IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. XX, NO. X, DECEMBER 20XX

An Incrementally Scalable and Cost-efficient Interconnection Structure for Data Centers

Junjie Xie, Yuhui Deng, *Member, IEEE,* Geyong Min, *Member, IEEE,* Yongtao Zhou

Abstract—The explosive growth in the volume of data storing and complexity of data processing drive data center networks (DCNs) to become incrementally scalable and cost-efficient while to maintain high network capacity and fault tolerance. To address these challenges, this paper proposes a new structure, called *Totoro*, which is defined recursively and hierarchically: dual-port servers and commodity switches are used to make Totoro affordable; a bunch of servers are connected to an intra-switch to form a basic partition; to construct a high-level structure, a half of the backup ports of servers in the low-level structures are connected by inter-switches in order to incrementally build a larger partition. Totoro is incrementally scalable since expanding the structure does not require any rewiring or routing alteration. We further design a distributed and fault-tolerant routing protocol to handle multiple types of failures. Experimental results demonstrate that Totoro is able to satisfy the demands of fault tolerance and high throughput. Furthermore, architecture analysis indicates that Totoro balances between performance and costs in terms of robustness, structural properties, bandwidth, economic costs and power consumption.

Index Terms—Data center network, Scalability, Network capacity, Cost efficiency, Fault tolerance.

For Manuscript Only

1 Introduction

With the rapid development of information digitization, a huge amount of data is being created every day in various fields. To process these explosively incremental data, large-scale data center networks (DCNs) are built and play a significant role in hosting various applications, such as Instant Messaging (IM), video service, Machine Learning (ML) and so forth. A modern DCN is not just a collection of servers and network devices but needs to be considered as a single computing unit, namely *Warehouse-Scale Computers* (WSC) [1]. Modern DCNs are distinguished from traditional ones by their more rigorous requirements:

1) **Scalability**: DCNs must physically support thousands and even millions of nodes to power the computational tasks and data storage [2]. In practice, DCNs are more likely to be built firstly with a part of integrated components because the investors often prefer to a low startup cost and then enlarge the scale as business expands [3]. Thus DCNs should enable incremental expansion efficiently and such expansion should minimize.

2) **High network capacity**: Cisco has studied the data center traffic and reported that 76.7% of the traffic remains within the data centers [4]. High network capacity is fundamental for a well-designed DCN to support such traffic. Two solutions are widely adopted: a) the "scale up" solution utilizes higher-end devices to upgrade the network capacity; b) the "scale out" solution connects more commodity devices to satisfy the performance requirements. The later has two advantages of economical efficiency and fault tolerance and thus represents a rising trend in this field.

3) **Fault tolerance**: As the scale of DCNs increases, failures become common in the cloud environment and have a significant impact on the running applications [5]. These damages make fault tolerance a big challenge in the cloud environment.

4) **Cost efficiency**: Costs in today's data center contain four major components: 45% goes to servers (CPU, memory, and storage systems), 25% goes to infrastructure (power distribution and cooling), 15% goes to power draw (electrical utility costs), and 15% goes to network (links, transit, and equipment) [6]. The design of DCNs must balance between performance and costs, especially the economic costs and power consumption.

However, legacy designs of data centers can not fully meet these requirements. In current practice, many data centers follow the legacy ThreeTier [7] structure in which servers are connected in a rack with Top-of-Rack switches at the edge level. Then edge switches are connected with aggregation switches to build the network architecture. On the top of the structure, ThreeTier provides the Internet services by core-routers or core-switches. However, the ThreeTier data centers have three noticeable weaknesses. Firstly, the top-level components often become the bandwidth bottleneck. Secondly, one failure of them can abruptly degrade the crossing traffic. Thirdly, it is expensive to update the top-level switches, leading to the sharp rise of costs. Adding redundant switches and links may lighten these issues without considering the cost. But the ThreeTier structure is still inherently short of adequate scalability and fault tolerance.

To overcome the disadvantages of the traditional ThreeTier structure, this study aims to develop an innovative solution to meet the requirements of well-designed DCNs: high scalability (especially incremental scalability), cost-

- *J. Xie, Y. Deng and Y. Zhou are with the Department of Computer Science, Jinan University, Guangzhou, China, 510632.*
  E-mail: xiejunjiejnu@gmail.com, tyhdeng@jnu.edu.cn
- *Y. Deng is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 100190.*
- *G. Min is with the College of Engineering, Mathematics and Physical Sciences, University of Exeter, Exeter, EX4 4QF, United Kingdom.*
  Email: g.min@exeter.ac.uk

effectiveness, high network capacity, and fault-tolerance.

The servers in the current market commonly own two Network Interface Card (NIC) ports: one for normal connection and the other for backup usage [8]. Servers with four or more ports have recently appeared, but such machines are usually high-end and thus are expensive. In terms of network connection, more redundant links make the structure more efficient and robust. Adopting high-end machines with more ports or adding more NICs to the existing machines may address the problems of fault-tolerance and bandwidth requirement. However, it is infeasible to build a large-scale DCN with a vast number of high-end machines due to their high cost [9]. Besides, updating hardware (e.g., adding more NICs or replacing 2-port NICs with 4-port ones) may affect the existing business or even destroy the original communication mechanism. Therefore, it is more desirable to construct scalable and fault-tolerant DCNs by utilizing the widely used and low-cost commodity servers with dual ports [1] [8] [10][11].

In this paper, we propose a new interconnection structure called *Totoro* [1], which adopts commodity servers with two ports. Totoro is recursively defined. When constructing a high-level Totoro, the low-level Totoros use half of their available backup ports for interconnections. Thus, there exist available (un-used) ports for each level structure. This feature makes the expansion of Totoro convenient. If the scale of DCNs needs to be expanded, more servers can be connected and integrated with the existing structures (plugging wires to the available ports) without modifying any existing hardware (e.g., rewiring or updating NICs) or software (e.g., adopting new routing mechanism). As a consequence, Totoro is incrementally scalable.

The method of using half of the available ports for expansion was firstly adopted by *FiConn* [8]. But there exist many significant differences between Totoro and FiConn. Totoro connects servers to switches and thus there are no direct links between any two servers, while FiConn connects servers directly to form a complete graph in each level. Since switches can forward data to several directions, the property of link multiplexing is intrinsical for Totoro, which offers more available ports and is conducive to connecting more redundant links. Compared to FiConn, another advantage is that the data flowing from one partition to another can be distributed to multiple links. This reduces the forwarding loads and makes the data transmission more efficient. We will further discuss and prove that the usage of switches achieves a lower price-performance ratio than FiConn. An existing structure sharing the similar wiring principle of using switches to connect servers is *BCube* [11]. However, it is extremely hard to expand a completely built BCube since it is mainly designed for modular data centers. The incremental scalability of BCube is not comparable to Totoro. More details will be discussed in Section 2 and 6. To sum up, the major contributions of this paper are listed as follows.

1) We propose a new and cost-effective network structure, Totoro, which is recursively defined and incrementally scalable. As only half of the available ports in the lower-level structures are used whenever the network is extended to construct a high-level structure, the available ports enable the incremental scalability without any rewiring, hardware replacement or routing alternation. Besides, the use of commodity servers and switches makes the DCNs affordable.

2) We develop a fault-tolerant and effective routing mechanism to handle multiple types of failures in DCNs. The proposed rerouting technology leverages a Base 2 Logarithms Model to bypass the fault domains via neighbor or remote partitions without trapping into the local dilemma. This model does not require any global information, and thus it can efficiently determine the target links to reroute the packets.

3) We investigate the important properties of Totoro, conduct the experiments of evaluating the path failure ratio and network throughput, analyse the structural robustness, bandwidth, cost and power consumption. The results demonstrate that Totoro is a robust and cost-efficient architecture design.

The remainder of this paper is organized as follows. Section 2 introduces the related work. Section 3 details the Totoro structure. Section 4 presents a distributed and fault-tolerant routing protocol for Totoro. Section 5 presents experiments to evaluate the performance and availability of Totoro. Section 6 presents the architecture analysis. Finally, Section 7 concludes this paper.

