Deconstructing Commodity Storage Clusters

Haryadi S. Gunawi, Nitin Agrawal,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler[‡]

Computer Sciences Department, University of Wisconsin, Madison

[‡] EMC Corporation, Hopkinton, Massachusetts

Abstract

The traditional approach for characterizing complex systems is to run standard workloads and measure the resulting performance as seen by the end user. However, unique opportunities exist when characterizing a system that is itself constructed from standardized components: one can also look inside the system itself by instrumenting each of the components. In this paper, we show how *intra-box* instrumentation can help one understand the behavior of a large-scale storage cluster, the EMC Centera.

In our analysis, we leverage standard tools for tracing both the disk and network traffic emanating from each node of the cluster. By correlating this traffic with the running workload, we are able to infer the structure of the software system (*e.g.*, its write update protocol) as well as its policies (*e.g.*, how it performs caching, replication, and load-balancing). Further, by imposing variable intra-box delays on network and disk traffic, we can confirm the causal relationships between network and disk events. Thus, we are able to infer the semantics of the messages between nodes without examining a single line of source code.

1 Introduction

The systems community has long understood the benefits of separating architecture and implementation [2]. Given this clean separation, clients are assured a consistent, standard interface, while designers have the freedom to innovate behind that interface. The common result is that a rather simple interface hides a growing amount of internal implementation complexity. This trend has occurred not only in the implementation of microprocessors behind their instruction sets, but in storage systems as well.

Modern storage systems have simple interfaces that hide a great deal of internal complexity. Many high-end storage servers continue to use the SCSI interface, a simple read/write interface exporting an array of blocks to the client. However, storage servers now implement a range of complex functionality and contain tens of processors, gigabytes of memory, and hundreds of disks. For example, the EMC Symmetrix 8000 storage server [19] implements redundant data paths, end-to-end checksumming, machinery to fence off portions of caches under failure, and disk scrubbing technology to discover latent errors proactively – all behind the standard SCSI interface.

Today, a fundamental change is occurring in storage systems. Whereas, in the past, storage systems were built from specialized parts, they are now being assembled from commodity components. High-end storage servers are now being built from collections of commodity PCs, each running a commodity operating system, and connected together by an Ethernet network [17, 20, 23, 28]. Hence, many modern storage systems are simply instances of a “cluster of workstations” [3, 31].

Across all domains, users often want to understand the behavior of the systems they use. This understanding enables critical evaluation of the design and implementation choices; it allows users to build better models of how the system behaves under different workloads and to tune and debug their performance; it enables administrators to identify when the system is not behaving correctly. Unfortunately, one of the drawbacks of standard interfaces is that they can hide interesting information about internal behavior. Currently, one can measure and evaluate systems using application-level benchmarks [9, 15, 33] or microbenchmarks [4, 14, 22, 27, 30]; however, these traditional approaches assume one can only observe the behavior of a system from its external interface.

The shift in storage system design to leverage commodity components greatly increases our ability to analyze how the storage system behaves. In effect, building a system from commodity components “opens the box” and allows users to directly observe what is occurring inside. Furthermore, users can often leverage existing, standardized tools to perform their analysis.

In this paper, we develop a set of *intra-box* techniques to analyze the structure and policies of commodity-based storage clusters. Our analysis contains two major components. First, we monitor and perturb network and disk traffic internal to the storage cluster in order to deduce the structure of the main communication protocols. Second, we build upon this protocol knowledge to dissect internal policy decisions, such as caching, prefetching, write buffering, and load balancing.

We apply these techniques to a new and important instance of a commodity storage cluster, the EMC Centera Content-Addressable Storage System. Centera is designed to provide low-cost, easy-to-manage, scalable storage for “fixed content” data, such as medical images, electronic documents, and email archives [17]. With content addressability, Centera can take advantage of the massive redundancy often present in data (*e.g.*, email attachments) and reduce capacity requirements.

One main contribution of this paper is our analysis of the Centera’s protocols and policies. Note that our results were achieved entirely without assistance from EMC; to verify their accuracy and relevance, we include a meta-analysis of our results from EMC (§5).

Overall, we find that the Centera chooses simplicity and reliability over sophisticated performance optimization, perhaps a good choice for an early implementation. Our analysis reveals the structure of the write update protocol as a standard two-phase commit with writes committed synchronously to the disks. As for policies, Centera uses caching and prefetching mechanisms of the commodity file systems within storage nodes, leaving more complicated caching schemes to client applications. We also derive some interesting properties of load balancing. For example, Centera storage nodes gather and disperse load information locally; global effects (such as network link performance) are not taken into account.

The other main contribution of this paper is the development of our intra-box techniques; although applied solely to Centera, we believe our techniques are widely applicable. One general lesson we draw is on the power of probe points within a system; observation of internal cluster network and disk traffic is crucial to our approach. Perhaps future I/O systems should consider architectural support to enable this type of detailed analysis.

The paper is structured as follows. We first present our methodology (§2), structural analysis of Centera protocols (§3), and policy inference (§4). We then analyze our results (§5), including accuracy confirmation by an EMC engineer, discuss related work (§6), and conclude (§7).

2 Methodology

In this section, we begin by presenting a general overview of the storage cluster under test, the EMC Centera. We then describe our intra-box techniques for analyzing both the structure and policies of distributed systems.

2.1 System Overview

The EMC Centera is a Content Addressable Storage (CAS) cluster designed for storing and retrieving fixed content information. The Centera handles the management of the physical storage resources transparently to applications and is designed as a highly scalable, "no-single-point-of-failure" platform. Centera uses *content addressing* in which applications access data objects (also known as BLOBs for "Binary Large Objects") with a 128- or 256-bit content address derived from the contents of the object (*e.g.*, via a hash function applied over the data).

The architecture of the Centera cluster is a Redundant Array of Independent Nodes (RAIN) [7]. Nodes are connected via a private LAN. Each node runs CentraStar software on a Linux kernel and operates as either a *storage node* or an *access node*. The storage nodes hold the objects on their local disks, whereas the access nodes manage read and write requests between an external *client* (*i.e.*, an application server) and the storage nodes.

The general protocol followed by Centera is as follows [11]. First, an application contacts an access node and delivers a data object (*i.e.*, a file) to the Centera with a write (*i.e.*, using the `FP_BlobWrite` API). The Centera calculates the unique content address (CA) for the object and records the CA along with other metadata for the object in a separate file, called a C-Clip Descriptor File (CDF). The CA for the CDF is then calculated and returned to the application after two copies of both the CDF and the BLOB have been stored. Data objects can be retrieved by contacting an access node via a read (*i.e.*, the `FP_BlobRead` API), providing the C-Clip's CA.

we evaluate a Centera cluster containing two access and six storage nodes, sometimes utilizing smaller clusters for experimental purposes. Each node runs Linux 2.4.19 and CentraStar version 2.0 and contains second-generation hardware: a 1-GHz Pentium 3 with 512 MB of memory, three Intel EtherExpress Pro 100 Mb/s Ethernet ports, and four 250 GB Maxtor 5A250J0 ATA disks. Each access node is connected to clients via 100 Mb/s Ethernet.

2.2 Intra-Box Analysis

The traditional approach for understanding the behavior of systems is to run standard workloads and measure the resulting performance as seen by the end user. However, unique opportunities exist when characterizing a system that is itself constructed from standardized components: one can also use *intra-box analysis* to look inside the system by instrumenting each of the constituent components.

