Trevi: Watering Down Storage Hotspots with Cool Fountain Codes

George Parisis[1,2], Toby Moncaster[2], Anil Madhavapeddy[2] and Jon Crowcroft[2]

School of Engineering and Informatics, University of Sussex[1] and Computer Laboratory, University of Cambridge[2]
G.A.Parisis@sussex.ac.uk[1] and {firstname.lastname}@cl.cam.ac.uk[2]

ABSTRACT

Datacenter networking has brought high-performance storage systems' research to the foreground once again. Many modern storage systems are built with commodity hardware and TCP/IP networking to save costs. In this paper, we highlight a group of problems that are present in such storage systems and which are all related to the use of TCP. As an alternative, we explore Trevi: a fountain coding-based approach for distributing I/O requests that overcomes these problems while still efficiently scheduling resources across both networking and storage layers. We also discuss how receiver-driven flow and congestion control, in combination with fountain coding, can guide the design of Trevi and provide a viable alternative to TCP for datacenter storage.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks, network communications*

General Terms

Design, Performance, Reliability

Keywords

Datacenter Storage; Fountain Coding; TCP Incast

1. INTRODUCTION

Datacenters bring new challenges to the design and operation of storage systems. Several conflicting requirements need to be met at the same time, including scalability, data integrity [12, 3] and resilience, consistency and line-speed performance. Often, the only cost-effective solution is to relax some of the requirements. Over the years, client-server systems [21, 5, 9] have been succeeded by distributed systems that share functionality across multiple nodes in the network. In some cases, metadata servers are used to resolve the location of data and help maintaining an updated view of the storage resources so that consistency and data resilience is preserved in case of failures [6, 19, 26, 10]. Other systems distribute said functionality to multiple nodes or even across *all* storage nodes in order to support decentralisation and ease of management [30, 20, 18, 25, 24, 31].

We present *Trevi*, a radical new storage mechanism that uses fountain codes [14, 27] to provide resilience, high performance, and efficient resource utilisation. Trevi runs on top of UDP or even raw Ethernet and does not depend on TCP. We leverage the inherent ability of fountain coding to multicast data blobs across storage nodes, and to utilise multiple replicas in parallel when reading data. Our approach is tolerant of losses and requires no retransmissions. Trevi requires no changes to the network stack; instead, we build on top of UDP or raw Ethernet frames, thus ensuring easier deployability.

Common to most existing systems is the necessity of balancing high throughput while keeping the deployment costs low. Consequently, they are built using commodity hardware and use TCP to communicate. The use of TCP leads to a number of known limitations that Trevi mitigates, as described next.

TCP Incast

A well known consequence of using TCP is TCP incast; *"a catastrophic TCP throughput collapse that occurs as the number of storage servers sending data to a client exceeds the ability of an Ethernet switch to buffer packets"* [22]. Incast is obvious for specific I/O workloads such as synchronised reads, but can also occur whenever severe congestion plagues the network, as TCP's retransmission timeouts are orders of magnitude higher than the actual Round Trip Times (RTTs) in datacenter networks. Several techniques have been proposed to

HotNets-XII, November 21–22, 2013, College Park, MD, USA.

mitigate TCP Incast [29, 33, 32], but their deployment is hindered due to requiring extensive changes in the OS kernel or the TCP protocol itself, or by requiring network switches to actively monitor their queues and report congestion to end-nodes. Adopting fountain coding eliminates the need for retransmission upon packet loss. Instead, additional encoded symbols are transmitted until the receiver can decode the missing data (§2.2).

Trading network resources for resilience

Existing systems either send multiple copies of the same data [10, 30, 20] or apply an erasure code to the data and send the encoded pieces to multiple storage nodes [1, 8] to support resilience. The first approach effectively divides the performance of every write request by the number of stored replicas since each one of them has to be unicast separately. The latter does better in terms of required storage space, but erasure blocks need to be updated when writing to one or more blocks of data, and their old value must be fetched before the update. With fountain coding, write requests can be multicast to all replica points, thereby minimising network overhead and increasing energy efficiency (§2.3). Trevi, operating as a transport mechanism, is orthogonal to the usage of erasure codes as a means to provide storage resilience and it can work when either k-way replication or erasure coding is employed in the storage system.

