that avoids *Incast* must also either avoid making multiple requests in parallel or adjust the throttle rate when necessary. This lack of generality may make these throttling techniques less palatable.

3.4 Staggering Data Transfers

Another approach to limiting the amount of data being transferred synchronously is to stagger the server responses so only a subset of servers are sending data at any one time. The control decisions required to produce this staggering effect can either be made by the client or at the servers.

A client may stagger data transfers by requesting only a subset of the total block fragments at a time. Alternatively, a client may achieve similar behavior with a smaller data block size spread across fewer servers, but this alternative would be impractical for a system that uses a fixed data block size or prefers a specific level of erasure coding. By staggering requests to a limited number of servers, or even maintaining a window of concurrent requests to servers, the client can avoid requesting too much data at once.

Servers can collectively skew their responses by making either random or deterministic localized decisions to delay their response to a request. Each server responsible for a data fragment can determine the total number of other servers responding to a request, either communicated directly from the client or based on the level of erasure coding. With this number, a server can wait an explicitly preassigned time before beginning its data transfer or choose a random period to skew its response. By choosing the right amount of skew and delaying the server's response, a request may actually perform better in the long run by avoiding costly TCP timeouts. Servers can also use the delay time to service other requests and prefetch data into the cache.

Previous studies of *Incast* have assumed that data is always cached in memory to isolate *Incast* as a mostly networking problem. In real systems, however, there may be some inherent staggering in transfer responses based on many factors, including the network switch configuration imposing different delays on each server and whether the requested data is in cache or on disk. For example, one server may have data cached in memory and be able to respond to a request packet immediately, while another server may require a disk seek and experience some disk head positioning delay potentially larger than an SRU transfer time. Thus, real systems may exhibit some of the benefits of staggering, although the existence of *Incast* in the real world suggests that this natural staggering effect is not sufficient.

3.5 Global Scheduling of Data Transfers

Since a client might be running many workloads and making multiple requests to different subsets of servers, any complete solution affecting when and how a server should respond cannot be made without information about all workloads. In these multiple workload situations, global scheduling of data transfers is required. One instantiation of such scheduling would use *SRU tokens*, where assuming each client has a dedicated leaf switch port into the network, a server cannot transfer data to a client unless it has that client's *SRU token*; after receiving a data request, a server may also prefetch data while it waits for the appropriate *SRU token*.

For example, the storage system may learn that it can only send data to a given client from k servers before experiencing *Incast*; the system would create k *SRU tokens* for each client in a global token pool. A client can send request packets to all servers containing data for its multiple requests, but only the k servers that have been allocated that client's *SRU tokens* can actually transfer the data. This will restrict the total number of simultaneous servers sending data to any given client to the optimal value k. The storage system might obtain the value of k either through manual configuration or real-time system measurements.

When a server has no more use for a token, the token should pass back into the token pool and be made available to other servers that have data to send to the token's associated client. There are several algorithms for deciding where and how to pass the token. A simple round-robin token passing approach might work for small networks but does not scale well. Instead, either the client or a separate logical entity might act as a *token authority*, issuing tokens when requested, maintaining a queue of servers waiting for each client's token, and receiving tokens back when the server relinquishes it or the token expires. A *token authority* can also help to load balance requests across the network by showing preference to servers with higher numbers of prefetched requests, to relieve those servers' pinned memory allocations. For petascale environments, the token authority should be physically distributed to handle token traffic. In all cases, the storage system will attempt to time-share the client's link among requests across single and multiple workloads.

4. SUMMARY

TCP *Incast* occurs when a client simultaneously asks for data from enough servers to overload the switch buffers associated with its network link. This paper discusses a number of application-level approaches to avoiding the *Incast* problem, and continuing research will explore their efficacy.

5. ACKNOWLEDGMENTS

We would like to thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, Network Appliance, Oracle, Seagate, and Symantec) for their interest, insights, feedback, and support. This research is sponsored in part by the National Science Foundation, via grants #CNS-0546551, #CNS-0326453 and #CCF-0621499, by the Army Research Office under agreement DAAD19-02-1-0389, by the Department of Energy under Award #DE-FC02-06ER25767, and by DARPA under grant #HR001107-10025. Elie Krevat is supported in part by an NDSEG Fellowship from the Department of Defense.

6. REFERENCES

[1] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proceedings of 4th USENIX Conference on File and Storage Technologies*, 2005.

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

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

[4] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), Apr. 1999.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *HotOS*, 2001.

[6] P. J. Braam. File systems for clusters from a protocol perspective. In *Second Extreme Linux Topics Workshop*, June 1999.

[7] J. Butler. Panasas Inc. Personal Communication, August 2007.

[8] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, Apr. 2004.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, 1998.

[11] G. Grider, H. 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*, Apr. 2006.

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

[13] D. Nagle, D. Serenyi, and A. 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.

[14] A. Phanishayee, E. Krevat, V. Vasudevan, D. Andersen, G. Ganger, G. Gibson, and S. Seshan. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. Technical Report CMU-PDL-07-105, Sept 2007.

[15] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of 1st USENIX Conference on File and Storage Technologies*, 2002.

[16] M. Wadekar. Enhanced ethernet for data center: Reliable, channelized and robust. In *15th IEEE Workshop on Local and Metropolitan Area Networks*, June 2007.