MDCube: A High Performance Network Structure for Modular Data Center Interconnection

Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, Yongguang Zhang
Microsoft Research Asia (MSRA), China
{hwu, lguohan, danil, chguo, ygz}@microsoft.com

ABSTRACT

Shipping-container-based data centers have been introduced as building blocks for constructing mega-data centers. However, it is a challenge on how to interconnect those containers together with reasonable cost and cabling complexity, due to the fact that a mega-data center can have hundreds or even thousands of containers and the aggregate bandwidth among containers can easily reach tera-bit per second. As a new inner-container server-centric network architecture, BCube [9] interconnects thousands of servers inside a container and provides high bandwidth support for typical traffic patterns. It naturally serves as a building block for mega-data center.

In this paper, we propose MDCube, a high performance interconnection structure to scale BCube-based containers to mega-data centers. MDCube uses the high-speed uplink interfaces of the commodity switches in BCube containers to build the inter-container structure, reducing the cabling complexity greatly. MDCube puts its inter- and inner-container routing intelligences solely into servers to handle load-balance and fault-tolerance, thus directly leverages commodity instead of high-end switches to scale. Through analysis, we prove that MDCube has low diameter and high capacity. Both simulations and experiments in our testbed demonstrate the fault-tolerance and high network capacity of MDCube.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network topology, Packet-switching networks

General Terms

Algorithms, Design

Keywords

Modular data center, Server-centric network, Commoditization

1. INTRODUCTION

Large data centers are built around the world to provide various online services. The deployment trend shows that the number of servers in data centers continues to grow. Companies like Amazon, Google, and Microsoft are building mega-data centers for cloud computing [13]. The recently proposed "shipping container" data center [13, 9, 11, 15, 16, 17] takes a modular approach, which is called Modularized Data Center (MDC). Servers, up to a few thousand, are interconnected and placed on multiple racks within a standard, say 40- or 20-feet, shipping container. Efforts are made to significantly reduce the costs of cooling, powering, and administration in a container.

The shipping container product is already available and deployed by companies including HP, Microsoft, and Sun [13]. Taken the structure shown in [13] as an example, future mega-data center may contain 400 containers, with each container houses 2500 servers. However, how to interconnect the building block containers to scale a data center from thousands to millions of servers is a challenging problem.

There are three challenges for inter-container interconnection. *First, high inter-container bandwidth requirement.* A data center needs to support many high bandwidth services, such as distributed file system [7, 4, 5] and distributed execution engine [6, 12]. They typically require high communication capacity among servers across different containers. The number of containers of an MDC may vary from several to hundreds. A scalable design should provide high network capacity in various settings. *Second, the cost of the interconnection structure.* To scale, existing approaches either scale up the intermediate networking devices, e.g., high end routers, with more capacity, or scale out the intermediate devices to a large number, e.g., in fat-tree [1] and VL2 [8]. However, either capacity is poor or enormous switch cost is introduced to a mega-data center with hundreds of containers. *Third, cabling complexity.* When the number of containers scales to hundreds, the required long cables between containers become a practical barrier for MDC scalability. The inter-container cabling complexity should not restrict the scalability of a well planned MDC.

With these challenges, we discuss the design spaces as follows. For scalability and high capacity reason, we cannot use hierarchical designs such as the tree structure, and we need to make containers equal in certain sense. For cost issue, it is better that we do not depend on high end switches, and we want to only use commodity switches without any change. This directly leads to our server-centric design on whole mega-data center, which makes us choose BCube [9] as one of the examples for inner container. For cabling issue, we choose to use 10Gbps switch-switch link to reduce the number of cables, which naturally fits the link rate hierarchy of Ethernet switches.

As a new network architecture designed for shipping-container-based data centers, BCube [9] is proposed for intra-container server interconnection. BCube is a server-centric structure and provides high network capacity for various typical data center applications such as MapReduce[6] and Dryad[12], distributed file systems, and reliable data broadcasting/ multicasting. Due to the desired properties of BCube (high network capacity, graceful performance degradation, and directly built from low-end COTS (commodity off-the-shelf) switches, see [9]), it is a natural step to use BCube as the building block for mega-data centers.

In this paper, we present MDCube, a high-capacity interconnection structure for a ***M***odularized ***D***ata center ***Cube*** network. Recent COTS switches provide dozens (e.g., 48) of Gigabit ports and several (e.g., 4) high-speed 10 Gigabit ports, while BCube only uses those Gigabit ones. Our idea for MDCube is to directly connect the high-speed ports of a switch in a container to those of the peer switches in another container. By treating each container as a virtual node and those switches inside a BCube network as interfaces for inter-container connection, all the container-based virtual nodes form a virtual cube network. MDCube therefore connects all the containers using fairly low cost optical fibers without extra high-end switches or routers. Moreover, MDCube ensures high-degree robustness because of the large number of paths between two containers. We note that it is possible to use switch-centric structures such as fat-tree as intra-container structure and still treat them as a virtual node in MDCube. But switch-centric structures need upgrading switches. In this paper, we only focus on the server-centric BCube structure, due to BCube's high performance compared with that of other server-centric structures such as DCell.

Our MDCube-based solution well addresses all the three challenges. First, MDCube offers high aggregate bandwidth between containers, which is achieved by the MDCube topology design and a novel hierarchical routing algorithm on top of the MDCube topology. Second, MDCube directly connects the high speed interfaces (ports) of switches in containers and thus the interconnection cost is only the optical fibers. The trade-off we pay is the slightly reduced all-to-all throughput of the whole mega-data center compared with the case that all servers are directly connected to a super crossbar. Third, the number of inter-container cables is half of the number of high-speed interfaces of switches, which is much smaller than the number of servers. Hence cabling complexity is an addressable issue. MDCube makes innovations in its direct interconnection structure, load balanced routing protocol, and efficient implementation.

We have designed and implemented an MDCube protocol suite. Due to the property of the MDCube structure, we are able to extend the fast packet forwarding engine in BCube to MDCube packet forwarding, which can decide the next hop of a packet by only one table lookup. The packet forward engine can be efficiently implemented in both software and hardware. We have built an MDCube testbed with 20 servers and 5 24-port Gigabit Ethernet switches. Experiments in our testbed demonstrated the efficiency of our implementation.

MDCube is a better structure for modular mega-data centers than recently proposed structures such as fat-tree [1] and DCell [10]. Compared to DCell, MDCube solves cabling issue, and provides much higher network capacity since it does not have performance bottlenecks; compared to fat-tree, MDCube can be directly built using COTS switches without any switch upgrade. See Section 7 for detailed comparisons.

The rest of the paper is organized as follows. Section 2 discusses research background and the design of MDCube. Section 3 presents MDCube and its properties. Section 4 designs the load balance and fault tolerant routing. Sections 5 and 6 present simulations and experiments, respectively. Section 7 discusses related work. Finally, Section 8 concludes the paper.

2. BACKGROUND AND MDCUBE DESIGN

The design of the MDC network architecture is driven by application needs, technology trends, as well as MDC deployment constraints and operational requirements. Instead of building raised floor rooms, mounting server racks and open room cooling systems, shipping-container-based building blocks [13, 9] with integrated computing, power and cooling systems provide an appealing alternative way to build mega-data centers.

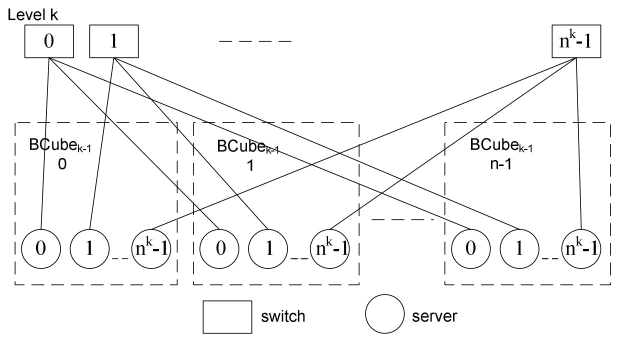
BCube [9] is a novel network structure for inner-container server interconnection. By installing a small number of network ports (e.g., 2, which is already a standard profile for existing servers in data centers) at each server, using COTS switches as dummy crossbars, and putting routing intelligence at the server side, BCube forms a server-centric network architecture. BCube provides high network capacity and smooth performance degradation when failures increase. Yet, designing network architecture for MDCs with BCube containers as building blocks is still challenging. The bandwidth between containers are highly dynamic and bursty [8], and many data center applications are bandwidth demanding. The design should be scalable as the number of containers starts from several and increases to hundreds or thousands and should only use commodity switches for cost saving. In what follows, we first briefly introduce the BCube structure and its properties, and then we overview the BCube-based MDCube design.

