CamCubeOS: A Key-based Network Stack for 3D Torus Cluster Topologies

Paolo Costa Austin Donnelly Greg O'Shea Antony Rowstron
Microsoft Research Cambridge, UK
{pcosta,austind,gregos,antr}@microsoft.com

ABSTRACT

Cluster fabric interconnects that use 3D torus topologies are increasingly being deployed in data center clusters. In our prior work, we demonstrated that by using these topologies and letting applications implement custom routing protocols and perform operations on path, it is possible to increase performance and simplify development. However, these benefits cannot be achieved using mainstream point-to-point networking stacks such as TCP/IP or MPI, which hide the underlying topology and do not allow the implementation of any in-network operations.

In this paper we describe CamCubeOS, a novel key-based communication stack, purposely designed from scratch for 3D torus fabric interconnects. We note that many of the applications used in clusters are key-based. Therefore, we designed CamCubeOS to natively support key-based operations. We select a virtual topology that perfectly matches the underlying physical topology and we use the keyspace to expose the physical locality, thus avoiding the typical overhead incurred by overlay-based approaches.

We report on our experience in building several applications on top of CamCubeOS and we evaluate their performance and feasibility using a prototype and large-scale simulations.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems; H.3.4 [**Information Systems**]: Information Storage and Retrieval

General Terms

Algorithms, Design, Performance

Keywords

Data center clusters, 3D torus topologies, key-based routing, in-network processing

HPDC'13, June 17–21, 2013, New York, NY, USA.

1. INTRODUCTION

The recent acquisitions of Cray Interconnect by Intel [45] and SeaMicro by AMD [41] confirmed the increasing trend of deploying HPC-inspired fabric interconnects in general-purpose data center clusters [49]. This trend has been mostly driven by the opportunity of drastically reducing space and power consumption. For instance, the SeaMicro SM10000-XE appliance has three times the density of today's servers and consumes half of the power while providing 12x more bandwidth [46]. An additional feature of these interconnect fabrics is that the switching functionality is distributed across the servers, typically using a 3D torus topology [29], like the one in Figure 1. This provides a tight integration between servers and network.

We explored the opportunities provided by this integration in the context of general purpose data centers [11] and we showed the benefits of implementing custom routing protocols [1] and performing in-network packet processing [10]. Unfortunately, most of these benefits are lost with current platforms, which use traditional networking stacks such as TCP/IP and MPI, which are oriented towards point-to-point server communication and completely hide the underlying topology, thus inhibiting the implementation of any in-network functionality.

To address these shortcomings, we developed CamCubeOS, a novel networking stack for 3D torus topologies. Many of the applications that run in a cluster are key-based, e.g., [6, 9, 14, 15, 24, 26], using keys to identify data or users. The CamCubeOS API has been designed to make it easier to develop these applications by supporting key-based routing along with traditional point-to-point, server-based, communication. CamCubeOS provides an API similar to the Key-Based Routing API (KBR) [12], which has been successfully used in many distributed hash tables (DHTs) [34, 38]. The main benefit of KBR is that the destination of a packet is represented by a key, which is mapped to a reachable server and is re-mapped to another server in case of failure. This means that, unlike for server-based communication in which the packet is dropped if the destination server is unreachable, with key-based routing the packet is *always* delivered because the system ensures that there will always be a valid mapping between keys and servers.

Current data center thinking is dominated by networking technology and abstractions designed to support the Internet and enterprise networking. CamCubeOS challenges these views, and demonstrates that the combination of HPC-inspired fabrics and key-based abstractions makes writing

applications easier and achieves higher performance than traditional setups.

A major issue when using key-based routing is how to expose the physical topology through the keyspace so as to minimize path stretch. CamCubeOS addresses this problem by making the physical and virtual topologies the same. The topology defines a keyspace: each server is assigned a coordinate in the 3D space defined by its location and neighbors in the 3D torus.

The workloads generally used in HPC are batch-based and computationally intensive. Normally, HPC clusters are *vertically* partitioned, i.e., applications are allocated on disjoint subsets of nodes. In contrast, CamCubeOS focuses on the workloads typically run in general-purpose data center clusters such as data analytics jobs, web services, and data stores. It uses a *horizontal* partitioning of the services; every server runs an instance of every service. We built a number of services on top of CamCubeOS, including a file distribution service, a `memcached`-inspired key-value cache layer, an extensible persistent key-value store, and Camdoop [10], a MapReduce-like system that, in common scenarios, achieves up to two orders of magnitude higher performance than Hadoop [42] and Dryad [23]. We also implemented a TCP/IP service to support legacy applications.

The CamCubeOS runtime controls when packets are sent on a link and uses fair or weighted-queues to control the number of packets that each service can send on a link. This provides good link bandwidth partitioning between services, which services can exploit to implement their own queuing policies as well as implementing their own transport protocols and routing protocols [1] if needed.

To evaluate the performance of our stack, we built a 27-server (3x3x3) 3D Torus prototype, using commodity servers interconnected through 1 Gbps Ethernet cables. Clearly, our prototype cannot match the performance of dedicated SeaMicro appliances and we use it as a proof-of-concept. We compared the performance of applications built on top of CamCubeOS against the performance of equivalent applications implemented using TCP/IP over the switch. The results demonstrates that the benefits of using CamCubeOS outweighs the overhead of servers participating in packet forwarding. For instance, the key-value store outperforms an equivalent switch-based application by a factor of 1.93 and the file distribution service achieves a throughput per member of 5.66 Gbps, which is more than five times higher than what can be achieved in a traditional, switch-based, setup.

2. BACKGROUND: CAMCUBE

CamCubeOS is part of CamCube, a research project that explores the benefits of using 3D torus topologies to interconnect clusters of servers for general-purpose applications.

2.1 Motivation

A 3D torus (or k-ary 3-cube) [29] can be visualized as a wrapped 3D mesh, with each server connected to six other servers, similar to the one depicted in Figure 1. This is a popular topology, used in high performance computing, e.g., the IBM BlueGene/L and Cray XT3/Red Storm, and in cluster appliances, e.g., the SeaMicro SM10000-XE. Its properties are well understood and it provides a high degree of multi-path, which makes the topology very resilient to link and server failure. There are also well known wiring

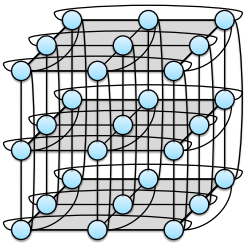


Figure 1: A 3-ary 3-cube (or 3D torus).

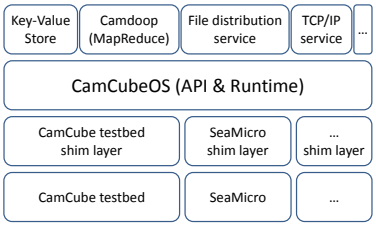


Figure 2: The CamCube software architecture.

techniques that allow efficient inter-connections using only short cables, each cable traversing a single server at most [2].

A key benefit of using a direct-connect topology like the 3D torus is that applications can implement their own custom routing protocols. Typically, both in switched-based network and HPC clusters, only one routing protocol, e.g., the IS-IS link-state protocol [28] for the former and a variant of greedy routing protocols [2, 36] for the latter, is available to applications. We demonstrated that in many cases running a custom routing protocol is beneficial both from the application and network perspective [1]. For example, applications that need to exchange large amount of data between two servers may be interested in using as many disjoint paths as possible so as to maximize the throughput between the two servers. In contrast, multicast applications desire to minimize the number of links used by the spanning tree to reduce the impact over network resources. We showed that custom routing protocols enable higher application-level performance, even when used concurrently, and reduce the network link stress.

A further advantage, beside enabling implementing custom *forwarding* decisions at each hop, is the ability of *processing* packets on-path. This means that applications can intercept packets on every server along the path from the source to the destination, inspect the *payload* (as opposed to just the *header*) of the packet and decide whether to drop the packet (possibly after storing its content in memory), to forward as is, to modify part of its payload, or to create a new packet. This functionality is really powerful as it enables sophisticated on-path operations, e.g., opportunistic caching or in-network aggregation [10].