Expensive switches to prevent packet loss

Realistic solutions to the TCP Incast problem are often solved by using network switches with large (and energy-hungry) memory for buffering packets in-flight. Packet loss or out-of-order receipt is much less important when using fountain coding. Consequently we can reduce the size of network buffers, or treat storage traffic as a lower QoS class with limited buffer space. Hence, our approach requires less energy to power the memory required for buffering storage requests (§2.2).

Lack of parallelism when multiple replicas exist

Systems that store copies of the same data in multiple locations (and where consistency among replicas is maintained or outdated replicas are flagged) only use a single storage node to fetch the data, leaving the rest of the nodes idle. Data reads are parallelised by striping a single blob to multiple disks and hoping that I/O requests are uniformly large. Usually, deep *read-ahead* policies are employed to force the system to fetch multiple stripes simultaneously, although this approach can be wasteful for workloads with small random reads. By contrast, fountain coding allows simultaneous multiple sources when reading data. This leads to more efficient utilisation of storage resources even when the I/O workload cannot itself be parallelised in situations such as smaller read requests involving a single stripe (§2.4).

No (or basic) support for multipath transport

Most datacenters now offer multiple equal (or near-equal) cost paths through their fabric [11, 2] to support multipath transport, but exploiting these is hard. Protocol extensions like multipath TCP [23] require extensive changes in the network stack. Other efforts seek to balance flows across different paths in a datacenter in a deterministic and rather static fashion [2, 13]. More dynamic approaches to balance packets across different paths to the same host are prohibitive because out-of-order packets can degrade TCP's performance significantly. Fountain coding schemes are much more welcoming to such dynamic balancing, since all symbols are useful and there is no notion of out-of-order packets. Unlike TCP (where balancing happens on a per-flow basis to avoid out-of-order packets throughout a flow's lifetime), in our approach encoded symbols can be balanced independently. This provides a lot more flexibility in the design of in-network multipath mechanisms (§2.2).

2. A STRAWMAN DESIGN

We start with a strawman design that focuses on how blobs are transferred between clients and servers. We abstract out details of the OS integration (e.g. as a distributed block device, file system or key-value store), and omit details of how blobs are resolved to storage nodes, failure recovery and system expansion. Trevi can be integrated in any storage systems where blobs or pieces of blobs are assigned to storage servers deterministically.

Our description is based on a simplified version of the Flat Datacenter Storage (FDS) system [18]. In FDS, data is logically stored in blobs. A blob is a byte sequence named with a 128-bit GUID. The GUID can be selected by the application or assigned randomly by the system. Reads from and writes to a blob are done in units called tracts. Every disk is managed by a process called a *tract server* that serves read and write requests which arrive over the network from clients. FDS uses a metadata server, but its role during normal operations is simple and limited: collect a list of the system's active tract servers and distribute it to clients. This list is called the *tract locator table (TLT)*. In a single-replicated system, each TLT entry contains the address of a single tract server. With k-way replication, each entry has k tract servers. To read or write a tract from a blob, a client first selects an entry in the TLT by computing an index into it by hashing the blob's GUID and adding the index of the tract in the blob. This process is deterministic and returns the same set of tract servers to all clients that hold an up-to-date version of the TLT.

The only extension required to support Trevi is the addition of a column that stores some multicasting information (e.g. an IP multicast group) to the TLT. When nodes fail, or new nodes join the system, the TLT, which is cached locally to all clients, is updated [18]. Note

that storage nodes subscribe to multicast groups in a deterministic fashion when they receive the TLT, and update their subscriptions when servers join or leave the storage network. Trevi's operation does not involve subscribing to and unsubscribing from multicast groups to transfer data among clients and servers.