To analyze the behavior of a distributed storage server we use two intra-box techniques: *observation* and *delay* of traffic to nodes and disks. Passive observation of network and disk traffic allows us to track correlations across different requests and thus gives us a general idea of the protocol structure and internal policies. However, observation alone does not enable us to definitively conclude the causality between different requests; to infer that one message depends upon a specific previous event, we must also delay messages and disk requests.

2.2.1 Observation

To derive the general protocol structure and policies, we begin by simply observing the traffic to each node or disk. In each case, the traffic can be observed using straightforward tools. To trace both TCP and UDP communication between nodes, we use `tcpdump`. To trace disk traffic, we insert into the kernel a pseudo-device driver associated with each disk and mount the file system upon it; the driver records the start time, end time, and block number of each read or write request.

To collect suitable observations, we run a simple workload on the client application server (*e.g.*, the workload repeatedly creates new objects) and trace all of the traffic seen at the network and at the disk. We assume that there is no other workload running on the Centera and we perform multiple trials so that we can filter out traffic that does not occur consistently. Analysis is performed off-line, to avoid interference with system activity.

Our analysis allows us to determine general properties of the communication and disk activity in the system:

which nodes send messages to which other nodes, which nodes read or write to local disk, the size of each network message, and the time delay between network and disk events. We also can infer properties that are specific to the Centera, *e.g.*, which nodes in the cluster act as access nodes versus primary and secondary storage nodes.

2.2.2 Delay

Passive observation of network and disk events allows us to correlate events with one another; however, observation alone does not enable us to determine the exact dependencies between events. More precisely, even when one repeatedly observes that event X occurs after event A, one cannot infer that X depends on A.

Using correlation to derive causality can be successful [1], but has a number of potential weaknesses. First, unrelated events may consistently occur in the same order simply due to performance characteristics of the system. Second, a large number of iterations are required to observe which event orderings occur consistently and to filter out background noise. Finally, it is difficult to discover that an event is dependent on multiple preceding events.

To derive the causal relationships across events, we delay each network and disk event and then observe which subsequent events are delayed as well. For example, if we delay the receipt of a message A and then observe that sending message X is also delayed, we infer that X is dependent (perhaps indirectly) on A; however, if X is not delayed, we infer that X is independent of A.

One complicating factor is determining the amount by which an event should be delayed; this is an important issue when later events may be dependent upon multiple events. For example, suppose that sending message X is dependent upon receiving messages A and B and that B usually arrives 20 ms after A. If we delay A by only 10 ms, B will still arrive later (at its usual time) and will be the event that triggers X; thus, we will incorrectly conclude that X is independent of A. To avoid this situation, we inject a relatively large delay of 500 ms that exceeds the other durations in the system (this value must be tuned to the system of interest). Thus, delaying A will cause all dependent events to be delayed as well, as desired.

To delay network traffic, we employ NistNet [10]; our modified version sits on top of `tcpdump` and delays only incoming packets. Our framework allows us to select the amount of delay and define criteria for which packets should be delayed. On Centera, most messages can be uniquely identified by their size; thus, we use size and protocol type (*i.e.*, TCP or UDP) to determine which messages should be delayed. We also utilize other fields in the protocol headers such as IP addresses and port numbers.

To delay disk traffic, we use the same pseudo-device driver as for tracing requests. One can configure the duration of the delay as well as which requests should be delayed (*e.g.*, reads or writes). Delayed requests are placed in a separate queue until the delay time expires.

3 Deducing System Structure

In this section, we apply intra-box analysis to derive the internal structural protocols employed by the EMC Centera. We begin by using passive observations to infer the basic protocol for both storing and accessing objects. We then actively delay network and disk events to determine the relationships across events when storing objects.

3.1 Passive Observations

In our first step, we passively trace the network and disk traffic within the EMC Centera. We analyze two distinct protocols: the events that occur when the user stores objects via writes and the events that occur when the user later accesses objects through reads. Most of our analysis is focused on writes because they are more complex.

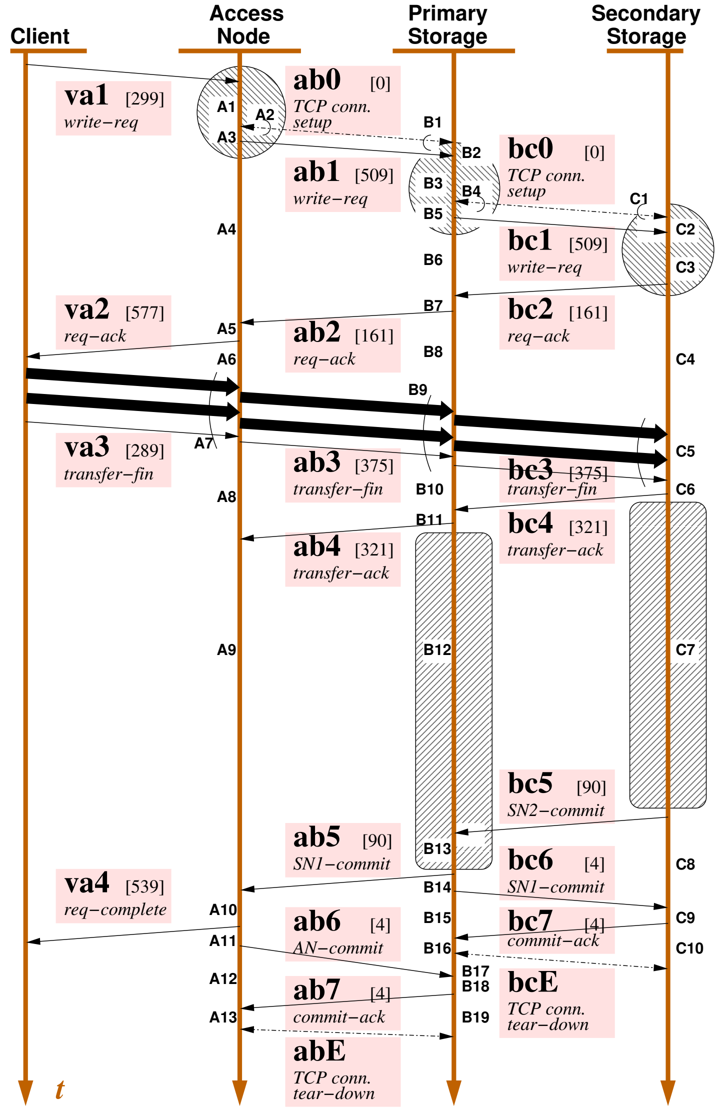
3.1.1 Object Write Protocol

Figure 1 is a pictorial representation of our findings for the Centera protocol when writing objects. The figure shows the network and disk events that occur between the client and Centera nodes for each request. Figure 2 reports the duration of intervals between each TCP message.

Our analysis as presented in the diagram allows us to infer the basic Centera protocol for storing objects. Note that in our description, we tend to say that nodes perform action X after action Y (*e.g.*, send a UDP message after a disk operation completes); however, until we have verified the causal relationship across events as we do in Section 3.2, these sequences are only conjectures.