2.2 CamCube Architecture

To simplify building CamCube applications, we developed CamCubeOS, a novel networking stack, which represents the core contribution of this paper. The overall CamCube software architecture is depicted in Figure 2. The top boxes represent the applications and services that we have implemented for CamCube. These include fully fledged applications, e.g., a key-value store (described in Section 5) and a MapReduce-like system (Section 6.2) but also lower-level services like the file distribution service (Section 6.1) that can be used by other applications to disseminate large files to a subset of the CamCube servers. We also developed a TCP/IP service (Section 6.3) that allows to run *unmodified* TCP/IP applications on top of CamCubeOS.

CamCubeOS consists of two parts: the CamCubeOS API, which exposes a key- and server-based interface, and the runtime, which handles the packet outbound queues and demultiplexes packets among applications and services. We

provide more details about both the API and the runtime in the next section.

To communicate between the CamCubeOS runtime and the specific underlying cluster hardware we use a shim layer. We have implemented a shim layer for the prototype 27-server testbed that we used in our evaluation and we are currently developing one for a SeaMicro appliance. The only functionality that we require from the underlying platform is the ability to send and receive packets to/from the six neighbors. No further functionality (e.g., multi-hop routing protocol) is assumed. Therefore, we expect that CamCubeOS be able to support most (if not all) 3D torus-based platforms currently available on the market with limited effort.

3. CamCubeOS OVERVIEW

We first describe the CamCubeOS API and then highlight the main features of the CamCubeOS runtime.

3.1 API

An early choice in the design of the CamCubeOS API was to support multi-hop key-based communication along with the traditional server-based communication. Services provide the reference to the packet (represented as a byte array) to be transmitted and specify whether the final destination is a key or server address. In the former case, the packet is always delivered to the server that is currently responsible for the given key. If, instead, a server address is used as destination, the packet is delivered only if that server is reachable, otherwise it is dropped. The motivation for selecting this model is that we wanted to make it easier to write key-based applications and, in particular, to simplify failure recovery.

One possible approach could have been to just run any structured overlay, e.g., Chord [38] or Pastry [34], over the underlying 3D torus network. This, however, would have created a mismatch between the overlay virtual topology and the physical network topology because each hop in the overlay would correspond to multiple links in the physical network. This would destroy locality, by increasing the physical hop count between two overlay neighbors, and induce fate sharing of links [20]. Typical solutions to this usually reduce failure resilience of the overlay, and still do not fully address the mismatch between the physical and virtual topologies. In contrast, in CamCubeOS we chose a virtual topology that matches the underlying 3D torus physical topology. We selected the virtual topology used by the Content Addressable Network (CAN) [33] structured overlay, a 3D torus topology. Hence, each link in the virtual overlay represents a single link in the physical topology.

In contrast to most structured overlays, node identifiers in CAN are not constant, and change over time as nodes join and fail. Many other structured overlays, like Chord and Pastry, have static node identifiers. This is necessary in CAN because it uses greedy routing over the keyspace, so failures with static identifiers can cause voids in the keyspace, that cause greedy routing to fail. In a structured overlay, non-static node identifiers are not ideal because the applications need to handle the mapping between nodes and keyspace, which usually requires migration of keys, as well as making maintaining consistency harder. In CamCubeOS we want the identifier of servers to be fixed. We assign a server identifier using its location in the initial physical topology. We use the 3D torus to define a 3D coordinate space. When a CamCube is first commissioned, a bootstrap service assigns each server an (x, y, z) coordinate representing its offset within the 3D torus from an arbitrary origin. The symmetry of the 3D torus means that any server can be the origin: it has no special role other than having the address *(0,0,0)*. The bootstrap service on each server exposes the coordinate and dimensions of the 3D torus to local services. It also provides a mapping between the one-hop neighbors and their coordinates. Intuitively, the coordinates of one-hop neighbors will each differ in only one axis and by +/-1 modulo the axis size. The assigned coordinate is the address of the server and, once assigned, it is never changed.

CamCube services are partitioned horizontally, i.e., all servers run an instance of the service. This means that the services running on all servers on the path to the destination are able to intercept and arbitrarily modify a packet or even drop it and create a new packet. For instance, a cache service can intercept a query packet along the path and, if a cache hit occurs, it can halt its propagation and immediately reply to the originating server.

Although the multi-hop routing functionality can be used to deliver packets end-to-end, in many cases, the key or server address specified is not the final destination of the packet but it is an intermediate destination. This allows services to implement custom routing protocols by leveraging the default routing protocol only between two intermediate destinations (possibly just between 1-hop neighbors). For instance, our key-value store (described in Section 5) uses intermediate destinations as cache locations and adopts a custom routing protocol to ensure that both query and reply packets are routed through them.

Services can also query the CamCubeOS API to retrieve up-to-date information about reachable and unreachable servers, in case they need fine-grained control over routing.

Finally, services can access the custom APIs offered by other services running on the same server. For example, a de-duplication backup service could use in its implementation the functionality offered by the key-value store and the file distribution service in addition to the standard CamCubeOS API.

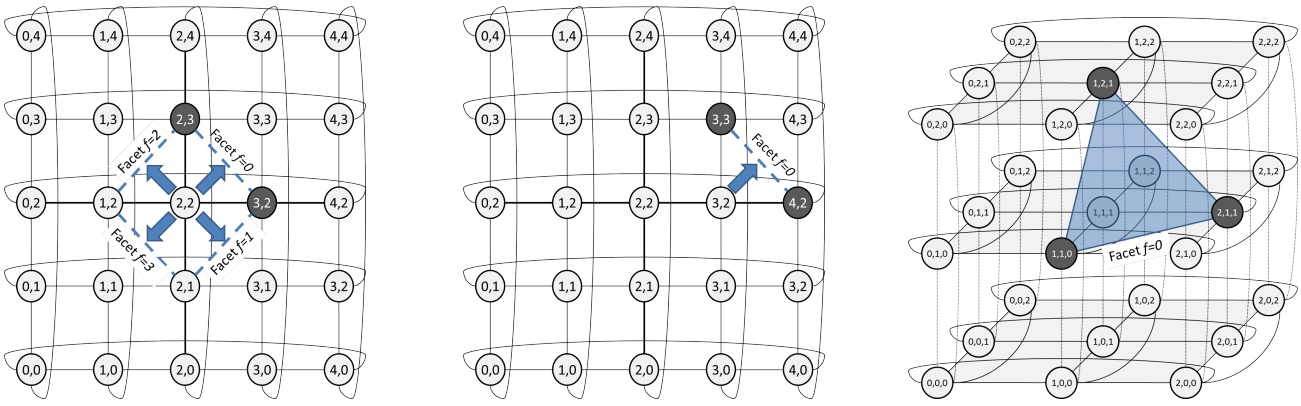
3.2 Keyspace Management

An important aspect of the CamCubeOS API is the keyspace management, especially in the presence of failures.

By default all keys are considered as 160-bit keys, but only the least significant 64 bits are used when routing a message to a key. Each key is mapped to a *root* server that is responsible for that key. The highest bits[1] of the 64-bit key generate an (x, y, z) coordinate. If the server with this coordinate–hereafter referred to as the *home server*–is reachable then it is the root.

If the home server is unreachable, then a naive solution would be to simply map the failed coordinate onto another single coordinate. Although this would preserve correctness, it would incur significant load skew because a single server would now be responsible for twice the number of keys of the other servers. A more elegant and efficient solution is to distribute the keys that the failed server was root for, over the set of one-hop neighbors. This maintains locality

[1]In our deployment, we use 4 bits per dimension, which can encode clusters of up to $(2^4)^3 = 4{,}096$ servers. If 8 bits per dimension were used, we could support clusters comprising more than 16 million servers.



(a) The facets of server $(2, 2)$. (b) The facet $f = 0$ of server $(3, 2)$. (c) The facet $f = 0$ of server $(1, 1, 1)$.

