2011 19th IEEE International Conference on Network Protocols

Building Mega Data Center from Heterogeneous Containers

Dan Li*, Mingwei Xu*, Hongze Zhao*, Xiaoming Fu†

*Computer Science Department of Tsinghua University †Georg-August-University of Goettingen
{tolidan,xmw}tsinghua.edu.cn, srzhz@hotmail.com, fu@cs.uni-goettingen.de

***Abstract*—Data center containers are regarded as the basic units to build mega data centers. In practice, heterogeneity exists among data center containers, because of technical innovation and vendor diversity. In this paper, we propose uFix, a scalable, flexible and modularized network architecture to interconnect heterogeneous data center containers. The inter-container connection rule in uFix is designed in such a way that it can flexibly scale to a huge number of servers with stable server/switch hardware settings. uFix allows modularized and fault-tolerant routing by completely decoupling inter-container routing from intra-container routing. We implement a software-based uFix stack on the Linux platform. Simulation and experiment results show that uFix enjoys high network capacity, gracefully handles server/switch failures, and brings light-weight CPU overhead onto data center servers.**

I. Introduction

Data center runs applications from on-line services such as Web mail, Web search, on-line games, to back-end infrastructural computations including distributed file systems [1], [2], structured storage [3] and distributed execution engine [2], [4], [5]. Nowadays, companies or organizations build mega data centers containing tens of thousands of, or even hundreds of thousands of servers [6], [7]. The role of a data center network (DCN) is to interconnect a massive number of data center servers, and provide routing to upper-layer applications. A desired DCN architecture should have high scalability, high fault tolerance, rich network capacity, and low deployment cost.

It has been observed that the tree architecture used in current data centers suffers from the problems of low bisection bandwidth, poor scalability and single failure point [8], [11]. Consequently, in recent years there has been a growing interest in the community to design new DCN architectures. These efforts can be roughly divided into two categories. The first category is switch-centric, which puts primary network intelligence onto switches. Specifically, either a Clos network is designed to replace the Tree hierarchy of switches, such as in PortLand [12] and VL2 [13], or shortcut links are added on the tree to enhance the network capacity [14], [15]. In contrast, the second category is server-centric, which leverages the rapid growth of server hardware such as multi-core CPU and multiple NIC ports to put the interconnection and routing intelligence onto servers. Switches are simply used as dummy crossbars or even completely eliminated. The latter category of architectures include DCell [8], FiConn [9], BCube [10], MDCube [18].

Containerized data centers are regarded as the basic units for building modern mega data centers. A containerized data center packs 1k∼4k servers as well as switches into a standard 20- or 40-feet shipping container. The container environment has the advantages in easy wiring, low cooling cost, high power density, etc. [10], [18]. In practice, heterogeneity can widely exist among data center containers. First, hardware innovation is rapid. It has been shown that data center owners usually incrementally purchase servers quarterly or yearly [16] to accommodate the service expansion. Hence, it is common that the containers bought at different times hold servers and switches with different hardware profiles. The heterogeneity can lie in the CPU capability, the memory space, the number of ports, etc. Second, there are a wide range of network architectures to build the data center containers. The containers bought from different vendors can use different network architectures for server connection inside.

In this paper, we propose uFix, a scalable, flexible, and modularized network architecture to interconnect heterogeneous data center containers. uFix requires that each server in containers reserves a NIC port for inter-container connection. Instead of using any strictly structured topology, the container interconnection in uFix is conducted in a flexible manner, so as to benefit smoothly incremental deployment of mega data centers. The connection rule is designed in such a way that with the growth of server population, we do not need to add any additional server hardware or switches, and there is no rewiring cost.

uFix provides modularized and fault-tolerant routing to accommodate the interconnection topology. The uFix inter-container routing module is completely decoupled from the intra-container routing modules. Only light overhead is necessary for routing signaling and failure handling. The advantages of uFix routing scheme are twofold. First, data center owners can buy uFix containers from different vendors without modifying the intra-container routing modules. Second, the *one-fit-all* inter-container routing can be loaded onto servers without knowing the details of intra-container routing schemes.

The work is supported by the National Basic Research Program of China (973 Program) under Grant 2011CB302900 and 2009CB320501, and the National Natural Science Foundation of China (No.61170291, No.61133006).

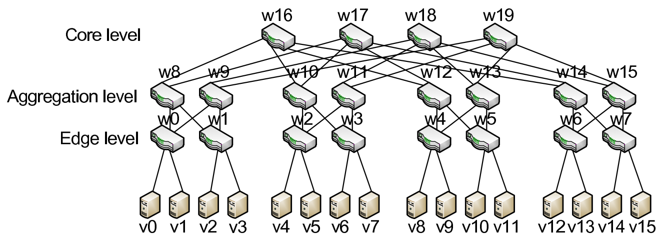


Fig. 1. A Fat-Tree architecture with 16 servers. It has three levels of 4-port switches.

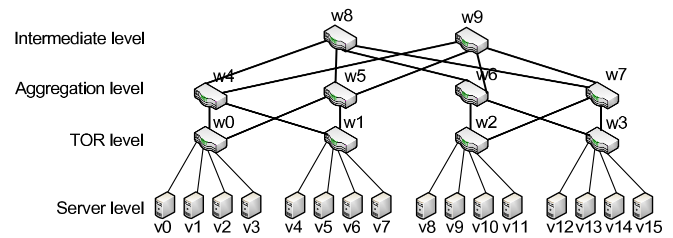


Fig. 2. A VL2 architecture with 16 servers. 10Gb switches are used in the top two levels.

We implement a software-based uFix stack on the Linux platform. By large-scale simulations and real experiments on a test bed, we find that uFix enjoys high network capacity by its interconnection rule, gracefully responds to server/switch failures, and adds little forwarding overhead to data center servers.

The rest of this paper is organized as follows. Section II discusses the related work and design goal of uFix. Section III presents the interconnection rule of uFix. Section IV designs uFix routing and forwarding schemes. Section V describes the uFix implementation and evaluation. Section VI concludes the whole paper.

II. Related Work and Design Goal

In this section, we introduce the related proposals on DCN architecture design, and discuss the design goals of uFix.

A. Related Work

As mentioned in the previous section, recently proposed DCN architectures can be divided into two categories, namely, switch-centric and server-centric. The former puts the interconnection and routing intelligence on switches, while the latter brings data center servers into the networking part and gets them involved in packet forwarding.