The object write protocol begins when the client contacts an access node directly, sending a 299-byte message ($va1$). The access node then issues a disk read, sets up a new TCP connection with the primary storage node ($ab0$), and communicates with it ($ab1$). The primary storage node also performs a disk read, sets up a connection with the secondary storage node ($bc0$), and communicates with it ($bc1$). Thus, each time a client stores an object, a new TCP connection is created between the access node and the appropriate storage nodes. In this structural analysis of the protocol, we do not determine *which* access node or *which* storage nodes are selected; this selection is a policy decision, which we will explore in Section 4.

A series of messages then propagates back from the secondary storage node all the way to the client ($bc2$, $ab2$, $va2$). After the client receives this response, it sends a variable number of messages to the access node. We infer that these messages contain the data object since their size varies with the size of the objects. The access node forwards this data to the primary storage node, which again forwards the data to the secondary storage node. After sending a transfer acknowledgment ($ab4$, $bc4$), each storage node writes to its local disk and communicates using UDP with two other Centera storage nodes ($udpX1$, $udpY1$). After these events, the storage nodes propagate a series of 90-byte ($ab5$, $bc5$) and 4-byte acknowledgments ($ab6$, $ab7$, $bc6$, $bc7$) to each other and to the access node. Finally, the access node informs the client that the request



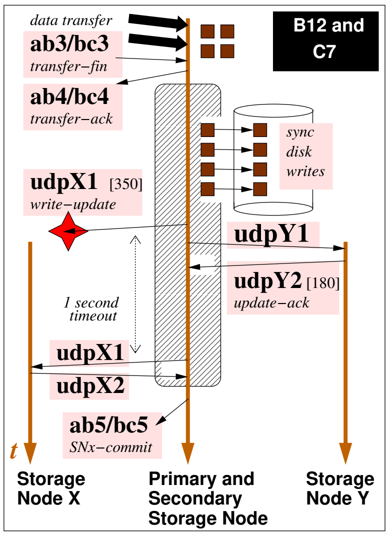


Figure 1: **Anatomy of the Centera Write Protocol.** *The figure on the left provides a pictorial representation of our findings and shows the network and disk events that occur between the client and Centera nodes on a 16-KB object write through the* `FP_BlobWrite` *API. TCP messages are labeled with three values. First, each message has a unique label identifying the endpoints and a simple message id; for example,* ab0 *indicates the initial message between nodes a and b, where a is the access node and b is the primary storage node. Second, the average size in bytes of each protocol message is shown. Third, each message is assigned a label reflecting our understanding of its semantic purpose in the protocol. The interval between TCP events on each Centera node is labeled with a unique identifier (e.g. A4); the duration of each of the intervals is reported in Figure 2. Single block disk reads sometimes occur around intervals A1, B3, and C3. The figure on the top shows in more detail events that occur around intervals B12 and C7. UDP messages shown above are designated with a label identifying the destination node and a simple message id (e.g.,* udpX1 *is the first UDP message sent to storage node X in this round of the protocol).*

is complete ($va4$) and the Centera nodes tear down their TCP connections (abE, bcE).

The timing results reported in Figure 2 reveal where most of the time is spent when storing 16-KB objects. We note that the client sees that the object has been stored after a latency of approximately 150 ms, which is roughly the sum of the times A1 to A10. The figure shows that most of this latency occurs during intervals A4, A7, and A9. Matching these intervals with Figure 1, we see that A4 corresponds to when the access node is waiting for the storage nodes to read from disk; A7 to when the data object is being transferred to the access node; and A9 to when the storage nodes are writing the object to disk.

3.1.2 Object Read Protocol

Again for reads, the client node communicates only with the access node, which in turn reads from disk, and establishes a new TCP connection with one of the storage nodes. After the storage node receives the request, it reads from disk and sends the data object back to the access node. The access node transmits this object to the client, exchanges acknowledgments with the storage node, tears down the TCP connection with the storage node, and finally informs the client that the request is complete (Figure not shown due to space limitation).

3.2 Delaying Events

Observing network and disk events enables us to learn much about the internal protocol of the Centera. However, from these observations, we cannot conclude which events must occur after one another. We now delay events to infer the dependencies across network and disk events for the write protocol. Due to space constraints we do not present similar analysis for the read protocol.

For each message in the protocol, we determine the set of events that it depends upon by delaying the preceding events on that node and observing which delay the subsequent sending of that message. We structure our discussion by investigating each type of event in turn: TCP traffic, UDP traffic, and completion of disk reads and writes.

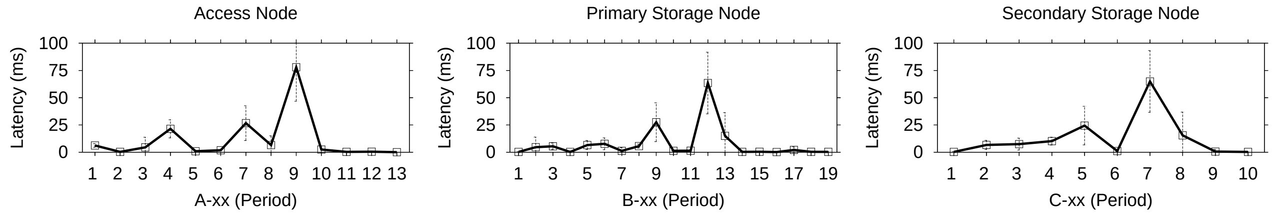


Figure 2: **Intervals when Writing Objects.** *The figures report the average duration for each of the intervals between TCP message events. Across the three graphs, we examine the access node, the primary storage node, and the secondary storage node, respectively. Within each graph, we examine the intervals within a node, where the x-labels match those used in Figure 1.*

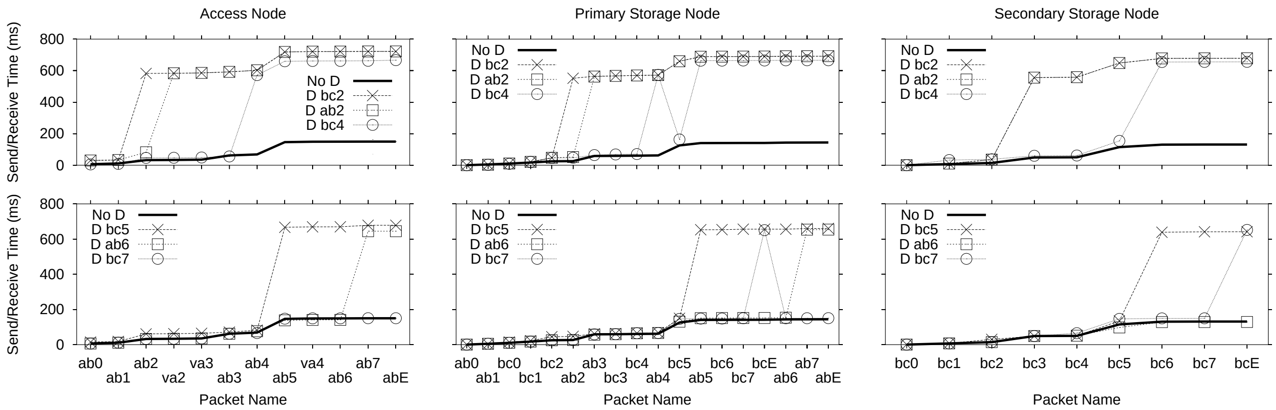


Figure 3: **Impact of Delaying TCP Packets on Write Protocol.** *We delay an interesting subset of the TCP packets (i.e., $bc2$, $ab2$, $bc4$, $bc5$, $ab6$, and $bc7$) by 500 ms. Across the three pairs of graphs, we consider each of the three involved Centera nodes: the access, primary storage, and secondary storage nodes. Along the y-axis, we report the send or receive time of each packet relative to the start of the request. A sharp increase in the send or receive time of packet X when packet A is delayed indicates that packet X depends on packet A. "D" stands for delay.*

TCP Traffic: We begin by investigating the most straightforward case of a dependency: whether an outgoing TCP packet depends on any of the previously-arriving TCP packets. Note that we do not need to determine if an outgoing message is dependent on multiple TCP packets from the same node due to the nature of TCP; because TCP is a reliable byte-stream protocol, the receiver is guaranteed to see packets in the same order that the sender transmitted them, even if some packets are delayed or lost and subsequently resent. For example, we check whether $ab5$ is dependent on receiving $bc5$ or on receiving $bc4$, but not whether it is dependent on receiving both; since $bc5$ is guaranteed to arrive after $bc4$, even if $ab5$ depends on both $bc4$ and $bc5$, then waiting for $bc5$ is sufficient.