2 RELATED WORK

As cloud computing has developed rapidly in recent years, studies on data center networks (DCNs) have attracted many research efforts from both academic and industrial communities [13] [14] [15].

Considering the weaknesses of the traditional ThreeTier structure, Fares *et al*. presented an improved ThreeTier structure, namely *FatTree* [16], which scales out with a large number of links and mini-switches. Using more redundant switches, FatTree achieves an oversubscription ratio of 1:1. Based on FatTree, *SEATTLE* [17] and *Portland* [18] were proposed to provide "plug-and-play" functionality via flat addressing and hierarchical addressing, respectively. But the scalability of FatTree is still limited by the ports of switches fundamentally. If FatTree needs to be expanded and the existing switches are fully utilized, switches must be replaced to offer more ports. This has negative effects because updating switches will break the existing business and cause steeper costs. In contrast, Totoro is not limited by any hardware (e.g., the number of servers or switch ports) and thus has no bound of scale. To expand the network, we only need add more machines and follow the building principle to connect them to the available backup ports. Besides, Totoro uses fewer switches than FatTree. Based on its connecting philosophy, Totoro needs the switches fewer than $2T/n$ while FatTree needs $5T/n$ switches (T indicates the total number of servers and n is the number of switch ports). It is worth noting that using fewer switches leads to the lower cost and energy consumption.

DCell [10] is a level-based, recursively defined interconnection structure with typical requirements of multiport (e.g., 3, 4 or 5) servers. DCell scales double exponentially

1. A preliminary short version of this paper [12] appears in the Proceedings of the 10th IFIP International Conference on Network and Parallel Computing (NPC-2013). We significantly extend the fault-tolerant routing algorithm, add the extensive experiments and enrich the architecture analysis in the current paper.

For Manuscript Only

with the server node degree. It is also fault-tolerant and has high network capacity. As a trade-off, DCell replaces the expensive core switches/routers with multi-port NICs and higher wiring cost. Compared to DCell, Totoro needs fewer ports, but more switches. As being discussed in the section of Introduction, most commodity servers in the current market are equipped with dual-port NICs [8]. Replacing NICs with more-port ones or adding more NICs undoubtedly increases the cost and deployment overhead. Therefore, Totoro adopts dual-port machines as the building foundation and thus significantly reduces the cost. By using switches to connect servers, all data flows from a node to another go through switches only, improving the ports' efficiency. Besides, Totoro adopts a naturally bottom-up resolution to expand incrementally, which is opposite to what DCell uses and makes the incremental deployment more convenient.

BCube [11] represents a wonderful attempt to design the network architecture for modular data center. It connects servers with multiple ports to mini-switches and there are no direct links between servers. BCube places routing intelligence on servers. It intrinsically supports various bandwidth-intensive applications and exhibits graceful performance degradation. Nevertheless, large-scale use of multi-port NICs inevitably leads to an expanding overhead, which will be proven in Section 6. Totoro adopts the similar method to connect servers with switches. There are no direct links between servers as well. Partial deployment of Totoro and BCube are also similar since they both use a full top-level switches. However, it is more convenient for Totoro to expand a completely deployed structure because there is no need to reserve a port or add NIC on each host. BCube is designed for mega data center and thus the incremental scalability of BCube is not comparable to Totoro.

FiConn [8] is also a new server-interconnection structure by adopting servers with two ports and low-end commodity switches to form the network infrastructure. FiConn grows double exponentially. The degree of server nodes in FiConn is always two, leading to a lower wiring cost than DCell. Routing in FiConn also makes a balanced use of links at different levels and is traffic-aware so as to better utilize the link capacities. Totoro shares the similar wiring principle with FiConn by using half of the available backup ports to form a higher-level structure, which provides the feature of incremental scalability. The difference between Totoro and FiConn is that Totoro connects servers with switches instead of direct wires. In FiConn, two partition flows communicate through a unique link. This brings high forwarding loads to the servers at each end of this link. Unlike FiConn, there are multiple links connecting two partitions directly in Totoro. All data flowing from one partition to the other can be distributed to these links, and thus reducing the forwarding load and making data transmission more efficient. Besides, the intrinsical property of link multiplexing saves the structure more available ports and is conducive to connecting more redundant links. We will further prove that the usage of switches gains a lower price-performance ratio than FiConn in Section 6.

Different from the existing work, this paper proposes a new interconnection structure called *Totoro* for DCN. The key features of Totoro can be summarized as follows:

1) **Incremental scalability:** Totoro supports not only largely physical interconnection but also flexibly incremental expansion;

2) **Cost-effectiveness:** Totoro achieves a lower price-performance ratio;

3) **High network capacity:** Totoro provides a high bisection width;

4) **Fault-tolerance:** Totoro offers a fault-tolerant and high-effective routing mechanism to handle multiple types of failures in data centers.

TABLE 1: The Denotations Frequently Used in this Paper.

Denotation	Meaning
n	The number of ports on a switch.
k	The top level in a Totoro.
$Totoro_i$	The ith level Totoro.
$Totoro_i[x]$	The xth $Totoro_i$ in a $Totoro_{i+1}$.
t_k	The total number of servers in $Totoro_k$.
$[a_k, a_{k-1}, ..., a_i, ..., a_1, a_0]$	A $(k+1)$-tuple to denote a server, where $a_i < n$ $(0 < i \leq k)$ indicates at which $Totoro_{i-1}$ this server is located and $a_0 < n$ indicates the index of this server in that $Totoro_0$.
$(u - b_{k-u}, b_{k-u-1}, ..., b_0)$	A combination of an integer and a $(k-u+1)$-tuple to denote a switch, where $u \leq k$ indicates that it is a level-u switch, $b_i < n$ $(0 < i \leq k-u)$ indicates at which $Totoro_{u+i-1}s$ this switch is located and b_0 indicates the index of this switch among level-u switches in that $Totoro_u$.
$P(src, dst)$ or $src \rightarrow dst$	A path from src to dst.

3 TOTORO INTERCONNECTION NETWORKS

The frequently used denotations in this paper are listed and explained in Table 1.

3.1 The Physical Structure of Totoro

Totoro consists of a series of commodity servers with dual ports and low-end n-port switches. Dual-port servers are commonly deployed in industry. Low-end switches without uplinks are inexpensive and affordable. These motivate us to build a modern data center at acceptable costs.

Totoro is recursively defined as follows. We connect n servers to an n-port switch to form the basic partition of Totoro, denoted by $Totoro_0$. The switch is called an *intra-switch*. Each server in $Totoro_0$ is connected to an intra-switch using one port; the rest ports are called *available* ports. If a $Totoro_0$ is considered as a virtual server, the number of available ports in a $Totoro_0$ is equal to n. Then each $Totoro_0$ is connected to $n/2$ switches using half of its available ports (i.e., $n/2$ ports). As each switch has n ports, it is connected to n $Totoro_0s$. Now we obtain a larger partition denoted by $Totoro_1$ (as shown in Fig. 1). Then we connect n $Totoro_1s$ with $n^2/4$ switches to form a $Totoro_2$ (see Fig. 2). In each $Totoro_1$, half of the available ports, i.e., $n^2/4$ ports, are used for connection. Generically, we connect n $Totoro_{k-1}s$ to $(n/2)^k$ switches to build a $Totoro_k$. A switch connecting different partitions is called

For Manuscript Only

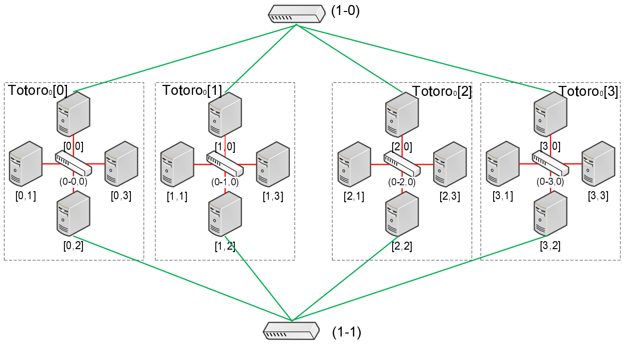


Fig. 1: A $Totoro_1$ structure with $n = 4$.