Figure 3: The facets in the 2D and 3D torus. Dark circles indicates the servers lying on facet 0.

and reduces load skew. To do this, when a root server is unreachable, the remainder of the 64-bit key is used to determine the coordinate of the server which will be the root. Conceptually, the 64-bit key can be thought as an (x, y, z, w) coordinate in a 4D space. The first three dimensions, x, y, and z, identify the home server while the fourth dimension w is used to select the neighbor to remap the key to if the current root server becomes unreachable.

This key-to-server mapping in the presence of failures should achieve the following design goals:

- *G1: Correctness.* If there is at least one server that has not failed, every key must have a valid (i.e., not failed) root server.
- *G2: Efficiency.* In case of failure, the keys of the failed server should be remapped to nearby servers (to achieve *locality*) while minimizing *load skew*.

A straw-man approach would be to simply represent the fourth dimension w as a ring like in Chord [38] and then assign each key range to a different reachable server. While this would satisfy $G1$ and would ensure good key distribution, it would completely disregard locality ($G2$), since the keys of a failed server F would end up on all servers, including severs located several hops away from F.

In CamCubeOS, we implemented a novel keyspace management solution that ensures correctness and that achieves high locality with a good load distribution. CamCubeOS provides a function `GetServersForKey(key,r)` that given a `key` and an integer `r` returns an ordered list of *reachable* servers $\mathcal{L} = [S_0, S_1, \ldots, S_{r-1}]$ with the following property. S_0 is the server that is currently responsible for `key`. S_1, is the one that would become responsible for `key` if S_0 failed. Likewise, S_2 would take over the key if both S_0 and S_1 should fail and so on.

If `GetServersForKey` is invoked with `r=1`, it just returns the current root of `key`. Invoking the function with higher values of `r` can be used to implement n-way replication. For example, `GetServersForKey(key,n)` returns the list $\mathcal{L}$ of the `n` servers that should store a replica of the object indexed by `key`. In this way, if the primary replica becomes unreachable, the new root server for `key` is one of the secondary replicas.

We now explain how we implemented the function `GetServersForKey` in CamCubeOS and how it achieves both design goals $G1$ and $G2$. To simplify the exposition, we start by describing it using a 2D torus and then we extend it to cover the 3D case.

2D torus. We first introduce the notion of a *facet*. Let us consider a home server $S = (x, y)$ and let us divide the coordinate space into four sub-spaces (i.e., squares in the 2D cases), each with a corner located in S. We call facets the lines connecting the neighbors of S in each sub-space. Figure 3(a) shows the four facets of server $(2, 2)$. The dark neighbors, $(2, 3)$ and $(3, 2)$, are the ones belonging to facet $f = 0$.

Given a key $k = (x, y, w)$, we can use some of the bits of w to select one of the four facets. Once we have selected a facet f, e.g., $f = 0$, we need to choose the order o in which the two neighbors should appear in $\mathcal{L}$. We can reuse the remainder bits of w to decide whether to pick first the neighbor on the x axis and then the one on the y axis ($o = xy$) or vice versa ($o = yx$). As an example, consider the scenario depicted in Figure 3(a) and a key $k = (2, 2, w)$ and suppose that the bits in w yield $f = 0$ and $o = xy$. In this case, the home server would be $S_0 = (2, 2)$ and S_1 and S_2 would be, respectively, $(3, 2)$ (i.e., the neighbor on the x axis) and $(2, 3)$ (i.e., the neighbor on the y axis).

The next servers S_i are retrieved by proceeding recursively in a breadth-first fashion. S_1 is taken as home server and its two neighbors on the chosen facet are appended at the end of the list. Next, the two neighbors on S_2's facet are appended to the list, unless they are already part of it. The process terminates when either `r` servers are retrieved or all reachable servers have been visited at least once. In our example, if we consider $S_1 = (3, 2)$ as the new home server and assume again $f = 0$ and $o = xy$, we have $S_3 = (4, 2)$ and then $S_4 = (3, 3)$ (see Figure 3(b)). Therefore, assuming no failures, `GetServersForKey(k,5)` would return the following ordered list of servers: $\mathcal{L} = [(2, 2), (3, 2), (2, 3), (4, 2), (3, 3)]$.

Since the keyspace is wrapped, eventually all servers will be visited at least once, which satisfies *G1*. Further, it is easy to see that this strategy ensures that, in case of failure, all the keys of the failed server are equally distributed among its four 1-hop neighbors (assuming all of them are reachable), which fulfills both the locality and the load-balance requirements of *G2*. This is particularly important if n-way replication is used because the closer the secondary replicas are to the primary, the higher bandwidth would be available.

In the 2D case, we have four facets per server and two possible orderings between the two neighbors of each facet.

Therefore, in total, for a home server $S = (x, y)$ we have eight possible different sequences of r-servers that can be generated, all starting from S. For efficiency, we pre-compute offline a lookup table containing these sequences and then, given a key $k = (x, y, w)$, we use the index $i = (w \mod 8)$ to retrieve the correct sequence for k.

3D torus. The main difference between the 2D and the 3D case is that in the latter the facet is a plane rather than a line. This is depicted in Figure 3(c), which shows the facet $f = 0$ for the home server $(1, 1, 1)$. This means that instead of two neighbors per facet, we now have three neighbors per facet. Therefore, there are six (as opposed to two for the 2D case) possible orderings to select these neighbors, corresponding to the six permutations of xyz. Further, instead of four facets per home server, we now have eight facets, one for each of the equal-size sub-cubes in which the cube can be divided. This yields $8 \times 6 = 48$ (as opposed to eight in the 2D case) possible entries in the lookup table.

3.3 Runtime

The runtime component is responsible for handling packet queuing and forwarding, and sharing the network resources across services. As already mentioned, CamCube services run on every server. Services are independent and can be registered and de-registered. Each service has a unique identifier. The packet header is a service identifier and when the runtime receives a packet it uses this to de-multiplex the packet and deliver it to the correct service. In most cases, services include their own identifier in the packet header but they could also include the identifier of a different service if the packet needs be received by a different service.

The runtime uses a simple link-state routing protocol and shortest paths to support key- and server-based multi-hop routing. Control traffic for maintaining the link-state is negligible, as failures are comparatively rare. When packets need be transmitted, the multi-hop routing protocol returns the set of outbound links that can be used to forward the packet towards a key or server, as required. In general, due to the high-degree of multi-path in the 3D torus, for a given destination the routing protocol is likely to yield multiple outbound links. Instead of arbitrarily selecting one of this, e.g., randomly, the runtime keeps track of the set of valid outbound links and transmits the packet over the link that becomes available first.

Each service maintains its own outbound packet queue. Services are polled, in turn, by the runtime for packets to be sent on each of the six outbound links, when there is capacity on the link. This means there is implicit per-link congestion control; if a link is at capacity then a service will not be polled for packets for that link. A service is able to control link queue sizes, packet drop policy, and packet prioritization. We provide a number of parameterizable default queue implementations, but services are free to implement their own. By default the runtime provides a fair queuing mechanism, meaning that each service is polled at the same frequency, and if s services wish to send packets on the same link then each will get $1/s$th of the link bandwidth. Partitioning the per-link bandwidth, combined with the fact that all packets on a single link are explicitly sourced by only two servers, means that they do not interfere. If services need to be partitioned into foreground and background tasks then a weighting queuing can be used.

4. THE CAMCUBE TESTBED

To evaluate the feasibility and performance of CamCubeOS and the services written on top of it, we built a small-scale testbed using commodity hardware. We now describe its setup and evaluate its base performance.