2.1 BCube Structure

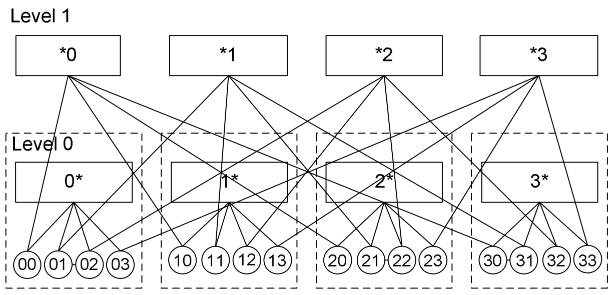
There are two types of devices in BCube: Servers with multiple ports and switches that connect a constant number of servers. BCube is a recursively defined structure. A BCube$_0$ is simply that n servers connect to an n-port switch. A BCube$_k$ ($k \geq 1$)) is constructed from n BCube$_{k-1}$s and n^k n-port switches. Each server in a BCube$_k$ has $k+1$ ports, which are numbered from level-0 to level-k. It is easy to see that a BCube$_k$ has n^{k+1} servers and $(k+1)$ levels of switches, with n^k n-port switches at each level.

The construction of a BCube$_k$ is as follows. We number the n BCube$_{k-1}$s from 0 to $n-1$ and the servers in each BCube$_{k-1}$ from 0 to n^k-1. We then connect the level-k port of the i-th server ($i \in [0, n^k-1]$) in the j-th BCube$_{k-1}$ ($j \in [0, n-1]$) to the j-th port of the i-th level-k switch, as we show in Figure 1(a).

We show a BCube$_1$ with $n = 4$, which is constructed from four BCube$_0$s and four 4-port switches, in Figure 1(b). We denote a server in a BCube$_k$ using an array $a_k a_{k-1} \cdots a_0$



(a)



(b)

Figure 1: (a) BCube is a leveled structure. A BCube$_k$ is constructed from n BCube$_{k-1}$ and n^k n-port switches. (b) A BCube$_1$ with $n = 4$. In this BCube$_1$ network, each server has two ports.

($a_i \in [0, n-1], i \in [0, k]$). Equivalently, we can use a *BCube server ID* bsID=$\sum_{i=0}^{k} a_i n^i$ to denote a server. We denote a switch using the an array $< l, s_k s_{k-1} \cdots\ s_{l+1} *\ s_{l-1} \cdots\ s_0 >$ $s_k s_{k-1} \cdots\ s_{l+1} *\ s_{l-1} \cdots s_0$ ($s_j \in [0, n-1], j \in [0, k], s_l = *$), where $l (0 \le l \le k)$ is the level of the switch. Similarly, we can use a *BCube switch ID* bwID=$\sum_{i=0}^{l-1} s_i n^i + \sum_{i=l+1}^{k} s_i n^{i-1}$ $+ln^k$ to denote a switch. Figure 1 shows that the construction guarantees that the i-th ($i \in [0, n-1]$) port of a switch $s_k s_{k-1} \cdots\ s_{l+1} * s_{l-1} \cdots s_0$ connects to the level-l port of server $s_k\ s_{k-1} \cdots s_{l+1} i s_{l-1} \cdots s_0$.

As described in [9], BCube has two proved properties as follows.

- The diameter, which is the longest shortest path among all the server pairs, of a BCube$_k$, is $2(k + 1)$.
- There are $k + 1$ edge-disjoint parallel paths between any two servers in a BCube$_k$.

Note that the path length is twice of that in [9], because one server-server hop (through a switch) is counted as two server-switch hops in this paper. In practice, k is a small integer, typically 1 and at most 3. BCube therefore is a low-diameter network.

2.2 Properties for BCube Switches

In this Subsection, we provide several additional properties of BCubes. These properties will be used to analyze the properties of MDCube and help us design the MDCube inter-container routing algorithm.

1. The longest shortest path length between two switches in a BCube$_k$ is $2(k + 1)$.
2. The longest shortest path length between a server and a switch in a BCube$_k$ is $2k + 1$.
3. There are n edge-disjoint parallel paths between any two switches in a BCube$_k$.
4. There are $k + 1$ edge-disjoint parallel paths between a switch and a server in a BCube$_k$ network (suppose $k + 1 < n$).

We prove the first property here and move the proofs of third and fourth properties to Appendix A and B. The proof of the second property is similar to that of the first one and thus skipped. Note that the edge-disjoint parallel paths in BCube between a switch and another switch (or a server) are useful to construct parallel paths crossing containers in an MDCube.

PROOF. Suppose the switches w_0 and w_1 are numbered as $a_k a_{k-1} \cdots\ a_{l+1} * a_{l-1} \cdots a_0$ at level l and $b_k b_{k-1} \cdots\ b_{t+1} *$ $b_{t-1} \cdots b_0$ at level t. By definition w_0 connects to a server $n_0 = a_k a_{k-1} \cdots a_{l+1} x a_{l-1} \cdots a_0$ and w_1 connects to a server $n_1 = b_k b_{k-1} \cdots b_{t+1} y b_{t-1} \cdots b_0$, $x, y \in [0, n-1]$. From the properties in Section 2.1, we know that there are shortest pathes of length at most $2(k + 1)$ between the two servers. Hence the path length between the two switches are at most $2(k + 1) + 2$. However, when selecting n_0, we can choose $x = b_l$ so that the shortest path from w_0 to w_1 is shorten by 2 hops. Similarly we can also choose $y = a_t$, but the path length is shorten only when $l \neq t$. Thus, we have longest shortest path length between switches at $2(k + 1)$. $\square$

2.3 MDCube Design Overview

BCube builds high capacity network structure using only COTS switches and commodity servers with multiple ports for better performance-to-price ratio [2]. However, BCube cannot directly scale out to millions of servers by adding more ports to servers and deploying more COTS switches. The fundamental barrier is that the number of non-local (inter-container) cables required by BCube increases linearly with the total number of servers. Note that similar problems exist for other high capacity structures like DCell and fat-tree. See Section 7 for detailed discussion.

How to scale to hundreds of containers while maintaining high interconnection capacity with reasonable cost is a big challenge. A conventional approach is to use high end switches to connect those high speed interfaces, but the high over-subscription ratio is not satisfactory for mega-data center. A recent approach VL2 [8] is to use multiple high end switches to build a Clos network as interconnection structure. Although it could support 10k servers with 148 148-port 10Gbps switches, the cost of such structure is unaffordably high for a mega-data center. For example, considering directly using VL2 to connect 1 million servers (each with 1Gbps link), then it needs 448 448-port 10Gbps aggregate and intermediate switches.

Observing the link speed hierarchy in existing network devices, i.e., 1Gbps for server-switch links and 10Gbps for switch-switch links, we are motivated to reduce the number of cables by using those high speed interfaces on commodity off-the-shelf switches. MDCube chooses to directly interconnect those 10Gbps links. The cables that connect switches in different containers are optical fibers since the range of copper fibers is not enough for space required by hundreds of containers. We expect our design follows the technique improvements well, e.g., when 10Gbps is used for server-switch links, 40Gbps or 100Gbps will be provided for switch-switch interfaces.

Besides the cabling complexity and cost challenges, there are several other technical challenges that MDCube is facing. First, the structure of MDCube. How to directly interconnect those high speed interfaces, while ensure the capacity still increases linearly with the number of servers. Second, the routing of MDCube. How to choose the path and how to determine the bandwidth ratio of those high speed switch-switch links over those server-switch links. Third, the flexibility of MDCube. Both small and large number of containers should be supported. And the requirement of incremental deployment is also possible. We will address these challenges in the next Section.

In addition to the above challenges, there are three technical details we need to address: the first is load balance for better performance; the second is fault tolerance as faults are inevitable for such large number of servers and switches; and the third is the routing to external networks. We will address these issues in Section 4.

