BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers*

Chuanxiong Guo[1], Guohan Lu[1], Dan Li[1], Haitao Wu[1], Xuan Zhang[1,2], Yunfeng Shi[1,3], Chen Tian[1,4], Yongguang Zhang[1], Songwu Lu[1,5]
1: Microsoft Research Asia, 2: Tsinghua, 3: PKU, 4: HUST, 5: UCLA
{chguo,lguohan,danil,hwu}@microsoft.com, xuan-zhang05@mails.tsinghua.edu.cn, shiyunfeng@pku.edu.cn, tianchen@mail.hust.edu.cn, ygz@microsoft.com, slu@cs.ucla.edu

ABSTRACT

This paper presents BCube, a new network architecture specifically designed for shipping-container based, modular data centers. At the core of the BCube architecture is its server-centric network structure, where servers with multiple network ports connect to multiple layers of COTS (commodity off-the-shelf) mini-switches. Servers act as not only end hosts, but also relay nodes for each other. BCube supports various bandwidth-intensive applications by speeding-up one-to-one, one-to-several, and one-to-all traffic patterns, and by providing high network capacity for all-to-all traffic.

BCube exhibits graceful performance degradation as the server and/or switch failure rate increases. This property is of special importance for shipping-container data centers, since once the container is sealed and operational, it becomes very difficult to repair or replace its components.

Our implementation experiences show that BCube can be seamlessly integrated with the TCP/IP protocol stack and BCube packet forwarding can be efficiently implemented in both hardware and software. Experiments in our testbed demonstrate that BCube is fault tolerant and load balancing and it significantly accelerates representative bandwidth-intensive applications.

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, Multi-path

*This work was performed when Xuan Zhang, Yunfeng Shi, and Chen Tian were interns and Songwu Lu was a visiting professor at Microsoft Research Asia.

1. INTRODUCTION

Shipping-container based, modular data center (MDC) offers a new way in which data centers are built and deployed [13, 16, 23, 24, 25]. In an MDC, up to a few thousands of servers are interconnected via switches to form the network infrastructure, say, a typical, two-level tree in the current practice. All the servers and switches are then packed into a standard 20- or 40-feet shipping-container. No longer tied to a fixed location, organizations can place the MDC anywhere they intend and then relocate as their requirements change. In addition to high degree of mobility, an MDC has other benefits compared with the data center facilities directly built from server racks. They include shorter deployment time, higher system and power density, and lower cooling and manufacturing cost.

In this work, we describe BCube, a high-performance and robust network architecture for an MDC. The design and implementation of BCube are driven by demands from data-intensive computing, recent technology trends, as well as special MDC requirements. Many data center applications require bandwidth-intensive, one-to-one, one-to-several (e.g., distributed file systems [12]), one-to-all (e.g., application data broadcasting), or all-to-all (e.g., MapReduce [6] and Dryad [17]) communications among MDC servers. BCube is designed to well support all these traffic patterns. A recent technology trend is to build data centers using commodity servers [2]. We go one step further by using only low-end COTS mini-switches. This option eliminates expensive high-end switches. Different from a traditional data center, it is difficult or even impossible to service an MDC once it is deployed. Therefore, BCube needs to achieve graceful performance degradation in the presence of server and switch failures.

The BCube network architecture takes the server-centric approach, rather than the switch-oriented practice. It places intelligence on MDC servers and works with commodity switches. BCube makes innovations in its server-centric interconnection structure, routing protocol, and efficient implementation.

In the BCube interconnection structure, each server is equipped with a small number of ports (typically no more than four). Multiple layers of cheap COTS mini-switches are used to connect those servers. BCube provides multiple parallel short paths between any pair of servers. This not only provides high one-to-one bandwidth, but also greatly improves fault tolerance and load balancing. BCube accelerates one-to-several and one-to-all traffic by constructing edge-disjoint complete graphs and multiple edge-disjoint server spanning trees. Moreover, due to its low diameter, BCube provides high network capacity for all-to-all traffic such as MapReduce.

BCube runs a source routing protocol called BSR (BCube Source Routing). BSR places routing intelligence solely onto servers. By taking advantage of the multi-path property of BCube and by actively probing the network, BSR balances traffic and handles failures without link-state distribution. With BSR, the capacity of BCube decreases gracefully as the server and/or switch failure increases.

We have designed and implemented a BCube protocol suite. We can design a fast packet forwarding engine, which can decide the next hop of a packet by only one table lookup. The packet forwarding engine can be efficiently implemented in both software and hardware. We have built a BCube testbed with 16 servers and 8 8-port Gigabit Ethernet mini-switches. Experiments in our testbed demonstrated the efficiency of our implementation. Experiments also showed that BCube provides 2 times speedup for one-to-x (abbreviation for one-to-one, one-to-several, and one-to-all) traffic patterns, and 3 times throughput for MapReduce tasks compared with the tree structure. BCube uses more wires than the tree structure. But wiring is a solvable issue for containers which are at most 40-feet long.

Recently, fat-tree [1] and DCell [9] are proposed as network structures to interconnect tens of thousands or more servers in data centers. BCube is better than these two structures for MDCs. Compared with DCell, BCube does not have performance bottlenecks and provides much higher network capacity; compared with fat-tree, BCube provides better one-to-x support and can be directly built using commodity switches without any switch upgrade. See Section 8 for detailed comparisons.

The rest of the paper is organized as follows. Section 2 discusses background. Section 3 presents BCube and its support for various traffic patterns. Section 4 designs BSR. Section 5 addresses other design issues. Sections 6 studies graceful degradation. Section 7 presents implementation and experiments. Section 8 discusses related work and Section 9 concludes the paper.

2. BACKGROUND

MDCs present new research opportunities as well as challenges. The size of a 40-feet container is $12m \times 2.35m \times 2.38m$, hence wiring becomes a solvable problem when we depart from the traditional tree structure; it is possible to use cheap commodity Gigabit Ethernet mini-switches for interconnection since the target scale is typically thousands of servers. Yet, designing network architecture for MDCs is also challenging. It is difficult or even impossible to service a container once it is sealed and deployed. The design should be fault tolerant and the performance should degrade gracefully as components failure increases. We now elaborate the requirements for MDCs in more details.

Bandwidth-intensive application support. Many data center applications need to move huge amount of data among servers, and network becomes their performance bottleneck [6, 12, 17]. A well designed network architecture needs to provide good support for typical traffic patterns. We describe several typical traffic patterns as follows.

One-to-one, which is the basic traffic model in which one server moves data to another server. Our design should provide high inter-server throughput. This is particularly useful when there exist server pairs that exchange large amount of data such as disk backup. Good one-to-one support also results in good several-to-one and all-to-one support.

One-to-several, in which one server transfers the same copy of data to several receivers. Current distributed file systems such as GFS [12], HDFS [4], and CloudStore [5], replicate data chunks of a file several times (typically three) at different chunk servers to improve reliability. When a chunk is written into the file system, it needs to be simultaneously replicated to several servers.

One-to-all, in which a server transfers the same copy of data to all the other servers in the cluster. There are several cases that one-to-all happens: to upgrade the system image, to distribute application binaries, or to distribute specific application data.

All-to-all, in which every server transmits data to all the other servers. The representative example of all-to-all traffic is MapReduce [6]. The reduce phase of MapReduce needs to shuffle data among many servers, thus generating an all-to-all traffic pattern.

Low-end commodity switches. Current data centers use commodity PC servers for better performance-to-price ratio [2]. To achieve the same goal, we use low-end non-programmable COTS switches instead of the high-end ones, based on the observation that the per-port price of the low-end switches is much cheaper than that of the high-end ones. As we have outlined in our first design goal, we want to provide high capacity for various traffic patterns. The COTS switches, however, can speak only the spanning tree protocol, which cannot fully utilize the links in advanced network structures. The switch boxes are generally not as open as the server computers. Re-programming the switches for new routing and packet forwarding algorithms is much harder, if not impossible, compared with programming the servers. This is a challenge we need to address.

Graceful performance degradation. Given that we only assume commodity servers and switches in a shipping-container data center, we should assume a failure model of frequent component failures. Moreover, An MDC is prefabricated in factory, and it is rather difficult, if not impossible, to service an MDC once it is deployed in the field, due to operational and space constraints. Therefore, it is extremely important that we design our network architecture to be fault tolerant and to degrade gracefully in the presence of continuous component failures.