4.1 Setup

The testbed consists of 27 servers, interconnected in a 3D torus (3x3x3) like the one in Figure 1. We use Dell Precision T3500 servers with quad-core Intel Xeon 5520 2.27 GHz processors and 12 GB RAM, running an unmodified version of Windows Server 2008 R2. Each server has a 32 GB Solid State Drive (Intel X-25E SATA). In the current prototype platform we use a 1 Gbps Intel PRO/1000 PT Quadport NIC and two 1 Gbps Intel PRO/1000 PT Dualport NICs in PCIe slots to provide sufficient per-server ports. This limits the server form factors we can use as this requires three PCIe slots. However, we have started to use Silicom PE2G6I PCIe cards with six 1 Gpbs ports per NIC, requiring only a single PCIe slot. All experimental results are qualitatively and quantitatively the same when using a Silicom card, but as we have only a small number of cards for evaluation, we configured all servers identically using the Intel cards. One port is connected to a dedicated commodity 48-port 1 Gbps NetGear GS748Tv3 switch that uses store-and-forward routing (as opposed to cut-through). This provides the external connectivity to the CamCube. Six of the remaining ports, two per multi-port NIC, provide the 3D torus network. In all the experiments, intra-CamCube traffic is routed using the 3D torus topology, and all inbound and outbound traffic to / from the CamCube uses the switch.

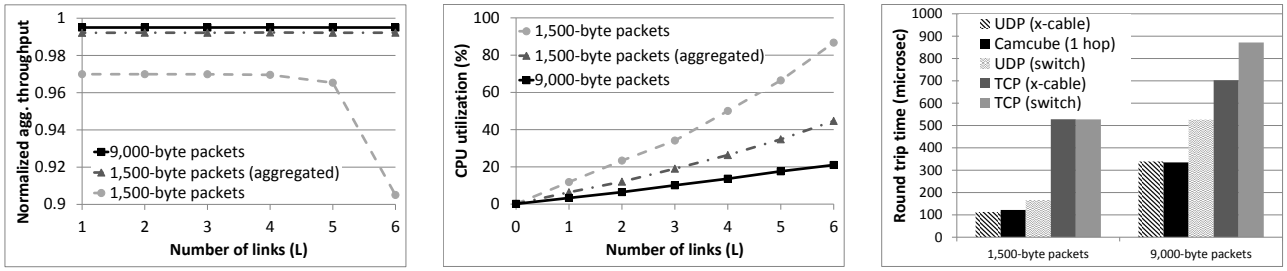
The Intel NICs support jumbo Ethernet frames of 9,014 bytes (including the Ethernet header). In the experiments, unless otherwise stated, we use jumbo frames and use default settings for all other parameters on the Ethernet cards.

Due to space constraints, we omit the full details of the shim layer implementation. However, we used techniques similar to those presented in [16,25]. In particular, we follow a zero-copy approach and use a pool of pre-allocated packet buffers. We also batch multiple asynchronous I/O requests to minimize the number of transfers across the kernel-user boundary. Further, whenever possible, we exploit aggregation at the packet level by concatenating multiple small packets into a single jumbo frame to reduce the overhead of Ethernet headers and the interrupts generated at the receiving server. Finally, we keep the transmit and receive paths separate by using two distinct threads per link to minimize interference between them. We also allow processing of incoming packets on the receive thread, provided the computation performed on the packet is small, to reduce the overhead of context switching on each packet.

In order to demonstrate how our platform scales we also present results using a packet-level discrete event simulator. The simulator allows the same codebase used on the testbed to be compiled for the simulator. It accurately simulates the Ethernet network links, assuming 1 Gbps links and with jumbo frames.

4.2 Benchmarking

This set of experiments benchmark the performance of our CamCube testbed. We also include simulation results to show the impact of server failure and scale.



(a) Normalized aggregate throughput. (b) CPU utilization. (c) Average round trip time.

Figure 4: Throughput and CPU utilization versus number of links and average round trip time.

4.2.1 Throughput and CPU overhead

For the purpose of performance analysis, we developed a benchmarking service that attempts to saturate $1 \leq L \leq 6$ links with full payload packets, and records the number of packets delivered per second per server across the servers inbound links and the CPU utilization. The benchmarking service ensures that when $L = k$, all servers have k inbound links receiving packets. This service runs the same code used by CamCubeOS to route packets. Therefore, this experiment allows us to quantify the network performance and CPU overhead of using the servers to route packets. We ran the experiments using standard packets (1,500 bytes) and using jumbo frames. We measure the CPU utilization by using Windows performance counters that provide, per core, an estimate of the CPU utilization. As we have a quad core processor, with hyper-threading, we have eight CPU readings per server. The CPU utilization per server is the average of these eight readings. We also confirmed that the per-core CPU utilization is not skewed across the cores.

Figure 4(a) shows the median aggregate throughput per server for $L = 1$ to 6 normalized with respect to the maximum achievable for that value of L. We do not show error bars, but across all experiments the maximum and minimum throughput observed per-server is within 1.28% of the median value. We define the aggregate throughput per server as the total number of packets per second received by the server and the total number of packets received by one-hop neighbors of that server that are sourced by this server. The maximum aggregate throughput for a server, when $L = 6$, is 12 Gbps. Passing packets across the kernel user-space boundary induces overhead, and intuitively, using larger packets is more efficient. Figure 4(a) shows that for jumbo frames all servers are able to sustain close to the maximum aggregate throughput. At 1500-byte packets the results show CamCubeOS is able to achieve about 0.97 of the maximum aggregate throughput for up to $L = 5$. This is less than the jumbo frames as we are not including the Ethernet header overhead, and at 1500-byte packets this overhead is higher and accounts for the difference. When $L = 6$ the 1500-byte packet experiment achieves approximately 0.91. If we use the packet-level aggregation optimization described above, where multiple packets are aggregated into a single jumbo frame, the throughput increases, as the Ethernet headers are removed, and close to maximum aggregate throughput is achieved for all values of L.

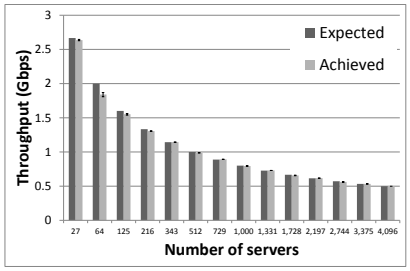
With servers using CPU resources to handle packets, we next look at the CPU load during the experiment. Figure 4(b) shows the average per-server CPU overhead for $L = 0$ to 6. We do not show error bars, but the maximum and minimum are within 10.6% of the median value for all results. When $L = 0$ there is no network load and the CPU utilization is close to zero. Figure 4(b) shows that for jumbo packets, the CPU utilization is low despite the overhead associated with passing each packet across the kernel user-space boundary. When all six links are being saturated, CPU utilization is less than 22%. Using 1500-byte packets incurs a higher CPU overhead, over 87% when saturating all links. When we use the packet aggregation optimization this drops to less than 45%. This demonstrates the effectiveness of packet aggregation optimization: it increases throughput and decreases CPU overhead.

4.2.2 Latency

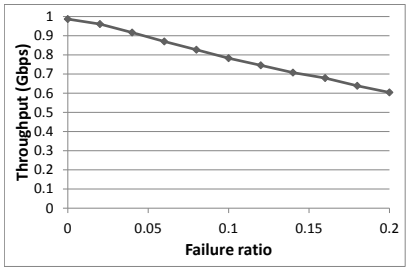
Next, we use a ping service to measure the increase in communication latency introduced by CamCubeOS. The ping service uses the routing service to route packets between two arbitrary servers. The source generates a packet and then forwards it towards the destination server. Each server on path receives the packet, modifies a counter in the payload, and then forwards it. When it reaches the destination, the destination reverses the source and destination identities in the packet and then routes the packet back towards the source. On-path the runtime and the ping service ensure that the packet is zero-copied. The source sends 10,000 ping packets, sequentially, for 1,500-byte and jumbo packets, and then reports the average round trip time (RTT).

To show the relative performance of the runtime, we also measured the performance of the standard Windows TCP/IP stack performing a ping operation between two servers using UDP and TCP over the switch for both packet sizes. The jumbo frame performance over the switch is poor, presumably because it is a commodity store-and-forward switch. Therefore, we also ran the TCP/IP experiments using a standard cross-over cable directly connecting two servers. In the case of TCP, the source creates a new socket for each ping. This is consistent with a recent analysis of data center network traces [19], which shows that most flows are smaller than 10 KB. If, instead, persistent TCP connections were used, we expect the RTTs to match those obtained with UDP and, hence, we did not run this configuration.