Figure 3 shows the impact of delaying an interesting subset of the TCP packets (*i.e.*, $bc2$, $ab2$, $bc4$, $bc5$, $ab6$, and $bc7$). Across the three pairs of graphs, we report the send or receive time of each packet for the three Centera nodes involved: the access, primary storage, and secondary storage nodes. We note that a sharp increase in the send or receive time of packet X when packet A is delayed indicates that packet X depends on packet A.

These measurements imply the following dependencies. First, in the initial request exchange, sending $ab2$ from the primary storage node depends upon first receiving $bc2$ from the secondary storage node; thus, $ab2$ serves as an acknowledgment that both the primary and secondary storage nodes have received the request. Second, sending $ab4$ from the primary storage node depends upon receiving $bc4$; thus, $ab4$ acknowledges that both the primary and secondary storage nodes have received the data object. Third, sending $ab5$ from the primary storage node depends upon receiving $bc5$ from the secondary; thus, $ab5$ serves as an acknowledgment that both storage nodes have written the object to disk (further details below).

Finally, the measurements show an interesting set of relationships after the request is complete. Specifically, we see that delaying $ab6$ impacts $ab7$ and delaying $bc6$ impacts $bc7$, but delaying $bc7$ does not impact $ab7$. Thus, in the protocol, packet $ab7$ serves only as an acknowledgment that the primary storage node has received the packet from the access node and not from the secondary storage node. This independence could not be confirmed without the delay technique because our passive measurements always observed that $ab7$ followed $bc7$.

UDP Traffic: We next isolate the events that are dependent upon arriving UDP traffic. Interesting UDP traffic occurs during intervals **B12** and **C7** on the primary and secondary storage nodes, respectively. A non-trivial amount of additional UDP traffic occurs throughout our measurements, but this UDP traffic is filtered because it does not occur at regular points in the write protocol. Furthermore, we have found that delaying other protocol events has no impact on this background UDP traffic; therefore, we can conclude that this other UDP traffic is not related to the write protocol.

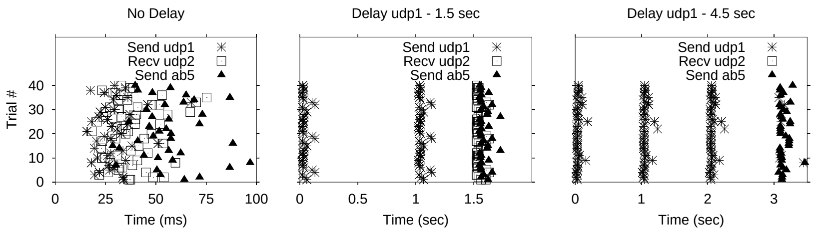


Figure 4: **Impact of Delaying UDP Packets on Write Protocol.** *The three graphs show the time at which the primary storage node sends and receives the UDP packets $udp1$ and $udp2$, respectively, and sends the TCP packet $ab5$. In the first graph, we consider the default case in which there is no extra delay; in the second graph, we delay $udp1$ by 1 second; in the third graph, we delay $udp1$ by 4.5 seconds. Each experiment is repeated 40 times.*

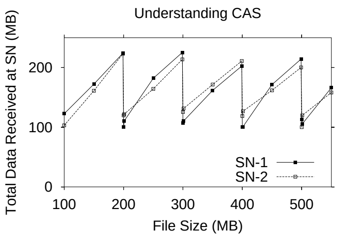


Figure 5: **Content Addressability.** *The figure plots the amount of data written to the storage nodes storing a file as the file size is increased. The file is filled with a single repeated byte, and hence is a candidate for space-savings due to content-based addressing.*

Figure 4 addresses the relationship between UDP traffic and other events. These measurements were performed on the primary storage node, but the relationships are identical for the secondary storage node. In each graph, for 40 independent trials, we show the time at which the two $udp1$ packets were sent to the other storage nodes, when the two $udp2$ packets were received, and when the TCP packet $ab5$ was sent back to the access node.

The first graph reports our observed timings for the default Centera system with no delays of UDP traffic. This figure shows that there is no ordering between sending a UDP packet to storage node Y and receiving the UDP packet from storage node X; in fact, $udpX1$ and $udpY1$ are sent in parallel. Thus, passive observations can be sufficient to show there is no dependency between messages.

The first graph does show that $udp2$ always arrives after $udp1$ is sent and that $ab5$ is always sent after $udp2$ arrives. However, to confirm whether these are true dependencies, we must delay UDP messages. In the second graph we delay the request $udp1$ by 1.5 seconds. These results show that $udp2$ and $ab5$ are also delayed. Thus, $udp2$ is indeed an acknowledgment of $udp1$. However, another interesting property is apparent as well. Since UDP is an unreliable protocol, the Centera software implements its own timeout-retry policy to resend messages when a response is not received. In this graph, we see that the Centera implements a timeout of one second for the $udp1$ packets, at which point it resends those packets.

We explore the impact of the timeout policy in more depth in the third graph; in these experiments we increase the delay of $udp1$ to 4.5 seconds. In these circumstances, the storage node never receives the acknowledgment packet of $udp2$. Instead, the Centera protocol performs three timeout-retry intervals and then stops retrying; at this point, the storage node sends the final $ab5$ TCP packet without receiving $udp2$. Thus, $ab5$ depends on either receiving both $udpX2$ and $udpY2$, or waiting for a time-out of approximately three seconds.

Disk Events: We now determine which TCP and UDP messages are dependent on the completion of disk reads or disk writes. Disk read operations occur on three Centera nodes when the request is first initiated (*i.e.*, during intervals **A1**, **B3**, and **C3**); disk write operations occur on the two storage nodes after the data object has been received (*i.e.*, during intervals **B12+B13** and **C7**).

Figure 6 shows whether delaying the disk reads and writes on the storage nodes impact the subsequent messages; across the three graphs, we examine the access node, primary storage node, and secondary storage node, respectively. We make two observations from these measurements. First, on the storage nodes, the TCP message immediately following the disk read is in fact dependent upon that disk read (*i.e.*, the send times of $bc0$ and $bc2$ both increase when the disk read is delayed); this confirms our initial intuition. On a related note, delaying the reads on both nodes causes subsequent events to be delayed by 1 second, further indicating that these reads are not overlapped. Second, on the storage nodes, delaying the disk writes also delays sending the $udpX1$ packet; further, given that the send time of $udpX1$ is delayed by nearly 2.5 seconds, the storage nodes appear to perform five disk writes in succession.