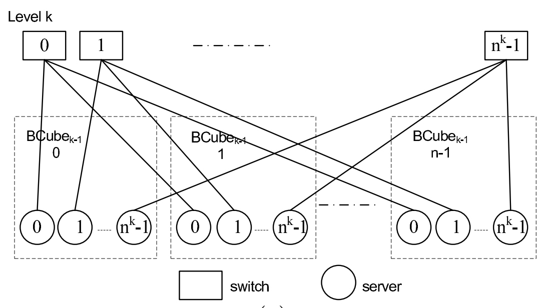
In our BCube architecture, we introduce a novel server-centric BCube network topology and a BCube Source Routing protocol (BSR) to meet the requirements of MDCs. We next present the structure and BSR sequentially.

3. THE BCUBE STRUCTURE

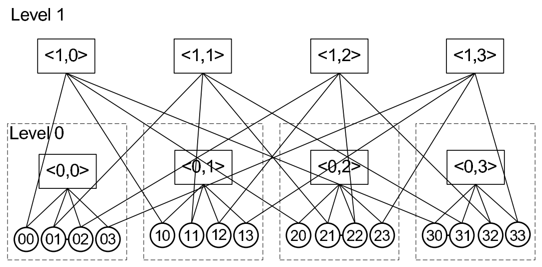
In this section, we first present the server-centric BCube structure and then analyze its one-to-x and all-to-all traffic support properties.

3.1 BCube Construction

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 n servers connecting to an n-port switch. A BCube_1 is constructed from n BCube_0s and n n-port switches. More generically, a BCube_k ($k \geq 1$)) is con-



(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.

structed 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 $N = n^{k+1}$ servers and $k+1$ level of switches, with each level having n^k n-port switches.

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 illustrated in Fig. 1(a). Fig. 1(b) shows a BCube$_1$ with $n = 4$, which is constructed from four BCube$_0$s and four 4-port switches. The links in BCube are bidirectional. In this section, we assume the link bandwidth as 1 for simplicity of presentation.

We denote a server in a BCube$_k$ using an address array $a_k a_{k-1} \cdots a_0$ ($a_i \in [0, n-1], i \in [0, k]$). Equivalently, we can use a *BCube address* $baddr = \sum_{i=0}^{k} a_i n^i$ to denote a server. We denote a switch using the form $< l, s_{k-1}s_{k-2} \cdots s_0 >$ ($s_j \in [0, n-1], j \in [0, k-1]$), where $l (0 \le l \le k)$ is the level of the switch.

From Fig. 1, we can see that the i-th port of a level-k switch $< k, s_{k-1}s_{k-2} \cdots s_0 >$ connects to the level-k port of server $i s_{k-1}s_{k-2} \cdots s_0$ ($i \in [0, n - 1]$). More generically, the construction ensures that the i-th port of a switch $< l, s_{k-1}s_{k-2} \cdots s_0 >$ connects to the level-l port of server $s_{k-1}s_{k-2} \cdots s_l i s_{l-1} \cdots s_0$.

The BCube construction guarantees that switches only connect to servers and never directly connect to other switches. As a direct consequence, we can treat the switches as dummy crossbars that connect several neighboring servers and let servers relay traffic for each other. With 8-port mini-switches, we can support up to 4096 servers in one BCube$_3$. BCube therefore meets our goal of using only low-end commodity switches by putting routing intelligence purely into servers.

The BCube structure is closely related to the generalized Hypercube [3]. In a BCube network, if we replace each switch and its n links with an $n \times (n - 1)$ full mesh that directly connects the servers, we get a generalized Hypercube. Compared to the generalized Hypercube, the server port number is much smaller in BCube. It is $k + 1$ in a BCube$_k$ and $(n - 1)(k + 1)$ in a corresponding generalized Hypercube. This implies that we reduce the server port number from 28 to 4 when $n = 8$ and $k = 3$. The incurred cost in BCube is the $k+1$ layers of switches. This is a tradeoff we make willingly, due to the low cost of the mini-switches.

```
/*
  A=$a_k a_{k-1} \cdots a_0$ and B=$b_k b_{k-1} \cdots b_0$; $A[i] = a_i$; $B[i] = b_i$;
  $\Pi = [\pi_k, \pi_{k-1}, \cdots, \pi_0]$ is a permutation
    of $[k, k-1, \cdots, 1, 0]$
*/
BCubeRouting(A, B, $\Pi$):
  $path(A, B) = \{A, \}$;
  $I\_Node = A$;
  for($i = k; i \ge 0; i - -$)
    if ($A[\pi_i] \ne B[\pi_i]$)
      $I\_Node[\pi_i] = B[\pi_i]$;
      append $I\_Node$ to $path(A, B)$;
  return $path(A, B)$;
```

Figure 2: BCubeRouting to find a path from A to B. The algorithm corrects one digit at one step. The digit correcting order is decided by the predefined permutation Π.

3.2 Single-path Routing in BCube

We use $h(A, B)$ to denote the Hamming distance of two servers A and B, which is the number of different digits of their address arrays. Apparently, the maximum Hamming distance in a BCube$_k$ is $k + 1$. Two servers are *neighbors* if they connect to the same switch. The Hamming distance of two neighboring servers is one. More specifically, two neighboring servers that connect to the same level-i switch only differ at the i-th digit in their address arrays. By utilizing this fact, we design BCubeRouting, as we illustrate in Fig. 2, to find a path from a source server to a destination server.

In BCubeRouting, A=$a_k a_{k-1} \cdots a_0$ is the source server and B=$b_k b_{k-1} \cdots b_0$ is the destination server, and Π is a permutation of $[k, k-1, \cdots, 1, 0]$. We systematically build a series of intermediate servers by 'correcting' one digit of the previous server. Hence the path length is at most $k+1$. Note that the intermediate switches in the path can be uniquely determined by its two adjacent servers, hence are omitted from the path. BCubeRouting is similar to the routing algorithm for Hypercube. This is not by chance, but because BCube and the generalized Hypercube share similarity as we have discussed in Section 3.1.

From BCubeRouting, we obtain the following theorem.

THEOREM 1. *The diameter, which is the longest shortest path among all the server pairs, of a BCube$_k$, is $k + 1$.*

In practice, k is a small integer, typically at most 3. Therefore, BCube is a low-diameter network.

```
/*A=$a_k a_{k-1} \cdots a_0$ and B=$b_k b_{k-1} \cdots b_0$; $A[i] = a_i$; $B[i] = b_i$;*/
BuildPathSet($A, B$):
    PathSet = { };
    for($i = k; i \geq 0; i - -$)
      if ($A[i] \neq B[i]$)
        $P_i$=DCRouting($A, B, i$);
      else /*$A[i] == B[i]$*/
        $C$= a neighbor of $A$ at level $i$; /*$C[i] \neq A[i]$*/
        $P_i$=AltDCRouting($A, B, i, C$);
      add $P_i$ to PathSet;
    return PathSet;

DCRouting($A, B, i$):
    $m = k$;
    for ($j = i; j \geq i - k; j - -$)
      $\Pi[m] = j \mod (k+1)$; $m = m - 1$;
    path = BCubeRouting($A, B, \Pi$);
    return path;

AltDCRouting($A, B, i, C$):
    path={$A$,};
    $m = k$;
    for ($j = i - 1; j \geq i - 1 - k; j - -$)
      $\Pi[m] = j \mod (k+1)$; $m = m - 1$;
    path += BCubeRouting($C, B, \Pi$);
    return path;
```

Figure 3: The algorithm to calculate the $k+1$ parallel paths between servers A and B.

3.3 Multi-paths for One-to-one Traffic

Two *parallel* paths between a source server and a destination server exist if they are node-disjoint, i.e., the intermediate servers and switches on one path do not appear on the other. The following theorem shows how to generate two parallel paths between two servers.