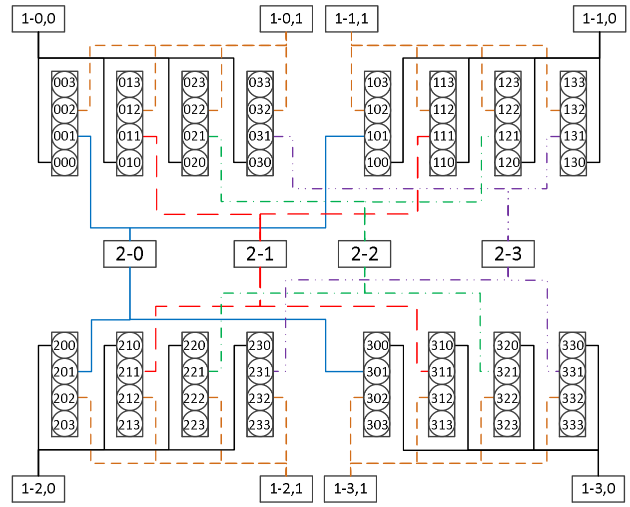


Fig. 2: Given 4-port switches, a $Totoro_2$ structure can be constructed from 4 $Totoro_1s$. Each $Totoro_1$ contains 4 $Totoro_0s$ and 4 servers are connected in each $Totoro_0$.

an *inter-switch*. In a $Totoro_k$, switches and links connecting different $Totoro_{k-1}s$ are called *level-k switches* and *level-k links*, respectively. In particular, the level of intra-switch is 0.

It is worth noting that there is no need to connect low-level switches to high-level switches in the proposed Totoro structure in order to build the higher-level Totoro. Therefore, no direct link between any two switches is required. This is one typical difference between Totoro and ThreeTier structure.

Now a case study is presented to demonstrate how to build a Totoro structure with two given structural parameters: n and k. n determines how many ports of a switch are used while k represents how many levels of the target architecture are. Take Fig. 2 as an example where $n = 4$ and $k = 2$. Firstly, we connect 4 servers to a 4-port switch to build a bottom partition, $Totoro_0$. In Fig. 2, a small circle represents a single server and a rectangle filled with 4 circles represents a $Totoro_0$. For clarity, the intra-switch is omitted in Fig. 2. Repeatedly, we can construct 4 $Totoro_0s$ in the same way. It can be seen that there exist 4 available ports in a $Totoro_0$ (each server owns one). Then half of them, i.e., 2 ports, are chosen to connect with 2 different switches, respectively. For example, we connect server $[0, 0, 0]$ to switch $(1 - 0, 0)$ and server $[0, 0, 2]$ to switch $(1 - 0, 1)$. For other $Totoro_0s$, we adopt the same method to get a $Totoro_1$ structure (see Fig. 1), containing 16 servers, 4 intra-switches (not shown in Fig. 2) and 2 level-1 switches. To build a $Totoro_2$, 4 $Totoro_1s$ are also required. If $Totoro_1$ is considered as a whole, it can be observed that there are 8 available ports in a $Totoro_1$. We also utilize half of them, i.e., 4 ports, to connect with 4 different switches, respectively. In Fig. 2, we connect server $[0, 0, 1]$ to switch $(2 - 0)$, server $[0, 1, 1]$ to switch $(2 - 1)$, server $[0, 2, 1]$ to switch $(2 - 2)$ and server $[0, 3, 1]$ to switch $(2-3)$. Similarly, other $Totoro_1s$ can be connected together and we finally obtain a higher-level Totoro, i.e., $Totoro_2$. Generically, when n-port switches are used to build a $Totoro_k$, the numbers of required servers, switches, and links are n^{k+1} (see Theorem 1 for details), $n^k \times (2 - 1/2^k)$, and $n^{k+1} \times (2 - 1/2^k)$, respectively.

The linking principle of Totoro is: $1/2^k$ of the links in a certain partition are connected to several k-level partitions (i.e., $Totoro_ks$). As k grows, the percentage of k-level links declines, which means that most of the links are provided to access the data stored nearby. This closely matches the fact that most of the relevant data is put together, also known as spatial locality [19].

In some other structures, like DCell or FiConn, there is only one direct link between two adjacent partitions. If this link is busy or disabled, the routing mechanism has to bypass this link with the help of other neighbor partitions. Distinctly, this creates more forwarding workloads for other servers in those partitions. Through comparison, the structure of Totoro reduces the accessing distance between servers in the fault situation because there are several inter-switches between two partitions. The servers in a $Totoro_i$ $(0 \leq i < k)$ can access servers in another $Totoro_i$ directly by $(n/2)^{i+1}$ paths without going through any other $Totoro_i$. For instance, server $[0, 1]$ in Fig. 1, needs to access server $[1, 1]$. Under the normal circumstances, we can choose the path $[0, 1] \to (0 - 0, 0) \to [0, 0] \to (1 - 0) \to [1, 0] \to (0-1, 0) \to [1, 1]$. Assume that one link between servers and the inter-switch fails (e.g., $[0, 0] \to (1 - 0)$), this path is unavailable now. In this case, another path $[0, 1] \to (0-0, 0) \to [0, 2] \to (1 - 1) \to [1, 2] \to (0 - 1, 0) \to [1, 1]$ can be chosen. As a result, the communication is still between two $Totoro_0s$ without going through any other. For instance, the path from server $[0, 1]$ to server $[1, 1]$ will not across $Totoro_0[2]$. This feature also naturally supports multi-path routing if we simultaneously activate all existing routing selections. For example, if the paths of $[0, 1] \to (0 - 0, 0) \to [0, 2] \to (1-1) \to [1, 2] \to (0-1, 0) \to [1, 1]$ and $[0, 1] \to (0-0, 0) \to [0, 0] \to (1-0) \to [1, 0] \to (0-1, 0) \to [1, 1]$ are both utilized, the throughput between server $[0, 1]$ and server $[1, 1]$ will double.

Observing the Totoro structure, it is clear that not all servers are connected to inter-switches. In our design philosophy, unused ports are left for extension. For a k-level Totoro using n-port, the number of available ports for expansion is $n^{k+1}/2^k$. Thus, the proposed Totoro is open and easy for extension. FiConn [8] makes use of all available backup server ports for interconnection, i.e., adding shortcut links to improve the bisection bandwidth. As a trade-off, Totoro does not adopt this method since the percentage of available backup ports is not high $(1/2^{k+1})$ and keeping the routing simple and consistent is quite important. Especially

For Manuscript Only

Algorithm 1: Totoro Building Algorithm

```
1  Function TotoroBuild (n, k)
2      t_k = n^{k+1}
3      for tid = 0 to (t_k − 1) do
4          s(s_k, ..., s_i, ..., s_0) = TotoroIDToTuple (tid)
5          intraSw = (0 − b_k, ..., b_i, ..., b_0)
6          for i = 1 to k do
7              b_i = s_i
8          b_0 = 0
9          Connect(s, intraSw)
10         for u = 1 to k do
11             if (tid − 2^{u−1} + 1) mod 2^u == 0 then
12                 interSw = (u − b_{k−u}, b_{k−u−1}, ..., b_1, b_0)
                   for i = u to (k − 1) do
13                     b_i = s_{i+1}
14                 b_0 = (tid/2^u) mod (n/2)^u
15                 Connect(s, interSw)
```

when upgrading the scale of data centers, removing and re-plugging shortcut links bring significant deployment complexity. We also advocate using low-end switches without uplinks and expanding Totoro by increasing the structural levels rather than updating the switches. This helps to reduce the device cost and the management cost in data centers.