2.1 Fountain coding-based blob transport

Fountain coding [14, 27] is central to our approach as it allows us to provide a storage service that is tolerant to packet loss, but without retransmissions or timeouts. It requires extra computing resources to encode and decode symbols and it has a small penalty in terms of bandwidth due to requiring a slightly larger number of encoded symbols than the initial number of fragments to decode the original information. This overhead can be as low as 5% [7] and proprietary raptor code implementations report a network overhead of less than 1%.[1]

As depicted in Figure 1, the sender calculates how many MTU sized fragments of the blob will be encoded in each symbol; this is called the *degree* of the encoded symbol. For example, the degree of encoded symbol *2* is 3. Selecting the degree is a crucial step in the process as it affects the efficiency of the information delivery in terms of the number of symbols required to decode the blob, and the decoding complexity.

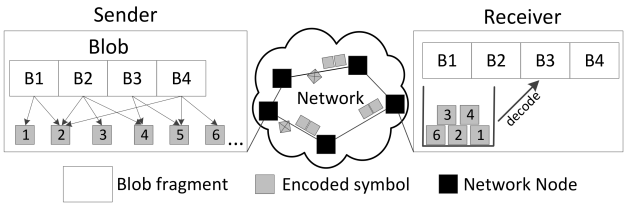

Figure 1: Fountain Coding Blobs

Different statistical distributions have been proposed for different coding techniques [14, 27]. The sender uniformly selects *degree* number of fragments and encodes them into the final symbol by XORing them; this set is called the symbol's *neighbours*. For instance, the neighbour set of symbol *2* includes fragments *B1*, *B2* and *B4*. A receiver utilizes symbols with degree 1 to partially or fully decode other symbols by XORing them with the decoded symbol. In Figure 1, the receiver utilizes the encoded symbol *1*, to decode fragment *B1*. It then XORs it with symbol *2*. *B2* is decoded using symbol *3*, which is also XORed with symbol *2*. Symbol *2*, then, only contains *B4*, which is decoded.

As shown in Figure 1, symbols may get lost on the way or travel via different paths. This is totally transparent to the endpoints. All encoded symbols contribute to the decoding process regardless the order they arrive, therefore there are no out-of-order symbols and, consequently, there is no need to keep or care about any sequencing information.

[1]Source: Qualcomm website article "why raptor codes are better than reed-solomon codes for streaming applications"

2.2 Receiver-driven flow control

Traditionally, the fountain coding transport model is *push* based. Senders start sending symbols and continue until all receivers have decoded the data and sent a notification to the sender or unsubscribed from a multicast group. In receiver-driven layered multicast [16], receivers play a more active role by subscribing to and unsubscribing from multicast groups that represent different coding layers, according to the network congestion.

In Trevi receivers actively manage the rate at which encoded symbols arrive (effectively providing flow and congestion control), by employing a receiver-driven *pull* communication scheme where a sender sends one or more symbols only when explicitly requested by a receiver. In §4.1, we discuss the applicability and benefits of a mixed, push/pull transport scheme.

Trevi receivers include a statistically unique label when requesting an encoded symbol so that the RTT can be calculated upon receiving a symbol sent in response to that request (and therefore carrying the same identifier). No action is taken when symbols are lost. We use labels instead of sequence numbers since packets can arrive out-of-order.

A receiver-driven approach for requesting symbols simplifies flow and congestion control and guarantees that no extra symbols are sent after the receiver decodes the initial data. A receiver adjusts the number of pending symbol requests (called the *window* of requests) to handle changes in: 1) the rate at which a storage server can store data, 2) the rate at which a sender can send data and 3) the congestion in the network.

The first point has not been solved in past systems, especially for storage servers with spinning disks. In such cases the network bandwidth can be much higher than the disk array's throughput. Hence, there is no point in a storage server requesting more data than the amount it can actually store, sparing the extra bandwidth for others in the network. TCP's flow and congestion control would allow to transfer more data than what a server can actually store. This data would be buffered in memory until stored but some network bandwidth could have been spared to other data transfers.