THEOREM 2. *Given that two servers $A = a_k a_{k-1} \cdots a_0$ and $B = b_k b_{k-1} \cdots b_0$ are different in every digit (i.e., $a_i \neq b_i$ for $i \in [0, k]$). BCubeRouting generates two parallel paths from A to B using two permutations $\Pi_0 = [i_0, (i_0 - 1) \mod (k+1), \cdots, (i_0 - k) \mod (k+1)]$ and $\Pi_1 = [i_1, (i_1 - 1) \mod (k+1), \cdots, (i_1 - k) \mod (k+1)]$ ($i_0 \neq i_1$ and $i_0, i_1 \in [0, k]$).*

The permutations Π_0 and Π_1 start from different locations of the address array and then correct the digits sequentially. This pattern ensures that the used switches are always at different levels for the same digit position, thus producing the two parallel paths. The formal proof of Theorem 2 is given in Appendix A.

From Theorem 2, we see that when the digits of A and B are different, there are $k+1$ parallel paths between them. It is also easy to observe that the number of parallel paths between two servers be upper bounded by $k+1$, since each server has only $k+1$ links. The following theorem specifies the exact number of parallel paths between any two servers.

THEOREM 3. *There are $k+1$ parallel paths between any two servers in a BCube$_k$.*

We show the correctness of Theorem 3 by constructing such $k+1$ paths. The construction procedure, BuildPathSet, is based on Theorem 2 and shown in Fig. 3. For two servers A and B, the paths built by BuildPathSet fall into two categories: the paths constructed by DCRouting using permutations start from digits $a_i \neq b_i$ and those constructed by AltDCRouting. There are $h(A, B)$ and $k + 1 - h(A, B)$

$$
\begin{array}{ll}
P_3 : \{0001, & <3,001>, 1001, <1,101>, 1011\} \\
P_2 : \{0001, & <2,001>, 0101, <1,011>, 0111, \\
 & <3,111>, 1111, <2,111>, 1011\} \\
P_1 : \{0001, & <1,001>, 0011, <3,011>, 1011\} \\
P_0 : \{0001, & <0,000>, 0002, <3,002>, 1002, \\
 & <1,102>, 1012, <0,101>, 1011\}
\end{array}
$$

Figure 4: An example showing the parallel paths between two servers A (0001) and B (1011) in a BCube$_3$ with $n = 8$. There are 2 paths with length 2 (P_3 and P_1) and 2 paths with length 4 (P_2 and P_0).

paths in the first and second categories, respectively. From Theorem 2 (and by removing the digits $a_i = b_i$ in all the servers), we can see that the paths in the first category are parallel.

Next, we show that paths in the second category are also parallel. Assume $a_i = b_i$ and $a_j = b_j$ for two different i and j. From Fig. 3, the i-th digit of all the intermediate servers in path P_i is a value $c_i \neq a_i$, whereas it is a_i in all the intermediate servers in path P_j. Similarly, the j-th digits of the intermediate servers in P_i and P_j are also different. The intermediate servers in P_i and P_j differ by at least two digits. The switches in P_i and P_j are also different, since a switch connects only to servers that differ in a single digit. Hence the paths in the second category are parallel.

Finally, we show that paths in different categories are parallel. First, the intermediate servers of a path in the second category are different from the servers in the first category, since there is at least one different digit (i.e., the i-th digit c_i). Second, the switches of a path in the second category are different from those in the first category (due to the fact that switches in the second category have c_i whereas those in the first category have a_i in the same position).

From BuildPathSet, we further observe that the maximum path length of the paths constructed by BuildPathSet be $k + 2$. The lengths of the paths in the first and second categories are $h(A, B)$ and $h(A, B) + 2$, respectively. The maximum value of $h(A, B)$ is $k + 1$, hence the maximum path length is at most $k+3$. But $k+3$ is not possible, since when $h(A, B) = k + 1$, the number of paths in the second category is 0. The parallel paths created by BuildPathSet therefore are of similar, small path lengths. It is also easy to see that BuildPathSet is of low time-complexity $O(k^2)$.

Fig. 4 shows the multiple paths between two servers 0001 and 1011 in a BCube network with $n = 8$ and $k = 3$. The Hamming distance of the two servers is $h(A, B) = 2$. We thus have two paths of length 2. These two paths are P_3 and P_1. We also have two paths of length $h(A, B) + 2 = 4$. These two paths are P_2 and P_0, respectively. For clarity, we also list the intermediate switches in the paths. It is easy to verify that all these paths are parallel, since an intermediate server or switch on one path never appears on other paths.

It is easy to see that BCube should also well support several-to-one and all-to-one traffic patterns. We can fully utilize the multiple links of the destination server to accelerate these x-to-one traffic patterns.

3.4 Speedup for One-to-several Traffic

We show that edge-disjoint complete graphs with $k + 2$ servers can be efficiently constructed in a BCube$_k$. These complete graphs can speed up data replications in distributed file systems like GFS [12].

THEOREM 4. *In a BCube$_k$, a server src and a set of servers $\{d_i | 0 \leq i \leq k\}$, where d_i is an one-hop neighbor of src at level i (i.e., src and d_i differ only at the i-th digit), can form an edge-disjoint complete graph.*

We show how we recursively construct such an edge-disjoint complete graph. Suppose src and $d_0 - d_{k-1}$ are in a BCube$_{k-1}$ B_0, and d_k is in another BCube$_{k-1}$ B_1. Assume that servers in B_0 have already formed a complete graph. We show how to construct the edges among d_k and the rest servers $d_0 - d_{k-1}$. The key idea is to find k servers $d'_0 - d'_{k-1}$, where $d'_i (0 \leq i < k)$ and d_k differ in the i-th digit. It is easy to see that the Hamming distance between d_i and d_k is two. We can then establish an edge between d_i and d_k via the intermediate server d'_i. This edge uses the level-k link of d_i and the level-i link of d_k. This edge is node-disjoint with other edges: it does not overlap with the edges in B_0 since it uses the level-k link of d_i; it also does not overlap with the edges in B_1 since it uses the level-i link of d_k. In this way, we can recursively construct the edges between d_{k-1} and $d_i (0 \leq i < k-1)$, using the level-$(k-1)$ links of $d_i (0 \leq i < k-1)$ and level-i link of d_{k-1}, etc.

From the construction procedure, we see that the diameter of the constructed complete graph is only two hops. For a server src, there exist a huge number of such complete graphs. src has $n-1$ choices for each d_i. Therefore, src can build $(n-1)^{k+1}$ such complete graphs.

In distributed file systems such as GFS [12], CloudStore [5], and HDFS [4], a file is divided into chunks, and each chunk is replicated several times (typically three) at different chunk servers to improve reliability. The replicas are chosen to locate at different places to improve reliability. The source and the selected chunk servers form a pipeline to reduce the replication time: when a chunk server starts to receive data, it transmits the data to the next chunk server.

The complete graph built in BCube works well for chunk replication for two reasons: First, the selected servers are located at different levels of BCube, thus improving replication reliability. Second, edge-disjoint complete graph is perfect for chunk replication speedup. When a client writes a chunk to r $(r \leq k+1)$ chunk servers, it sends $\frac{1}{r}$ of the chunk to each of the chunk server. Meanwhile, every chunk server distributes its copy to the other $r-1$ servers using the disjoint edges. This will be r times faster than the pipeline model.

3.5 Speedup for One-to-all Traffic

We show that BCube can accelerate one-to-all traffic significantly. In one-to-all, a source server delivers a file to all the other servers. The file size is L and we assume all the links are of bandwidth 1. We omit the propagation delay and forwarding latency. It is easy to see that under tree and fat-tree, the time for all the receivers to receive the file is at least L. But for BCube, we have the following theorem.

THEOREM 5. *A source can deliver a file of size L to all the other servers in $\frac{L}{k+1}$ time in a BCube$_k$.*

We show the correctness of Theorem 5 by constructing $k+1$ edge-disjoint server spanning trees. Edge-disjoint means that an edge in one spanning tree does not appear in all the other ones. Fig. 5 shows how such $k+1$ server spanning trees are constructed. BuildMultipleSPTs constructs the $k+1$ spanning trees from the $k+1$ neighbors of the source. A

```
/*Servers are denoted using the address array form.
  $src[i]$ denotes the $i$-th digit of the address array of $src$.*/
BuildMultipleSPTs($src$):
  for($i = 0; i \leq k; i++$)
    $root$=src's level-$i$ neighbor and $root[i] = (src[i] + 1) \bmod n$;
    $Tree_i = \{root,\}$;
    BuildSingleSPT($src, Tree_i, i$);

BuildSingleSPT($src, T, level$):
Part_I:
  for($i = 0; i \leq k; i++$)
    $dim = (level + i) \bmod (k+1)$;
    $T_2 = \{\}$;
    for (each server A in T)
      $C = B = A$;
      for ($j = 0; j < n-1; j++$)
        $C[dim] = (C[dim] + 1) \bmod n$;
        add server $C$ and edge $(B, C)$ to $T_2$;
        $B = C$;
    add $T_2$ to $T$;
Part_II:
  for (each server $S \neq src$ and $S[level] = src[level]$)
    $S_2 = S$; $S_2[level] = (S[level] - 1) \bmod n$;
    add server $S$ and edge $(S_2, S)$ to T;
```

Figure 5: Build the $k+1$ edge-disjoint server spanning trees from a source server in a BCube$_k$.