3.2 Totoro Building Algorithm

A server in Totoro can be indicated in two ways: *Totoro tuple* and *Totoro ID*. Totoro tuple is a $(k+1)$-tuple $[a_k, a_{k-1}, ..., a_i, ..., a_1, a_0]$, which indicates where this server is located and can help calculate the common partition of two servers. In routing algorithm, it is a vital step to find out the common partition of the source server and the destination server. For example, servers $[0, 0]$ and $[0, 1]$ in Fig. 1 are in the same $Totoro_0[0]$ in terms of their common prefix (i.e., $[0]$). Totoro ID is an unsigned integer, taking a value from $[0, t_k)$. Totoro ID will be used in the header of packets to identify a server uniquely, performing like IP Address. Note that, the mapping between Totoro tuple and Totoro ID is a bijection.

In addition, a switch is denoted as a combination of an integer and a $(k-u+1)$-tuple $(u - b_{k-u}, b_{k-u-1}, ..., b_1, b_0)$. Note that, b_0 is identically equal to 0 when $u = 0$. Because there is only one intra-switch in a $Totoro_0$. Algorithm 1 presents how Totoro can be built. The key step in this algorithm is to determine the level of the outgoing link of this server (Line 11). The function **Connect** represents the operation that a server is connected to a switch manually. The time complexity of Algorithm 1 is $O(k \times t_k)$ where t_k denotes the total number of nodes in a $Totoro_k$.

Considering the fact that the linking philosophy and address configuration of Totoro are slightly more complex than ThreeTier structures, some automatic address configuration mechanisms, e.g., [20] would be introduced to make the deployment faster and easier.

3.3 Incremental Deployment

Incremental deployment of interconnection networks becomes a common requirement due to the scalability requirement. To incrementally deploy an interconnection network, three important aspects should be considered: 1) no rewiring, 2) no hardware replacement, and 3) no software modification. These requirements ensure that the existing applications will not be affected and can be achieved in the proposed Totoro structure.

When n-port switches are used, a k-level Totoro remains $n^{k+1}/2^k$ ports for expansion and thus there is no need to change the existing structure. A straightforward way to gradually construct Totoro is the "bottom-up" approach. Totoro firstly builds the complete low-level structures and connects them to the top-level switches. We also make sure that all k-level links are connected in each $Totoro_{k-1}$ and deploy full top-level switches. This approach provides the full network capacity at the top level but the ports of top-level switches will not be fully utilized. Since low-cost switches are adopted, this approach is affordable.

3.4 Properties of Totoro

To investigate the scalability of Totoro, Theorem 1 reveals that the number of servers, t_k, in Totoro scales exponentially as the level increases.

Theorem 1. In $Totoro_k$, the total number of servers is

$$t_k = n^{k+1}. \quad (1)$$

Proof: A $Totoro_0$ has $t_0 = n$ servers. n $Totoro_0s$ are connected to n-port inter-switches to form a $Totoro_1$. Hence, there are $t_1 = n \times t_0$ servers in a $Totoro_1$. In general, a $Totoro_i$ $(1 \leq i \leq k)$ consists of n $Totoro_{i-1}s$ and has $t_i = n \times t_{i-1}$ servers. Finally, the total number of servers in $Totoro_k$ is $t_k = n^{k+1}$. □

The proposed Totoro is suitable for different sizes, from thousands to millions of nodes. In accordance with the wiring philosophy, a $Totoro_k$ always remains $t_k/2^k$ ports for extension. Henceforth, the total number of Totoro, t_k, can be infinite in theory as the structural level k increases.

Theorem 2 shows that the average node degree of Totoro, denoted by $degree_{avg}$, approaches to 2 when k grows, but will never reach 2.

Theorem 2. In $Totoro_k$, the average node degree is

$$degree_{avg} = 2 - (\frac{1}{2})^k. \quad (2)$$

Proof: Let c_i $(1 \leq i \leq k)$ denote the number of available ports in $Totoro_i$. A $Totoro_0$ has $c_0 = n$ available ports. By using half of the available ports in each $Totoro_0$, n $Totoro_0$s are connected to n-port inter-switches to form a $Totoro_1$ which has $c_1 = n \times c_0/2 = n^2/2$ available ports. In general, a $Totoro_i$ has $c_i = n \times c_{i-1}/2$ available ports. Finally, a $Totoro_k$ has $c_k = n \times c_{k-1}/2 = n \times (n/2)^k$ available ports. In other words, there are c_k one-degree servers while the others are two-degree. Therefore, the total node degree in $Totoro_k$ is $degree_{total} = 2 \times t_k - n \times (n/2)^k$. In combination with Theorem 1, the average node degree is $degree_{avg} = degree_{total}/t_k = 2 - (1/2)^k$. □

Theorem 2 demonstrates that Totoro is always incomplete and highly scalable by using available backup ports. In addition, a low node degree means that fewer links are required, leading to the lower deployment cost.

For Manuscript Only

Algorithm 2: Totoro Routing Algorithm

```
Function TRoute (src, dst)
    if src == dst then
        return NULL
    lcl = getLCL(src, dst)
    if lcl == 0 then            // in the same Totoro_0
        return P(src, dst)
    else
        P(m, n) = getNearestPath(src, dst, lcl)
    return TRoute(src,m)+P(m,n)+TRoute(n,dst)
```

TABLE 2: The Mean Value and Standard Deviation of the Path Length in TRA and SPA.

n	k	t_k	M_k	TRA		SPA	
				Mean	StdDev	Mean	StdDev
24	1	576	6	4.36	1.03	4.36	1.03
32	1	1024	6	4.40	1.00	4.39	1.00
48	1	2304	6	4.43	0.96	4.43	0.96
24	2	13824	10	7.61	1.56	7.39	1.32
32	2	32768	10	7.68	1.50	7.45	1.26

Theorem 3. The bisection width (BiW) of $Totoro_k$ is

$$BiW = \frac{t_k}{2^{k+1}}. \quad (3)$$

Proof: Bisection width denotes the minimal number of links to be removed so as to partition a network into two parts of equal size. Considering the linking philosophy, there exist $(n/2)^k$ top-level (i.e., k-level) switches in a $Totoro_k$. We divide these top-level switches into two equivalent sets, indicated by S_A and S_B. We also divide all nodes into two equal sets, indicated by N_A and N_B. Then we unlink all switches in S_A from nodes in N_B and keep the connection between S_A and N_A. Similarly, we unlink all switches in S_B from nodes in N_A and keep the connection between S_B and N_B. Now the network is divided into two equal parts. In the above process, half of the links on each switch have been unplugged, i.e., $(n/2)^k \times n/2 = (n/2)^{k+1} = t_k/2^{k+1}$ links are removed (Note that each switch has n links). Hence, Theorem 3 has been proved. □

A larger bisection width implies a higher network capacity and a more resilient structure against failure. A low-level Totoro can hold a large number of servers. Thus, Totoro has a relative large bisection width. We will further compare Totoro and other structures in Section 6.

4 Totoro Routing

In DCNs, how to reroute the packets to bypass the failures becomes a vital problem [5] [21]. The fashionable approach that shares global link states is impracticable due to the huge volume of traffic caused by sending link states. As the servers deployed in DCNs are all commodity servers, it is extremely difficult to finish this computational task with an $O(n^3)$ time complexity of thousands or even millions of nodes at short notice.

Since Totoro is layered and the connection is regular (see Algorithm 2), we design *Totoro Routing Algorithm* based on *Divide and Conquer Algorithm* [10] instead of the shortest path algorithm. Then the whole network is partitioned into some domains, *Totoro Broadcast Domain*s. Link states are limited in such a domain rather than spread globally. In combination of these two strategies, a fault-tolerant routing mechanism, namely *Totoro Fault-tolerant Routing*, is proposed to deal with several common failure scenarios.