1) Switch-Centric Architecture: The switch-centric architecture proposals either use a totally new switch infrastructure to replace the tree structure, or make enhancements upon the tree to improve the bisection bandwidth. PortLand [12] (Fig. 1) and VL2 [13] (Fig. 2) both use low-end, commodity switches to form a three-layer Fat-Tree architecture. Each server still uses one NIC port to connect an edge-level switch. The Fat-Tree infrastructure [11], which is in fact a Clos network, can provide 1:1 oversubscription ratio to all servers in the network. The difference of VL2 from PortLand is that higher-speed switches, e.g., those with 10GE ports, are used in higher levels of the Clos network to reduce the wiring complexity.

Instead of completely giving up the traditional tree architecture, the tree enhancement approaches add shortcut links besides the Tree links, either by 60Gbps wireless link [14], or by reconfigurable optical circuits [15]. The reasoning behind is that the full bisection bandwidth provided in many advanced DCN architectures is usually an overkill for mega data centers, since it is not actually required in the scale of entire data center.

2) Server-Centric Architecture: In server-centric DCN architectures, a server uses multiple NIC ports to join the networking infrastructure and participates in packet forwarding. As demonstrated in Fig. 3, in DCell [8], a recursive, level-based structure is designed to connect servers via mini-switches and multiple server ports. DCell is highlighted by its excellent scalability, i.e., the number of servers supported increases double-exponentially with the number of server NIC ports. FiConn [9] goes one step further to limit the number of server NIC ports as two, since most of current data center servers have two built-in ports. The architecture is shown in Fig. 4. Compared with DCell, FiConn not only eliminates the necessity to add server NIC ports during data center expansion, but also reduces the wiring cost. The downside is that the network capacity in FiConn is lower than DCell, and the number of server NIC ports is restricted to two.

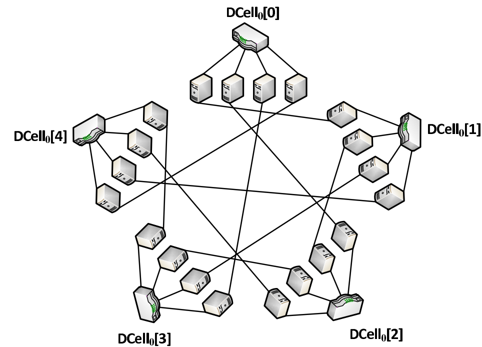


Fig. 3. A DCell(4,1) architecture with 20 servers.

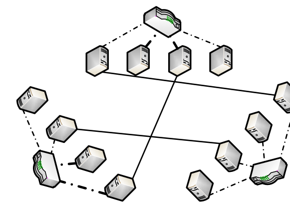


Fig. 4. A FiConn(4,2) architecture with 12 servers.

BCube [10] targets at building a data center container, typically with 1k∼4k servers. BCube is also a recursive structure, which is shown in Fig. 5. Each server uses multiple ports to connect different levels of switches. The link resource in BCube is so rich that 1:1 oversubscription ratio is guaranteed. MDCube [18] designs an architecture to interconnect

the BCube-based containers, as shown in Fig. 6. The inter-container connection and routing in MDCube are closely coupled with the intra-container architecture, so as to provide high bisection width and great fault tolerance.

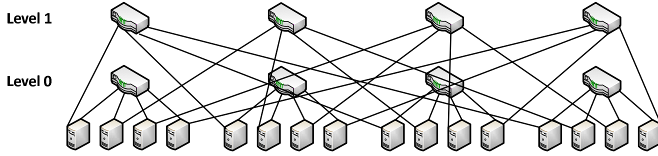


Fig. 5. A BCube(4,1) architecture with 16 servers.

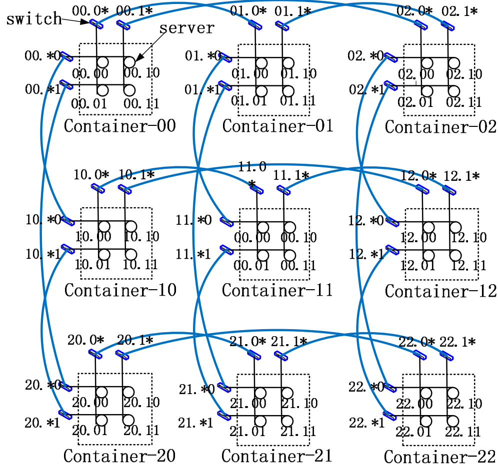


Fig. 6. An MDCube architecture with 36 servers.

B. Design Goals of uFix

It is envisioned that future mega data center is composed of data center containers, because of the multi-facet advantages of containerized data centers, including easier wiring, lower cooling cost, higher power density, etc [10], [18]. Data center containers are purchased periodically to accommodate the service expansion. Due to the hardware innovation and vendor diversity, it is common that containers are heterogeneous, in terms of both the server hardware configuration and the intra-container topology.

In this paper, we discuss how to build mega data centers from heterogeneous containers. We have the following design goals for the uFix architecture.

Scalable Interconnection of Heterogeneous Data Center Containers: uFix is designed for a mega data center, which is expected to be composed of a large number of heterogeneous containers. Note that after a data center container is sealed, the hardware configurations on servers and switches are relatively fixed. When the data center scales up, the data center owners are reluctant to open the containers and change the server/switch hardware settings, such as adding NIC ports or CPU cores. For inter-container connection, a number of NIC ports (from servers or switches) are exposed outside for wiring. Hence, uFix is expected to expand to large scale with stable physical settings on servers/switches in containers.

Support for Incremental Server Expansion: The data center size is usually incrementally expanded to accommodate service growth. Though strictly structured topologies can provide high network performance by fully leveraging the server/switch configurations, they usually cannot smoothly handle incremental growth of data center scale. For example, the Fat-Tree structure composed of k-port switches can achieve 1:1 oversubscription ratio when the number of servers is $\frac{k^3}{4}$. But if the number of servers marginally exceeds this magic value, it is still an open issue how to connect them with network performance guarantee. In uFix, since the interconnection units are containers, we need to pay special attention to gracefully embracing server expansion.

Decoupling Inter-container Architecture from Intra-container Architecture: We expect that data center owners will purchase containers at different times from different vendors. Due to the rapid technical innovation, various interconnection topologies and routing schemes can be used within these containers. An inter-container interconnection architecture will be operationally expensive if it needs to accommodate the specific intra-container structures and routings. It is desirable if a solution can regard every container as a black box, and completely decouple inter-container architecture from intra-container architectures. This kind of *generic* inter-container architecture not only makes container interconnection easy, but also benefits the independent development of intra-container technologies.