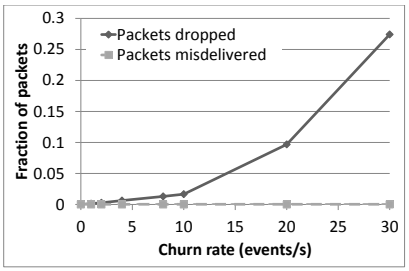
Figure 4(c) shows the average RTT for the different configurations, split by 1,500 and 9,000 byte packets. For CamCubeOS we show the results for a single hop. In all cases, the CamCubeOS performance is comparable to UDP using the cross-over cable. As would be expected, the TCP latency is much higher due to the additional round trips needed to create and close a TCP socket. To support arbitrary routing, multi-hop routing will be required for CamCubeOS. To explore this performance we also configured the ping service to



(a) Per-server throughput versus number of servers with an all-to-all traffic pattern.



(b) Median throughput per-server versus server failure ratio with an all-to-all traffic pattern (512 servers).



(c) Fraction of packets dropped and misdelivered versus churn rate (512 servers).

Figure 5: Simulation results.

use a 15-hop path. This is the maximum hop count between two servers in a 1,000-server CamCube (i.e., a 10x10x10, with an average path length of 7.5 hops). The average RTT is 2.22 ms for 1,500-byte packets and 5.87 ms for jumbo packets, respectively about 4.2 times and 6.73 times slower than TCP. This is the worst case and demonstrates the latency overhead of routing through servers. In an 8x8x8 CamCube, e.g. 512 servers, the maximum hop count is 6 and the average only 3. The latency would be comparable to using TCP in a traditional 512 switch-based cluster. Further, as many services induce locality often the average hop count is further reduced. As we demonstrate in Section 5, a service built on top of CamCubeOS can significantly outperform the same service running on a traditional switched setup.

4.2.3 Scaling

Next, we explore how our testbed would scale with the number of servers. We do this in simulation with an all-to-all traffic pattern, emulating the shuffle phase in a MapReduce job. Each server sends packets to $N-1$ servers, and receives packets from $N-1$ servers. All packets are routed on shortest paths using the multi-hop routing protocol. To keep the experiment simple we do not use a flow control protocol, but instead calculate the expected throughput per server, given N servers, and then have each server generate packets at that rate. In the experiment we measure the achieved throughput per-server, which includes only packets delivered to the server, *not* packets forwarded by the server.

Figure 5(a) shows the expected and achieved throughput per server as we scale the number of servers. The achieved and expected results are close and, as N increases, the expected throughput drops. At 27 servers the throughput is 2.7 Gbps, at 512 servers this has dropped to 1 Gbps and at 4,096 servers this is 500 Mbps. Recall that in a traditional data center this would represent an over-subscription rate of only 1:2 at 4,096 and full bisection bandwidth at 512 (1:1 over-subscription). At core routers over-subscription is normally much higher, 1:20 is not unusual and can be as high as 1:240 [19], and 1:4 at a rack level is considered good.

4.2.4 Server failure sensitivity

So far we have not considered the impact of server and link failures. Figure 5(b) shows the median throughput per server with 512 servers using the same all-to-all traffic pattern used in the previous experiment, when up to 20% of the servers are failed. Failed servers are selected randomly. The results show that the throughput drops linearly with the number of failures. When 20% of the servers have failed throughput has dropped by 38.77%.

4.2.5 Maintaining keyspace consistency

The final experiment in this benchmarking section evaluates the keyspace consistency. We run an experiment on the simulator using 512 servers, and we induced server churn. The churn rate represents the number of servers that join and fail per second. The inter-failure times are selected randomly from an exponential distribution with a mean of 1/(failure rate), where the failure rate is (churn rate)/2. When a server fails, after a failure duration period, it rejoins the CamCube. The failure duration is set as a function of a target fraction of servers to have failed at any point in time and in these experiments this was set to 10% of the servers failed. The experiment runs for 10 minutes of simulated time. Every active server generates packets destined to a random key at a rate of 1,000 per second. All packet deliveries and drops are recorded. We use an oracle, which has global knowledge, to determine if a packet has been correctly delivered to the server responsible for the key (e.g. the key root server).

Figure 5(c) shows the fraction of packets misdelivered and dropped. Delivering a packet to the wrong server (misdelivered) is an issue with correctness, as the server will handle the packet as though it was responsible for the key. Across all experiments we had no misdelivered packets. Dropped packets impact performance, as the source of the packet will probably need to re-transmit the packet. Even at high churn rates for a data center (two failures per second), the packet drop rate is below 1%. For higher churn rates, the packet drop rate increases up to 27.41% but these correspond to extreme, unrealistic, scenarios in which on average 15 servers would fail every second.

5. KEY-VALUE STORE

To show how to build an application on top of CamCubeOS, we describe and evaluate on our testbed the implementation of a key-value store. Key-value stores such as Amazon Dynamo [15], Google BigTable [9] and Facebook Haystack [6] are widely used and often represent the critical component of a complex system. To achieve high performance, we augmented our persistent key-value store service with a caching service, similar to memcached [26], to cache the results of popular queries.

In this section, we first distill the main design features of the caching and the store service and then we evaluate the

performance of the two services combined against a similar setup using a switch-based network.

5.1 Caching Service

Typically, distributed in-memory key-value caches have an API that allows read, write and delete operations on key-value pairs, with the key specified as a string and the value as a byte array. For example in an image store the key could be `image:user3:picture.jpg`, which encodes service, user and image name.

We have implemented a distributed key-value cache that exploits caching techniques from structured overlay applications to improve performance [32, 35]. In particular, the string key is converted into a 160-bit key, *memId*, by taking its SHA1 hash. The key-value pair is stored on the server that is responsible for the key. Hashing ensures that the memIds will be uniformly distributed across the keyspace. On writes the value is simply routed, using key-based routing, to the root server for the memId where the value is stored in memory. On a read the request is routed to the root server, using key-based routing, which looks up the associated value and returns it or produces an error message.

To demonstrate the flexibility of CamCubeOS, we extended the basic cache to handle hot spots, which are not supported in memcached. This is done by dynamically creating replicas of cached values on-demand, as done in many structured overlay applications. We use a function that, given a memId, generates k additional keys that are uniformly distributed in the keyspace. On reads, the lookup source generates the k keys, and then selects the closest in the keyspace from the k keys and memId. The read is routed, using key-based routing, to this key. When a server, C, receives the lookup and it is the destination, it checks if the key-value pair is locally cached. In case of a cache miss, if the server is not the root for memId, the key is routed to memId. When the root for memId receives the read it sends a response to C and records that C has a copy. When the response is received by C, it is cached (so that future requests for the same key can be handled locally) and routed to the original server that performed the lookup request.

When a server is notified by the CamCubeOS API that another server has failed it flushes all cached items for which the failed server would be the root for the memId. If a new value is associated with a key, then the root ensures all other cached copied are flushed, by explicitly contacting servers it knows requested copies of the key to cache.

This is an example of a service inducing locality, to limit load skew caused by hot keys, and to reduce the average lookup hop count. Having the k additional keys uniformly distributed means that average number of hops traversed to perform a lookup is lower.

5.2 Key-value Store Service

Our implementation of a key-value store maintains r replicas of each value inserted. Each value is associated with a 160-bit objId, and a replica is stored on the r servers returned by `GetServersForKey` (see Section 3.2). Therefore, unlike in the key-value cache, the r replicas will be stored on servers that will, under normal operation, be one-hop neighbors in the network. If the server responsible for objId fails, one of the $r-1$ replicas will become the primary, and the replica selected will be the one responsible for the keyspace in which objId lies after the failure. The store is configurable to use a disk-based or memory-based store for the key-value pairs. By default we use $r = 3$. It supports versioning of key values, and provides insertion, lookup, delete and a compare and swap operation.