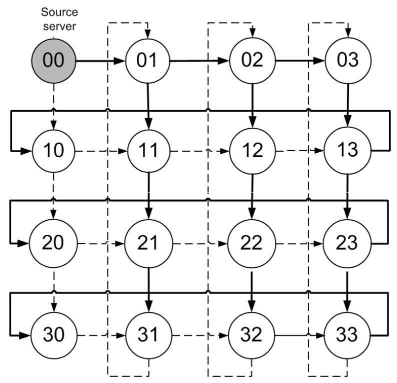


Figure 6: The two edge-disjoint server spanning trees with server 00 as the source for the BCube$_1$ network in Fig. 1(b).

level-i neighbor differs from the source in the i-th digit. We then systematically add servers into the tree starting from that level. Intuitively, the trees are edge-disjoint because a server is added to different trees using links of different levels. The formal proof is omitted due to space limitation. Fig. 6 shows the two edge-disjoint spanning trees with server 00 as the source for the BCube$_1$ network of Fig. 1(b).

When a source distributes a file to all the other servers, it can split the file into $k+1$ parts and simultaneously deliver all the parts via different spanning trees. Since a receiving server is in all the spanning trees, it receives all the parts and hence the whole file. The time to deliver a file of size L to all is therefore $\frac{L}{k+1}$ in a BCube$_k$. No broadcast or multicast is needed in BuildMultipleSPTs. Hence we can use TCP to construct the trees for reliable data dissemination.

3.6 Aggregate Bottleneck Throughput for All-to-all Traffic

Under the all-to-all model, every server establishes a flow with all other servers. Among all the flows, the flows that receive the smallest throughput are called the *bottleneck flows*. The aggregate bottleneck throughput (ABT) is defined as the number of flows times the throughput of the bottleneck flow. The finish time of an all-to-all task is the total shuffled data divided by ABT. ABT therefore reflects the network capacity under the all-to-all traffic pattern.

THEOREM 6. *The aggregate bottleneck throughput for a BCube network under the all-to-all traffic model is* $\frac{n}{n-1}(N-1)$*, where* n *is the switch port number and* N *is the number of servers.*

See Appendix B on how we derive the number. An advantage of BCube is that BCube does not have performance bottlenecks in the all-to-all traffic model since all the links are used equally. As a result, the ABT of BCube increases linearly as the number of servers increases. As we will show in Section 6, the ABT of BCube decreases gracefully under both server and switch failures.

4. BCUBE SOURCE ROUTING

We require that the routing protocol be able to fully utilize the high capacity (e.g., multi-path) of BCube and automatically load-balance the traffic. Existing routing protocols such as OSPF and IS-IS [20] cannot meet these requirements. Furthermore, it is unlikely that OSPF and IS-IS can scale to several thousands of routers [20]. In this paper, we design a source routing protocol called BSR by leveraging BCube's topological property. BSR achieves load balance and fault tolerance, and enables graceful performance degradation.

4.1 The BSR Idea

In BSR, the source server decides which path a packet flow should traverse by probing the network and encodes the path in the packet header. We select source routing for two reasons. First, the source can control the routing path without coordinations of the intermediate servers. Second, intermediate servers do not involve in routing and just forward packets based on the packet header. This simplifies their functionalities. Moreover, by reactively probing the network, we can avoid link state broadcasting, which suffers from scalability concerns when thousands of servers are in operation.

In BSR, a flow can change its path but only uses one path at a given time, in order to avoid the packet out-of-order problem. A flow is a stream of packets that have the same values for a subset of fields of the packet header, such as the five-tuple (*src*, *src_port*, *dst*, *dst_port*, *prot*). We treat a duplex flow as two separate simplex flows, since the network conditions along opposite directions may be different.

When a new flow comes, the source sends probe packets over multiple parallel paths. The intermediate servers process the probe packets to fill the needed information, e.g., the minimum available bandwidth of its input/output links. The destination returns a probe response to the source. When the source receives the responses, it uses a metric to select the best path, e.g., the one with maximum available bandwidth. In this paper, we use available bandwidth to optimize application throughput since we focus on bandwidth-intensive applications. However, it is possible to use other metrics such as end-to-end delay.

When a source is performing path selection for a flow, it does not hold packets. The source initially uses a default

```
PathSelection(src, dst):
Source:
  when a flow arrives or probing timer timeouts:
      goodPathSet = { };
      pathSet = BuildPathSet(src, dst);
      while (pathSet not empty)
         path = pathSet.remove();
         if (ProbePath(path) succeeds)
            goodPathSet.add(path);
         else
            altPath = BFS(pathSet, goodPathSet);
            if(altPath exists) pathSet.add(altPath);
       return SelectBestPath(goodPathSet);

Intermediate_server: /*receiver is not pkt.dst*/
  when a path probe pkt is received:
      if (next hop not available)
         send path failure msg to src; return;
      ava_band = min(ava_band_in, ava_band_out);
      if (ava_band < pkt.ava_band)
         pkt.ava_band = ava_band;
       forward(pkt);

Destination: /*receiver is pkt.dst*/
  when a path probe pkt is received:
      if (ava_band_in < pkt.ava_band)
         pkt.ava_band = ava_band_in;
      reverse the path in pkt; pkt.type = response;
      route pkt back to src;
```

Figure 7: The path selection procedure of BSR.

path selected from the parallel path set. After the path selection completes and a better path is selected, the source switches the flow to the new path. Path switching may result in temporal packet out-of-order. Because path probing can be done in a short time and TCP is still in its three-way handshaking or slow-start phase, this one-time switching does not pose performance problem.

4.2 The PathSelection Procedure

The detailed, path probing and selection procedure is given in Fig. 7. It has three parts, which describe how source, intermediate, and destination servers perform.

When a source performs PathSelection, it first uses BuildPathSet to obtain $k+1$ parallel paths and then probes these paths. The PathSelection tries its best to find $k+1$ parallel paths. Thus, if one path is found not available, the source uses the Breadth First Search (BFS) algorithm to find another parallel path. The source first removes the existing parallel paths and the failed links from the BCube$_k$ graph, and then uses BFS to search for a path. When links are of equal weights, BFS is a shortest-path routing algorithm. The newly found path is parallel to the existing pathes, since all existing paths are removed before BFS. When BFS cannot find a path, we know that the number of parallel paths must be smaller than $k+1$.

BFS is very fast for a BCube network that has thousands of servers. For a BCube network with $n=8$ and $k=3$, the execution time of BFS is less than 1 millisecond in the worst case (using a 2.33GHZ Intel dualcore CPU).

When an intermediate server receives a probe packet, if its next hop is not available, it returns a path failure message (which includes the failed link) to the source. Otherwise, it updates the available bandwidth field of the probe packet if its available bandwidth is smaller than the existing value. Its available bandwidth is the minimum available bandwidth of its incoming and outgoing links. We need to do this because two adjacent servers A and B in BCube are indirectly connected via a switch S. Hence the available bandwidth of A's output link is not necessarily equal to that of B's input link.

When a destination server receives a probe packet, it first updates the available bandwidth field of the probe packet if the available bandwidth of the incoming link is smaller than the value carried in the probe packet. It then sends the value back to the source in a probe response message.

All the servers maintain a failed link database by overhearing the path failure messages. Links are removed from the database by timeout or by a successful probe response that contains that link. This database is used in the path selection procedure as we have described.

4.3 Path Adaptation

During the lifetime of a flow, its path may break due to various failures and the network condition may change significantly as well. The source periodically (say, every 10 seconds) performs path selection to adapt to network failures and dynamic network conditions.