3. MDCUBE

3.1 MDCube Construction

MDCube is designed to interconnect multiple BCube containers by using the high-speed (10Gb/s) interfaces of switches in BCube. To support hundreds of containers in a mega-data center, we assume optical fibers are used for these high-speed links. Each switch contributes its high-speed interfaces as a virtual interface for its BCube container. The virtual interface can be achieved by port trunking, e.g., four 10Gbps high-speed interfaces of a 48 port Gigabit switch can be bundled into a virtual interface at 40Gbps. Thus, if each BCube container is treated as a virtual node, it will have multiple virtual interfaces (the exact number is the number of switches). For each switch, it only observes its directly connected servers and its directly connected peer switch as well as servers connecting to that peer. Therefore, MDCube is server centric, and switches are dumb crossbars.

We can use the virtual interfaces of the BCube containers to form a basic complete graph. Suppose the number of containers to be connected is M, and there is a direct link between any two containers, then each container needs $M-1$ virtual interfaces. To be scalable when the number of containers grows, we extend the virtual complete graph topology to a generalized cube by introducing dimensions. The switches of a BCube container are divided into groups, serving as interfaces connecting to different dimensions. A container is identified by an ID that is mapped to a multi-dimensional tuple. Each container connects to other neighbor containers with different tuple on one dimension.

We build a $(D+1)$ dimensional MDCube as follows. We have $M = \prod_{d=0}^{D} m_d$, where m_d is the number of containers on dimension d. A container is identified by a $(D+1)$-tuple cID$=c_D c_{D-1} \cdots c_0 (c_d \in [0, m_d - 1], d \in [0, D])$. Each container houses $\sum_{d=0}^{D}(m_d - 1)$ switches, as $(m_d - 1)$ is the number of switches on dimension d. In MDCube, each switch is identified by its container ID and its switch ID in its BCube container: $\{cid, bwid\}$, $cid \in [0, M-1]$, $bwid \in [0, \sum_{d=0}^{D}(m_d - 1) - 1]$. Each switch contributes one trunked interface for MDCube's interconnection. There are two types of links in MDCube: one is the normal link by which switch connects to servers, and the other is the inter-container high-speed link between switches in different containers.

```
/* $D+1$ is the dimension size of MDCube;
   $m_d$ is the number of containers on dimension $d$;
*/
**BuildMDCube**$(D, m_D, m_{D-1}, \cdots, m_0)$:
  for(int $d = D; d \geq 0; d--$)
    for(int $i = 0; i < m_d - 2; i++$)
      for(int $j = i+1; j < m_d - 1; j++$)
        $\text{cID}_1 = c_D \cdots c_{d+1} i c_{d-1} \cdots c_0$;
        $\text{cID}_2 = c_D \cdots c_{d+1} j c_{d-1} \cdots c_0$;
        $bwid_1 = j - 1 + \sum_{x=0}^{d-1}(m_x - 1)\}$;
        $bwid_2 = i + \sum_{x=0}^{d-1}(m_x - 1)\}$;
        connect $\{\text{cID}_1, bwid_1\}$ and $\{\text{cID}_2, bwid_2\}$;
  return;
```

Figure 2: The procedure to build an MDCube from BCubes

The construction of MDCube is shown in Figure 2. There is a link on dimension d between two containers that have different identity tuple only on dimension d. To illustrate the MDCube's construction, as an example we show a 2-d MDCube networks built from 9 BCube$_1$ containers with $n = 2, k = 1$ in Figure 3. Another 1-d MDCube built from 5 BCube$_1$ containers is shown in Figure 8, which is the topology used in our experimental testbed. Note there are superficial similarities between MDCube and generalized Hypercube [3] at container level. MDCube is a hierarchical structure built from BCube, which is a server-centric structure. This results in both different structural properties and routing strategies.

Although the properties of MDCube shown later work for any integer D, this paper focuses on 1-d MDCube (virtual mesh) and 2-d MDCube (virtual generalized hypercube) cases, since the number of servers supported in 2-d is already over 1 million. Take MDCube constructed by BCube$_1$ with 48 port switches $(n = 48, k = 1)$ as an example: each BCube container houses $n^{k+1} = 48^2 = 2304$ servers and $n(k+1) = 48 \times 2 = 96$ switches, so MDCube supports up to $n(k+1)+1 = 97$ containers (0.22M servers) on 1-d, and up to $(\frac{n(k+1)}{2}+1)^2 = 49^2 = 2401$ containers (5.5M servers) on 2-d.

3.2 Single-path Routing in MDCube

By exploring the multi-dimensional structure of MDCube at container level, we design a single-path routing algorithm for MDCube as shown in Figure 4. The procedure is to correct the tuples of the container ID one by one to reach the destination container, and the order of such correction is controlled by a permutation Π_D. Function GetLink returns the switch pairs that directly connect the two containers using the high-speed link defined in Figure 2. For the routing

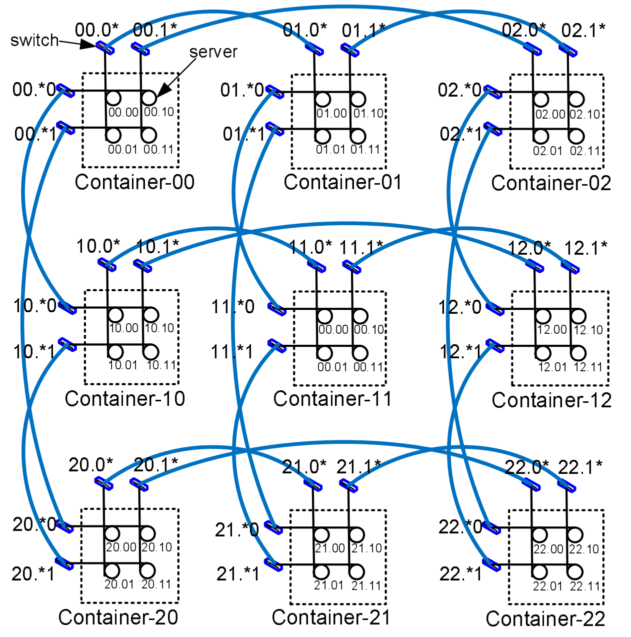


Figure 3: A 2-D MDCube is constructed from 9=3*3 BCube$_1$ Containers with n=2, k=1.

between two servers or switches in the same container, we call BCubeRouting. Note that [9] only defines the route between servers, here we define route for switches as a path through any of its directly connected servers. The procedure shown in Figure 4 can be easily implemented at the server side. Hence the routing can be carried out without the involvement of switches.

Theorem 1. *For a $(D+1)$-dimensional MDCube constructed from BCube$_k$, the path length in MDCubeRouting between any two servers is at most $h = 4k+3+D(2k+3)$.*

Proof. The procedure taken in MDCubeRouting is to correct the container ID $(D+1)$tuple one by one, so that the number of intermediate container between source and destination container is at most D. For each intermediate container, the path is from an entry switch to an exit switch. From the first property of BCube switches in Section 2.2, the total hops for intermediate container is thus $2D(k+1) + (D+1)$. The adding of $(D+1)$ is to count the hops on high-speed links when containers are concatenated. Then we add hops in source and destination container from the second property, and the result is $4k+3+D(2k+3)$. $\square$

Note that here the path length is increased by 2 (or 3) whenever a server is traversed for inner-container crossing one switch (or inter-container crossing two switches). Theorem 1 gives an upper bound for the maximum path length among all the shortest-paths. Therefore, the diameter of an MDCube network is also at most $4k+3+D(2k+3)$.