The key-value store is failure tolerant, leveraging the remapping of keys to servers on failure. In the presence of failures, the meta-data is immediately updated to reflect the failure, but re-replication of data is delayed by t seconds, currently $t = 180$, so that servers being rebooted do not generate significant data migration. If a failed server rejoins after t seconds, a recovery protocol updates the stale data on the server in the background. This can involve the transfer of a significant amount of data, and the key-value store is able to continue to service read and write operations concurrently with the recovery protocol.

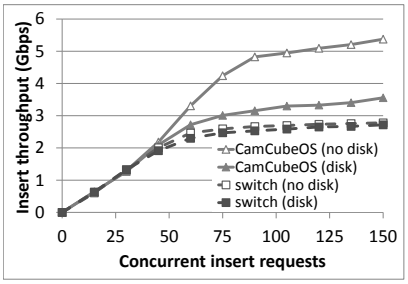
The key-value store can be used with the key-value cache, and normally, the memId = objId. The cache service is optimized so that, if it is caching items that are stored in the key-value store, it does not locally duplicate the storage.

5.3 Evaluation

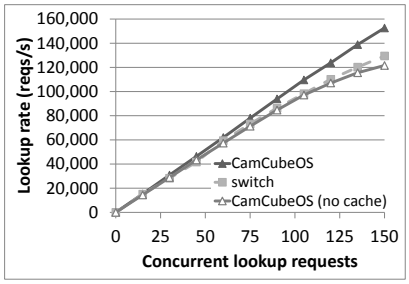
In order to evaluate the performance of the key-value store and memory cache, we implemented a simple image store on top of them. Images are stored in a disk-backed key-value store using three-way replication. External clients can insert images into the store, which also causes a small thumbnail image to be automatically generated and inserted into the memory cache. Clients outside the CamCube can lookup the original or thumbnail image. Images and thumbnails are associated with a unique 160-bit key. Similar image stores underpin many social networking-like applications.

In order to generate a workload to drive the experiments we attached ten additional servers to the switch used by CamCube (the *load generators*). Each load generator has a single 1 Gbps link to the switch, therefore supporting an aggregate throughput of 10 Gbps. Each load generator can issue up to 15 concurrent requests, providing an aggregate peak load of 150 concurrent requests. Whenever a request is satisfied a new request is immediately generated, and a request can be an image insertion or lookup. All CamCube servers are front-end servers (*FEs*) that accept incoming TCP connections from the load generators. Load generators randomly select a FE and send a request to it.

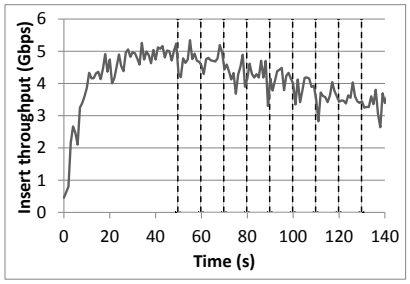
We compare against a switch-based configuration, inspired by traditional cluster architectures, in which servers communicate only through the switch, using a standard TCP/IP network stack. In this configuration, each server uses only 1 Gbps link. The reason is that in a multi-tier tree network topology, adding a second network interface to each server increases the bandwidth over-subscription. Maintaining the bandwidth over-subscription requires the uplink capacity from the top of rack switch to be doubled (as well as the capacity of aggregation/core switches). This would be expensive. Without doing this, the increased bandwidth within the rack would only help in-rack communication, not across racks. This would provide no benefit for workloads like MapReduce. We therefore decided, as has prior work, e.g., [3, 21, 22], to keep the currently used network topology, without increasing the number of links per server. To provide a fair comparison, we created a shim that allowed the image store to use TCP for communication. In this way, the same core key-value store code is used in the two configurations and just the communication layer is different.



(a) Insert throughput.



(b) Lookup requests per second.



(c) Insert throughput when failing a server every 10 s (dotted lines indicate the time of each failure).

Figure 6: Image store performance for insertion and lookup workloads.

We present results for two simple workloads: an insertion workload where all requests are to insert an image, and a lookup workload. For the insertion workload the load generators have 233 JPEG image files, cached in memory, with an average size of 1.47 MB and the minimum and maximum size of 0.50 MB and 2.63 MB, respectively. A unique key is generated per insertion and an image selected at random. For the lookup workload 23,300 images are inserted, and we then generate requests for thumbnail images, selecting uniformly at random from the set of images inserted.

Figure 6(a) shows the aggregate insertion throughput at the load generators versus the number of concurrent insertion requests, varied from 15 to 150. We show the results for the TCP switch-based version (*switch*) and CamCubeOS. We found that for the CamCubeOS version the bottleneck in performance was the low disk I/O rate and, hence, we show performance with and without the disk write (labeled *disk* and *no disk*, respectively). We use only a single disk per server, and adding further disks would increase the disk bandwidth rates per server to the point where they were not the bottleneck. With and without disk writes enabled, the CamCubeOS implementation outperforms the switch-based version. At 150 concurrent requests, the insert throughput achieved by CamCubeOS is 1.31 times (disk) and 1.93 times (no disk) higher than the switch-based version. The reason is twofold. First, in the switch-based version, the server uplink bandwidth is used for external and internal traffic while for CamCubeOS two distinct networks are used. Second, as explained in Section 3.2, CamCubeOS ensures that, under normal operation, the secondary replicas are one-hop neighbors of the primary replica. This means that the primary replica can use two distinct links to copy the data to the secondary replicas. In contrast, in the switch-based version the traffic to the secondary replicas is sent through the same single uplink that is also handling the external traffic, which significantly reduces the overall throughput.

Figure 6(b) shows the achieved lookup rate in requests per second versus the number of concurrent thumbnail lookup requests. The thumbnails are retrieved from the in-memory cache service, and are on average 3.55 KB in size. To understand the impact of the distributed key-value cache, we run two configurations of the image store on CamCubeOS, one using the key-value store only (*disabled cache)* and one using also the distributed key-value cache. When running without the cache, the CamCubeOS version achieves a slightly lower rate of lookup requests per second than the switch-based version. However, if cache is enabled, the CamCubeOS version is able to scale linearly with the number of concurrent requests and sustain the full load of the 10 generators. The reason is that caching exploits locality and reduces latency by decreasing the hop count of each lookup request. At 150 concurrent requests, the median latency for CamCubeOS with distributed cache is 0.83 ms and the 95th percentile is 1.70 ms (respectively 0.97 ms and 2.13 ms with the cache disabled) while the switched based implementation has a median latency of 0.95 ms and a 95th percentile of 2.22 ms.

We also measured the CPU utilization for all CamCube servers. Across all insert workloads, the median CPU utilization among all servers is always lower than 33% and the 95^{th} percentile is always lower than 77% (respectively 16% and 19% for the lookup workloads). This shows that, although all CamCube traffic is being routed hop-by-hop through the servers, the CPU is not the bottleneck.

Last, to evaluate the impact of failures when running on CamCubeOS, we ran an experiment in which we permanently failed 9 randomly selected servers, one every 10 s. We used the insert workload as this is the most critical for our system because it is both computationally and bandwidth intensive. We disabled disk writes and we generated a load of 150 concurrent requests. In Figure 6(c), we plot the insert throughput (one second moving average) versus the experiment time. We let the system run for 50 s to reach a steady state and then we started failing servers. Results show that server failures have small impact on the insert throughput: after 9 servers have been removed from the system, the insert throughput is only 38.85% lower. The reason is that, as explained in Section 3.2, the load of the failed server is evenly distributed across its one-hop neighbors and, hence, the impact of a failure is minimized. Also, the availability of multiple paths partly compensate for the loss of the links attached to the failed servers.