No existing DCN architectures can achieve all the design goals above, since they focus on optimizing a homogeneous set of servers, by assuming that each server uses the same number of NIC ports for interconnection and bears almost equivalent forwarding burden. The most related architecture with uFix is MDCube. However, the containers connected in MDCube are all BCube containers, and the inter-container architecture is closely coupled with the intra-container BCube architecture.

III. uFix Interconnection

In this section, we discuss how uFix connects the heterogeneous data center containers to form mega data centers.

A. Interconnection Overview

A containerized data center packs servers and switches into a standard 20- or 40-feet shipping container, and offers networking, power, and cooling services to them. The servers inside a container, with a typical size of 1k$\sim$4k, are usually homogeneous so as to conduct easy interconnection when manufactured. Note that a specific network architecture usually matches certain server configuration. For example, server-centric data center architectures often require multiple NIC ports for interconnection, as well as high CPU capacity for servers to participate in packet forwarding. But for switch-centric architectures, the requirement on server hardware is relatively low. Hence, if the servers have more than 4 ports and strong CPU, BCube is a nice choice to achieve full bisection bandwidth within the container. As for servers with limited CPU capacity and NIC ports, tree or Fat-Tree can be used.

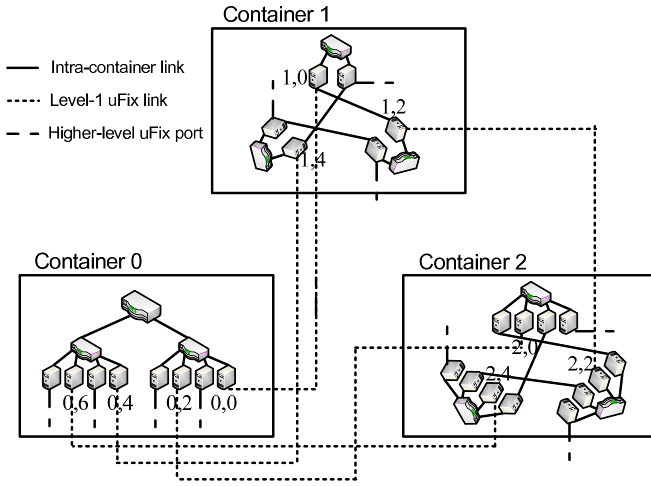


Fig. 7. A level-1 uFix domain constructed upon three containers. Container 0 uses a Tree structure composed of 8 servers, Container 1 is a DCell_1 with 6 servers, and Container 2 is a FiConn_1 with 12 servers. There are in total 5 level-1 uFix links connecting the 3 uFix containers.

One possible way for interconnecting containers is to use existing structured topologies such as Fat-Tree or BCube, treating each container as a unit. However, the aggregate bandwidth within a container is very large, and this kind of interconnection will make the inter-container bandwidth quite low, even using higher-speed switches for inter-container connection. Besides, this kind of interconnection requires additional switches and brings in significant hardware investment.

Therefore, we turn to directly interconnecting the containers by the devices inside the containers. There are two design choices, namely, switch based connection and server based connection. We choose server based interconnection for two reasons. First, we can leverage the increasing NIC port density on servers for network connection [10], [9], and do not require additional switches. Second, it is much easier to program on servers to load inter-container intelligence, especially because low-end switches are commonly used in current data center design. Note that we can use the forwarding engine such as ServerSwitch [19] on servers to offload the forwarding overhead on server CPU.

In order to accomplish server based interconnection in uFix, it is *required* that in each container, a number (at least 2) of servers with available NIC ports are reserved. We call these servers *uFix proxy candidates*. Note that each uFix proxy candidate only needs to reserve one NIC port. If uFix is widely accepted in future, this requirement can be regarded as a standard for manufacturing data center containers. The NIC port reserved in the uFix proxy candidate for inter-container connection is called a *uFix port*, while the other ports on the server are called *container-specific ports*.

B. Interconnection Algorithm

There are two challenges for inter-container connection in uFix. First, the server settings in a container are relatively stable after the container is manufactured, thus we need to scale uFix with fixed uFix ports in a container. Second, it is desirable for uFix to gracefully support incremental expansion of data center size. In other words, uFix data center can be gradually built up in fine granularity, without breaking the interconnection rule. It is difficult to use previously proposed strictly structured topologies to achieve this goal.

We interconnect containers in uFix in a flexible way. We iteratively define a *level-l uFix domain* as follows.

- A level-0 uFix domain is a container.
- A level-l ($l > 0$) uFix domain comprises a number of lower-level uFix domains, with the highest level of $l-1$.

The process of connecting lower-level uFix domains to form a level-l uFix domain is called a *level-l uFix interconnection*. The new links added during level-l uFix interconnection are called *level-l uFix links*. A server with a level-l uFix link is called a *level-l uFix proxy*. For a level-l uFix proxy, the highest-level uFix domain it belongs to before the level-l uFix interconnection is called the *designated domain* of the level-l uFix proxy. Note that each uFix proxy has only one designated domain, but multiple uFix proxies can share the same designated domain.

Now we describe the procedure of a level-l uFix interconnection. There are m lower-level uFix domains to connect, which we call *pre-connecting uFix domains*. They are denoted sequentially from 0 to $m-1$, and the highest level among them is $l-1$ ($l > 0$). Generally speaking, the new level-l uFix domain is formed by directly connecting part of the uFix proxy candidates in the m pre-connecting uFix domains. Suppose the total number of uFix proxy candidates in pre-connecting uFix domain i ($0 \leq i < m$) is x_i, and the number of uFix proxy candidates in uFix domain i used for level-l uFix interconnection is u_i ($u_i \leq x_i$). We define *level-l uFix interconnection degree* as $g = \frac{\sum_{i=0}^{m-1} u_i}{\sum_{i=0}^{m-1} x_i}$. There are two requirements.

$$\sum_{i=0}^{m-1} x_i - \sum_{i=0}^{m-1} u_i \geq 2 \tag{1}$$

$$u_i \geq m-1, \text{for any } i \tag{2}$$

Inequation (1) aims for higher-level extensibility of the resultant level-l uFix domain, for which we reserve at least two uFix proxy candidates. Hence, during the expansion of uFix scale, no additional server-side hardware, additional switches, or rewiring is required. It not only saves the building cost of data centers, but also makes incremental deployment much easier. Inequation (2) guarantees that the pre-connecting uFix domains can be connected as an *intensive mesh* if every uFix domain is regarded as a virtual node. Here the intensive mesh indicates that there is at least one level-l uFix link between every two pre-connecting uFix domains. In this way, we can well control the diameter of the resultant level-l uFix domain. If Inequation (2) cannot be satisfied, the m pre-connecting uFix domains will not be connected in the same level. Instead, a hierarchical solution is used. Note that each uFix proxy

candidate server will be used only once during the inter-container connection process, since it reserves only one NIC port.