The data rate of a sender is variable because it may serve multiple requests from receivers at the same time. Our approach ensures that senders will be requested to send encoded symbols at a rate that they can actually cope with. This rate can be achieved by adjusting the window of pending symbol requests when the RTT increases. Note that the RTT also increases when symbols are buffered in switches, but in both cases the window of pending requests should be decreased.

Congestion can be inferred and avoided by monitoring variations of RTTs for each symbol request. Receivers react when congestion occurs in the network by decreasing the number of pending symbols' requests. Additionally, losses in the network can be estimated since a receiver can know for which requests respective symbols did not arrive. It is worth highlighting that the notion of the window, as introduced above, is different from the classic TCP flow and congestion windows. There are no timeouts and no retransmissions in Trevi, and instead some internal timeouts which are only necessary to remove stale requests from the current window and update the loss statistics. These timers are adjusted based on the monitored RTTs but do not trigger any retransmission requests. In the worst case, if such a timer expires and an encoded symbol arrives after the respective request was removed, the receiver just increases the timer value for the upcoming requests; there is no penalty for the early timer expiration because the encoded symbol will be used in the decoding process just like any other symbol (there are no out-of-order symbols!).

We are still developing the details about the pace at which receivers populate the window of requests when starting to receive a new blob, as well as how they adjust its size when one of the three conditions mentioned above change and, therefore, we will not elaborate more in this paper but we envision approaches similar to [4] and [28].

Packet losses are less important than in TCP; we do not identify packets with sequence numbers, we do not ask for specific packets and, consequently, there is no notion of retransmissions because of timeouts. Hence, there is no need to extensively buffer packets and desperately try to deliver them to their destination. Less buffering means cheaper and energy-efficient switches and/or more buffering for other TCP traffic which can be clearly isolated from Trevi's traffic.

2.3 Multicasting data

Write requests for a (part of a) blob are first resolved utilising the hash of a blob's identifier to locate it in the Tract Locator Table (TLT) (step 1 in Figure 2). This gives the client the addresses of individual servers as well as the multicast group(s) to which it should send the tracts in the write request. Tract storage servers are assumed to be already subscribed to the right multicast groups, according to the information in the TLT (all entities in the storage network have the same view of the TLT [18]).

When a client needs to write a tract to a number of servers, it first sends a *prepare* notification to all replica points (step 2). This notification must be sent in a reliable way, either via a separate control TCP connection or by employing a retransmission mechanism for this packet, which includes the identifier of the specific tract (an encoded symbol can be piggybacked in the

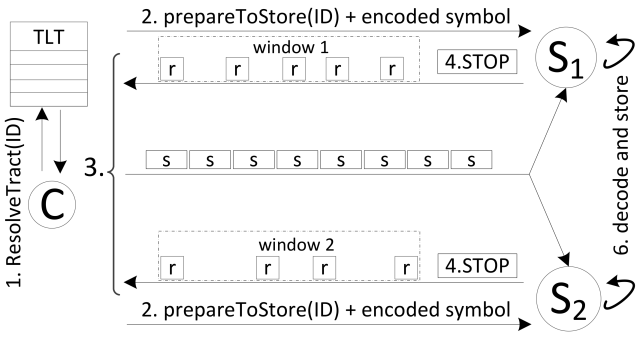

Figure 2: Multicasting Write Requests

notification). Upon receiving this notification, storage servers start requesting encoded symbols from the client (for write requests, servers are the receivers of symbols) (denoted as r in step 3).

It is important to note that although servers run their own flow and congestion control window, the client always sends encoded symbols based on the requests coming from the slowest server. The rest of the servers slow down the rate at which they request symbols to match the incoming rate (the separate windows of requests converge to the one of the slowest storage server). This way the client is able to multicast encoded symbols, denoted as s in step 3, to all servers at a rate defined by the slowest server (this feature is beneficial for the network because the multicasting rate is smoothed by the slowest server). Replication is by definition a synchronised operation which is completed only when all replicas acknowledge the reception of a tract. In §4.3, we describe a potential optimisation in case one storage server is straggling.