Conclusion: Delaying network and disk events within the Centera write protocol allows us to identify which events are dependent upon which others. In two cases, this analysis would not have been possible with passive observations and correlations alone.

The first case occurs at the end of the write protocol: when the primary storage node sends a *commit-ACK* ($ab7$) message to the access node, this indicates that it received the *AN-commit* ($ab6$) message from the access node, not that it received the *commit-ACK* ($bc7$) from the second storage node. Relying on correlations alone, we could not determine that *commit-ACK* from the primary was independent of the *commit-ACK* from the secondary, because the one always occurred after the other.

The second case occurs on the storage nodes after they have written to disk: when the storage nodes send a TCP message, this indicates that the data has been committed to disk and that the node has attempted to communicate using UDP with other storage nodes, but not that the UDP messages have succeeded. Again, relying only on correlations, we would have inferred that storage nodes must receive UDP replies before sending the TCP commit.

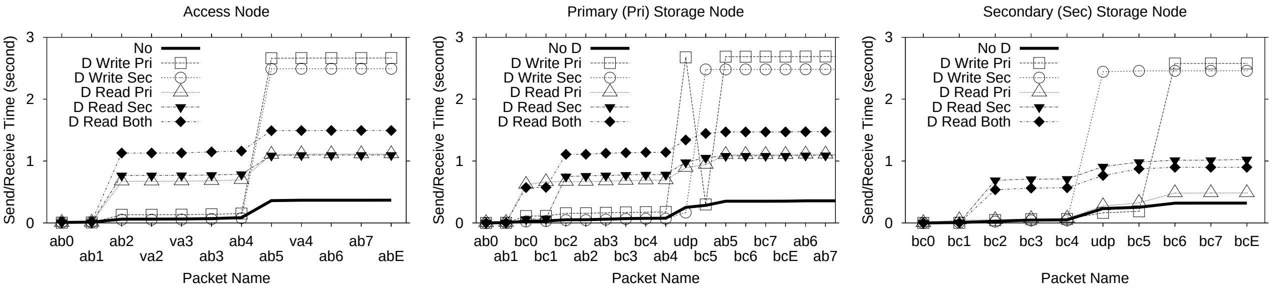


Figure 6: **Impact of Delaying Disk Activity on Write Protocol.** *We delay reading or writing from disk by 500 ms. Across the three graphs, we consider each of the three involved Centera nodes: the access, primary storage, and secondary storage nodes. Along the y-axis of each graph, we report the send or receive time of each packet relative to the start of the request;* UDP *indicates the sending of* $udpX1$. *A sharp increase in the send or receive time of packet* X *when the disk event* A *is delayed indicates that packet* X *depends on disk event* A.

4 Inferring Policies

In the previous section, we analyzed the protocol structure of the Centera for write and read operations. In this section, we infer the policy decisions within each of those protocols. In our analysis, we focus on the most important functionality one would expect in a storage system: replication, load balancing, caching, and prefetching.

One key to our approach is that we can utilize the derived structure of the write (Figure 1) and read (not shown) protocols to fine-tune our analysis. For example, in our analysis of caching, we use this information to enable fine-grained accounting of disk accesses, thus enabling us to filter out traffic in the system that is unrelated to the current request.

4.1 Object Write Policies

We begin by analyzing the decisions that occur during the write protocol, originally shown in Figure 1. We continue to assume that we are able to observe and delay both network and disk traffic.

4.1.1 Content Addressability

We begin by inferring a very basic decision: how large are the units of data? Centera segments each file into multiple BLOBs, each of which is stored, replicated, and accessed independently. A BLOB is the unit of granularity at which both content hashing (and hence duplicate detection) is performed and storage is allocated across storage nodes.

To determine the size of a BLOB, we write a file containing the same byte repeated throughout. When the size of the file being written is exactly twice the BLOB size, the file will internally be comprised of two identical BLOBs; thus, the amount of traffic should be halved. Indeed, this behavior should be observed at all multiples of the BLOB size.

Figure 5 shows the amount of data transferred to the storage nodes (both primary and secondary) across the network as we increase the size of the file being written. As one can see, the amount of data transferred climbs steadily from around 100 MB to around 200 MB as the x-axis increases from 100 MB to 200 MB, at which point it drops to 100 MB again. This cyclical pattern repeats at 200 MB, 300 MB, and so on, indicating that the unit of content addressability is 100 MB.

4.1.2 Level of Replication

The Centera uses data replication to protect against data unavailability or corruption in the face of failures. One fundamental choice is the *level of replication*, or number of copies, for data objects. This level is readily apparent from our previous structural analysis. Figure 1 shows that two replicas are made of the object being written. Further experimentation (not shown) across a range of object sizes reveals that the level of replication is two for all objects.

4.1.3 Load Balancing

We now dissect the load balancing strategy under writes. When a write enters the system, Centera first chooses the primary storage node for the data; the primary storage node then chooses the secondary location. We now infer the load balancing policy: what factors determine which storage nodes are selected? Many factors may influence the decision of which two nodes to place a given data item upon, including current performance or the amount of available space. In our analysis, we focus on four performance factors: CPU utilization, disk usage, network connections, and network delay. We vary these factors one at a time in a controlled manner. For CPU, we run a high priority while loop with a varying fraction of sleep time. The network delay is varied using modified NistNet. For disk usage, we generate background traffic using a file copy program. We also open varying number of TCP connections between the primary and secondary nodes.

We then observe internal message traffic to determine whether the induced load has an impact on Centera's placement decisions. Figure 7 plots the amount of data written to each node under different load experiments.

From the figure, we can see that three factors influence the selection of nodes under writes, heavily skewing writes to other unloaded nodes: the CPU load, the disk load, and the number of network connections to that node. Interestingly, we also observe that increasing the network delay of an incoming link to a storage node does *not* affect load balancing; performance for writes decreases dramatically (not shown) when we increase the latency of the incoming link to a storage node, because Centera does not incorporate this delay into its load balancing strategy.

We hypothesize that Centera is collecting performance

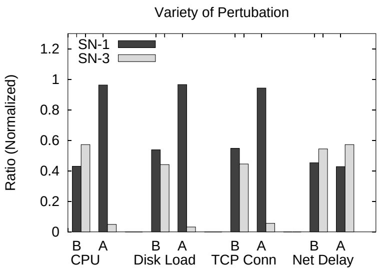


Figure 7: **Write Load Balancing.** *The results from four experiments are shown. In each, we first run the system in normal mode for some time (labeled B for Before), and then add load to one resource of a storage node (labeled A for After). Along the y-axis, we plot the normalized ratio of traffic across the two primary storage nodes in a four storage node configuration. Before the load addition, we expect and see that writes are roughly balanced across the two nodes; after, we expect to see an imbalance, skewing towards the unloaded primary node.*

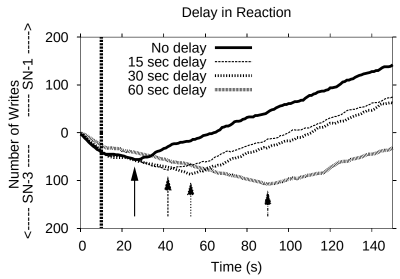


Figure 8: **Impact of delaying UDP message traffic.** *The graph illustrates the impact of delaying distribution of load information. The y-axis plots the difference of number of writes to SN-1 and SN-3. Initially writes are served by SN-3. The bold vertical line at t=10sec marks the CPU load addition to SN-3. The arrows point to the times when the writes switch from the loaded node (SN-3) to the unloaded node (SN-1) due to the write load balancing strategy. We vary the UDP traffic delay by 15, 30, and 60 seconds.*