5.4 Source Code Size

CamCubeOS is designed to provide an easy platform on which to design and implement services. It is very difficult to quantify this, but as an indication, we counted the number of lines code for our key-value store (counting semi-colons). The distributed key-value cache service, including its ability to handle hot spots by exploiting physical locality, and the persistent key-value store required only 1,335 lines of C# in total. In contrast memcached version 1.4.5, which does not provide support for replication nor hot spots, is written in 4,671 lines of C. The point-to-point or point-to-key reliable transport service used by the key-value store is 569 lines of C#, respectively. Finally, the image store application is only

151 lines of C#, of which the majority is to implement front end server functionality, as most of the other functionality is achieved using the key-value store and the distributed key-value cache service.

6. BUILDING SERVICES ON CamCubeOS

Many of the applications running in modern data center clusters are key-based and adopt the so-called *partition-aggregate* model [4]. These include key-value stores, e.g., [15, 24, 26], large-scale data analytics platforms such as MapReduce [42] and Dryad/DryadLINQ [23, 40] as well as real time stream processing and web applications, e.g., [7, 43, 48].

These applications represent a great match for CamCubeOS as they can naturally leverage its key-based API and its efficient support for *custom forwarding* and *arbitrary processing* of packets on path rather than relying on inefficient, application-level overlay networks.

In the previous section, we demonstrated the benefits of CamCubeOS in the context of the key-value store application. In this section, we provide further examples of its flexibility. First, as an example of custom forwarding, we report on the design and evaluation of a file distribution service. Next, we illustrate the benefits of in-network processing by briefly discussing the design and implementation of Camdoop. Finally, to show the support for legacy applications, we discuss the TCP/IP service, which enables running existing unmodified TCP/IP applications on top of CamCubeOS.

6.1 File Distribution Service

The file distribution service allows to transmit a file to an arbitrary subset of CamCube servers using a multicast tree. The key-based API offered by CamCubeOS greatly simplifies this task. The tree is built using techniques similar to those used in prior work on building multicast trees on structured overlays [8]. The tree vertexes are represented by keys in the keyspace and parent and children are chosen such that the edges of the trees map onto a single link. Key-based routing is used to route packets from vertex to vertex.

Traditionally, application-level multicast are inefficient due to the mismatch between the physical and the logical topology, which causes link sharing (multiple logical links mapped to the same physical link) and path stretch (a single logical link is mapped to multiple physical links). The CamCubeOS API, instead, exposes the key locality to the developers and makes it easier to ensure that, when there are no failures, there is a one-to-one correlation between the next hop in the keyspace and a link in the physical topology. This allows the full 1 Gbps per server to be achieved.

Building a single distribution tree limits the throughput to the data rate of a single link, i.e., 1 Gbps. In order to increase throughput the service builds six *disjoint* trees, allowing all six links per server to be used. Without failures this enables a file distribution rate of 6 Gbps. In general, if a server is distributing a file to the $N-1$ other servers in the CamCube then $6(N-1)$ (unidirectional) links will be utilized, with each link being traversed by a single tree.

This further shows the benefits of knowing the topology and control the routing. However, one of the main benefits of the key-based abstraction supported by CamCubeOS is that it minimizes the effort of maintaining the tree connected in the presence of failures. When a server fails, then the key-based routing will ensure that packets are still delivered to the vertex in the keyspace, but the vertex will be mapped to a different server. This means that the packets for that edge will now traverse one or more physical links. If there are multiple paths available, these will be exploited.

	Throughput (Gbps)		
Single instance	5.662		
Three concurrent instances	1.852	1.854	1.848

Table 1: Minimum per-member throughput for multicast with a single and three concurrent instances.

6.1.1 Evaluation

To evaluate the performance of this service, we multicast a 750 MB file to a subset of the servers. In the experiments, we measure the achieved throughput by determining the time from when the first packet is sent to when each group member has successfully received the entire file (including any time required for re-transmissions due to packet loss). We then divide 750 MB by the elapsed time to determine the multicast throughput in Gbps. To ensure bandwidth is the bottleneck resource, the file is cached in memory on the source, and the group members store the file in memory.

The link-level queue management provided by CamCubeOS allows the per-link bandwidth to be efficiently shared across competing services. To demonstrate this, we also ran multiple instances of the file distribution service concurrently and we measured the throughput achieved by each.

Table 1 compares the *minimum* throughput per member when running the service in isolation and when running three instances of the file distribution service concurrently, each distributing a 750 MB file to 26 servers. The first observation is that when running in isolation the multicast service achieves a throughput close to the theoretically maximum achievable of 6 Gbps. We also ran with group sizes of 1, 6, and 13 members and obtained similar results, but we omit these for space reasons. In a traditional data center files can be distributed using unicast, IP multicast or application-level multicast. If the servers have 1 Gbps links to the ToR switch, as is normal, this will be the upper throughput bound for any of these approaches.

The results also show that when running three concurrent instances the bandwidth is split evenly between the three instances. Each instance achieves approximately one third (1.9 Gbps) of the throughput achieved by a single instance. This demonstrates that CamCubeOS ensures fair-sharing of the bandwidth across multiple services. It would be trivial to configure weighted-sharing of links.

6.2 Large-scale Data Analytics

For completeness, we briefly report on Camdoop, a CamCube service to run MapReduce jobs. We extensively describe its design and evaluation in [10]. Here, instead, we focus on how it benefited from the CamCubeOS API.

MapReduce-like systems such as Apache Hadoop [42] and Microsoft Dryad [23, 40] are used daily by large-scale companies as well as small and medium businesses to process large amount of data. These systems typically operate in a partition-aggregate fashion. Input data is partitioned across multiple servers and locally processed. These intermediate results are then merged and aggregated together. These systems significantly stress network resources, due to the large amount of data that needs to be shipped across the network during the aggregation phase.

A common property of MapReduce-like jobs is that the output size is often a small fraction of the input size, due to the high degree of aggregation occurring during the process. We leveraged this property in Camdoop by performing partial aggregation of packets on path. This drastically reduces the traffic (and, hence, the job running time) because at each hop, only a fraction of the data received is forwarded. This enables achieving a speed-up of up to two orders of magnitude compared to Hadoop and Dryad/DryadLINQ [10].

CamCubeOS greatly simplified the implementation of Camdoop. Camdoop uses six aggregation *disjoint* trees, built in a way similar to the file distribution service. Due to the ability of CamCubeOS to intercept packets on path, the Camdoop service can easily receive the packets, aggregate their content in a new packet and forward it to the upstream server. Since each vertex is represented by a key, little effort is required to deal with fault tolerance because we leveraged the CamCubeOS key-based routing to keep the tree connected in the presence of link or server failures.

We used the key-based API also to map tasks to servers, using the task ID as the key. In case of failure, tasks are re-assigned using the re-mapping scheme described in Section 3.2. This ensures that tasks get re-scheduled on neighboring servers, thus preserving the tree locality.

6.3 Legacy Applications

Although CamCubeOS has been designed to efficiently run key-based services, it can also support generic workloads, including legacy TCP/IP applications. To support legacy applications we have created a TCP/IP service that enables running unmodified TCP/IP applications on CamCubeOS. To achieve this, we bound the TCP/IP stack to a virtual Ethernet interface. All packets sent to this interface, are intercepted and delivered to the CamCubeOS runtime, which is also able to inject packets into the bottom of the TCP/IP stack. The TCP/IP service encapsulates and tunnels the intercepted IP packets across the CamCube to the destination server, with the exception of ARP requests. These are spoofed, and a MAC address generated that encodes the destination. Applications can use the standard TCP/IP stack and socket API and are unaware that they are running on top of a 3D torus, which is presented as a single layer 2 IP network.

The 3D torus introduces multi-path, and when TCP/IP tunneling this is a challenge. It increases the aggregate throughput between servers, and provides a high resilience to failures, but causes out of order packet delivery. Traditionally, TCP handles out of order packet delivery badly, and it leads to throughput collapse which we also observed. Making multi-path TCP work is a current active area of research [31]. The TCP/IP service uses small buffers at the destination, which allows the TCP/IP service to reorder the packets of each flow.