4.1 Totoro Routing Algorithm (TRA)

TRA is based on Divide and Conquer Algorithm and is more simple and efficient. Suppose the source server (denoted by src) is in a $Totoro_{i-1}$ $(0 < i \leq k)$ partition and the destination server (denoted by dst) is in another $Totoro_{i-1}$ $(0 < i \leq k)$ partition. These two $Totoro_{i-1}s$ belong to the same $Totoro_i$ $(0 < i \leq k)$. Thus, there must be at least one level-i path between these two $Totoro_{i-1}s$ to connect each other. To find out the path from src to dst in Totoro: firstly, we need to find out one such level-i path (denoted by $P(m, n)$); we suppose servers m and src are in the same $Totoro_{i-1}$ while servers n and dst are in the another $Totoro_{i-1}$; then, the problem is divided into two sub-problems, i.e., to work out the path from src to m and the path from n to dst; we use the same method to gain $P(src, m)$ and $P(n, dst)$ recursively; in this process, if the beginning and the ending of a path are found in the same $Totoro_0$, the directed path between them is returned; finally, we join $P(src, m)$, $P(m, n)$ and $P(n, dst)$ for a full path.

The function **TRoute** in Algorithm 2 follows the whole process mentioned above. The function **getLCL** returns the $Lowest\ Common\ Level$ (LCL) of two nodes. The function **getNearestPath** picks a level-lcl path nearest to the given source host. For example, in Fig. 1, **getNearestPath**($[0, 0]$, $[1, 1]$, 1) returns $P([0, 0], [1, 0])$ rather than $P([0, 2], [1, 2])$. The time complexity of Algorithm 2 is $O(2^k)$ where k denotes the top level of $Totoro$. Considering k is always smaller than 4 because a low-level Totoro can hold a large number of servers and the larger k is not required, the actual time complexity is acceptable.

Denoting the distance between the server and its direct neighbor switch as 1, the maximum distance between two servers, M_k, can be given by the following theorem.

Theorem 4. The maximum distance between two servers in $Totoro_k$ is

$$M_k = 2^{k+2} - 2. \quad (4)$$

Proof: Algorithm 2 reveals that the routing algorithm divides the path into two sub-paths, which are connected by an intermediate link. The length of intermediate link is 2. Thus, we can easily get $M_k = 2 \times M_{k-1} + 2$, which is further transformed into $M_k + 2 = 2 \times (M_{k-1} + 2)$. Through using the induction, this theorem can be proved. □

In fact, Theorem 4 reveals the upper bound of network diameter. The shorter the network diameter is, the more effective the routing mechanism will be. The performance of routing algorithm can be directly evaluated according to the path length. Table 2 lists the mean values and the standard deviations of the path length by using TRA and

For Manuscript Only

SPA (Shortest Path Algorithm[2]) for Totoro with different n and k. In terms of the mean value and standard deviation, we observe that the differences are both small, indicating that the performance of TRA is close to SPA under the conditions of different sizes. Although SPA is globally optimal, its computation complexity is as high as $O(n^3)$ and thus it is not suitable for routing in data center. However, SPA gives the upper bound of routing performance (e.g., path length, and path failure ratio) [10]. Such comparison suggests that the proposed TRA is efficient enough. Thus, we build the fault-tolerant routing algorithm in Totoro based on TRA since it is much simpler than SPA.

Considering the following case:, suppose $[0,0]$ in Fig. 1 needs to access $[1,1]$, but link $[0,0] \rightarrow (1-0)$ and link $[1,2] \rightarrow (1-1)$ both fail (e.g., the corresponding ports are unavailable or there is something wrong with the wires). In this case, no matter which intermediate path is chosen (i.e., $[0,0] \rightarrow [1,0]$ or $[0,2] \rightarrow [1,2]$), TRA will fail to find a path from $[0,0]$ to $[1,1]$. In fact, however, there still exist available paths, such as $[0,0] \rightarrow [0,2] \rightarrow [2,2] \rightarrow [2,0] \rightarrow [1,0] \rightarrow [1,1]$. Therefore, TRA is not fault-tolerant and we need a more powerful and robust mechanism to solve this problem. Since the network state, e.g., link state is crucial for a routing mechanism, we naturally decide to utilize it to make a fault-tolerant routing mechanism. Instead of sharing link states in the whole network, the structure is divided into several partitions for efficiency. Each partition is called a TBD, which is detailed below.

4.2 Totoro Broadcast Domain (TBD)

In this subsection, the definition of Totoro Broadcast Domain (*TBD*) is introduced to break up the network. Firstly, we define a variable called *bcl* (Broadcast Level) for broadcast domain, which means that a $Totoro_{bcl}$ is a TBD. The server in a TBD is called *inner-server* while the server connected to TBD with an outgoing link whose level is larger than *bcl* is called *outer-server*. Take Fig. 1 as an example, and assume $bcl = 0$. Then $[0,0]$, $[0,1]$, $[0,2]$ $[0,3]$ and $(0-0,0)$ can be regarded as a TBD. $[1,0]$, $[2,0]$, $[3,0]$, $[1,2]$, $[2,2]$ and $[3,2]$ are outer-servers of this TBD.

Servers detect the state of links connecting them and broadcast the link state information to its intra-switch and inter-switch (if it has) periodically. If a server receives a packet of link states, it handles the packet based on the following steps: If this packet has ever been received, then just drop it. Otherwise, save the link states and determine whether the packet comes from inter-switch. If this is the case, broadcast it to the intra-switch. If not, broadcast it to inter-switch if this server is connected to an inter-switch with a link whose level is less than or equal to *bcl*.

As a result, all inner-servers get the link states of every inner-server and every outer-server while outer-servers only own the states of the links that connect inner-servers and themselves. The reason is that we will regard an outer-server as a proxy in the failure scenarios and data will only flow from inner-servers to outer-servers. Hence, outer-servers do not need to get the link states among inner-servers. Note that, inner-servers and outer-servers are not fixed in different TBDs. That is to say, as an outer-server in one TBD (e.g., $Tototro_0[1]$), a server (e.g., $[0,0]$) will never share the link states to inner-servers (e.g., $[1,0]$). But as an inner-server in another TBD (e.g., $Tototro_0[0]$), this server (e.g., $[0,0]$) will share link states with outer-servers (e.g., $[1,0]$).

2. Shortest path algorithm is widely used in link state protocols like OSPF and IS-IS. In this paper, we implement SPA by using Floyd-Warshall algorithm.

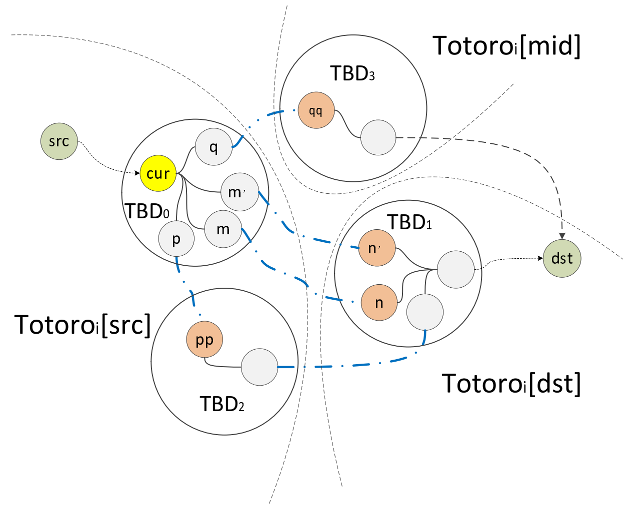


Fig. 3: Totoro Fault-tolerant Routing.

4.3 Totoro Fault-tolerant Routing (TFR)

To combine TRA and TBD, we propose a distributed, fault-tolerant routing protocol for Totoro. In a real-world situation, there are four common types of failures: link, server, switch and rack. By *Using Redundant Links* and *Rerouting Through Neighborhoods*, TFR displays the excellent fault-tolerance capacity to handle these four types of failures. The evaluation will be detailed in Section 5.