We can see a tradeoff for the setting of a level-l uFix interconnection degree. On one hand, higher degree means more level-l uFix links, which brings greater inter-domain networking capacity and better fault tolerance in the resultant level-l uFix domain. On the other hand, smaller degree leaves more uFix ports in the resultant level-l uFix domain for higher-level interconnection. The data center owner has the flexibility to set the tradeoff by taking the wiring complexity, the inter-domain network capacity as well as the extensibility for higher-level connection into account.

Fig. 8 describes the procedure to connect different uFix domains. It includes two steps. In the first step called *link-counting*, we calculate the number of level-l uFix links to be added between any two pre-connecting uFix domain p and q, denoted as e_{pq}. The second step is called *link-connecting*, in which we choose specific level-l uFix proxy candidates from each pre-connecting uFix domain and complete the wiring.

The link-counting step is illustrated in Lines 1-16. First, we compute the number of full meshes that can be formed among the pre-connecting uFix domains (Lines 1-6). There might be remaining level-l uFix proxy candidates in a uFix domain i, and we try to add links among them in the following way (Lines 7-16). If there are still remaining level-l uFix proxy candidates among at least two uFix domains (Line 7), we continue to add level-l uFix links. In each round, we choose the uFix domain with the most remaining level-l uFix proxy candidates, say, uFix domain p (Line 8), and add one level-l link between uFix domain p and each of the other uFix domains (Lines 9-15).

Lines 17-30 describe the link-connecting step, which selects the actual level-l uFix proxy candidates from each pre-connecting uFix domain to complete the wiring. Given the interconnection degree and the number of links to add between any two uFix domains calculated by the link-counting step, we try to avoid connection hot spots by *interleaving port selection* and *interleaving connection*.

Interleaving Port Selection: Given the level-l uFix interconnection degree, we try to disperse the level-l uFix links added on each uFix domain among all the level-l uFix proxy candidates. For instance, assume the available level-l uFix proxy candidates on a uFix domain are servers 0-200[1], and we use only 50 servers for level-l uFix interconnection. Our choice is to spread the level-l uFix proxy candidates chosen on servers 0, 4, 8, 12, ..., 196, instead of choosing the servers 0-49.

Interleaving Connection: We try to disperse the level-l uFix links added towards the same uFix domain among all the uFix ports used. Using the example above, in the 50 actually selected level-l uFix proxy candidates from a pre-connecting ufix domain, assume 25 servers are to connect uFix domain 1

[1]We assume that the servers with closer sequence numbers are physically located more closely, which holds in most cases in currently proposed DCN architectures.

/*Input:
m: number of pre-connecting uFix domains;
x_i: number of available uFix ports in uFix domain i;
u_i: number of available uFix ports in domain i planned to use for level-l interconnection.*/

uFixInterConnection(){
Link-counting:
01 *meshN* = $\lfloor \frac{\mathbf{min}\{u_i\}}{m-1} \rfloor$;
02 for ($i = 0$; $i < m$; i++){
03 v_i = u_i-*meshN*$*(m-1)$;
04 for ($j = 0$; $j < m$ && $j \neq i$; j++)
05 e_{ij} = *meshN*;
06 }/*for*/
07 while ($\sum v_i >$**max**$\{v_i\}$){
08 p = **argmax**$_i\{v_i\}$;
09 $U = \{i|0 \leq i < m$ && $v_i > 0$ && $i \neq p\}$;
10 while ($v_p > 0$ && $U \neq \phi$){
11 q = **argmax**$_i\{v_i|i \in U\}$;
12 $U = U \setminus q$;
13 v_p--; v_q--;
14 $e_{pq}++$; $e_{qp}++$;
15 }/*while*/
16 }/*while*/
Link-connecting::
17 for ($i = 0$; $i < m$; i++){
18 $f_i = 0$; $c_i = 0$
19 $g_i = \frac{u_i - v_i}{x_i}$;
20 }/*for*/
21 while ($\sum e_{ij} > 0$){
22 for ($i = 0$; $i < m$; i++){
23 for ($j = i+1$; $j < m$ && $e_{ij} > 0$; j++){
24 connect available uFix proxy candidate f_i in domain i and available uFix proxy candidate f_j in domain j;
25 c_i++; c_j++; $e_{ij}--$; $e_{ji}--$;
26 while ($f_i == 0 \parallel \frac{c_i}{f_i} > g_i$) f_i++;
27 while ($f_j == 0 \parallel \frac{c_j}{f_j} > g_j$) f_j++;
28 }/*for*/
29 }/*for*/
30 } /*while*/
}/*uFixInterConnection*/

Fig. 8. The procedure to interconnect uFix domains.

and the other 25 servers are to connect uFix domain 2. We do not let servers 0, 4, 8, ..., 96 connect uFix domain 1, and let servers 100, 104, 108, ..., 196 connect uFix domain 2. Instead, we choose servers 0, 8, 16, ..., 192 to connect uFix domain 1, while servers 4, 12, 20, ... 196 to connect uFix domain 2.

In the algorithm of Fig. 8, we use f_i to denote a level-l uFix proxy candidate in uFix domain i (Line 18), and g_i to represent the ratio of the number of actually used level-l uFix proxy candidates over that of all available level-l uFix proxy candidates in domain i (Line 19). We divide the link-connecting step into multiple rounds (Lines 21-30). In each round, we add one level-l link between every two uFix domains i and j if the value of e_{ij} allows. After adding a

level-l link in a uFix domain i, we may skip a number of next available level-l uFix proxy candidates based on the value of g_i (Lines 26-27).

Network Diameter: The primary limitation on the growth of server population in uFix comes from the network diameter, which is required to be low for sake of end-to-end delay requirement of data center applications.

Theorem 1: If the highest diameter of a container is d_0, the diameter of a level-l uFix domain, d_l, is upper-bounded by $d_0 * 2^l + 2^l - 1$. Here the diameter represents the number of hops.

Proof: We prove it by induction.

For $l = 0$, the upper-bound certainly holds true.