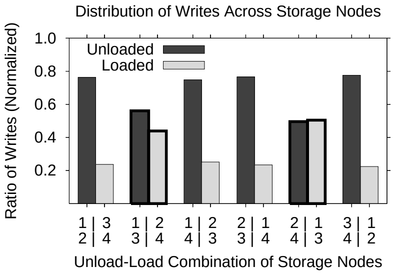


Figure 9: **Write Constraints.** *The graph shows the percentage of writes directed to a particular pair of nodes in a system configured with four storage nodes. Each pair of storage nodes is either "loaded" (i.e., has a CPU load induced upon it) or "unloaded" (i.e., does not have a load induced upon it) for the given experiment, as varied across the x-axis (e.g. pair of nodes (1, 2) and (3, 4) are unloaded and loaded respectively in the first pair of bars). The y-axis plots the percentage of writes that are directed to the loaded and unloaded pair.*

statistics on each storage node, distributing that information throughout the system periodically, and basing its load balancing decisions upon it. To confirm our belief, we run another experiment, in which we again increase the load on a particular storage node (the CPU load in this case), but also delay UDP message traffic within the cluster. As we saw in our protocol analysis, TCP is used within Centera for writes and reads, and UDP for virtually all other inter-node communication. Hence, by slowing various UDP messages down, we hope to slow the spread of load information, and confirm our hypothesis.

Figure 8 reveals the method by which load information is dispersed. From the figure, we can see that the longer UDP message traffic is delayed, the longer it takes for the load balancing decision to be affected by the increased CPU load on a particular storage node. Hence, we confirm our hypothesis about load information dispersal.

Finally, additional constraints appear to determine the primary and secondary copies of a data item. To isolate these constraints, across experiments, we placed identical CPU loads on different pairs of the four storage nodes. Figure 9 shows the results. In most cases, when a greater CPU load is placed on a pair of nodes, then a greater fraction of writes are sent to the unloaded storage nodes. However, in some cases, the number of writes does not adjust to the CPU load. In particular, when either the pair of nodes (1, 3) or nodes (2, 4) are unloaded, then the load balancing policy does not react: writes are allocated roughly evenly across the loaded and unloaded pairs.

Hence, the Centera ensures that a data item has one copy on either node 1 or node 3, and the other copy on either node 2 or node 4. Our inspection of power distribution within Centera reveals the reason: each pair of these nodes uses a separate power supply. Hence, while Centera write load balancing is sensitive to performance factors, it is constrained by factors that influence reliability, such as the power source.

4.1.4 Caching and Buffering

Another important performance optimization present in most storage systems is *write buffering*, also known as *write behind*. By transforming writes into asynchronous operations, application-perceived latency is greatly reduced, as copying data into an in-memory buffer is much faster than committing it to disk [26]. The trade-off comes in terms of reliability: by delaying the commit to disk, the chance of data loss under failure increases.

Our protocol analysis shown in Figure 1 revealed that the access and client nodes are notified only after the disk write has been committed on both storage nodes. Hence, we conclude the Centera performs all write operations *synchronously*. Thus, the Centera developers chose safety and reliability over performance.

4.2 Object Read Policies

We now turn our attention to the read protocol. Although reads are not as complex, their performance characteristics may be crucial for some applications.

4.2.1 Caching

We begin by determining whether or not caching of data objects is performed within the Centera read protocol. To demonstrate the benefits of intra-box techniques, we begin by assuming that we *cannot* access the internals of the Centera and can only observe performance at the client.

We begin with a simple workload that repeatedly reads the same file. By comparing the difference in time between the first read and subsequent reads, in many environments, one can determine whether caching is present [27]. From the identical latency numbers in the first row of Table 1a, one might conclude that there is no caching taking place; in this case, one would be wrong.

	Client Latency (s)	
	1st	*2nd ...*
No Delay	0.52	0.51
w/ Delay	**32.39**	0.55

(a)

	Data Read (MB)	
	1st	*2nd ...*
AN-Cli	4.04	4.04
SN-AN	4.04	4.04
Disk-SN	4.04	**0.00**

(b)

Table 1: **Read caching.** *The table on the left shows the time taken to read a 4 MB file from Centera; the first column shows the time for the first read and the second column shows the average time for the second and subsequent reads. The second row shows the same experiment, except with a large (800ms) disk delay induced. The table on the right shows the breakdown of traffic from access node to client, storage node to access node, and from the disk to the storage node.*

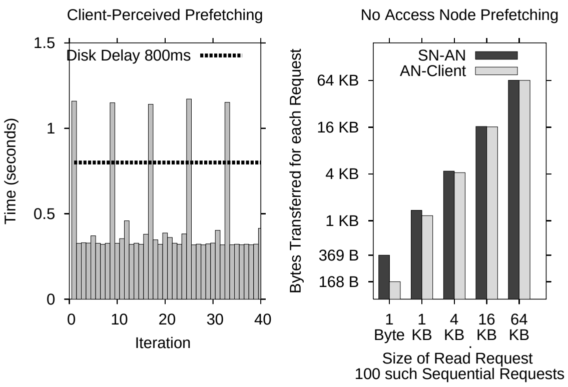


Figure 10: **Prefetching.** *The graph on the left shows the time taken for each 1 KB sequential read of a file. The graph on the right shows the amount of bytes transferred across the network (either from the storage node to the access node, or from access node to client) under such a workload; however, in the rightmost graph, we run multiple tests, varying the size of the request from 1 byte up to 64 KB.*

We now leverage our ability to observe and delay events inside Centera. In particular, we insert a substantial disk delay for every read that is sent to the disk of a storage node. The second row in Table 1a illustrates this, as the numbers now show a large difference between the time taken for the first read and subsequent reads of the same file. Hence, caching must be taking place within Centera.

The presence of caching is not observable in Centera because the 100 Mbit/s Ethernet delivers data as quickly as the IDE disks. By inserting delay into the disks, we change the relative ratios between the network and disk, and hence can observe that caching is taking place within the system. However, the experiment does not reveal *where* in the system caching is occurring.

To complete our read caching analysis, we monitor both network and disk traffic during the previously-described experiment. The results of this analysis are presented in Table 1b. The table shows how much data is transferred from the access node to the client, the storage node to the access node, and the disk to the storage node for both the first and subsequent file accesses. The table shows that, across requests, the same amount of data is transferred between the access node and the client and between the storage node and the access node; thus, the client and the access nodes are not performing caching. However, the table shows that no data is transferred between the disk and the storage node on the second and subsequent requests; therefore, the storage node performs in-memory caching.

4.2.2 Prefetching

Prefetching is an important optimization for storage systems [24]. In these experiments, we determine whether Centera performs prefetching, and if so, which components perform the prefetching.

In our first experiment, we read data sequentially from a file in small chunks (*i.e.*, 1 KB), and time each read at the client. We again slow down the disks to exacerbate the difference between on-disk and in-memory accesses. The first graph of Figure 10 shows the results of this experiment. From the graph, we observe that the first 1 KB read takes a significant amount of time, followed by seven requests that are completed more rapidly. From this client-perceived timing result, we can conclude that prefetching is taking place within Centera; specifically, Centera prefetches an 8 KB block when the first 1 KB is read.