4.3.1 Using Redundant Links

Although TRA is efficient, it cannot deal with failures efficiently as discussed above. Assume that source server src needs to access the destination server dst, the current server is cur and the selected path, $P(m,n)$, fails. In this case, the failure cannot be detected until the packet arrives at server m. This may cause a lot of useless forwarding. So, what if there is enough intelligence to find out an available $P(m,n)$?

Note that the key of TRA is to figure out the *Lowest Common Level* (Algorithm 2, Line 4), denoted by lcl, and a nearest level-lcl link (Algorithm 2, Line 8), denoted by $P(m,n)$, between two $Totoro_{lcl-1}s$ where src and dst are located respectively. Before determining the routing path, we must make sure that $P(m,n)$ can be found out in its TBD so as to know its state. Here, the following constraint is given to bcl to provide this feature.

Theorem 5. The constraint to bcl is

$$n^{bcl+1} \geq 2^k, \quad (5)$$

i.e.,

$$bcl \geq \log_n 2^k - 1. \quad (6)$$

Theorem 5 implies that there is at least one outgoing link with $level \leq k$ in a TBD. In other words, a TBD contains

For Manuscript Only

links of all levels, from 0 to k. Thus, we can always find out $P(m,n)$ and its state which is shared within this TBD. Note that, the value of bcl should be as small as possible because sharing link states in a large TBD will cause huge traffic loads.

In order to further improve the routing efficiency, we replace TRA with Dijkstra algorithm within TBD. This will not create heavy burden on the server since the number of servers in a TBD is not large. Among TBDs, we still use TRA for routing.

As shown in Fig. 3, by virtue of TRA, server cur finds that the whole path consists of $P(cur,m)$, $P(m,n)$, and $P(n,dst)$. Nevertheless, there is no need to work out the whole path in the routing calculation. Instead, we can work out the next hop only. Hence, the above process can be simplified to identify the path from server cur to server n, which is an outer-server of TBD_0. Thus, Dijkstra algorithm is adopted to find out the next hop with link states shared in TBD_0. Furthermore, we add a *proxy* field to the packet header, representing a temporary destination. After working out $P(m,n)$, server n is set as the proxy. If this field is not empty, intermediate servers just need to find out the next hop to the proxy through the use of Dijkstra algorithm rather than TRA. After the packet arrives at the proxy, TRA will be used again to find out the next proxy. This strategy can help reduce the overhead of routing calculation. If the chosen proxy is unreachable (e.g., $P(m,n)$ fails), we just pick out another link $P(m',n')$ whose level is the same as $P(m,n)$ and set server n' as the proxy. Through redundant links, the packet is rerouted to a reachable proxy to bypass the failure.

In conclusion, TRA is used to find out the proxy through the nearest path firstly. In case of failure, the packet is rerouted to another reachable proxy through redundant links. Moreover, if there exist several available links, TRA can choose one of them according to a random algorithm or the link load. After that, Dijkstra algorithm is adopted to determine the next hop to the proxy server.

4.3.2 Rerouting Through Neighborhoods

We use redundant links to bypass a failed link. However, if all required level-i links in the current TBD are unavailable, we cannot find out a path from the current server to dst because some servers or inter-switches may fail simultaneously. Server failure and switch failure are also common in a long-running cloud platform. As failures are associated and occur closely, we need a strategy to "escape from" the local area.

Observing the structure of Totoro, we find that TBDs are associated by inter-links, whose levels vary from bcl to k. Naturally, we can utilize the adjacent TBDs to bypass the failures. Take Fig. 3 as an example, TBD_0 has three neighborhoods: TBD_1, TBD_2 and TBD_3. TBD_1 is the destination of TBD. TBD_2 is connected to TBD_0 by a link whose level is smaller than i and thus TBD_2 and TBD_0 belong to the same $Totoro_i[src]$, leading to a benefit that there still exist links connecting to the destination TBD with the required i-level (i.e., LCL) after the packets are rerouted to TBD_2. So the packets can be delivered to the destination TBD directly.

Since the outage of data center is inevitable [21], a worse scenario may happen that a row of racks are all down if their power is cut off. In this case, intermediate TBDs may be unreachable and rerouting the packets to neighbor TBDs in the same low-level substructure is useless. Therefore, we adopt a more aggressive method to reroute packets to a neighbor TBD which is far from the trouble spot. Take TBD_3 as an example, it is connected with TBD_0 by a link whose level is greater than or equal to i and thus TBD_3 and TBD_0 belong to different $Totoro_i$s. If $Path(m,n)$, $Path(m',n')$ and $Path(p,pp)$ all fail, the higher-level $Path(q,qq)$ can be chosen to bypass failures.

In addition, there exist two more unavoidable problems: 1) how to quantitatively determine the level of rerouting links? 2) how to limit the rerouting times due to their huge cost? Here we define a variable *RTR* (Remaining Times to Retry) and a calculation model to solve these two problems. RTR indicates how many times the rerouting technique can be retried. Whenever the packet is rerouted, RTR needs to be decreased. If it reduces to 0, the packet will be dropped. We naturally believe that the more times the packets are rerouted, the worse the situation must be. Hence, a smaller value of RTR indicates that a higher-level rerouting link should be used. However, rerouting through a higher-level link will cause a longer path and heavier forwarding workload and thus we should reduce the use of higher-level links. To meet the above requirements, we leverage a **Base 2 Logarithms Model** to calculate the required rerouting level, rl, as follows:

Theorem 6.

$$rl = min(lcl + \lfloor \log_2 RTR\_MAX \rfloor - \lfloor \log_2 RTR \rfloor, k), \quad (7)$$

where RTR_MAX is the maximum value of the initial RTR and lcl is the required level (i.e., the current LCL).

If we assume $lcl = 2$ and $RTR\_MAX = 8$, the rerouting levels will be 2, 3, 3, 3, 3, 4, 4, 5 with the decrement of RTR. As observed, this model will select lower rl (e.g., level-2 and level-3) many times and skip to higher level (e.g., level-4 and level-5) faster (i.e., after a few retrying times) if it still fails. This implies that the lower-level rerouting links will be tried more while the higher-level ones will be adopted less. Even though this model is simple, the experimental results will prove that it is efficient enough. Note that, if there are more than one link with the required level, one of them is chosen in accordance with a random algorithm or the link load. Furthermore, TFR is not loop free. Frequent rerouting may form a ring. Besides RTR, the field of *TTL* (Time To Live, hop count of the packet) in IP header will be also used to prevent packets from persisting. If either TTL or RTR reduces to 0, TFR just drops this packet and sends an unreachable message to the source server, if necessary.

4.3.3 Algorithm

Algorithm 3 shows the detailed procedure of TFR. Let pkt and $pkt.dst$ denote the packet and its destination. If this host is the packet destination, deliver it to the upper layer (Line 3). Otherwise, check whether this host is the proxy. If yes, clear the $proxy$ field (Line 5). The empty $proxy$ field means that a new proxy will be set in the following steps if