Finally, each server sends a *stop* notification containing the identifier of the stored tract, which must arrive to the client reliably (step 4). The client stops sending encoded symbols after receiving such notifications from all storage servers that store the specific tract.

In our approach data replication happens with the minimum network overhead and more energy-efficiency by just multicasting data to a deterministically chosen set of nodes. Existing systems [30, 18, 20] select nodes for storing data deterministically, and therefore the only requirement is to have these nodes subscribing to a multicast group specific to the dataset assigned to them.

2.4 Multisourcing data

Reading tracts out of storage nodes follows similar lines. After resolving the nodes that store a specific tract (step 1 in Figure 3), a client sends a *get blob* request to all these nodes (step 2). All servers acknowledge the reception of the request (step 3) and a symbol can be piggybacked in the acknowledgement packet (Figure 3).

After receiving acknowledgements from all servers, the

Figure 3: Multisourcing Read Requests

client simultaneously requests encoded symbols from all storage servers that hold an updated version of the tract (for read requests, the client is the receiver of symbols). Some systems [18, 30, 20] provide mechanisms to ensure that nodes with outdated data are never selected to fetch data. Trevi adopts a similar approach to ensure that such nodes will never be chosen. As shown in Figure 3, the client keeps separate windows of pending symbols' requests for each storage server.

Each storage node creates and sends encoded symbols in an independent and uncoordinated way in response to requests from the client (step 4). The only requirement here is that symbols created by storage nodes have a very high probability of being different, so we ensure storage nodes randomise the seed used when calculating the degree of each symbol. Different seeds will produce statistically different encoded symbols. Note that there is no need for any kind of synchronisation for this scheme to work. Servers transmit symbols at different rates (defined by the client's flow control mechanism) and each server only contributes the number of symbols that it is able to produce and transmit. Servers never send an encoded symbol unless they are requested to do so.

Finally, the client reliably sends a *stop* request (step 5) to (separately) let each server know that it decoded the requested blob. The whole procedure ends when the client passes the decoded blob to the application or the file or block subsystem (step 6).

The multisource transmission provided by Trevi allows storage resources to be fully utilised even when the I/O workload cannot be parallelised (e.g. for smaller read requests involving a single stripe). More specifically, all storage nodes that hold an updated version of some data can contribute to the transmission of the data to a client. This feature provides a second, inherent level of load balancing when fetching data, the first being the striping of blobs to multiple storage nodes.

3. THE PRICE TO PAY

In the previous sections we discussed what the problems are with existing storage systems that are based on commodity hardware and operate on top of TCP and how Trevi deals with them. In this section we discuss the potential downsides of our approach, which are all related with the fountain coding technique. We believe, though, that none of these issues is significant in a datacenter storage context.

CPU Overhead. Fountain coding involves encoding and decoding of information on the sender and receiver side, respectively. Here, the overhead comes from generating random numbers according to the used statistical distribution (e.g. the Robust Soliton Distribution [14]) and, mainly, from XORing several pieces of the initial information to produce each encoding symbol to be transmitted. We are confident that this overhead will not be prohibitive with respect to Trevi's applicability. First, modern hardware in datacenter networks consists of fast, multi-core CPUs that could easily cope with the encoding and decoding processes. Second, the process itself is highly parallelisable, thus one could take advantage of the multiple cores or even offload it to hardware (e.g. GPU or NetFPGA). Finally, an opportunistic approach, where a master replica decodes and stores the original blob while other servers serve the statistically-required number of symbols to decode the blob, can be used to minimise the overall CPU overhead of the storage system.

Network Overhead. As mentioned in §2.1, fountain coding involves a constant penalty in terms of network overhead. This overhead is not significant given that in Trevi, TCP incast, which can severely degrade the I/O performance, is mitigated, and that we save network resources by multicasting write requests.