When an intermediate server finds that the next hop of a packet is not available, it sends a path failure message back to the source. As long as there are paths available, the source does not probe the network immediately when the message is received. Instead, it switches the flow to one of the available paths obtained from the previous probing. When the probing timer expires, the source will perform another round path selection and try its best to maintain $k+1$ parallel paths. This design simplifies the implementation by avoiding packets buffering.

When multiple flows between two servers arrive simultaneously, they may select the same path. To make things worse, after the path selection timers expire, they will probe the network and switch to another path simultaneously. This results in path oscillation. We mitigate this symptom by injecting randomness into the timeout value of the path selection timers. The timeout value is a constant plus a small random value. Our experiment in Section 7.5 showed that this mechanism can efficiently avoid path oscillation.

5. OTHER DESIGN ISSUES

5.1 Partial BCube

In some cases, it may be difficult or unnecessary to build a complete BCube structure. For example, when $n = 8$ and $k = 3$, we have 4096 servers in a $BCube_3$. However, due to space constraint, we may only be able to pack 2048 servers.

A simple way to build a partial $BCube_k$ is to first build the $BCube_{k-1}$s and then use a partial layer-k switches to interconnect the $BCube_{k-1}$s. Using Fig. 1(b) as an example, when we build a partial $BCube_1$ with 8 servers, we first build two $BCube_0$s that contain servers 00-03 and 10-13, we then add two switches $< 1, 0 >$ and $< 1, 1 >$ to connect the two $BCube_0$s. The problem faced by this approach is that BCubeRouting does not work well for some server pairs. For example, BCubeRouting will not be able to find a path between servers 02 and 13 no matter which routing permutation is used, because 02 and 13 are connected to non-existing layer-1 switches. Of course, we still can establish paths between 02 and 13 by enlarging the path length. For example, 02 can reach 13 via path {02, 00, 10, 13}. But this approach reduces network capacity.

The root cause for why server 02 cannot reach server 13 is that we do not have switches $< 1, 2 >$ and $< 1, 3 >$. Hence, our solution to partial BCube construction is as follows. When building a partial $BCube_k$, we first build the needed $BCube_{k-1}$s, we then connect the $BCube_{k-1}$s using a *full* layer-k switches. With a full layer-k switches, BCubeRouting performs just as in a complete BCube, and BSR just works as before.

An apparent disadvantage of using a full layer-k switches is that switches in layer-k are not fully utilized. We prefer this solution because it makes routing the same for partial and complete BCubes, and most importantly, the mini-switches are cheap and affordable. In this paper, we choose $n = 8$ and $k = 3$ and use these parameters to build a partial BCube with 2048 servers. $n = 8$ implies that we only need cheap COTS mini-switches. $k = 3$ means that each server has 4 ports, which provides significant speedups for one-to-x and enough fault-tolerance and load-balance.

5.2 Packaging and Wiring

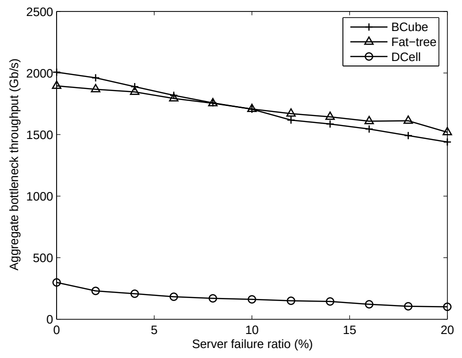
We show how packaging and wiring can be addressed for a container with 2048 servers and 1280 8-port switches (a partial BCube with $n = 8$ and $k = 3$). The interior size of a 40-feet container is $12m \times 2.35m \times 2.38m$. In the container, we deploy 32 racks in two columns, with each column has 16 racks. Each rack accommodates 44 rack units (or 1.96m high). We use 32 rack units to host 64 servers as the current practice can pack two servers into one unit [23], and 10 rack units to host 40 8-port switches. The 8-port switches are small enough, and we can easily put 4 into one rack unit. Altogether, we use 42 rack units and have 2 unused units.

As for wiring, the Gigabit Ethernet copper wires can be 100 meters long, which is much longer than the perimeter of a 40-feet container. And there is enough space to accommodate these wires. We use 64 servers within a rack to form a BCube$_1$ and 16 8-port switches within the rack to interconnect them. The wires of the BCube$_1$ are inside the rack and do not go out. The inter-rack wires are layer-2 and layer-3 wires and we pace them on the top of the racks. We divide the 32 racks into four super-racks. A super-rack forms a BCube$_2$ and there are two super-racks in each column. We evenly distribute the layer-2 and layer-3 switches into all the racks, so that there are 8 layer-2 and 16 layer-3 switches within every rack. The level-2 wires are within a super-rack and level-3 wires are between super-racks. Our calculation shows that the maximum number of level-2 and level-3 wires along a rack column is 768 (256 and 512 for level-2 and level-3, respectively). The diameter of an Ethernet wire is 0.54cm. The maximum space needed is approximate $176cm^2 < (20cm)^2$. Since the available height from the top of the rack to the ceil is 42cm, there is enough space for all the wires.

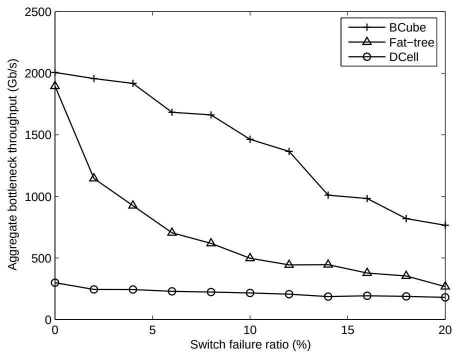
5.3 Routing to External Networks

So far, we focus on how to route packets inside a BCube network. Internal servers need to communicate with external computers in the Internet or other containers. Since we have thousands of servers in an MDC, the total throughput to or from external network may be high. We assume that both internal and external computers use TCP/IP.

We propose *aggregator* and *gateway* for external communication. An aggregator is simply a commodity layer-2 switch with 10G uplinks. We can use a 48X1G+1X10G aggregator



(a)



(b)

Figure 8: The aggregate bottleneck throughput of BCube, fat-tree, and DCell under switch and server failures.

to replace several mini-switches and use the 10G link to connect to the external network. The servers that connect to the aggregator become *gateways*. When an internal server sends a packet to an external IP address, it will choose one of the gateways. The external IP to gateway mapping can be manually configured or dynamically adjusted based on network condition. The packet is then routed to the gateway using BSR. After the gateway receives the packet, it strips the BCube protocol header (which will be explained in Section 7.1) and forwards the packet to the external network via the 10G uplink. The paths from external computers to internal servers can be constructed similarly.

We can add more aggregators to handle large traffic volume and introduce load-balancing mechanisms to evenly distribute the traffic among the gateways. These are topics for our future work.

6. GRACEFUL DEGRADATION

In this section, we use simulations to compare the aggregate bottleneck throughput (ABT) of BCube, fat-tree [1], and DCell [9], under random server and switch failures. Recall that ABT is the throughput of the bottleneck flow times the number of total flows in the all-to-all traffic model (Section 3.6). ABT reflects the all-to-all network capacity. We show that only BCube provides high ABT and graceful degradation among the three structures. Graceful degradation states that when server or switch failure increases, ABT reduces slowly and there are no dramatic performance falls. We assume all the links are 1Gb/s and there are 2048 servers. This setup matches a typical shipping-container data center.

For all the three structures, we use 8-port switches to construct the network structures. The BCube network we use is a partial BCube$_3$ with $n = 8$ that uses 4 full BCube$_2$. The fat-tree structure has five layers of switches, with layers 0 to 3 having 512 switches per-layer and layer-4 having 256 switches. The DCell structure is a partial DCell$_2$ which contains 28 full DCell$_1$ and one partial DCell$_1$ with 32 servers. We use BSR routing for BCube and DFR [9] for DCell. For fat-tree, we use the routing algorithm described in [1] when there is no failure and we randomly re-distribute a flow to an available path when the primary path fails. The results are plotted in Figures 8(a) and (b) for server and switch failures, respectively.