For Manuscript Only

Algorithm 3: Totoro Fault-tolerant Routing Algorithm

```
1  Function TotoroFaultTolerantRoute (this, pkt)
2      if pkt.dst == this then
3          deliver(this, pkt) and return TRUE
4      if pkt.proxy == this then
5          pkt.proxy = NULL
6      if pkt.ttl −− ≤ 0 then
7          drop(pkt) and return FALSE
8      next = getNextByDijkstra(pkt.dst)
9      if next == NULL then
10         if pkt.proxy == NULL then
11             lcl = getLCL(this, pkt.dst)
12             pList = getPathsByLevel(this, pkt.dst, lcl)
13             next = selectAProxy(pkt, pList)
14             if next == NULL then
15                 pList = getReroutingPaths(this, pkt, lcl)
16                 next = selectAProxy(pkt, pList)
17         else
18             next = getNextByDijkstra(pkt.proxy)
19     if next != NULL then
20         send(next, pkt) and return TRUE
21     drop(pkt) and return FALSE
```

necessary (Lines 13 and 16). Then check the field of Time-To-Live ttl and reduce it. If ttl is less than or equal to 0, drop the packet (Line 7). After that, try to get the next hop on the path from the current host to the destination by using Dijkstra algorithm (Line 8). If this host and the destination node are in the same TBD, **getNextByDijkstra** ($O((V+E)\log V)$) where V and E represent the nodes and edges of a TBD will return the next hop. Otherwise, further calculation is required. Firstly, check whether the $proxy$ field is empty (Line 10). If not, we just work out the next hop to the proxy node (Line 18). Otherwise, get the LCL between the current host and destination node (Line 11). Then find out all available paths that connect TBDs in which this host and the destination are located, respectively, with the given level lcl (Line 12). The function **getPathsByLevel** ($O(2^k)$) is based on Algorithm 2 but it returns multiple paths between the given source and destination. $pList$ is sorted according to the distance (or link states). Then invoke the function **selectAProxy** (Line 13), which searches the given $pList$, sets the $proxy$ filed, and returns the next hop. If it fails (Line 14), rerouting should be adopted. The function **getReroutingPaths** ($O(2^k)$) firstly checks the value of RTR, then calculates the required rerouting level based on Theorem 6 and gets those eligible paths. Following the above steps, the packet is sent if the next hop is identified (Line 20). Otherwise, the algorithm drops the packet (Line 21). The time complexity of Algorithm 3 is $O(2^k+(V+E)\log V)$, which mainly depends on the scale of a TBD and the rerouting time.

4.4 Addressing and Forwarding

Totoro uses a 32-bit address to identify a unique server. The i-th ($0 \le i \le 3$) byte in the address indicates the i-th value a_i of Totoro tuple (see Table 1), which is also the index in i-level structure. Note that a 4-level Totoro ($k = 3, n = 48$) can support as many as five millions servers.

TABLE 3: Network Parameters in the Experiments.

$T_{n,k}$ / $S_{n,k}$	Network	n	k	t_k
$T_{12,2}$ / $S_{12,2}$	$Totoro_{12,2}$	12	2	1728
$T_{16,1}$ / $S_{16,1}$	$Totoro_{16,1}$	16	1	256
$T_{16,2}$ / $S_{16,2}$	$Totoro_{16,2}$	16	2	4096

Since most applications are based on TCP/IP, the function of routing and forwarding in Totoro can be implemented as a 2.5-Layer driver between IP layer and the link layer without affecting end-host applications. We need to add a header between IP header and Ethernet header, including fields used in TFR like *source address*, *destination address*, *proxy address*, *RTR* (Remaining Times to Retry) and so forth. Totoro address is mapped one-to-one to IP address. When a packet is sent from IP layer, the Totoro 2.5-Layer driver translates the IP address into Totoro address, utilizes TFR to calculate the proxy, attaches the Totoro header and delivers the packet to the Ethernet. When a packet arrives, Totoro driver will use TFR to determine whether delivering the packet to the IP layer (the current host is the destination) or rerouting the packet to the next hop by the *proxy* field. Before delivering the packet to the upper layer, Totoro header should be detached.

The software-based solution which is introduced above has been proven to be available in DCell [10] and BCube [11]. Considering the CPU overhead, hardware-based solutions like CAFE [22] and ServerSwitch [23] are also desirable candidates for implementing Totoro's routing and forwarding in real data center environments. They can handle self-defined packets through simple APIs and easy configurations. In addition, commodity switches used in Totoro are not required to be programmable. There is no any modification about them.

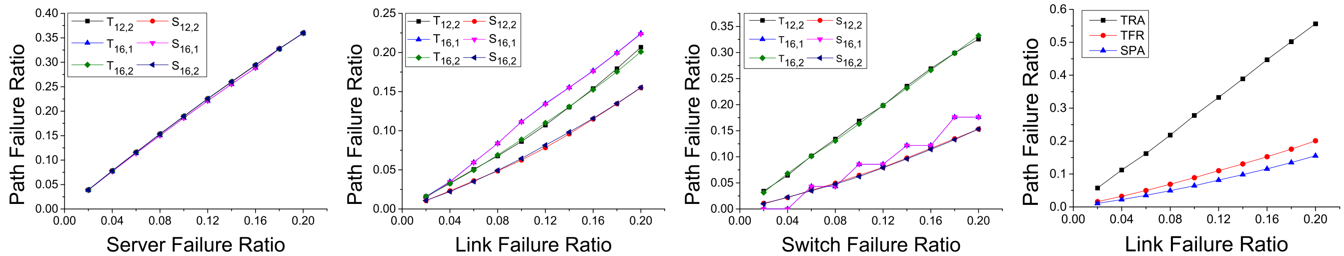
5 Experiments and Results

5.1 Fault Tolerance

In the experiments, we compare TFR and SPA which offers a performance upper bound under the structure of Totoro. The network parameters are shown in Table 3. Two arguments are considered: the switch ports (n) and the structural level (k). The design of the simulation experiments aim at studying how these two structural arguments affect the routing performance. The network scales vary from hundreds to thousands of servers. $T_{n,k}$ corresponds to the experiment which runs TFR on the Totoro structure with given n and k while $S_{n,k}$ corresponds to the experiment which runs SPA. Besides, each $Totoro_0$ is considered as a rack. Failures are generated randomly and the failure ratios vary from 2% to 20%. To achieve reliable experimental results, nodes route packets to all the other nodes 20 times in each simulation experiment and the final result is given by the average of the 20 running results. In each scenario, the numbers of packets to be sent in $Totoro_{12,2}$, $Totoro_{16,1}$ and $Totoro_{16,2}$ are about $3M$, $65K$ and $18M$, respectively.

Fig. 4(a) depicts the results of the path failure ratio under server failures. It shows that the performance of TFR is almost identical to that of SPA, regardless of the structural

For Manuscript Only



(a) (b) (c) (d)

Fig. 4: Evaluation of Path Failure Ratios

arguments of n and k. SPA is globally optimal and is always able to find out a path from the source to the destination if it exists. The remarkable performance of TFR benefits from the rerouting technique, through which, TFR can maximize the usage of redundant links when server failures occur.

Fig. 4(b) plots the path failure ratio versus link failure ratio. It can be observed that the path failure ratio of TFR increases as the link failure ratio rises. The proposed TFR is almost identical to that of SPA in the 1-level structure (i.e., $Totoro_{16,1}$). However, in the 2-level structure (i.e., $Totoro_{12,2}$ and $Totoro_{16,2}$), it cannot perform as well as SPA when the link failure ratio increases and the performance gap between them becomes larger and larger. For instance, in $Totoro_{16,2}$, the gap is 1% $(0.03 - 0.02)$ when the link failure ratio is 4%. It rises to 4% $(0.15 - 0.11)$ when the link failure ratio increases to 16%. This is because the link failure results in only very few nodes to be disconnected. SPA can achieve a good performance even when the link failure ratio is high. But TFR is not globally optimal and not guaranteed to find out an existing path. In fact, TFR is good enough when the link failure ratio is not large (i.e., lower than 10%). Furthermore, we also observe that TFR with a higher structural level has a lower path failure ratio. For example, when the link failure ratio is 12% with given $n = 16$, the path failure ratio is 13% when $k = 1$ (i.e., 1-level structure) while it is 10% when $k = 2$ (i.e., 2-level structure). This fact indicates that the fault tolerance of TFR is more apparent in a Totoro with more levels.

Fig. 4(c) depicts the result of the path failure ratio versus switch failure ratio. It shows that TFR achieves the performance equivalent to SPA in the 1-level structure (i.e., $Totoro_{16,1}$). But the performance gap between TFR and SPA becomes larger and larger with the increase of switch failure ratio in the 2-level structures (i.e., $Totoro_{12,2}$ and $Totoro_{16,2}$). It can also be observed that the path failure ratio of SPA becomes lower in a higher-level Totoro. It means that more redundant high-level switches help bypass the failure rather than become the single point of failure. For this reason, our next work is devoted to improving the performance of TFR under switch failure. Note that, the ladder-shaped polygonal line of $Totoro_{16,1}$ does not imply that the path failure ratio is strongly associated with a certain range of failure ratios. This is caused by the small number of switches in $Totoro_{16,1}$ and Totoro has the same number of failed switches in a range of ratios.