Memory Overhead. In Trevi, a sender needs to have fast access to a blob of data as long as it creates new symbols (in response to respective requests). This implies that a blob must be in memory until all receivers successfully received and decoded the blob. This requirement could potentially have an impact on the required amount of memory to support multiple I/O requests in parallel. However, more control of the storage buffer cache (using direct I/O and a userspace cache, or even libOS techniques [15]) makes it possible to partially map larger blobs into memory to allow encoding to be suspended if the memory is required elsewhere. This helps to mitigate the tail of requests for a given blob, especially if there are stragglers in the storage cluster.

Energy Efficiency. We expect that Trevi will have a positive effect on energy consumption, although we will have to extensively evaluate this perspective with the system we are currently implementing. Trevi adds some energy intensive functionality; more processing power is required to encode and decode data, and slightly more data need to be transmitted compared to a regular TCP blob transfer. Additionally when multiple servers contribute to the transfer of data to a client multiple

disks spin so that a number of encoded symbols can be served. However, data can be multicast instead of multi-unicast and, because Trevi is tolerant to symbols' loss, buffers' size in network switches can be reduced.

4. FLOW CONTROL REFINEMENTS

The receiver-driven flow control of our strawman design can be refined in several ways.

4.1 Predictive Flow control

In order to minimise the overhead because of requesting encoded symbols separately, as described in §2, we could use a simple push-pull flow control scheme. When transmitting data, the source knows how many symbols approximately need to be transmitted in the absence of loss. This number depends on the statistical distribution that is used to calculate the degree of each symbol. Initially the source is set to send this much data and then pause. If no symbols have been lost, the source is notified by all receivers that the blob is decoded, and then it simply stops. In the opposite case, receivers start issuing pull requests for additional required symbols, as described in §2.

4.2 Priority and Scavenging

Fountain coding is inherently resilient to loss. This makes it suitable for scavenger-type QoS. In such systems, scavenger traffic receives a very low priority which means it will be preferentially dropped in the presence of any congestion. But in the absence of congestion, such scavenger traffic can be sent at near-line rate. In order to make best use of such a system the sender must rapidly detect how busy the network is. One way to do this would be to modify the measurement approach used for PCN.

Pre-Congestion Notification [17] is a measurement based admission control system for real time traffic. Traffic traversing each path through the network is viewed as a single combined flow, called an ingress egress aggregate. Central to PCN is the concept of a virtual queue – a simple mechanism that behaves like a queue with a slightly slower drain rate than the real queue.[2] As the virtual queue passes a lower threshold the queue is defined as being in a pre-congested state. If the virtual queue continues to grow it eventually passes a second threshold which indicates that the real queue is about to start to fill. In PCN, crossing either of these thresholds causes arriving packets to be marked, and the aggregate rate of marks is used to decide whether to admit new flows or to drop existing flows.

A similar virtual queue technique could be applied to our fountain storage system. This would allow the storage control nodes to assess how much other traffic is competing with the storage traffic. This can then be used to determine the safe rate at which to send data. This is particularly relevant for the case of multi-sourcing data from many replicas to one client machine. In this instance there is a very real risk of causing the final network queue nearest to the client to become congested. Simply running a virtual queue on this and using this as one of the parameters in the destination-driven flow control would significantly reduce this risk.

[2]Since 2010, all Broadcom router chipsets have natively supported a form of virtual queue called a threshold marker.

4.3 Optimising for slow writes

If one storage node is writing data much more slowly than the others in its group then it will have a disproportionate effect on the rate at which all others can write. There are two potential solutions to this problem. Firstly any node that is significantly slower could be removed from the multicast group. Secondly, the sender may choose to ignore the slow node and simply go faster than it can cope. If the node becomes overwhelmed it can simply unsubscribe from that multicast group and mark that tract as unreadable. Both of these approaches have implications for the degree of replication within the system, but may significantly improve performance.