As with caching, we also wish to unearth *where* in the system prefetching occurs. The second graph of Figure 10 plots the network traffic for the same experiment. Along the x-axis, we vary the size of each read to the file, and along the y-axis, we plot how much data was transferred per request. The graph shows that the amount of data transferred between Centera nodes is just slightly more than the size of the requested data. Specifically, if the client requests x bytes from Centera, $x+368$ bytes will be sent from the storage node to the access node, and $x+167$ bytes will be sent from the access node to the client.

From these results, we draw two conclusions. First, given that few extra bytes are passed between Centera nodes, prefetching is not occuring across the network. Thus, the prefetching must be occuring at the storage node. Second, some extra information (roughly 200 bytes) is passed in a header from the storage node to the access node node that is not then passed on to the client.

4.2.3 Load Balancing

We now examine load balancing under reads. For load balancing, writes have a great deal of flexibility: in a large-scale system, a primary copy may go to any half of the nodes and the secondary to any node in the other half. However, reads are more constrained: the read in can go to only one of the two storage nodes where the data is located. In these experiments, we seek to understand the factors Centera uses to determine which copy of data is accessed. We again examine the performance factors of CPU utilization, disk usage, network connections, and network delay.

Surprisingly, we found that the Centera read balancing policy is completely insensitive to any and all loads we induced (not shown). Hence, even if a node responds much more slowly to read requests than other nodes, read requests are still just as likely to be directed its way.

5 Analysis

We now analyze the design and implementation of the Centera storage server. In each subsection, we present perspectives from Wisconsin followed by EMC; by adding the EMC perspective, we offer insight as to the accuracy and relevance of the Wisconsin analysis.

5.1 Protocol Structure

Wisconsin: The protocol structure reveals many basic elements of the Centera design. First, we can observe a basic two-phase commit protocol under writes [18]. Second, we also see that the generation of the secondary copy is handled by the primary storage node; a different implementation would have the access node send the data to both storage nodes itself. The trade-off here is clear and likely a reasonable one – in Centera, latency is potentially higher, but the load on the access node is decreased.

Finally, we can also see how TCP and UDP are used for different purposes in the Centera communication system. TCP is used for the most important aspects of data transfer, both on writes and reads. UDP, in contrast, is presumably used for traffic such as periodic heartbeats and, as we see in load balancing, for propagating load information. We also observe that a new TCP connection is created per data transfer; although not a large cost in the current generation system, future Centera implementations should consider caching connections to storage nodes and avoiding costly three-way TCP handshake and teardown.

EMC: The analysis correctly identifies the majority of protocol features and illuminates Centera design principles and the workload characteristics. Centera is designed for on-line archival of fixed content with mostly-write operations of medium- and large-sized objects. Therefore, reliability and write-ingest is more important than read performance and lower latency. The extra latency for writes introduced by using a storage node as a relay is typically very small; congestion occurs rarely because the internal network has dual paths with two switches and is shielded from outside traffic. Finally, the cost of setting up and tearing down a TCP connection for each write was initially deemed negligible, given targeted object sizes. Moreover, it provided a simple and scalable solution for clusters of 8–256 nodes. More recent CentraStar versions reuse TCP connections and tune the number of open connections based on the cluster size and load.

The udp messages in the write protocol are updates of a distributed hash table, translating content address to location in constant time. Even if the message is not delivered after three attempts, the write transaction reports success, logs this exceptional case, and retries the update at a later time (not observed). The occasional disk reads observed in the **A1**, **B3**, and **C3** intervals are not directly related to the write transaction. However, with *a priori* knowledge of the content address, the write protocol first performs a lookup in the distributed hash table; if found, no data is transferred to the cluster (not observed in the analysis).

5.2 Read Caching and Prefetching

Wisconsin: Our analysis reveals that only the storage nodes perform caching and prefetching. Indeed, one would expect these nodes to do so, as each runs a commodity file system. We also saw that no caching or prefetching is performed on either the client or access nodes. This decision seems reasonable, as there would probably be little benefit to the client (in terms of latency) if data were fetched from an access node instead of a storage node; in both cases, the data must still cross the network. As for the client host, one must remember that the user application which accesses the data is also running upon that host. Perhaps the designers did not want to consume precious memory resources on the client node for caching and prefetching.

EMC: With emphasis on leveraging commodity components (i.e., storage node's file system and disk drive caches) and less emphasis on access latencies, the extra hop on the internal network does not warrant the impact (both in complexity and performance) on access nodes. For file server-like environments, where repeated reads of the same objects are more likely, Centera offers a separate gateway, which sits in front of the cluster. It translates NFS and CIFS requests to Centera API operations and implements caching, so it avoids accesses to the cluster altogether. The design goal is to provide a light-weight Centera API library; applications can use their existing caches or implement their own. In short, both the gateway and application-specific caching reduce the need for caching at the access nodes.

5.3 Write Caching and Buffering

Wisconsin: Our analysis shows that Centera is a synchronous system under writes – no write buffering is performed. Once again, Centera leans towards simplicity and reliability; if a write completes successfully, this means that is has been reliably committed to two disks on different storage nodes. However, synchronous writing is slow. Hence, a next-generation Centera might consider other options to improve performance, such as a NVRAM that is found in other higher-end EMC products.

EMC: Through a patch developed with the Linux community, Centera ensures that data is reliably written to the media. Using (arguably non-commodity) NVRAM would increase complexity when handling exceptional states as well as hardware costs. Other EMC products that include NVRAM make the trade-off in favor of increased performance of read-modify-write workloads.

5.4 Replication and Load Balancing

Wisconsin: Our investigation of Centera replication reveals its uniform approach: all objects can be found on two disks in the system. Perhaps more control could be given to applications, enabling them to create more copies of particularly valuable data.

The Centera was further found to perform load balancing across storage nodes under writes. However, all decisions are based on what is locally observable by storage nodes. Hence, as we demonstrated by inducing a delay on an incoming network link, the Centera approach to load balancing may not perform well when the load is not measurable from the perspective of the storage node. In the future, we believe it is important to gather information for load balancing decisions at higher levels in the system, perhaps measuring each write from the perspective of an access node, and using this measured history to make more robust placement decisions. We also saw that Centera takes the power distribution network into account. Hence, the system is built similar to "orthogonal" RAID designs [13], which take the many sources of failure into account when placing data and its replicas across disks.

Finally, despite the clear presence of load balancing machinery under writes, the Centera does not seem to perform load balancing for reads. Hence, if a node is performing poorly, read performance of the system as a whole suffers. Future versions of Centera should consider correcting this oversight.

EMC: Network delays are not an issue as there are two paths from any node to any other node and the load is balanced between the two paths. Load observations local to the node are propagated to other nodes both by periodic broadcasts (observations of additional UDP traffic) and piggybacked onto various messages (observations of extra data transfers in Section 4.2.2). Access nodes use this information when selecting storage nodes.

Because of the nature of content addresses and the distribution of data across storage nodes, reads are likely to be spread equally across all the nodes, balancing the load in aggregate. Given the emphasis on write operations and the fact that network delays due to congestion almost never occur, the CentraStar version 2.0 analyzed here did not employ load balancing of *individual* read operations. This has been added to later versions and is similar to the load balancing of writes. The intra-box analysis could have also observed that each node balances load across its internal disks. Finally, the current hardware uses a different power distribution system which eliminates the constraints on placing replicas on nodes.