The results show that when there is no failure, both BCube and fat-tree provide high ABT values, 2006Gb/s for BCube and 1895Gb/s for fat-tree. BCube is slightly better than fat-tree because the ABT of BCube is $\frac{n(N-1)}{n-1}$, which is slightly higher than that of fat-tree, N. But DCell only provides 298Gb/s ABT. This result is due to several reasons. First, the traffic is imbalanced at different levels of links in DCell. Low-level links always carry much more flows than high-level links. In our simulation, the maximum numbers of flows in the level-0 - level-2 links are 14047, 9280, and 5184, respectively. Second, partial DCell makes the traffic imbalanced even for links at the same level. In our simulation, the maximum and minimum numbers of flows in the level-0 links are 14047 and 2095, respectively. This huge difference is because there are level-0 links that seldom carry traffic for other servers in a partial DCell.

Fat-tree performs well under server failures but its ABT drops dramatically when switch failure increases (e.g., 1145Gb/s at 2% switch failure and 704Gb/s at 6% switch failure). Our analysis revealed that the dramatic drop is caused by low-level switch failures. In fat-tree, switches at different layers have different impact on routing performance. When a level-1 switch fails, an affected server has only $\frac{n}{2} - 1$ choices to re-route, whereas it has $(\frac{n}{2})^2 - 1$ to re-route for a level-2 switch failure. Hence the failures of low-level switches make the traffic imbalanced in fat-tree and degrade the performance dramatically.

BCube performs well under both server and switch failures. Compared with fat-tree, switches at different layers are equal in BCube (recall that the multi-paths we build in Section 3.3 use switches at different levels equally). In BCube, live servers always have 4 live links under the server failure model whereas some live servers may have less than 4 live links under switch failures. This difference results less balanced traffic and therefore smaller ABT under the switch failure model. But the degradation is graceful. The ABT value is 765Gb/s even when the switch failure ratio reaches 20% (as a comparison, it is only 267Gb/s for fat-tree).

7. IMPLEMENTATION AND EVALUATION

7.1 Implementation Architecture

We have prototyped the BCube architecture by designing and implementing a BCube protocol stack. We have implemented the stack as a kernel driver in the Windows Servers 2003 and 2008. The BCube stack locates between the TCP/IP protocol driver and the Ethernet NDIS (Network Driver Interface Specification) driver. The BCube driver is located at 2.5 layer: to the TCP/IP driver, it is a NDIS driver; to the real Ethernet driver, it is a protocol driver. TCP/IP applications therefore are compatible with BCube since they see only TCP/IP.

The BCube stack has the following key components: the BSR protocol for routing, the neighbor maintenance protocol which maintains a neighbor status table, the packet sending/receiving part which interacts with the TCP/IP stack, and the packet forwarding engine which relays packets for other servers.

The BCube packet header is similar to that of DCell [9]. Each packet includes a BCube header between the Ethernet header and IP header. The header contains typical fields including source and destination BCube addresses, packet id, protocol type, payload length, and header checksum. We use a 32-bit integer for server address. Similar to DCell, we also maintain a fixed, one-to-one mapping between an IP address and a BCube address.

Different from DCell, BCube stores the complete path and a next hop index (NHI) in the header of every BCube packet. If we directly use the 32-bit addresses, we need many bytes to store the complete path. For example, we need 32 bytes when the maximum path length is 8. In this paper, we leverage the fact that neighboring servers in BCube differ in only one digit in their address arrays to reduce the space needed for an intermediate server, from four bytes to only one byte. We call this byte NHA. NHA is divided into two parts: DP and DV. DP indicates which digit of the next hop is different from the current relay server, and DV is the value of that digit. In our current implementation, DP has 2 bits and DV has 6 bits, the path (NHA array) has 8 bytes, and the BCube header length is 36 bytes.

7.2 Packet Forwarding Engine

We have designed an efficient packet forwarding engine which decides the next hop of a packet by only one table lookup. The forwarding engine has two components: a neighbor status table and a packet forwarding procedure. The neighbor status table is maintained by the neighbor maintenance protocol. Every entry in the table corresponds to a neighbor and has three fields: NeighborMAC, OutPort, and StatusFlag. NeighborMAC is the MAC address of that neighbor, which is learnt from neighbor maintenance protocol; OutPort indicates the port that connects to the neighbor; and StatusFlag indicates if the neighbor is available.

Every entry of the neighbor status table is indexed by the neighbor's NHA value. The number of entries is 256 since NHA is one byte. The number of valid entries is $(k+1)(n-1)$, which is the number of neighbors a server has. One entry needs 8 bytes and the entire table only needs 2KB memory. The table is almost static. For each entry, OutPort never changes, NeighborMAC changes only when the neighboring NIC is replaced, and StatusFlag changes only when the neighbor's status changes.

When an intermediate server receives a packet, it gets the next hop NHA from the packet header. It then extracts the status and the MAC address of the next hop, using the NHA value as the index. If the next hop is alive, it updates the MAC addresses, NHI and header checksum of the BCube header, and forwards the packet to the identified output port. The forwarding procedure only needs one table lookup.

We have implemented the forwarding engine in both hardware and software. Our hardware implementation is based on NetFPGA [21] and part of the implementation details is given in [10]. Due to the simplicity of the forwarding engine design, our NetFPGA implementation can forward packets at the line speed and reduce the CPU forwarding overhead to zero. We believe that hardware forwarding is more desirable since it isolates the server system from packet forwarding. But the the PCI interface of NetFPGA limits the sending rate to only 160Mb/s. Hence we mainly use the software implementation, which uses server CPU for packet forwarding, in the rest experiments.

There are other BCube driver components (such as available bandwidth estimation and BCube broadcast), the BCube configuration program, and the NetFPGA miniport driver. The BCube driver contains 16k lines of C code, the NetFPGA miniport driver has 9k lines of C code, and the BCube NetFPGA implementation contains 7k lines of Verilog (with 3.5k lines of Verilog for the forwarding engine).

7.3 Testbed

We have built a BCube testbed using 16 Dell Precision 490 servers and 8 8-port DLink DGS-1008D Gigabit Ethernet mini-switches. Each server has one Intel 2.0GHz dualcore CPU, 4GB DRAM, and 160GB disk, and installs one Intel Pro/1000 PT quad-port Ethernet NIC. The OS we use is Windows Server 2003.

In our experiments, there is no disk access. This is to decouple the network performance from that of disk I/O. We turn off the xon/xoff Ethernet flow control, since it has unwanted interaction with TCP [9]. Next, we study the CPU overhead when using CPU for packet forwarding. After that, we show BCube's support for various traffic patterns.

7.4 CPU Overhead for Packet Forwarding

In this experiment, we form part of a BCube_3 with five servers. The servers are 0000, 0001, 0010, 0100, 1000. We set up four TCP connections $0001 \rightarrow 0100$, $0100 \rightarrow 0001$, $0010 \rightarrow 1000$, $1000 \rightarrow 0010$. All the connections send data as fast as they can. Server 0000 needs to forward packets for all the other four servers. We vary the MTU size of the NICs and measure the forwarding throughput and CPU overhead at server 0000.

Fig. 9 illustrates the result. The result shows that when MTU is larger than 1KB, we can achieve 4Gb/s packet forwarding, and when we increase the MTU size, the CPU usage for packet forwarding decreases. When MTU is 1.5KB, the CPU overhead is 36%. When MTU is 9KB, the CPU overhead drops to only 7.6%. Our result clearly shows that per-packet processing dominates the CPU cost.

The experiment demonstrates the efficiency of our software forwarding. In the rest of our experiments, each server forwards at most 2Gb/s and MTU is set to be 9KB. Hence packet forwarding will not be the bottleneck. Again, the software implementation is to demonstrate that BCube accelerates representative bandwidth-intensive applications. Ideally, packet forwarding needs to be offloaded to hardware, to reserve server resources (e.g., CPU, memory I/O, PCI/PCI-E bus I/O) for other computing tasks.