Considering that MDCube is at most 2-dimensional (D=1) in practice, the longest shortest path length in MDCube is at most $6k+6$. Since k is 1 for BCube$_1$, the longest shortest path in any 2-dimensional MDCube constructed by BCube$_1$ is at most 12.

```
/* Both src and dst are denoted as {cid, bsid};
   $\Pi_x = [\pi_x, \pi_{x-1}, \cdots, \pi_0]$ is a permutation of of $[0, 1, \cdots, x]$
*/
MDCubeRouting(src, dst, $\Pi_D$):
   $c1$=src; $c2$=$c1$; path=();
   for($i = D; i \geq 0; i--$)
      if(the $\pi_i$-th entry of $c2$.cid and dst.cid are different)
         change $\pi_i$-th entry of $c2$.cid to that of dst.cid
         $(sw_1, sw_2)$= GetLink($c1$.cid, $c2$.cid);/*switches pair*/
         $path1$=BCubeRouting($c1$, $sw_1$);
         $path = path + path1$+$(sw_1, sw_2)$;
         $c1$=$sw_2$; $c2$=$c1$;
   if ($c2$.cid == dst.cid) /*in the same BCube*/
      $path1$= BCubeRouting($c2$, dst);
      $path = path + path1$;
   return path;
```

Figure 4: MDCubeRouting to find a path from src to dst. It corrects one digit at one step at container level by a predefined permutation Π_D.

Theorem 2. *There are $(k+1)$ parallel paths between any two servers in an MDCube built from BCube$_k$s (by assuming the high-speed links between containers provide higher bandwidth than $(k+1)$ normal speed link).*

The proof of Theorem 2 is from the third property of BCube switches in Section 2.2 for intermediate containers and the fourth property for source and destination container. Since the high-speed links usually provide bandwidth that is much higher than normal links, we can treat it as a fat pipe, e.g., 40Gbps high-speed link can be treated as 40 1Gbps normal links. The BCube$_k$ usually has k at most 3, so k is much smaller than the number of links in pipe. Note that here the paths are not fully edge disjointed because a high-speed link is treated as multiple small links. In Section 4.1 we will discuss how to provide fully edge disjoint parallel paths using detour routing for load balance.

3.3 Traffic Distribution in MDCube

We elaborate the traffic distribution on MDCube under the all-to-all communication model, which is used in MapReduce [6] alike applications in MDC. We have theorem as follows. Its proof is given in Appendix C.

Theorem 3. *Consider an all-to-all communication model for a $(D+1)$-dimensional MDCube Built from BCube$_k$s, where any two servers have one flow between them. When MDCubeRouting is used, the number of flows carried on a BCube normal link is around $(2k+1+D(\frac{k^2}{k+1}+1))\frac{(N-1)}{k+1}$, and the number of flows carried on a high-speed link on dimension i is $\frac{tN}{m_i}$, where t and N are the numbers of servers in a BCube and an MDCube, respectively.*

The proof of Theorem 3 shows that the requirement of normal links and high-speed links are different for the all-to-all communication traffic pattern. The bottleneck link is determined by the ratio of capacity of high-speed links over that of normal links. Take the number of flows we obtained, we can estimate the ratio required as $r = \frac{t(k+1)}{m_i(2k+1+D(\frac{k^2}{k+1}+1))}$.

As an example, we estimate the ratio of an MDCube built from BCube$_k$ with k=1, then the ratio is r=$\frac{n}{3}$ for 1-d MDCube, and r=$\frac{n}{2.25}$ for 2-d square MDCube. Let's consider BCube$_1$ built from 48-port switches, i.e., n=48, then the required high-speed link capacity on all-to-all communication pattern is 16Gbps for 1-d MDCube, and 21.3Gbps for 2-d MDCube. As we know, by port trunking, we provide 40Gbps high-speed link using four 10Gbps high-speed links, so that the high-speed links are not bottleneck for both 1-d and 2-d MDCube. Note that commodity 48-port Gigabit switches with 4 10Gbps high-speed ports are available.

To evaluate the capacity of MDCube, we use the metric **ABT** (Aggregate Bottleneck Throughput), defined in [9] as the throughput of the bottleneck flow times the total number of flows in the all-to-all traffic model, implemented in MapReduce like applications in data center. Large ABT means shorter all-to-all job finish time. The ABT of MDCube is constrained by normal links and equals to $N(\frac{2k+1}{k+1} + D\frac{k^2+k+1}{(k+1)^2})^{-1}$. It's $\frac{N}{1.5+0.75D}$ when k=1.

3.4 Incomplete MDCube

In a complete MDCube, all the high-speed interfaces of switches are used as in Figure 2. However, a user may deploy containers with the number ranging from several to hundreds. In this paper, we focus on three related issues: 1) how to provide high throughput between containers when the number of switches is much larger than the container number; 2) how to deal with factoring of M (the number of containers in an MDCube); 3) incremental deployment of MDCube.

The interconnection described in Figure 2 works for the case when the number of switches g in BCube is larger than the number of switches $G = \sum_{d=0}^{D}(m_d - 1)$ used in interconnection. However, it may result in low performance since the interfaces on the rest switches are not used. Take the MDCube in Figure 3 as an example, if there are only three containers, say container-00, container-01 and container-02, then it's a 1-d MDCube. There is only 2 paths between two containers yet each container has two idle interfaces.

Our solution is to duplicate the cabling $\lceil g/G \rceil$ times so that the aggregate bandwidth between containers are maintained compared to a complete MDCube. The procedure in the last step of function BuildMDCube that connects $bwid_1$ to $bwid_2$, is changed to connecting $(bwid_1 + iG)$ to $(bwid_2 + iG)$, $0 \leq i < \lceil g/G \rceil$. In this way, the rich connectivity of MDCube is maintained as the case $g = G$. Moreover, the function GetLink in Figure 4 should return one of the link from $\lceil g/G \rceil$ candidates.

Our MDCube construction method provides many ways to organize M containers. For example, when $M = 64$, we can arrange the containers into 1-d (1×64) or 2-d (8×8, or 2×32, etc). So there is a problem on how to decide the number of dimensions and choose the value of m_i for each dimension i.

We have two principles on how to choose the factoring of M. First, 1-d MDCube is preferred over 2-d if the number of switches is enough because 1-d has higher ABT. Second, when 2-d is used, factoring of $m_i = \sqrt{M}$ is preferred, with the least number of switches required so that duplication is maximized. Different m_i is possible but not recommended.

MDCube provides the flexibility on incremental deployment. When the number of containers is small, a 1-d MDCube is flexible to add/remove containers. For a 2-d MDCube with hundreds of containers, we believe that for a well planned mega-data center, it is more likely that several columns or rows of containers are added into a mega-data center instead of a single container. Note that rewiring does happen, but only affects the wires to those added/removed containers.

4. LOAD BALANCED AND FAULT TOLERANT ROUTING

MDCubeRouting described in Section 3.2 explores the hierarchical and dimensional properties of MDCube. It works well for the balanced all-to-all communication pattern. However, it may use bandwidth inefficiently when facing bursty traffic patterns, since it tends to only choose the shortest path at container level. Take 1-d MDCube (virtual mesh) as example, suppose we backup all files from container A to container B, since the direct link $A \rightarrow B$ is always selected, it may result in bottleneck even if other longer paths like $A \rightarrow C \rightarrow B$ through detour routing are idle. Moreover, MDCubeRouting is not fault tolerant when the selected inter-container link breaks. In MDCube, we take a holistic approach to design a load-balancing and fault tolerant routing algorithm. We divide the routing into inter- and inner-container parts. We use detour routing to achieve high throughput by balancing the load among containers, and the fault tolerant routing by handling inter- and inner-container routing failures at MDCube and BCube respectively.

4.1 Detour Routing for Load Balance

To leverage the rich connectivity between containers in MDCube, we design a detour routing algorithm shown in Figure 5. The idea of our detour algorithm is to initiate the routing by a random, container-level jump to a neighboring container, then perform MDCubeRouting by correcting the first random jump at the final step.

Taking a 1-d MDCube as an example, our detour routing algorithm is similar to Valiant Load Balance (VLB). However, in VLB a random node is selected for detour routing, while in our algorithm, a random container is selected. Our choice is based on MDCube topology and the goal to balance the load between containers: if simply a random node is selected, then the path from the source to it and from it to the destination may be unnecessarily overlapped at that container, wasting bandwidth. Similar arguments hold for multi-dimensional MDCube.