5.5 Content Addressability

Wisconsin: We observed that Centera uses a BLOB size of 100 MB, potentially missing out on opportunities for capacity savings achieved by using smaller BLOBs [25]. Of course, smaller BLOBs imply more metadata for BLOB tracking, which may not be desirable. Hence, if an application wishes to maximize its usage of content addressability, it must do so itself, not expecting the system to find more detailed content similarity among the objects.

EMC: Implementing single-instance storage at the object level (or 100 MB chunks) allows efficient storage and management of hundreds of millions of objects. Many applications take advantage of the single-instance feature and combine it with fast lookup to potentially eliminate unnecessary data transfer.

6 Related Work

Our intra-box techniques are similar to a recent line of work in the performance debugging of complex systems [1, 5, 6, 8, 12]. The major difference between our work and all related work is the level of detail that we can infer; because we assume knowledge of how storage systems generally function (*e.g.*, that they support caching, prefetching, and other domain-specific functions), we are able to discover specific structural and policy details that more general techniques cannot. Further, the goals of our work and much of the related work differ; specifically, while our work seeks to understand the structure and policies of a storage system, other approaches are primarily aimed at performance debugging.

More specifically, Aguilera *et al.* infer causal paths in distributed systems using message level traces. Their techniques are particularly useful for finding a component that is a performance bottleneck. However, their approach is limited in that they assume each message is dependent upon the arrival of exactly one previous message; more complex dependencies are found in storage systems. Similarly, Chen *et al.*[12] detect failures and diagnose performance problems using runtime path analysis; unlike Aguilera *et al.*and our own analysis, Chen *et al.* assume the existence of message tags within the system to help track dependencies. One advantage of these two approaches over our intra-box techniques is that they are able to run while the system of interest is online and running a real workload. In contrast, our approach must be applied to a quiesced system with controlled workloads. In the future, we hope to extend our approach within operational systems.

Previous research characterizing the behavior of storage systems has operated in different domains. Some work has focused on a single disk [29, 32, 34]. For example, Worthington *et al.* identify various characteristics of a disk, such as the mapping of logical block numbers to physical locations, the size of the prefetch window, the prefetching algorithm, and the caching policy [34]. In our own previous work, we characterize traditional RAID systems, for which we are able to automatically infer the number of disks, chunk size, level of redundancy, and layout scheme [16].

Some related work has taken a similar approach to ours in slowing components down to learn more about the behavior of a system [5, 8, 21]. For example, Brown *et al.*[8] use table locking to infer the dependence of various higher-level queries on database tables. In comparison, we slow network and disk traffic to better understand various aspects of the storage system under test. Our communication slowdown mechanism is similar to that presented by Martin *et al.*[21]; however, their approach is used to learn which aspects of network performance affects application performance, whereas we use network slowdown to infer dependencies within components of our storage cluster.

7 Conclusion

In this paper, we have shown how intra-box techniques can be applied to deconstruct the protocols and policies of a modern commodity-based storage cluster, the EMC Centera. Through our analysis, we can infer much about the design and implementation of the system, without access to a single line of the source code. In general, we believe our study demonstrates the power of probe points within the system – by observing and slowing down various system components, much can be learned about the structure of a complex system.

As systems continue to grow in complexity, we believe that intra-box techniques are a much-needed addition to the toolbox of systems analysts. Not only should such techniques be developed further; rather, we hope that systems themselves are built with the intra-box approach in mind – the more externally visible probe points, the better. By "opening up the box", such systems will be more readily understood, analyzed, and debugged. The result will be a generation of higher performing, more robust, more reliable computer systems.

Centera is now in its third generation of hardware and four CentraStar releases have taken place since version 2.0. Therefore, some of the observations made here may no longer apply. Nonetheless, this work, and in particular the slow-down causality analysis, helped EMC fine-tune some aspects of the Centera protocols.

8 Acknowledgments

We would like to thank Lakshmi Bairavasundaram, Todd Jones, James Nugent, Florentina Popovici, Vijayan Prabhakaran, and Muthian Sivathanu for their helpful discussions and comments on this paper. We would also like to thank Ana Bizarro for her assistance in setting up access to the Centera. Finally, we thank the anonymous reviewers for their many helpful suggestions.

This work is sponsored by NSF CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM, Network Appliance, and EMC.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP '03*, Bolton Landing, NY, 2003.

[2] G. Amdahl, G. Blaauw, and J. F.P. Brooks. Architecture of the IBM System 360. *IBM Journal of Research and Development*, 8(2):87–101, April 1964.

[3] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[4] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *ISCA '95*, Santa Margherita Ligure, Italy, 1995.

[5] S. Bagchi, G. Kar, and J. Hellerstein. Dependency Analysis in Distributed Systems Using Fault Injection. In *12th International Workshop on Distributed Systems*, Nancy, France, October 2001.

[6] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Real-Time Modeling and Performance-Aware Systems. In *HotOS IX*, Lihue, Hawaii, May 2003.

[7] V. Bohossian, C. C. Fan, P. S. LeMahieu, M. D. Riedel, L. Xu, and J. Bruck. Computing in the RAIN: A Reliable Array of Independent Nodes. In *IEEE Transactions on Parallel and Distributed Computing*, 2001.

[8] A. Brown, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. In *The 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[9] J. F. Cantin and M. D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. Computer Architecture News (CAN), September 2001.

[10] M. Carson and D. Santay. NIST Network Emulation Tool. snad.ncsl.nist.gov/nistnet, January 2001.

[11] CAS-Community. http://www.cascommunity.org.

[12] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *NSDI '04*, San Francisco, CA, March 2004.

[13] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[14] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *SIGMETRICS '93*, pages 1–12, Santa Clara, CA, May 1993.

[15] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *ISCA '93*, San Diego, CA, May 1993.

[16] T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *ASPLOS XI*, pages 59–71, Boston, Massachusetts, October 2004.

[17] EMC. EMC Centera: Content Addressed Storage System. http://www.emc.com/, 2004.

[18] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[19] J. L. Hennessy and D. A. Patterson, editors. *Computer Architecture: A Quantitative Approach, 3rd edition*. Morgan-Kaufmann, 2002.

[20] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *ASPLOS VII*, Cambridge, MA, October 1996.

[21] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *ISCA '97*, Denver, CO, May 1997.

[22] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX 1996*, San Diego, CA, January 1996.

[23] Panasas, Inc. Panasas Active-Scale Storage Cluster. http://www.panasas.com, 2004.

[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *SOSP '95*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[25] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *USENIX '04*, Boston, MA, June 2004.

[26] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[27] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.

[28] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building reliable enterprise storage systems on the cheap. In *ASPLOS XI*, Boston, Massachusetts, October 2004.

[29] J. Schindler and G. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.

[30] C. Staelin and L. McVoy. mhz: Anatomy of a micro-benchmark. In *USENIX '98*, pages 155–166, New Orleans, LA, June 1998.

[31] T. Sterling, editor. *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[32] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[33] S. Woo, M. Ohara, E. Torrie, J. P. Shingh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA '95*, Santa Margherita Ligure, Italy, 1995.

[34] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *SIGMETRICS '95*, pages 146–156, Ottawa, Canada, May 1995.