6.3.1 Evaluation

To understand the performance of running TCP/IP applications on CamCubeOS, we ran an experiment in which every server simultaneously transferred 1 GB of data to each of the other 26 servers using TCP. This creates an all-to-all traffic pattern, similar to the one generated by the MapReduce shuffle phase. We obtained a median aggregate TCP inbound throughput of 1.49 Gbps per server. This is higher than the maximum throughput achievable in a conventional cluster where servers have 1 Gbps uplinks to the switch. However, the maximum throughput achievable is 2.7 Gbps, but out of order packet delivery induced by multi-path limits the throughput achieved.

7. RELATED WORK

Torus-based fabric interconnects have been very popular in the High Performance Computing (HPC) and have recently started being deployed also in data center clusters, using proprietary technologies, e.g., SeaMicro Freedom fabric [47], as well as open industry standards such as HyperTransport [44], a consortium including, among others, AMD, Broadcom, Cisco, Dell, HP, NVIDIA, Oracle, and Xilinx.

Compared to existing switched-based networks, these solutions enable reducing capital and operational costs while delivering higher throughput [49]. These systems, however, still rely on traditional networking stacks like TCP/IP or MPI, which completely hide the topology and provide an end-to-end abstraction. This makes it impossible to perform efficient in-network packet processing, which, as we demonstrated in this paper, can significantly improve performance. For instance, the IBM Project Kittyhawk [5] proposes using the BlueGene/P supercomputer in a data center, and then runs unmodified TCP/IP applications, by making the 3D torus topology appear as a flat layer 2 IP network, as does our TCP/IP service. In contrast, CamCubeOS explicitly exposes the topology to allow services to exploit it.

Another key difference between CamCubeOS and mainstream networking stacks is that CamCubeOS natively supports a key-based API. Many of the applications running in clusters are key-based, e.g., [15, 24, 26]. CamCubeOS makes it easier to implement these applications, because it removes the burdens of managing the key space and ensuring consistency in the presence of failures. HPC clusters usually do not handle failures, e.g. BlueGene/L can tolerate three failed links [30], and indeed they use routing protocols that do not necessarily converge with link failure [13,18,39]. Failures are handled by stopping the system and restarting it [17,27]. In contrast, CamCubeOS uses key-based routing to mask failures and simplify failure recovery.

In the networking community there have been several proposals for new networking topologies for the data center, including direct-connect or hybrid ones, e.g., [21, 22, 37]. The goal of these proposals is to increase the bisection bandwidth, often motivated by MapReduce-like workloads, but they still assume that TCP/IP is used on top. CamCubeOS challenges this view and shows that these new topologies provide a great opportunity to rethink established practices in networking in order to achieve high performance and reduce development complexity.

8. CONCLUSIONS

CamCubeOS explores the question: are the current communication abstractions for clusters, derived from networking principles used in enterprise networks and the Internet, the best? Based on the observation that many clusters run key-based applications and the increasing availability of 3D torus fabric interconnects, CamCubeOS is designed from the ground up to support developing and running key-based services. It represents a very different design point from traditional data centers, but the results show that it is feasible and efficient.

Acknowledgements. We thank Thomas Zahn for his contribution to an early version of CamCubeOS, and the anonymous reviewers and our shepherd, Prasenjit Sarkar, for their insightful feedback and advice.

9. REFERENCES

[1] Abu-Libdeh, H., Costa, P., Rowstron, A., O'Shea, G., and Donnelly, A. Symbiotic Routing in Future Data Centers. In *SIGCOMM* (2010).

[2] Adiga, N. R., Blumrich, M. A., Chen, D., Coteus, P., Gara, A., Giampapa, M. E., Heidelberger, P., Singh, S., Steinmacher-Burow, B. D., Takken, T., Tsao, M., and Vranas, P. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development 49*, 2 (2005).

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

[4] Alizadeh, M., Greenberg, A. G., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center TCP (DCTCP). In *SIGCOMM* (2010).

[5] Appavoo, J., Uhlig, V., and Waterland, A. Project Kittyhawk: building a global-scale computer: Blue Gene/P as a generic computing platform. *OSR 42*, 1 (2008).

[6] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a Needle in Haystack: Facebook's Photo Storage. In *Usenix OSDI* (2010).

[7] Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., and Aiyer, A. Apache Hadoop Goes Realtime at Facebook. In *SIGMOD* (2011).

[8] Castro, M., Druschel, P., Kermarrec, A.-M., and Rowstron, A. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC 20*, 8 (2002).

[9] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. In *OSDI* (2006).

[10] Costa, P., Donnelly, A., Rowstron, A., and O'Shea, G. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *NSDI* (2012).

[11] Costa, P., Zahn, T., Rowstron, A., O'Shea, G., and Schubert, S. Why Should We Integrate Services, Servers, and Networking in a Data Center? In *WREN* (2009).

[12] Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., and Stoica, I. Towards a Common API for Structured Peer-to-Peer Overlays. In *IPTPS* (2003).

[13] Dally, W. J., and Seitz, C. L. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE ToC 36*, 5 (1987).

[14] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI* (2004).

[15] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP* (2007).

[16] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. Routebricks: Exploiting parallelism to scale software routers. In *SOSP* (2009).

[17] Duato, J. A Theory of Fault-Tolerant Routing in Wormhole Networks. *IEEE TPDS 8*, 8 (1997).

[18] Glass, C. J., and Ni, L. M. The turn model for adaptive routing. *SIGARCH Comput. Archit. News 20*, 2 (1992).

[19] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[20] Gummadi, K. P., Gummadi, R., Gribble, S. D., Ratnasamy, S., Shenker, S., and Stoica, I. The impact of DHT routing geometry on resilience and proximity. In *SIGCOMM* (2003).

[21] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM* (2009).

[22] Guo, C., Wu, H., Tan, K., Shiy, L., Zhang, Y., and Luz, S. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *SIGCOMM* (2008).

[23] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).

[24] Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. *OSR 44*, 2 (2010).

[25] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. The Click modular router. In *SOSP* (1999).

[26] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcached at Facebook. In *NSDI* (2013).

[27] Nordbotten, N. A., Flich, J., Skeie, T., Gomez, M. E., Lopez, P., Robles, A., Duato, J., and Lysne, O. A routing methodology for achieving fault tolerance in direct networks. *IEEE ToC 55*, 4 (2006).

[28] Oran, D. OSI IS-IS Intra-domain Routing Protocol. IETF RFC 1142.

[29] Parhami, B. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Publishers, 1999.

[30] Puente, V., and Gregorio, J. A. Immucube: Scalable Fault-Tolerant Routing for k-ary n-cube Networks. *IEEE TPDS 18*, 6 (2007).

[31] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. How Hard Can It Be? Designing and implementing a deployable Multipath TCP. In *NSDI* (2012).

[32] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI* (2004).

[33] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A Scalable Content-addressable Network. In *Proceedings of SIGCOMM* (2001).

[34] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware* (2001).

[35] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP* (2001).

[36] Scott, S. L., and Thorson, G. Optimized Routing in the Cray T3D. In *PCRCW* (1994).

[37] Shin, J.-Y., Wong, B., and Sirer, E. G. Small-world Datacenters. In *SOCC* (2011).

[38] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM* (2001).

[39] Valiant, L. G., and Brebner, G. J. Universal schemes for parallel communication. In *STOC* (1981).

[40] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Úlfar Erlingsson, Gunda, P. K., and Currey, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI* (2008).

[41] AMD Completes Acquisition of SeaMicro. `http://bit.ly/OuOaHm`.

[42] Apache Hadoop. `http://hadoop.apache.org/`.

[43] Google Distribution of Requests . `http://bit.ly/hUTaVQ`.

[44] HyperTransport Consortium. `http://www.hypertransport.org`.

[45] Intel acquires Cray Interconnect. `http://intel.ly/I8VIAR`.

[46] SeaMicro SM10000-XE. `http://bit.ly/KlIyAp`.

[47] SeaMicro Technology Overview. `http://bit.ly/MAwJ4S`.

[48] Storm. `http://storm-project.net`.

[49] Why Torus-Based Clusters? `http://bit.ly/MBuGOE`.