If the upper bound holds for any level-(l-1) uFix domain, i.e., there is $d_{l-1} \leq d_0 * 2^{l-1} + 2^{l-1} - 1$. Then, for a level-$l$ uFix domain, any two servers can communicate with each other in the following way: first, traversing its level-(l-1) uFix domain to reach the uFix proxy with a level-l uFix link towards the destination level-(l-1) uFix domain; second, crossing the level-l uFix link; third, traversing the destination level-(l-1) uFix domain to arrive at the destination server. Hence, there is $d_l \leq 2 * d_{l-1} + 1 \leq d_0 * 2^l + 2^l - 1$. ■

The data center containers are usually low-diameter structures. For example, the diameter of Fat-Tree structure is 6, while that of a BCube(8,3) is 8. When building uFix, data center owners can intentionally control the uFix levels if network delay is a major concern, by setting the interconnection degree in the uFix interconnection of each level. For example, in a level-3 uFix network with $d_0 = 6$, the maximum diameter of the network, d_3, is 55. However, the actual number of hops for communication between two servers should be much less than d_3 because of two reasons. First, the network diameters of many containers are less than d_0. Second, the rich inter-container links guarantee that there are multiple paths between two servers, hence the longest path is usually not used in practice.

Wiring: The wiring is not an issue within containers. We only discuss inter-container links in uFix. We can put these links on top of the containers. Gigabit Ethernet copper wires can be 100 meters long, which can connect tens of containers. It fits most cases for inter-container connection. Data center owner has the flexibility for putting uFix links among uFix domains. For instance, in Fig. 7, other than the interconnection approach shown in the figure, container 0 can also use ports [0,0] and [0,4] to connect container 2, while using ports [0,2] and [0,6] to connect container 1. In practice, this kind of flexibility can greatly reduce the wiring complexity compared with strictly structured topologies, which require hard wiring rules. When the uFix domain is large, we can intentionally choose physically-close ports for inter-container connection. In case when the distance between containers is so far that it exceeds the limit of copper wires, the optical fibres are introduced. Note that during the extending process of uFix, *no* rewiring is required since uFix can connect uFix domains of any levels. For example in Fig. 7, when we extend the level-1 uFix domain to higher-level uFix domains, we do not need to change all the existing links. Instead, we only adds links in the remaining uFix proxy servers which are selected for higher-level interconnection.

<table>
<tr><td colspan="2">IP</td></tr>
<tr><td colspan="2">uFix routing and forwarding</td></tr>
<tr><td rowspan="2">uFix port</td><td>DCell, BCube, FiConn (optional)</td></tr>
<tr><td>Container-specific ports</td></tr>
</table>

Fig. 9. uFix routing and forwarding engine. It works below IP layer and above container-specific forwarding layer.

IV. UFIX ROUTING AND FORWARDING

In this section we design the routing and forwarding mechanism in uFix, based on the container interconnection rule presented in Section III.

A. Basic Idea

There are two key challenges to design uFix routing and forwarding mechanism. First, we need to decouple inter-container architecture from intra-container architectures, which is one of our design goals. Specifically, the inter-container routing and forwarding in uFix is carried on without understanding the syntax and detail of intra-container architectures. Second, we need to carefully address uFix proxy failures, since failures are the norm instead of the exception in data centers. To some extent, the relationship between intra-container and inter-container routing/forwarding in uFix is analogous to intra-AS and inter-AS routing/forwarding in the Internet. We do not use BGP routing because in the Internet oriented routing/forwarding framework, the inter-AS forwarding engine and intra-AS forwarding engine are the same, i.e., IP forwarding based on longest-prefix matching. The inter-container routing scheme in uFix should assist in the inter-container forwarding scheme which is decoupled from intra-container forwarding.

We design a progressive routing and forwarding scheme upon the hierarchical uFix structure. The inter-container routing and forwarding functionality is realized by a *uFix routing and forwarding module* inserted onto uFix servers, without updates on intra-container routing modules. Fig. 9 illustrates the routing and forwarding framework on a uFix server. The uFix routing and forwarding module works below IP layer but above the container-specific routing layer. In addition, uFix routing and forwarding module can directly send/receive packets to/from the uFix port.

B. Route Signaling

We aggregate containers within a uFix domain into an IP segment. The IP segment shares an IP prefix as long as possible and the uFix domain is uniquely mapped to the IP prefix. Besides, the IP segment of a lower-level uFix domain must be the subset of that of the higher-level uFix domain it belongs to. In practice, a private IPv4 address space such as 10.*.*.* has a space sized of approximately 16 million.

Even considering the address waste during IP segmentation, the space is sufficient for most data centers. For even larger data centers, IPv6 address can be configured.

Each uFix proxy *periodically* sends an *IP prefix exchange* message to its physically neighboring (directly connected) uFix proxy. The message just advertises the IP prefix of its designated domain. After a uFix proxy receives the *IP prefix exchange* message from its neighboring uFix proxy, it immediately generates an *IP prefix advertisement* message and broadcasts the message within its own designated domain. The IP prefix advertisement message generated by a level-l uFix proxy i contains two items: the IP prefix of the designated domain of its neighboring uFix proxy j, which is obtained by the last received IP prefix exchange message from j, and the uFix proxy level of i, i.e., l. When broadcasting the IP prefix advertisement message, a uFix proxy uses the broadcast IP address of its designated domain as the destination IP address, so servers beyond its designated domain will not receive this message.

Each server maintains a *uFix proxy table*, which is composed of the following fields: IP prefix, uFix level and uFix proxy set. When a server receives an IP prefix advertisement message, it updates the uFix proxy table as follows. If the IP prefix advertised in the message is not contained in the uFix proxy table, an entry for this IP prefix is added. Then, the uFix level field of this entry is filled with the uFix proxy level in the message, and the source IP address of the message is added into the uFix proxy set for this entry. But if the IP prefix advertised in the message is already in the uFix proxy table, the uFix level field of the corresponding entry is updated, and similarly, the source IP address of the message is added into the uFix proxy set.

We use $U_l(s)$ to denote the level-l uFix domain a server s belongs to. Fig. 10 illustrates an example of a uFix proxy table on a certain server s_i, for which $U_0(s_i)$ is 10.1.1.0/24. From the table, we can see that there are 4 level-1 uFix proxies in $U_0(s_i)$. The neighboring uFix proxies of two of them share a designated domain of 10.1.0.0/24, while the neighboring uFix proxies of the other two share a designated domain of 10.1.2.0/23. We can thus infer that there are in total three containers in $U_1(s_i)$, and the IP prefix of $U_1(s_i)$ is 10.1.0.0/22. In $U_1(s_i)$, there are 9 level-2 uFix proxies, connected to three lower-level uFix domains in $U_2(s_i)$. Similarly, there are 16 level-3 uFix proxies in $U_2(s_i)$, connecting four other lower-level uFix domains in $U_3(s_i)$.