Another difference of our detour routing compared with VLB is that we only choose a random jump to a neighboring container (at one dimension) instead of choosing a fully random container (at all dimensions). This is because the paths obtained from our random jump are parallel (not overlapped) on intermediate containers. First consider a 1-d MDCube: it's a complete graph, and each path takes at most one intermediate container so that a randomly selected container results in parallel detour paths. Second consider a 2-d MDCube: a random jump on dimension $i (i \in [0, 1])$ changes the i-th entry of container ID, and it maintains that value on later correction except the final step to destination. Thus, paths started from different jumps are parallel. For multiple dimensional MDCube, the claim still holds but it needs a special $\Pi_{D\setminus i}$, e.g., a sawtooth sequence (ceiled at D) one like [i+1, i+2, $\cdots$, D-1,D, 0,1,$\cdots$, i]. The above property leads to the following Theorem.

THEOREM 4. *For a $(D + 1)$ dimensional MDCube constructed by M BCube Containers, and the i-th dimension houses m_i containers so that $M = \prod_{i=0}^{D} m_i$, in total there are*

```
/*  Both src and dst are denoted as {cid, bsid};
    $\Pi_{x\setminus i} = [\pi_{x-1}, \pi_{x-2}, \cdots, \pi_0, i]$ is a special permutation of
      $[0, 1, \cdots, x]$, where $i$ is pushed to the last one
*/
MDCubeDetourRouting(src, dst)
  inode=src; path1=();
  $i$=random(0,$D$);/*a random value in [0,$D$]*/;
  do
    $j$=random(0,$m_i$-1);
    /*$m_i$ is the number of containers in dimension $i$*/;
  until ($j \neq \pi_i$-th entry of src.cid) /*not src container itself*/
  change $\pi_i$-th entry of inode.cid to $j$;
  $(sw_1, sw_2)$= GetLink(src.cid, inode.cid);
  $path1$=BCubeRouting(src, $sw_1$)+$(sw_1, sw_2)$;
  path=path1+MDCubeRouting($sw_2$, dst, $\Pi_{D\setminus i}$);
  return path;
```

Figure 5: MDCubeDetourRouting to find a detour path from src to dst. A neighbor is randomly selected for the first step. MDCubeRouting is called later for the rest hops, and the first random jump is correct at the final step.

$\sum_{i=0}^{D}(m_i - 1)$ *intermediate container non-overlapped paths between any source destination server pairs in different containers.*

4.2 Traffic Distribution with Detour Routing

We are interested in the traffic distribution on MDCube when the servers of two containers have an all-to-all communication model, e.g., running MapReduce. This in practice makes senses since the number of servers of one container may be not enough while that of the whole MDC may be too large for a task. We have the following Theorem, and its proof is in Appendix D.

THEOREM 5. *Consider an all-to-all communication model for all servers of two containers in a $(D+1)$-dimensional MDCube built by BCube$_k$. When MDCubeDetourRouting is used, the number of flows carried on a BCube normal link is nearly $\frac{(3k+2)t}{k+1}$, and the number of flows carried on a high-speed link is $\frac{nt}{k+1}$, where t is the total number of servers in a BCube$_k$ in MDCube.*

From the proof of Theorem 5 we can estimate the ratio of capacity of high-speed link over that of normal speed link required for all-to-all traffic between two containers. We have ratio $r = \frac{n}{(3k+2)}$ and it translates to 9.6Gbps requirement on high-speed link for the case of k=1 and n=48. The ABT of two containers is constrained by normal links and thus it's $2t\frac{2k+2}{3k+2}$.

4.3 Fault Tolerant Routing

When either intermediate servers or switches are failed, the algorithm described in Figure 5 is difficult to implement since it's unlikely to guarantee that the failed device information is known immediately. Thus, directly extending BCube's source routing protocol to MDCube faces scalability challenge. However, we still want to control which intermediate containers are traversed to achieve multiple parallel paths to destination container for high aggregate throughput. The idea of our routing algorithm dealing with failures is similar to the inter- and intra-domain routing for the Internet, but we leverage the structure of MDCube so that the source node balances the load on candidate parallel paths at container level, and BCube handles failures in containers.

We implement a hierarchical routing approach to decouple the inter- and inner-container routing in MDCube: 1) the source node only chooses which containers to traverse implemented by a loosely controlled source routing protocol. 2) the path inside a container is maintained by BCube itself so that failed servers or switches are bypassed without notifying source node. However, for failures happened between containers that cannot be handled by BCube itself, e.g., the fiber link connecting two containers is broken, a route error message is generated and sent back to the source node to trigger rerouting at container level using other parallel paths. Note that from Theorem 4, if another random container is selected, then it guarantees that it will not overlap with the previous intermediate container.

After source node chooses its intermediate container, it stores the IDs of those containers into its MDCube header (extended from BCube header), which lies between Ethernet and IP header. Those container IDs are used when crossing BCube routing domain. Since BCube fully controls the routing inside a container, a connection that traverses multiple containers may experience packet reordering because an intermediate container may route the packet using different paths in BCube. In addition, there are many direct server pairs that are connected by the same inter-container link. Hence there exist many paths through this inter-container link. This may result in packet reordering.

To select a fixed path from those available ones, we tag the packet with a flow_id field at source node, and later path selection and caching is based on the flow_id so that a connection is always on the same path. The first packet with a new flow_id of a source/destination pair triggers a route selection in BCube, and the path selected is cached and indexed by source/destination plus flow_id. When failure happens and reroute in BCube is triggered, the new route in BCube replaces the old entry to keep path consistent. Note that alternatively we can let intermediate node inspect the 5-tuple (source/ destination IP and port, protocol), but we leave this flexibility to source node.

4.4 Routing to External Networks

We follow the approach proposed for BCube [9] to route the packets to or from the external networks. In [9], proxy servers that connect to the aggregate switches serve as gateways. The high-speed links of those aggregate switches are connected to routers to external networks.

In MDCube, we reserve multiple switches in each BCube with highest switch IDs and not used for interconnection of BCubes. Such reservation will not confuse inter-container MDCube routing since those switches do not appear in MDCube interconnection. We then use their high-speed links to connect routers to external networks. Servers under those switches are gateways to the other servers in the same BCube. The number of those switches and gateway servers are configurable based on the load generated by a container.

5. SIMULATION RESULTS

In this Section, we use simulations to evaluate the performance of MDCube. First, we study the ABT (aggregate bottleneck throughput) of MDCube with multiple containers. We have analyzed the ABT of two containers in Section 4. We now use simulation to study the ABT as the number

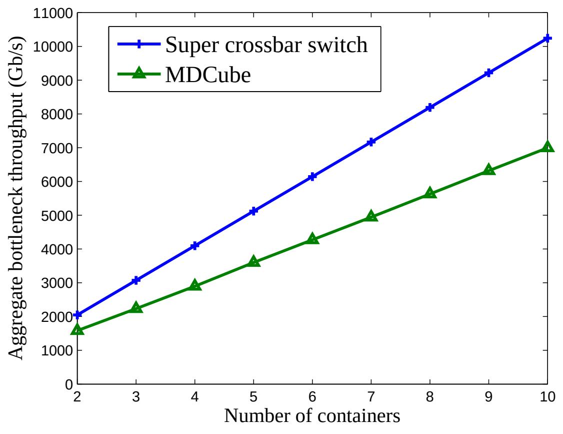


Figure 6: The ABT of MDCube with increased number of containers.

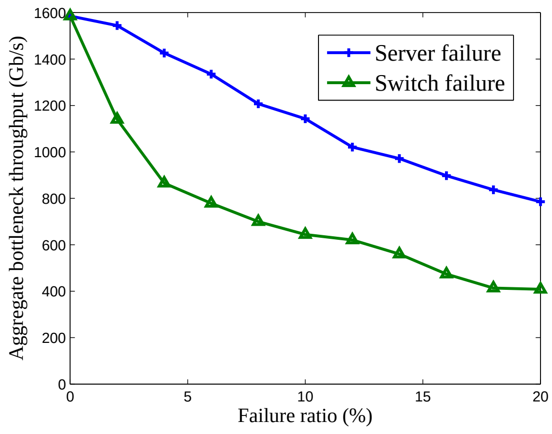


Figure 7: The ABT of two containers in MDCube under failures.

of containers increases. ABT generates an all-to-all traffic pattern. This is to emulate the reduce-phase of MapReduce: each reducer fetches data from all the mappers, resulting in an all-to-all traffic pattern. Second, we study the ABT of two containers under various server/switch failure ratios. In a large data center, both servers and switches are facing failures that can't be fixed immediately. We are interested in MDCube's performance to see whether our decoupled inter- and inner-container routing works well under high failure ratio.