7.5 Bandwidth-intensive Application Support

We use 16 servers to form a BCube_1. This BCube_1 has 4 BCube_0s. Each BCube_0 in turn has 4 servers. Our testbed

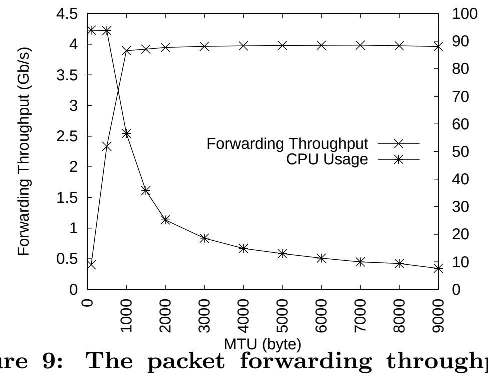


Figure 9: The packet forwarding throughput and CPU overhead with different MTU sizes.

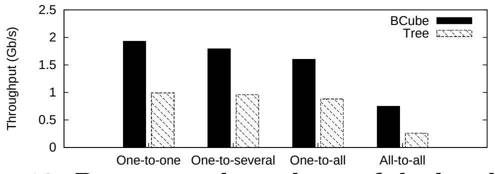


Figure 10: Per-server throughput of the bandwidth-intensive application support experiments under different traffic patterns.

uses two ports of each NIC and four ports of each mini-switch. Fig. 1(b) illustrates the topology. We perform experiments to demonstrate BCube's support for one-to-one, one-to-several, one-to-all, and all-to-all traffic patterns. In all our experiments, we set MTU to 9KB. In the one-to-x experiments, the source server sends 10GB data to the rest servers. In the all-to-all experiment, each server sends a total 10GB data to all the other servers. We compare BCube with a two-level tree. In the tree structure, 16 servers are divided into 4 groups and servers in each group are connected to a first-level mini-switch. The first-level switches are connected to a second-level mini-switch. Fig. 10 summarizes the per-server throughput of the experiments. Compared with the tree structure, BCube achieves near 2 times speedup for the one-to-x traffic and 3 times higher throughput for the all-to-all traffic.

One-to-one. We set up two TCP connections $C1$ and $C2$ between servers 00 and 13. The two connections use two parallel paths, $P1$ {00, 10, 13} for $C1$ and $P2$ {00, 03, 13} for $C2$, respectively. The total inter-server throughput is 1.93Gb/s. The total throughput is 0.99Gb/s in the tree structure, due to the fact that the two connections need to share the single network port.

In this experiment, we also study how BSR reacts to failures. For example, when we shut down server 03, server 00 discovers that server 03 is not available using its neighbor maintenance protocol (in three seconds). Then server 00 switches $C2$ to $P1$, which is the available path cached for $C2$ during the previous probing. At next BSR probing, $C2$ finds a new path $P3$ {00, 02, 22, 23, 13} and switches to it. The total throughput becomes 1.9Gb/s again.

One-to-several. In this experiment, we show that the complete graph can speedup data replication. Server A (00) replicates 10GB data to two servers B (01) and C (10). In our complete graph approach, A splits the data into two parts and sends them to both B and C, respectively. B and C then exchange their data with each other. We compare our approach with the pipeline approach using the tree structure. In the pipeline approach, A sends the data to B, and B sends the data to C. We need 89 seconds using the pipeline approach and only 47.9 seconds using our complete graph. This is 1.9 times speedup.

One-to-all. In this experiment, the server 00 distributes 10GB data to all the other 15 servers. We compare two methods. The first is the pipeline approach using the tree structure in which server i relays data to $i + 1$ ($i \in [0, 14]$). The second is our edge-disjoint spanning tree-based approach which we depict in Fig. 6. We measure the average throughput. In the pipeline approach, we get approximate 880Mb/s throughput, whereas we can achieve 1.6Gb/s throughput using our spanning tree approach.

All-to-all. In the experiment, each server establishes a TCP connection to all other 15 servers. Each server sends 10GB data and therefore each TCP connection sends 683MB. This is to emulate the reduce-phase operations in MapReduce. In the reduce phase, each Reduce worker fetches data from all other workers, resulting in an all-to-all traffic pattern. This experiment is similar to that presented in Section 7.3 of [9]. Fig. 11 plots the total throughput of BCube and the tree structure.

The data transfer completes at times 114 and 332 seconds for BCube and the tree, respectively. The per-server throughput values of BCube and the tree are 750Mb/s and 260Mb/s, respectively. BCube is about 3 times faster than the tree. The initial high throughput of the tree is due to the TCP connections among the servers connecting to the same mini-switches. After these connections terminate at around 39 seconds, all the remaining connections have to go through the root switch. The total throughput decreases significantly. There are no such bottleneck links in BCube. Compared with the result in DCell, the per-server throughput of a DCell_1 with 20 servers is 360Mb/s (Section 7.3 of [9]), which is less than 50% of BCube.

We further observe that the traffic be evenly distributed in all the 32 links in BCube. The average throughput per-link is 626Mb/s, with a standard deviation of 23Mb/s. (The sending/receving throughput per-server is 750Mb/s and the forwarding throughput per-server is 500Mb/s, hence the average throughput per-link is 626Mb/s by spitting the sending/receving and forwarding throughput into two links.) This shows that BSR does a fine job in balancing traffic. We have counted the number of path probings and path switchings. On average, every TCP connection probed the network 9.5 times and switched its path 2.1 times, and every probe completed in 13.3ms. The result shows that BSR is both robust and efficient.

8. RELATED WORK

Though network interconnections have been studied for decades [11, 18], to the best of our knowledge, none of the previous structures meets the MDC requirements and the physical constraint that servers can only have a small number of network ports. Switch-oriented structures (where servers connect to a switching fabric), such as tree, Clos network, Butterfly, and fat-tree [1, 19], cannot support one-to-x traffic well and cannot directly use existing Ethernet switches. Existing server-centric structures (where servers directly connect to other servers) either cannot provide high network capacity (e.g., 2-D and 3-D meshes, Torus, Ring)

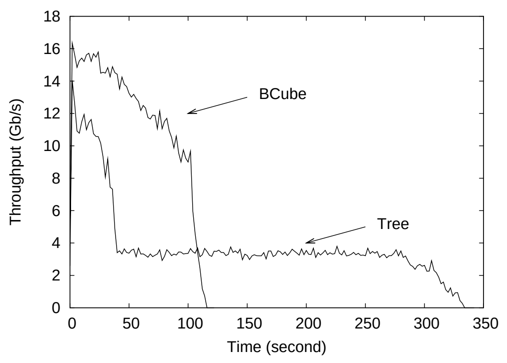


Figure 11: Total TCP throughput under BCube and tree in the all-to-all communication model.

	Tree	Fat-tree	DCell+	BCube
One-to-one	1	1	$k'+1$	$k+1$
One-to-several	1	1	$k'+1$	$k+1$
One-to-all	1	1	$\leq k'+1$	$k+1$
All-to-all(ABT)	n	N	$\frac{N}{2^{k'}}$	$\frac{n(N-1)}{n-1}$
Traffic balance	No	Yes	No	Yes
Graceful degradation	bad	fair	good	good
Wire No.	$\frac{n(N-1)}{n-1}$	$N\log_{\frac{n}{2}}\frac{N}{2}$	$(\frac{k'}{2}+1)N$	$N\log_{n}N$
Switch upgrade	No	Yes	No	No

+A level-2 ($k'=2$) DCell with $n=8$ is enough for shipping-container. Hence k' is smaller than k.

Table 1: Performance comparison of BCube and other typical network structures.

or use a large number of server ports and wires (e.g., Hypercube and de Bruijn).

In recent years, several data center network structures have been proposed [1, 7, 8, 9]. Table 1 compares BCube and other three typical structures: tree, fat-tree [1, 7], and DCell [9]. In the table, n is the number of switch ports and N is the number of servers. For one-to-x, we show the speedup as compared with the tree structure. For all-to-all, we show the aggregate bottleneck throughput.

As we show in Table 1, BCube provides much better support for one-to-x traffic than both tree and fat-tree. Tree provides the lowest aggregate bottleneck throughput since the throughput is only the capacity of the root switch. For the same n and N, the layer of switches needed by fat-tree is $\log_{\frac{n}{2}}\frac{N}{2}$, which is larger than that of BCube, $\log_{n}N$. Hence the path length and the number of wires in fat-tree are larger than those of BCube. Moreover, fat-tree does not degrade gracefully as switch failure increases (see Section 6) and it needs switch upgrade to support advanced routing and packet forwarding functionalities.