From the construction process, we can conclude that the uFix proxy table on a server s has the following characteristics. First, the IP prefixes of every two entries do not have overlapping space. Second, the combination of IP prefixes of all entries in the uFix proxy table and that of $U_0(s)$ is just the IP prefix of the highest-level uFix domain comprising s. Third, for a level-l entry from the uFix proxy table, the IP address of each uFix proxy in the proxy set lies within the IP prefix of a lower-level entry.

IP prefix	uFix level	Proxy set			
10.1.16.0/20	3	10.1.0.3	10.1.4.3	10.1.8.3	10.1.12.3
10.1.32.0/20	3	10.1.1.3	10.1.5.3	10.1.9.3	10.1.13.3
10.1.48.0/20	3	10.1.2.3	10.1.6.3	10.1.10.3	10.1.14.3
10.1.64.0/18	3	10.1.3.3	10.1.7.3	10.1.11.3	10.1.15.3
10.1.4.0/22	2	10.1.0.2	10.1.1.2	10.1.2.2	
10.1.8.0/22	2	10.1.0.127	10.1.1.127	10.1.3.2	
10.1.12.0/22	2	10.1.0.252	10.1.1.252	10.1.2.127	
10.1.0.0/24	1	10.1.1.1	10.1.1.17		
10.1.2.0/23	1	10.1.1.9	10.1.1.25		

Fig. 10. The uFix proxy table on a server from the container with IP prefix of 10.1.1.0/24. The highest uFix domain level is 3.

C. Forwarding Procedure

uFix forwarding takes a progressive way. When a source server s_s initiates sending a packet to a destination server, the uFix forwarding module on it intercepts the packet. Assume the destination IP address of the packet is s_d. The uFix forwarding module first looks up the uFix proxy table to find the entry whose IP prefix contains s_d. We first discuss the case when s_d is beyond $U_0(s_s)$. Based on the characteristic of uFix proxy table, the entry is unique and we assume its uFix level is l. Typically, there are multiple uFix proxies in the proxy set of this entry. To avoid out-of-order packets for the same flow, we use a Hash function to map the five tuple of the packet to a uFix proxy in the proxy set, say, p_l. Then, we use p_l as the destination IP address and look up the uFix proxy table recursively as above until we finally get a uFix proxy within $U_0(s_s)$, p_c. We denote the inverse list of uFix proxies we get as $P_0 = \{p_c, ..., p_l\}$. If s_d is within $U_0(s_s)$, we have P_0 =*NULL*. By concatenating P_0 and s_d, we get $P_s = P_0 \cup \{s_d\}$. Next, s_s adds a ***uFix routing field*** in the packet to contain P_s and changes the destination IP address of this packet to p_c. After that, s_s passes the packet down to the container-specific routing layer (if any) or directly to the MAC layer.

When an intermediate server s_i receives a uFix packet, it forwards the packet based on the following scenarios.

Case 1: If s_i is not the entry server of $U_0(s_i)$, exit server of $U_0(s_i)$ or the final destination server, the packet will be forwarded by the container-specific routing layer. In this case, the packet will not be delivered to the uFix forwarding module.

Case 2: If s_i is the entry server of $U_0(s_i)$, it will receive the packet from its uFix port and the packet will be directly delivered to the uFix forwarding module on s_i. If s_i is the exit server of $U_0(s_i)$ or the final destination server, the packet arrives at s_i via intra-container routing and it will also be delivered to the uFix forwarding module for upper-layer processing. In one sentence, whether s_i is the entry server of $U_0(s_i)$, the exit server of $U_0(s_i)$, or the final destination server, the uFix forwarding module will receive the packet.

In this case, the uFix forwarding module on s_i extracts the

```
/*Function getPLFromTable(v) returns the uFix proxy list towards
server v by looking up the uFix proxy table on s. If v is within the
container including s, the function returns NULL.*/

uFixFromUpLayer(s, pkt){
01    dst = destination IP address of pkt;
02    P_0 = getPLFromTable(dst);
03    P_s = P_0 ∪ dst;
04    pkt.uFixField = P_s;
05    change destination IP address of pkt as P_s[0];
06    pass pkt to lower layer;
07    return;
}/*uFixFromUpLayer*/

uFixFromDownLayer(s, pkt){
01    P' = pkt.uFixField;
02    if (P'[0] == s){
03        if (|P'| == 1){
04            deliver pkt to upper layer;
05            return;
06        }/*if*/
07        else{
08            remove P'[0] from P';
09            forward pkt via uFix port;
10            return;
11        }/*else*/
12    }/*if*/
13    P_0 = getPLFromTable(P'[0]);
14    P_s = P_0 ∪ P';
15    pkt.uFixField = P_s;
16    change destination IP address of pkt as P_s[0];
17    pass pkt to lower layer;
18    return;
}/*uFixFromDownlayer*/
```

Fig. 11. uFix forwarding module on a server s. *uFixFromUpLayer*() processes packets passed from IP layer, and *uFixFromDownLayer*() is responsible for packets delivered from lower container-specific layer or uFix port.

uFix routing field from the packet, gets the proxy list, P', and checks the first item of P', namely, $P'[0]$.

Case 2.1: If $P'[0] = s_i$ and there is only one item in P', s_i must be the final destination server of the packet. Then s_i delivers the packet to upper layers.

Case 2.2: If $P'[0] = s_i$ and there is more than one items in P', it indicates that s_i is the exit server (uFix proxy) of $U_0(s_i)$. In this case, s_i removes $P'[0]$ from the proxy list, and directly forwards the packet to the neighboring uFix proxy via the uFix port. Note that the outgoing packet does not have any container-specific header.

Case 2.3: If $P'[0]! = s_i$, it is an indication that s_i is the entry server of $U_0(s_i)$. In this case, s_i first recursively looks up the uFix proxy table to find the proxy list towards $P'[0]$, denoted as P_0, just as the original source server of the packet, s_s, does. s_i concatenates P_0 and P' to get a new proxy list P_i, and modifies the uFix routing field as P_i. Next, s_i changes the destination IP address of the packet as the first item of P_i, and passes the packet down to the container-specific routing layer (if any), or directly to the MAC layer.

Fig. 11 describes the procedure of uFix forwarding on a uFix server s for packet *pkt*, as presented above. *uFixFromUpLayer*() processes packets passed from IP layer, and *uFixFromDownLayer*() is responsible for packets delivered from lower container-specific layer or uFix port.

D. Failure Handling

Failure is common in data centers since they are constructed upon commodity devices. We discuss the failure handling on uFix proxies for inter-container routing and forwarding.