In our simulation, we use a 2-d MDCube network built from BCube$_1$ with n=32, k=1. There are 33*33= 1089 containers, and each houses 32*32=1024 servers, so that the total number of servers is over 1.1 million. The normal link rate is 1Gbps for links between servers and switches, while the high-speed link rate is 10Gbps for links between switches.

5.1 ABT versus Number of Containers

We increase the number of containers to check ABT provided by MDCube in Figure 6. As a comparison, we also show the performance of a case where all servers are connected to a super crossbar switch, i.e., it always achieves 100% line rate of 1Gbps per server. Hence, the theoretical ABT of a super crossbar switch increases linearly as the number of servers increases, serving as a performance upper bound.

For MDCube, the achieved aggregate throughput is 1584 Gbps for 2 containers (2048 servers), so that the per server throughput is 0.77Gbps with interface line rate at 1Gbps. The 0.77Gbps is very close to the analysis result $\frac{2k+2}{3k+2}$=0.8 (k=1) in Section 4.2. The performance ratio of MDCube over 100% line rate bound drops slowly with increased number of containers: 0.68Gbps per server for 10 containers. Here we only give the ABT of multiple containers. We will compare other performance metrics and cost in Section 7.

5.2 ABT under Failures

In [9], the graceful performance degradation of BCube under both server and switch failures have been demonstrated, compared with that of fat-tree and DCell. In this simulation, we focus on the case of all-to-all traffic pattern with two containers in a 2-d MDCube, and evaluate it by randomly choosing servers or switches from the whole MDCube as the failed ones in Figure 7.

Graceful performance degradation states that when server or switch failure ratio increases, ABT decreases slowly and there are no dramatic performance falls. For server failures, either resulted from server crash or hardware failure, we find that the ABT degrades smoothly for reasonable failure ratio: for failure ratio of 2%, the ABT drops by 2.6%(from 1584 to 1543). ABT drops by half at a high failure ratio of 20%.

Switch failure has higher impact on the ABT than server failure, and similar phenomena are also observed in [9] for fat-tree, DCell and BCube networks. In MDCube, a failed switch not only breaks all inner-container links for servers connected to it, but also the inter-container link using it. Note that in our simulation, the maximum failure ratio of 20% rarely happens in a well managed data center, where an Ethernet switch usually has MTBF (Mean Time Between Failure) at years.

6. EXPERIMENTAL EVALUATION

6.1 Testbed

We have designed and implemented an MDCube stack as a kernel driver in the Windows server 2003 network stack. The MDCube stack implements the load balance and fault tolerant routing in Section 4 and a fast packet forward engine which is extended from BCube[9]. We have built a 1-d MDCube testbed using 20 Dell OptiPlex 755 and 5 24-port Cisco 3560 Gigabit Ethernet switches. Each server has one Intel 2.0GHz dualcore CPU, 2GB DRAM, and 160GB disk, and an Intel Pro/1000 PT quad-port Ethernet NIC. We turn off the xon/xoff Ethernet flow control, since it has interaction with TCP [10].

For MDCube, it has 5 BCube$_1$ (n=2) containers and each houses 4 servers. We use VLAN to divide a physical switch into multiple virtual 4-port Gigabit mini-switches. Each mini-switch connects 2 servers and the rest two ports are bundled together to form a high-speed inter-container link. Each server uses two ports of its quad-port NIC. Figure 8

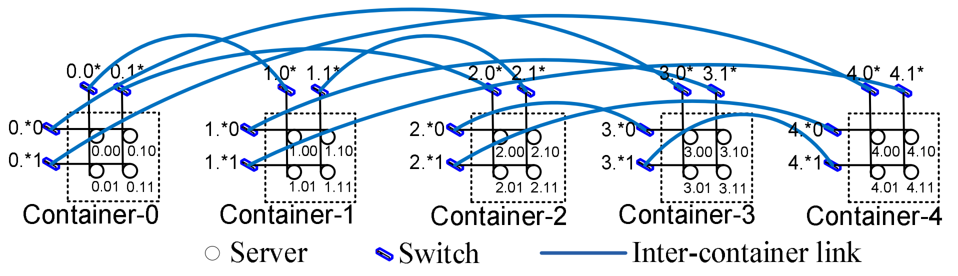


Figure 8: The 1-D MDCube testbed built from 5 BCube$_1$ containers with n=2, k=1.

illustrates the topology. We use a ***tree*** topology as a comparison. It connects the 20 servers in two layers: 4 servers in a container connect to a Gigabit leaf switch, and then a Gigabit root switch connects to those 5 leaf switches.

6.2 Bandwidth-intensive Application Support

To evaluate capacity of MDCube, we design four experiments which generate typical traffic patterns required by many bandwidth intensive applications. In all the experiments, we set MTU to 9KB and each server transmits 10GB data in total.

One-to-one. We set up two TCP connections *tcp*1 and *tcp*2 between servers 3.11 and 4.00. The two connections use random jump to two parallel paths, *path*1 {3.11, 1.00, 1.01, 4.10, 4.00} and *path*2 {3.11, 3.01, 0.10, 0.11, 4.00}, respectively. The total inter-server throughput is 1.92Gb/s. The throughput is 0.99Gb/s in the Tree, limited by one Gigabit NIC per-server.

C-to-C. In this experiment, we show that load balance can speedup data replication between two containers. Every server in Container 0 replicates 10GB data to its counterpart server (same server ID) in Container 1. Every server splits its 10GB data into 4 blocks and uses 4 TCP connections because every container has 4 inter-container links. We compare our approach with the tree structure and we achieve 2.1 times speedup. MDCube can achieve 4 times speedup in theory by assigning paths. Our result is sub-optimal since we use random jump which causes the inner-container links to be bottlenecks.

C-to-C (All-to-All). In this experiment, all the 8 servers in container 0 and 1 generate all-to-all traffic. Every server sends 10GB data to all other 7 servers. The per-server throughputs of MDCube and the tree are 460Mb/s and 300 Mb/s, respectively. Figure 9 plots the details for total throughput of MDCube and that of the tree. MDCube is about 1.5 times faster than the Tree. The initial high throughput of the Tree is due to the TCP connections among the servers under the same leaf switches, while later root switch becomes the bottleneck. There is no such bottleneck in MDCube.

All-to-all. In this experiment, each server establishes 19 TCP connections to all the other 19 servers. The per-server throughput values of MDCube and the Tree are 243Mb/s and 189Mb/s, respectively. MDCube is almost 1.3 times faster than the tree.

Figure 10 summarizes the per-server throughputs of the experiments. We note that while the first and fourth experiments are similar to that presented in [9], the second and third experiments are unique to MDCube and show the

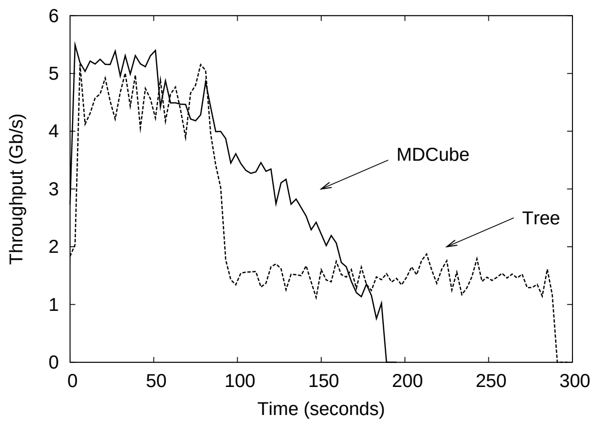


Figure 9: Total TCP throughput of MDCube and tree in the C-to-C(a2a) pattern of 2 containers.

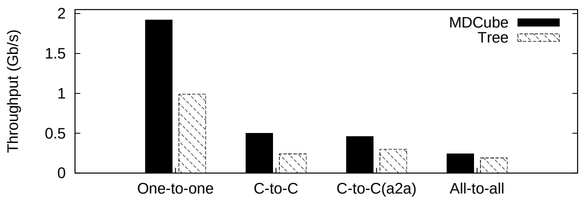


Figure 10: Per-server throughput under different traffic patterns.

high inter-container capacity of MDCube. Compared with the tree structure, MDCube achieves 1.3-2 times speedup for these traffic patterns. Note that as the scale of network increases, the gain of MDCube will be much more significant. The all-to-all capacity of MDCube increases linearly whereas the tree structure has severe bottleneck.