BCube is also better than DCell[9] for MDCs. DCell builds complete graphs at each level, resulting doubly exponential growth. As a result, DCell targets for Mega data centers. The traffic in DCell is imbalanced: the level-0 links carry much higher traffic than the other links. As a result, the aggregate bottleneck throughput of DCell is much smaller than that of BCube (see Section 6). Furthermore, though DCell has multiple parallel paths between two

	Cost(k$)			Power(kw)			wires
	switch	NIC	total	switch	NIC	total	No.
Tree	55	10	4161	4.4	10	424	2091
Fat-tree	92	10	4198	10	10	430	10240
DCell	10	41	4147	1.2	20	431	3468
BCube	51	41	4188	5.8	20	435	8192

Table 2: Cost, power, and wiring comparison of different structures for a container with 2048 servers. The total cost is the sum of the costs of the switches, NICs, and servers.

servers, the paths are of different lengths. This makes one-to-x speedup in DCell difficult to achieve. Of course, the benefits of BCube are not offered for free: BCube uses more mini-switches and wires than DCell. But mini-switches are affordable and wiring is a solvable issue for a container-based data center, as we have addressed in Section 5.

VL2 [8] extends the fat-tree structure by using 10G Ethernet to form its switching backbone. BCube and VL2 share several similar design principles such as providing high capacity between all servers and embracing end-systems. BCube uses only low-end switches and provides better one-to-x support at the cost of multi-ports per-server. VL2 is able to decouple IP address and server location by introducing a directory service. VL2 uses randomization whereas BCube uses active probing for load-balancing.

Table 2 presents construction cost, power consumption, and wire numbers of the four structures for a container with 2048 servers. We list the costs and power consumptions of switches, NICs, and the total numbers (which are the sums of those of switches, NICs, and servers). Each server costs $2000 and needs 200W power supply. For Tree, we use 44 48-port DLink DGS-3100-48 (which is the cheapest 48 port switch we can find) to form a two-level switch fabric. For other structures, we use 8-port DLink DGS-1008D switches. DCell, BCube, and fat-tree use 256, 1280, and 2304 switches, respectively. DCell and BCube need a 4-port NIC for each server, whereas Tree and fat-tree only use one-port NIC. A DGS-3100-48 costs $1250 and needs 103W. A DGS-1008D costs $40 and needs 4.5W. A one-port NIC and 4-port NIC cost $5 [15] and $20, and need 5W and 10W [22], respectively. Table 2 shows that the networking cost is only a small fraction of the total cost. This result is consistent with that in [14]. The construction and power costs of BCube and fat-tree are similar, but BCube uses a smaller number of wires than fat-tree. The performance and cost study clearly show that BCube is more viable for shipping-container data centers.

9. CONCLUSION

We have presented the design and implementation of BCube as a novel network architecture for shipping-container-based modular data centers (MDC). By installing a small number of network ports at each server and using COTS mini-switches as crossbars, and putting routing intelligence at the server side, BCube forms a server-centric architecture. We have shown that BCube significantly accelerates one-to-x traffic patterns and provides high network capacity for all-to-all traffic. The BSR routing protocol further enables graceful performance degradation and meets the special requirements of MDCs.

The design principle of BCube is to explore the server-centric approach to MDC networking in both topology design and routing, thus providing an alternative to the switch-oriented designs. In our future work, we will study how to scale our server-centric design from the single container to multiple containers.

10. ACKNOWLEDGEMENT

We thank Danfeng Zhang for his work on the NetFPGA Windows driver, Byungchul Park for helping us perform part of the one-to-several experiment, Lidong Zhou for generously granting us exclusive access to the testbed servers, Zhanmin Liu, Wei Shen, Yaohui Xu, and Mao Yang for helping us build the testbed. We thank Victor Bahl, Albert Greenberg, James Hamilton, Frans Kaashoek, Jim Larus, Mark Shaw, Chuck Thacker, Kushagra Vaid, Geoff Voelker, Zheng Zhang, Lidong Zhou, the anonymous SIGCOMM reviewers, and the members of the Wireless and Networking Group of Microsoft Research Asia for their feedback and comments.

11. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *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*, 2004.

[7] A. Greenberg et al. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *SIGCOMM PRESTO Workshop*, 2008.

[8] A. Greenberg et al. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, Aug 2009.

[9] C. Guo et al. DCell: A Scalable and Fault Tolerant Network Structure for Data Centers. In *SIGCOMM*, 2008.

[10] G. Lu et al. CAFE: A Configurable pAcket Forwarding Engine for Data Center Networks. In *SIGCOMM PRESTO Workshop*, Aug 2009.

[11] J. Duato et al. *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann, 2003.

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

[13] J. Hamilton. An Architecture for Modular Data Centers. In *3rd CIDR*, Jan 2007.

[14] J. Hamilton. Cooperative Expandable Micro-Slice Servers (CEMS). In *4th CIDR*, Jan 2009.

[15] J. Hamilton. Private communication, 2009.

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

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

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

[19] C. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Computers*, 34(10), 1985.

[20] J. Moy. *OSPF: Anatomy of an Internet Routing Protocol.* Addison-Wesley, 2000.

[21] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable Router Architecture for Experimental Research. In *SIGCOMM PRESTO Workshop*, 2008.

[22] Silicom. Gigabit Ethnet Server Adapters. http://www.silicom-usa.com/default.asp?contentID=711.

[23] Rackable Systems. ICE Cube Modular Data Center. http://www.rackable.com/products/icecube.aspx.

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

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

APPENDIX

A. PROOF OF THEOREM 2

From permutation Π_0 (Π_1), we establish path P_0 (P_1) by first correcting digits of A from position i_0 (i_1) to 0, then from k to $i_0 + 1$ ($i_1 + 1$). We denote P_0 and P_1 as $\{A, N_1^0, N_2^0, \cdots N_m^0, \cdots N_k^0, B\}$ and $\{A, N_1^1, N_2^1, \cdots N_m^1, \cdots N_k^1, B\}$.

We show that the intermediate server N_m^0 ($1 \le m \le k$) of P_0 cannot appear in P_1. First, N_m^0 cannot appear in P_1 at different locations other than m, otherwise, we can reach N_m^0 from two shortest pathes with different path lengthes, which is impossible. Second, N_m^0 cannot appear in P_1 at position m, because P_0 and P_1 start by correcting different digits of A. Therefore, N_m^0 cannot appear in P_1. Similarly, any intermediate server N_m^1 cannot appear in P_0.

We next show that the switches in the paths are also different. First, the switches in a single path are different, this is because these switches are at different layers. Then assume that switch S_0 in P_0 and switch S_1 in P_1 are the same, and we denote it as $< l, s_{k-1} \cdots s_l s_{l-1} \cdots s_0 >$. Due to the fact that servers in P_0 and P_1 are different, we have four servers in the two pathes that connected via this switch. But the only two servers it connects in P_0 and P_1 are $s_{k-1} \cdots s_l a_l s_{l-1} \cdots s_0$ and $s_{k-1} \cdots s_l b_l s_{l-1} \cdots s_0$. The contradiction shows that S_0 and S_1 cannot be the same. Therefore, P_0 and P_1 are two parallel paths.

B. PROOF OF THEOREM 6

In order to get the aggregate bottleneck throughput, we first calculate the average path length from one server to the rest servers using BCubeRouting. For a server A, the rest $n^{k+1} - 1$ servers in a BCube$_k$ can be classified into k groups. Group$_i$ contains the servers that are i hops away from A. The number of servers in Group$_i$ ($i \in [1, k+1]$) is $C_{k+1}^i (n-1)^i$. By averaging the path lengths of these different groups, we get the average path length is $ave\_plen = \frac{1}{n^{k+1}-1} \sum_{i=1}^{k+1} \{i C_{k+1}^i (n-1)^i\} = \frac{(n-1)N}{n(N-1)}(k+1)$.

Since links are equal in BCube, the number of flows carried in one link is $f\_num = \frac{N(N-1)ave\_plen}{N(k+1)}$, where $N(N-1)$ is the total number of flows and $N(k+1)$ is the total number of links. The throughput one flow receives is thus $\frac{1}{f\_num}$, assuming that the bandwidth of a link is one. The aggregate bottleneck throughput is therefore $N(N-1)\frac{1}{f\_num} = \frac{n}{n-1}(N-1)$.