IP prefix advertisement is a natural way to maintain the fresh states of uFix proxies. However, the frequency of sending IP prefix advertisement message is set relatively low to avoid broadcast storm. Another naive solution is that every server actively probes the uFix proxies in the uFix proxy table to update their states. Unfortunately this is also infeasible. In many cases, there are a large number of uFix proxies in the uFix proxy table, and this approach will cause considerable probing overhead to the network. As a result, we choose an *on-demand failure handling* solution which is driven by traffic demands. Hence each server does not need to probe and maintain the status of all uFix proxies. It works on a uFix server s as follows.

If s is the source server or the entry server of a container, when it calls *getPLFromTable*(.) (refer to the algorithm of Fig. 11) to select the uFix proxy list towards server v for the first packet of a flow, it selects the highest-level, say, level-l uFix proxy towards v, and triggers a probe to it. If receiving no response from v, another level-l uFix proxy is randomly selected from the same uFix proxy set and probed. When a higher-level proxy is successfully probed, we begin to select the lower-level proxies. If all proxies in a uFix proxy set fail, the selection is backed off to the higher-level uFix proxy set to choose another uFix proxy. If all proxies in the highest-level uFix proxy set fail, a *uFix Route Error Message* is sent back to the source server to route for another round from scratch. Since the round-trip time for probing between data center servers are usually in the orders of microseconds, the duration of on-demand failure handling on server s will be affordable. A timeout threshold is also set to trigger the sending of uFix route error message.

During the transmission of the flow, server s also makes periodical probes to the selected uFix proxies. If failure happens during transmission, s makes another round of uFix proxy selection to switch the flow to a live path.

If s is the exit proxy of a container and it detects the failure of its uFix link or uFix neighboring server (this can be obtained by keep-alive message between every two neighboring uFix proxies), it plays as if it is the entry proxy of the container and conducts similar process as an entry proxy, so as to choose another uFix proxy for packet forwarding.

V. Implementation and Evaluation

In this section we present the implementation of uFix, as well as the evaluations on uFix performance by both simulations and experiments.

A. Implementation

We have implemented a software based uFix stack on Linux Platform. Note that container-specific routing protocols may need to parse the destination/source IP address of a packet to make corresponding forwarding decisions. Hence, we cannot add another packet header between IP header and container-specific routing header. Otherwise, it will destroy the parsing of destination/source IP address on the container-specific routing module, which violates our design goal of decoupling inter-container architecture from intra-container architecture. In our implementation, we overcome this problem by adding a special option type in the IP option header to carry the uFix routing field.

We use a Linux NetFilter module to realize the uFix forwarding functionality, which is easy to be loaded into or unloaded from the system kernel. For packet forwarding, the uFix module intercepts packets from/to the uFix ports, looks up the uFix proxy table and modifies the uFix routing field in the IP option header. For packet sending/receiving, the uFix module also intercepts local outgoing/incoming packets, and adds/removes the uFix routing field before delivering the packet to lower/higher layer. There are three major components in our implementation architecture, namely, uFix proxy table maintenance, uFix forwarding engine, and the packet sending/receiving part interacting with the TCP/IP stack.

B. Simulations

We conduct simulations to evaluate the uFix interconnection and routing schemes.

Simulation Setting: We set the simulation environment as a level-1 uFix domain composed of 3 containers, namely, a BCube(8,2) with 512 servers, a DCell(4,2) with 420 servers, and a Tree architecture with 512 servers. The level-1 uFix interconnection degree is set as a variable parameter. The data transmission speed at all server NICs is 1Gbps. The speed of all switch ports in BCube and DCell is 1Gbps. For switches in the Tree, the lowest-level switches use 8 ports with 1Gbps to connect servers, and use a 10Gbps port to connect upper-layer switches. The ports of all higher-level switches in the Tree are 10Gbps.

We generate all-to-all traffic pattern, in which each uFix server sends packets to all other ones. Since uFix does not support full bisection bandwidth within the entire data center, we just use all-to-all communication to measure the performance of the interconnection scheme as well as the fault-tolerant routing in uFix. We consider the metric of Aggregate Bottleneck Throughput (ABT), which is defined as the throughput of the bottleneck flow multiplying the number of all flows in the network. ABT is a good indication of the task finish time for all-to-all communication with nearly equal amount of data to shuffle among servers.

Interleaving Port Selection and Interleaving Connecting: We first compare the ABTs with and without our interleaving port selection/interleaving connecting approach in uFix interconnection scheme. Fig. 12 shows the simulation results. We find that interleaving port selection and interleaving connecting

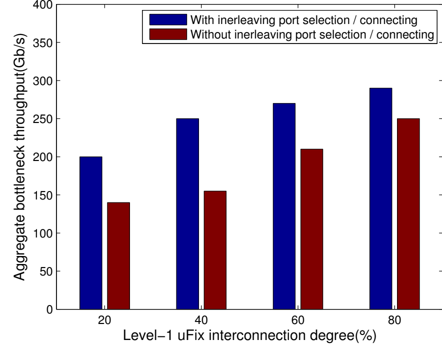
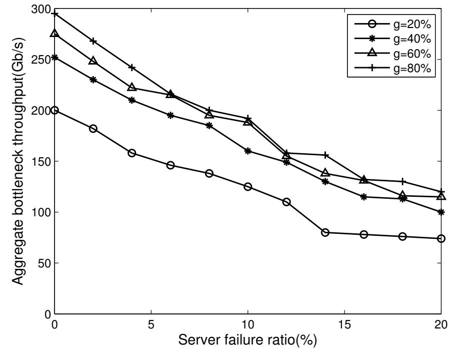
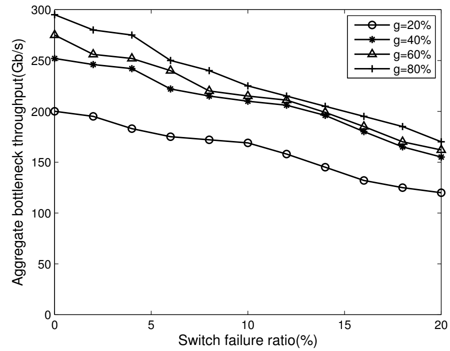


Fig. 12. ABT with/without interleaving port selection and interleaving connecting.



(a)



(b)

Fig. 13. ABT against failures (g: level-1 uFix interconnection degree). (a) server failure; (b) switch failure.

can improve the ABT by 11.5∼53.3% in different settings of level-1 uFix interconnection degrees. This is because by interleaving port selection and interleaving connecting, traffic is spread more equivalently into different locations of a uFix domain, which helps raise the bottleneck throughput. Another observation is that higher level-1 uFix interconnection degree brings higher ABT. It follows our expectation since more level-1 uFix links among servers are provided to carry the traffic.