6.3 TCP Connection under Failures

We use experiment to illustrate the impact of failure on an existing TCP connection in Figure 11. For *tcp*1 in the one-to-one case, when we first unplug the inner-container link between server 1.00 and switch 1.*0, server 3.11 finds that server 1.00 is not available using its neighbor maintenance protocol (by 3 continuous hello messages lost in three seconds). Then server 3.11 changes *tcp*1 to path {3.11, 1.10,

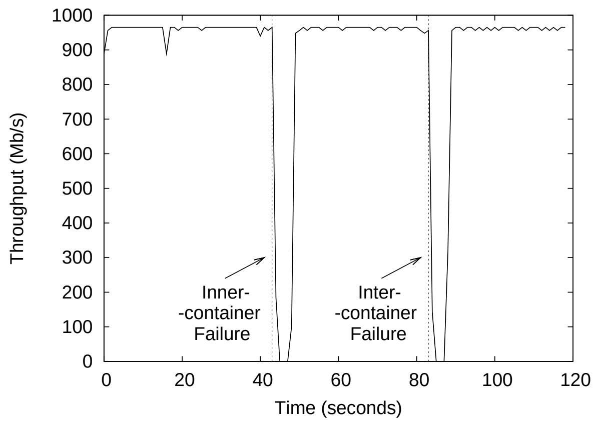


Figure 11: TCP connection experiences failures.

	fat-tree	DCell	BCube	MDCube
One-to-one	1	$k'+1$	$\log_n N$	$\log_n t$
ABT(MDC)	N	$\frac{N}{2^{k'}}$	$\frac{n(N-1)}{n-1}$	$\frac{N}{1.5+0.75D}$
Traffic balance	Yes	No	Yes	Yes
Graceful degradation	fair	good	good	good
Switch upgrade	Yes	No	No	No
Inner- cable NO.	$N\log_{\frac{n}{2}}\frac{t}{2}$	$N(\frac{k''}{2}+1)$	$N\log_n t$	$N\log_n t$
Inter- cable NO.	$N\log_{\frac{n}{2}}\frac{N}{t}$	$N(\frac{k'-k''}{2})$	$N\log_n\frac{N}{t}$	$\frac{N}{2n}\log_n t$
Switches NO.	$\frac{N}{n}\log_{\frac{n}{2}}\frac{N}{2}$	$\frac{N}{n}$	$\frac{N}{n}\log_n N$	$\frac{N}{n}\log_n t$

* n is port number of switches. t is the number of servers in a container, while N is the number of all . DCell has $k' = \log_2\log_n N$ and $k'' = \log_2\log_n t$. MDCube has $\log_n t = k+1$.

Table 1: Performance and cost comparison of MDCube and three other network structures.

1.11, 4.10, 4.00}. Then we unplug the inter-container link between container 1 and 4. Then, the source server notices the failure from destination unreachable message from server 1.10 and randomly select another container 2 to forward the traffic. The new path for *tcp*1 becomes {3.11, 3.10, 2.00, 2.01, 4.00}. In our current implementation, the unreachable message strictly follows the reverse direction of the forward path to ensure that the unreachable message can reach the source with high probability. The recovery times of both the inner-container and inter-container failure are dominated by the 3 seconds neighbor failure detection time. Note we set the period of hello message as 1 second to reduce the protocol overhead since we treat link broken as rare case. We believe further improvement can be made to detect failure faster in mega-data center networks.

7. RELATED WORK

Network interconnections have been studied for several decades. They can be roughly divided into two categories. One category is switch-centric and the other is server-centric. In the switch-centric approaches such as Clos network, Butterfly and fat-tree [1], servers connect to a switching fabric. Since the routing is implemented in those switches, existing COTS switches need upgrade. In the server-centric approaches such as mesh, torus, ring, Hypercube [14] and de Bruijn, servers usually connect to other servers directly, resulting in either limited capacity or large server port number. There are also hierarchical structures like CCC (cube-connected-cycle)[14] which trades network capacity for small server port number.

To interconnect thousands of containers, MDCube treats each container as a virtual node, and each switch in any container as a virtual interface, which makes it similar to generalized hypercube at container level. However, as routing decision is made by server instead of container, we decouple inter and inner containers routing and use random intermediate container to balance inter-container load. We consider MDCube at 1-d or 2-d to support millions of servers in a mega-data center.

DCell [10] is a multi-layer, server-centric structure that leverages COTS switches on its construction. The basic layer is a switch connected to n servers, where n is the port number of the switch. And a layer k DCell$_k$ is constructed by multiple DCell$_{k-1}$ treating each one as a virtual node with multiple interfaces. Each server in DCell has multiple ports and the total number of servers in DCell grows super exponentially. As we show in Table 1, DCell uses the least number of switches.

Fat-tree[1] is a switch-centric structure, motivated by reducing over-subscription ratio and removing single failure point. Since switches are concatenated, the effective port number for scaling out is half (except the root layer), which makes it uses the largest number of switches comparing with DCell, BCube, and MDCube.

We compare MDCube, fat-tree, DCell and BCube in detail in Table 1. For performance, we focus on throughput of one-to-one (server to server) and ABT for all-to-all communications. For one-to-one, fat-tree only achieves throughput 1 since each server has 1 port. DCell, BCube, and MDCube leverage multi-port and improve throughput linearly. The ABT of MDCube is lower than that of fat-tree and BCube, but still scales linearly as N increases. Take an MDCube built from BCube$_1$s as an example, the ABTs are $\frac{2N}{3}$ for D=0 and $\frac{4N}{9}$ for D=1, respectively. The ABT of MDCube of multiple containers is closer to that of fat-tree, e.g., $\frac{8}{5}t$ compared to $2t$ for 2 containers. Moreover, MDCube can balance traffic well and its ABT degrades gracefully as failure rate increase, as shown in Section 5.

For cost analysis, we focus on both cabling and switch costs. Since the basic building blocks are containers, the cabling cost is divided into two types: local (copper) cables for inner-container connections and remote (fiber) cables for inter-container connections. The cabling cost inside containers is similar for all structures, although fat-tree is the largest and DCell is the smallest. For inter-container cables, as the cost of inter-container cabling is dominate by the price of fiber ports (interface), we focus on the number instead of length of cables. The number of cables of MDCube is the smallest, since it reduces the cables by nearly $2n$ times. Taking MDCube built from 48-port switches as an example, $2n$=98 is two order of magnitudes.

To make a fair cost comparison of the four structures on switches, we choose to use the same n-port switches to connect N servers. We observe DCell uses the least number of switches, while fat-tree uses the largest. Take the MDCube constructed by BCube$_1$ (k=1, t=n^2) as an example, MDCube uses only twice number of switches than that in

DCell. Given its low switch and cabling costs, MDCube is more appealing for mega-data centers.

Note that VL2 [8] is a three-layer folded Clos network but needs large number high-end intermediate switches for a mega-data center, e.g., 448 448-port switches with 10G ports to support 1 million servers. It has fat-tree like performance while the number of cables is reduced by also using 10G link. The tradeoff made is the cost of those high-end switches.

8. CONCLUSIONS

In this paper, we have presented the design, implementation, and evaluation of a novel network architecture MDCube, to scale BCube-container-based data center to mega level. By directly connecting the high-speed interfaces of the COTS switches in the BCube containers, MDCube forms a virtual generalized hypercube at the container level. MDCube can interconnect hundreds or thousands BCube containers with its 1-d or 2-d structure and provide high network capacity for typical data center applications. Compared with mega-data center designs directly built from a single structure like BCube or fat-tree, the number of inter-container cables is reduced by nearly two orders of magnitudes.

MDCube is also fault-tolerant and load-balancing in nature due to its specific structure design and the routing protocol on top of its network topology. The MDCube routing is a hierarchical two-level approach, with BCube routing for inner container and a randomized algorithm for inter container routing. The routing algorithm well balances traffic load and provides high capacity for container-to-container communications. Our analysis, simulation, and implementation experiences demonstrate that MDCube is an attractive and practical modular design for mega-data centers, due to its high capacity for inner- and inter- container communications, its fault tolerance, and that it only uses low-end COTS switches and its manageable cabling complexity.

9. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. SIGCOMM*, 2008.

[2] L. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, March-April 2003.

[3] L. Bhuyan and D. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE trans. Computers*, April 1984.

[4] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/current/hdfs_design.pdf.

[5] CloudStore. Higher Performance Scalable Storage. http://kosmosfs.sourceforge.net/.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*, 2004.

[7] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *ACM SOSP'03*, 2003.

[8] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. SIGCOMM*, 2009.

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

[10] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault Tolerant Network Structure for Data Centers. In *Proc. SIGCOMM*, 2008.

[11] IBM. Scalable Modular Data Center. http://www-935.ibm.com/services/us/its/pdf/smdc-eb-sfe03001-usen-00-022708.pdf.

[12] M. Isard, M. Budiu, and Y. Yu. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *ACM EuroSys*, 2007.

[13] Randy H. Katz. Tech Titans Building Boom, Feb. 2009.

[14] F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays. Trees. Hypercubes.* Morgan Kaufmann, 1992.

[15] Rackable Systems. Rackable Systems ICE Cube^{TM} Modular Data Center. http://www.rackable.com/products/icecube.aspx.

[16] Verari Systems. The Verari FOREST Container Solution: The Answer to Consolidation. http://www.verari.com/forest_spec.asp.

[17] M. Waldrop. Data Center in a Box. *Scientific American*, July 2007.

APPENDIX

A. PROOF OF THE 3RD BCUBE PROPERTY

Proof. Suppose two switches w_0 and w_1 are numbered as $a_k a_{k-1} \cdots a_{l+1} {*} a_{l-1} \cdots a_0$ at level l and $b_k b_{k-1} \cdots b_{t+1} {*} b_{t-1} \cdots b_0$ at level t. If $l = t$, then we got n parallel paths between them because these two switches are connected to n sub BCube$_{k-1}$. Next without lossing generality we consider the case for $t > l$.

We construct a path, $path_i (i \in [0, n-1])$, from server $n_0^i = a_k a_{k-1} \cdots a_{l+1} i a_{l-1} \cdots a_0$ to server $n_1^i = b_k b_{k-1} \cdots b_{t+1} j b_{t-1} \cdots b_0$, where $j = i + a_t - b_l$. The bits correction of any path is from level k to 0, but we leave t and l as the last two levels. That is to say, $path_i$ is from n_0^i to $n_2^i = b_k b_{k-1} \cdots b_{t+1} a_t t_{t-1} \cdots b_{l+1} i b_{l-1} \cdots t_0$, then to n_1^i.

Here we distinguish two cases:

1) $i = b_l$, so that $j = a_t$: we have $n_2^i = n_1^i$, i.e., destination is reached so that path is $n_0^i \rightarrow n_2^i (n_1^i)$,

2) $i \neq b_l$: the penultimate server $n_3^i = b_k b_{k-1} \cdots b_{t+1} j t_{t-1} \cdots b_{l+1} i b_{l-1} \cdots b_0$ is routed, and path is $n_0^i \rightarrow n_2^i \rightarrow n_3^i \rightarrow n_1^i$. The parallel properties of these paths are easy to verify: for $path_i$, we maintain the l-th bit entry value as i until finally corrected at destination, for both intermediate servers and switches. □

B. PROOF OF THE 4TH BCUBE PROPERTY

We prove this property by construct such $k+1$ paths, shown in Figure 12.

C. PROOF OF THEOREM 3

Proof. For BCube$_k$, there are totally $t = n^{k+1}$ servers and $g = (k+1)n^k$ switches. For a $(D+1)$-dimensional MDCube, the number of containers connected is $M = \prod_{d=0}^{D} m_d$.

```
/* $sw = b_k b_{k-1} \cdots b_{l+1} * b_{l-1} \cdots b_0$; a switch at level $l$
   $svr = a_k a_{k-1} \cdots a_0$; a server at arbitrary location
*/
BuildServerSwitchPaths($sw, svr$):
  for ($i = k; i \geq 0; i--$)
    $j = (a_l + i - l)$ mod $n$; /*when $i = l$, we have $j = a_l$*/
    $src = b_k b_{k-1} \cdots b_{l+1} j b_{l-1} \cdots b_0$; /*servers connect to $sw$*/
    if ($a_i \neq b_i$)
      $\Pi = [i-1, \cdots, 0, k, \cdots, i+1, i]$;
      $path_i$ = BCubeRouting($src, svr, \Pi$);
    else
      if ($i == l$ AND $src == svr$)
        $path_i = ()$;/*when $svr$ connects to $sw$ on level $l$*/
      else
        $src2$ = a neighbor of $src$ at level $i$;
        $\Pi = [i-1, \cdots, 0, k, \cdots, i]$;
        $path_i = \{src,\}$;
        $path_i$ += BCubeRouting($src2, svr, \Pi$);
```

Figure 12: The algorithm to construct the $k+1$ paths between a server and a switch.

The total number of servers is $N = tM$. The total number of flows in MDCube is $N(N-1)$. The number of normal directional links is $2N(k+1)$, while the number of high-speed links is gM.

In a BCube$_k$, the averaged path length in hops between a server and switch is $h_1 = \frac{1}{n^{k+1}} \sum_{i=0}^{k} [(2i+1) n C_k^i \ (n-1)^i] = \frac{(2k+1)n^{k+1} - 2kn^k}{n^{k+1}} = 2k + 1 - \frac{2k}{n}$.

Similarly the averaged path length between two switches is $h_2 = \frac{2}{(k+1)n^k - 1} [\sum_{i=1}^{k} (i+1) C_k^i (n-1)^i + \sum_{i=0}^{k-1} (i+1) kn C_{k-1}^i (n-1)^i] = \frac{2k^2(n-1)n^{k-1}}{(k+1)n^k - 1} + 2 \approx \frac{2k^2}{k+1} \frac{n-1}{n} + 2$.

Similar to the proof procedure of Theorem1, the averaged number of normal link for a flow traversed is $2h_1 + D'h_2$, where D' is the averaged number of intermediate containers traversed. Note we ignore the case for inner-container flow given the number of containers is tens to hundreds, and have $D' \approx D$.

Since the links are symmetric, the number of flows carried on all the normal links should be identical. Then, the number of flows carried on a normal link is around $\frac{(2h_1 + Dh_2)N(N-1)}{2N(k+1)} = (2k + 1 - \frac{2k}{n} + D(\frac{k^2}{k+1} \frac{n-1}{n} + 1)) \frac{(N-1)}{k+1} \approx (2k + 1 + D(\frac{k^2}{k+1} + 1)) \frac{(N-1)}{k+1}$.

The number of flows carried on a high-speed link on dimension i is $t^2 \prod_{d=0}^{i-1} m_d \prod_{d=i+1}^{D} m_d = t^2 \frac{M}{m_i} = \frac{tN}{m_i}$ □

D. PROOF OF THEOREM 5

Proof. We use the same symbols in Appendix C. There are $2t^2$ flows crossing containers, and $2t(t-1)$ flows inside containers. The traffic through intermediate containers cross multiple switches, and the number of parallel paths n between switches are large enough to support it. Thus, we only need to consider the traffic on source and destination container.

From Theorem 1, the averaged number of hops (on normal links) for an inter-container flow traversed is h_1 for source and destination container. While the normal links a inner-container flow takes is $h_0 = 2(k+1)\frac{t(n-1)}{n(t-1)} \approx 2(k+1)\frac{n-1}{n}$[9]. Since the links are symmetric, the number of flows carried on all the normal links should be identical. Then, the averaged number of flows carried on a normal link is $\frac{2t^2 2h_1 + 2t(t-1)h_0}{2*2t(k+1)} \approx \frac{(3k+2)t}{k+1}$.

The number of high-speed links traversed by an inter-container flow is 2, one for source and one for destination container respectively, except those are directly connected. The high-speed links are also symmetric as used randomly. Thus, the number of flows carried on a high-speed link is at most $\frac{2t^2 * 2}{4g} = \frac{nt}{k+1}$. □