5. CONCLUSIONS

In this paper we presented Trevi, a new storage mechanism based on fountain coding. Trevi overcomes the limitations present in all storage systems that are based on TCP. We described a strawman design which highlighted the main features of our approach, and also presented an initial design for a receiver-driven flow and congestion control mechanism that can better utilise storage and network resources in a datacenter storage network. Finally, we explored approaches for flow control refinements over our base mechanism.

We are implementing a complete storage system which incorporates Trevi based on the flat datacenter storage system [18] and built as a userspace application. We are also studying several strategies for adjusting the request window for the receivers according to changes in the storage workload and in the network, as described in §4. We will evaluate these mechanisms in multi-level datacenter topologies, and experiment with different flow control strategies using large scale simulations. Finally, we will explore the design space by using our mechanism in SDN-enabled topologies.

6. ACKNOWLEDGMENTS

We would like to thank Malte Schwartzkopf, Steve Hand, Andrew Moore and the anonymous HotNets reviewers for their valuable feedback on earlier drafts of this paper. The research leading to these results has received funding from the European Union's Seventh Framework Programme FP7/2007-2013 under Trilogy 2 project, grant agreement no 317756.

7. REFERENCES

[1] M. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proc. of DSN 2005*, 2005.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[3] R. J. Anderson. The Eternity service. In *Pragocrypt*, 1996.

[4] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. In *SIGCOMM*, 1994.

[5] P. Breuer, A. Lopez, and A. Ares. The Network Block Device. *Linux Journal*, March 2000.

[6] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for Linux clusters. In *USENIX ALS*, 2000.

[7] P. Cataldi, M. Shatarski, M. Grangetto, and E. Magli. Implementation and performance evaluation of LT and raptor codes for multimedia applications. In *IIH-MSP*, 2006.

[8] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran. Decentralized erasure codes for distributed networked storage. *IEEE Transactions on Information Theory*, 52:2809–2816, 2006.

[9] L. Ellenberg. DRBD 9 and device-mapper: Linux block level storage replication. In *the Linux System Technology Conference*, 2009.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.

[11] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. *ACM SIGCOMM CCR*, 39(4), 2009.

[12] S. Hand and T. Roscoe. Mnemosyne: Peer-to-Peer steganographic storage. In *IPTPS*, 2002.

[13] C. Hopps. Analysis of an equal-cost multi-path algorithm. *RFC 2992*, 2000.

[14] M. Luby. LT Codes. In *Proc. of FOCS*, 2002.

[15] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *ASPLOS*, 2013.

[16] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *SIGCOMM*, 1996.

[17] M. Menth, F. Lehrieder, B. Briscoe, P. Eardley, T. Moncaster, et al. A survey of PCN-based admission control and flow termination. *Communications Surveys & Tutorials, IEEE*, 12(3):357–375, 2010.

[18] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *USENIX OSDI*, 2012.

[19] Oracle. The Oracle Clustered File System. http://oss.oracle.com/projects/ocfs/.

[20] G. Parisis, G. Xylomenos, and T. Apostolopoulos. DHTbd: A reliable block-based storage system for high performance clusters. In *CCGRID*, 2011.

[21] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *SANE 2000*, 2000.

[22] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *USENIX FAST*, 2008.

[23] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *Proc. of USENIX NSDI*, 2012.

[24] Y. Saito, S. Frolund, A. C. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *ASPLOS*, 2004.

[25] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *of USENIX FAST*, 2002.

[26] P. Schwan. Lustre: Building a file system for 1,000-node clusters. In *Linux Symposium*, 2003.

[27] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, 2006.

[28] K. Tan and J. Song. A Compound TCP approach for high-speed and long distance networks. In *IEEE INFOCOM*, 2006.

[29] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM*, 2009.

[30] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *USENIX SOSP*, 2006.

[31] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *USENIX FAST*, 2008.

[32] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data center networks. In *Proceedings of CoNEXT*, 2010.

[33] Y. Zhang and N. Ansari. On mitigating TCP incast in data center networks. In *Proc. of IEEE INFOCOM*, 2011.