Fault Tolerance: Then we evaluate the fault tolerant routing in uFix. The failure ratio (of servers and switches) is set from 0% to 20%. Fig. 13(a) and Fig. 13(b) show the results for server failures and switch failures respectively. Graceful

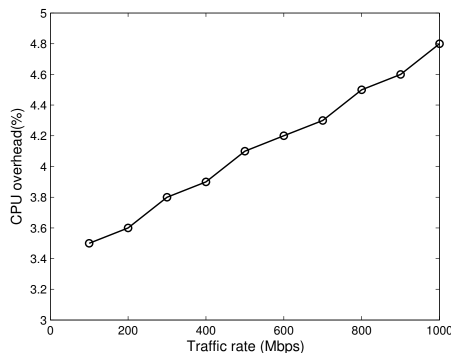


Fig. 14. CPU overhead of uFix proxy against packet forwarding rates.

performance degradation is achieved in both types of failures. This is due to the multiple connections among containers and accordingly the large number of paths between any pair of servers. The performance degradation with switch failure is even smoother, because server failure results in less flows in the all-to-all traffic pattern. But in face of switch failure, the rich connectivity in uFix ensures that there is always a path for the flow between two servers.

C. Experiments

We carry out experiments to study the packet forwarding speed and uFix forwarding load, on a test bed composed of 16 servers. Each server has one 2.8G Intel(R) Core(TM)2 Duo CPU, 2GB DRAM, and two Realtek RTL8111/8168B PCI Express Gigabit Ethernet NICs. We install Ubuntu Linux 9.10 (karmic) with kernel 2.6.31-14-generic on all servers. The servers are divided into two parts. 4 servers are connected as a tree, and the other 12 servers form a FiConn_1 network. The switches used in both Tree and FiConn are H3C S5100-24P-SI with Gigabit Ethernet ports. A uFix link is added to connect a server s_t from the tree and a server s_f in the FiConn. Hence, s_t and s_f are the two level-1 uFix proxies. Servers in the FiConn network are installed with both FiConn and uFix stacks, while those in the tree are only equipped with uFix under TCP/IP.

We generate traffic from tree servers to FiConn servers, and measure the CPU load on the uFix proxy s_f. The MTU is set as 6000 bytes. The traffic passing s_f is tuned as different rates. Fig. 14 shows the result. We can find the CPU utilization on s_f is less than 5%, even when it forwards with the speed close to 1Gbps. When the traffic rate increases, uFix consumes moderately more CPU power. It demonstrates that the packet forwarding burden uFix brings is quite light-weighted, affordable on most data center servers. For uFix networks with larger scale, the CPU overhead will not be much higher, as long as the link rates are limited to 1Gbps. If we employ the hardware-based server forwarding engines such as ServerSwitch [19], the forwarding overhead will be even lower.

VI. Conclusion

In this paper we proposed uFix, which aims to inter-connect heterogeneous data center containers to build mega data centers. In uFix, the inter-container connection rule is designed in such a way that we do not need to update the server/switch hardware within containers when the data center size grows. In addition, it benefits the incremental deployment of data centers by flexibly connecting the containers. The uFix routing and forwarding intelligence is completely decoupled from the intra-container architectures, which makes the inter-container routing easy to deploy and helps the independent development of intra-container architectures. We have implemented a software based uFix stack on Linux platform. Our evaluation results show that uFix exposes high network capacity, gracefully handles server/switch failures, and brings affordable a forwarding overhead to data center servers.

References

[1] S. Ghemawat, H. Gobio, and S. Leungm, "The Google File System", In *Proceedings of ACM SOSP'03*, 2003

[2] Hadoop, http://hadoop.apache.org/

[3] F. Chang, etc., "Bigtable: A Distributed Storage System for Structured Data", In *Proceedings of OSDI'06*, Dec 2006

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

[5] M. Isard, M. Budiu, Y. Yu, etc., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks", In *Proceedings of ACM EuroSys'07*, 2007.

[6] Google Data Center FAQ, http://www.datacenterknowledge.com/archives/2008/03/27/google-data-center-faq/, Mar 2008

[7] Yahoo Opens New Nebraska Data Center, http://www.datacenterknowledge.com/archives/2010/02/18/yahoo-opens-new-nebraska-data-center/, Feb 2010

[8] C. Guo, H. Wu, K. Tan, etc., "DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers", In *Proceedings of ACM SIGCOMM'08*, Aug 2008

[9] D. Li, C. Guo, H. Wu, etc., "Scalable and Cost-effective Interconnection of Data Center Servers using Dual Server Ports", *IEEE/ACM Transactions on Networking*, 19(1):102-114, 2011.

[10] C. Guo, G. Lu, D. Li, etc., "BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers", In *Proceedings of ACM SIGCOMM'09*, Aug 2009

[11] M. Al-Fares, A. Loukissas and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", In *Proceedings of ACM SIGCOMM'08*, Aug 2008

[12] R. Mysore, A. Pamboris, N. Farrington, etc., "PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric", In *Proceedings of ACM SIGCOMM'09*, Aug 2009

[13] A.Greenberg, J. Hamilton, N. Jain, etc., "VL2: A Scalable and Flexible Data Center Network", In *Proceedings of ACM SIGCOMM'09*, Aug 2009

[14] S. Kandula, J. Padhye and P. Bahl, "Flyways To De-Congest Data Center Network", In *Proceedings of HotNets'09* , Oct 2009

[15] G. Wang, D. Andersen, M. Kaminsky etc., "c-Through: Part-time Optics in Data Centers", In *Proceedings of ACM SIGCOMM'10* , Aug 2010

[16] A. Greenberg, J. Hamilton, D. Maltz, etc., "The Cost of a Cloud: Research Problems in Data Center Networks", In *ACM SIGCOMM Computer Communication Review*, 39(1):68-73, 2009

[17] Dell Powerage Servers. *http://www.dell.com/content/products /category.aspx/servers*

[18] H. Wu, G. Lu, D. Li, etc., "MDCube: A High Performance Network Structure for Modular Data Center Interconnection", In *Procedings of ACM SIGCOMM CoNEXT'09*, Dec 2009

[19] G. Lu, C. Guo, Y. Li, etc., "ServerSwitch: A Programmable and High Performance Platform for Data Center Networks ", In *Proceedings of NSDI'11*.