The results under rack failures are very similar to those of the server failures shown in Fig. 4(a). To evaluate the effects of our rerouting technology, we further compare TFR with TRA (i.e., the original routing in Algorithm 2 for Totoro without any rerouting technology) and SPA under link failures. The experimental results in Fig. 4(d) reveal that TFR greatly benefits from the proposed rerouting strategy.

It must be emphasized that the legacy SPA is impracticable in the real data centers due to its large traffic loads of sharing link states and its high computation complexity. But SPA offers the upper bound of routing performance and is used to compare the performance of our proposed TFR.

5.2 Throughput

We develop a flow-level simulator based on the approach [24] to evaluate the throughput of Totoro [3]. The Maximum Segment Size (MSS) is set to be $1500B$. In the current DCNs, the intra-rack RTT is approximately $100\mu s$ [25]. Hence, the flow's RTT is set as the result of $100\mu s$ multiplied by the number of switches along the path from the source to the destination. We use a synthetic flow workload from [26], which contains 80000 flows with the total size of $4TB$. The flow sizes vary from $1KB$ to $1GB$. The source and destination of each flow are randomly chosen from 0 to 4096. Besides, all flows are launched within 135 seconds.

Furthermore, we build Totoro and five state-of-the-art DCN structures, namely ThreeTier, FatTree, DCell, BCube and FiConn. Specifically, 16-port switches are adopted to construct a 2-level Totoro. ThreeTier structure uses 16-port switches in each level. Each ToR switch in ThreeTier has sixteen $1Gbps$ downlinks and five $1Gbps$ uplinks (i.e., $3.2 : 1$ oversubscription). Each Aggregate connects sixteen ToR switches and has one $10Gbps$ uplink to the core (i.e., $8 : 1$ oversubscription). Each switch used in FatTree has 26 ports. DCell is built as a 2-level structure with 8-port switches. BCube and FiConn utilize 16-port switches and both have 2-level structures. The number of nodes in Totoro, ThreeTier, FatTree, DCell, BCube and FiConn are 4096, 4096, 4394, 5256, 4096, and 5328, respectively. Except the uplinks of Aggregation switches in ThreeTier, data rates of other links in the experiment are all $1Gbps$.

The throughput results for DCNs are depicted in Fig. 5. FatTree and BCube complete the data transmission firstly while the ThreeTier structure takes about 65 seconds longer. As FatTree and BCube both use vast switches and abundant links to connect servers, they can achieve

3. The simulator is available in http://dsc.jnu.edu.cn/projects/totoro/totoro-exp.tar.gz now.

For Manuscript Only

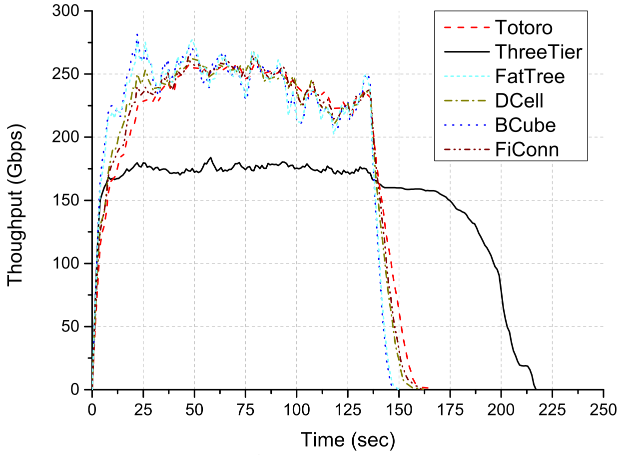


Fig. 5: Throughput Comparison of DCNs.

the highest bandwidth. Although their wirings are much sparser than FatTree and BCube, Totoro, DCell and FiConn also achieve comparable throughput, which only takes 10 seconds longer. During the transmission, the highest throughput of Totoro, FatTree, DCell, BCube and FiConn all exceed $250Gbps$. However, the highest throughput of ThreeTier hovers around $160Gbps$, equal to the total link capacity of the core switch. This is mainly because the core-level switch of ThreeTier becomes the bottleneck. It is also notable that ThreeTier will degrade more seriously since the congestion in the core may cause queues to build up the buffer of each lower-level sender.

6 Architecture Analysis

6.1 Robustness

Bilal et. al. [27] proposed a *Deterioration* metric σ_M, which can be calculated as the difference between the average metric value at various failure percentages and the initial metric without failure M_0, divided by M_0. σ_M can be represented as

$$\sigma_M = \left| \frac{1}{M_0} \left(\frac{\Sigma_{i=1}^{n} M_i}{n} - M_0 \right) \right|, \tag{8}$$

where M_i is the value of metric M when i percent of the nodes fail. A lower σ_M represents the higher robustness.

Here we calculate σ_M of a $Totoro_2$ ($n = 14$) with 2744 servers in total for six graph metrics namely:

Cluster size $max(v)$**:** the size of the largest connected component. A larger $max(v)$ means a more robust structure;

Average shortest-path length $\langle l \rangle$**:** the average length of all the shortest paths among all node-pairs. The smaller $\langle l \rangle$ represents the higher robustness;

Average nodal degree $\langle d \rangle$**:** the average degree of all nodes. A larger $\langle d \rangle$ exhibits the better robustness (see the performance analysis for more details);

Algebraic connectivity $\mu_{|v|-1}$**:** the second smallest Laplacian eigenvalue. The larger value translates to the higher robustness;

Symmetry ratio $\frac{\varepsilon}{D+1}$**:** the quotient between the number of distinct eigenvalues of the network adjacency matrix and the network diameter. The lower the value, the better the robustness (see the performance analysis for more details);

TABLE 4: Comparison of Robustness in terms of σ_M.

Metric	FatTree	ThreeTier	DCell	Totoro (σ_M)
$max(v)$	Middle	Low	High	Middle (0.8034)
$\langle l \rangle$	Middle	Low	High	High (0.6121)
$\langle d \rangle$	Middle	Low	High	High (0.3163)
$\mu_{\|v\|-1}$	High	Low	Middle	Middle (0.0176)
$\frac{\varepsilon}{D+1}$	Middle	Low	High	High (0.7770)
λ_1	Middle	Low	High	High (0.1142)

Spectral radius λ_1**:** the largest eigenvalue of the network adjacency matrix. The structure with a larger spectral radius is considered more robust.

We only consider the *targeted attack*, in which the most vital nodes [4] are removed to disconnect the network, taking into account 1% to 6% of the nodes failure. Note that $\langle l \rangle$, $\mu_{|v|-1}$, $\frac{\varepsilon}{D+1}$ and λ_1 are merely calculated for the largest connected component. Thus we can observe that the robustness increases during the higher nodes failure for $\langle l \rangle$, $\mu_{|v|-1}$ and $\frac{\varepsilon}{D+1}$. This also proves that the classic metrics are not appropriate for quantifying the DCN robustness.

We also collect approximate $\sigma_M s$ of other three state-of-the-art DCN structures from [27], namely DCell, FatTree and ThreeTier. Since such data is received from the similar scale of DCNs, i.e., about 2K nodes, we compare Totoro with them and present the qualitative results in Table 4. Like DCell, Totoro shows higher robustness than FatTree and ThreeTier for almost all metrics. For $max(v)$, DCell outperforms Totoro because the wiring number in DCell is much denser than Totoro. However, the comparison results still confirm that Totoro is inherently fault-tolerant.

6.2 Topological Properties

To evaluate the topology, we compare Totoro with the traditional ThreeTier structure and several recent structures, such as FatTree, DCell, BCube and FiConn. Let T, n and k denote the total number of servers, the number of ports on a switch and the structural level, respectively. Specially, ThreeTier structure adopts Cisco data center ThreeTier model topology [7], which consists of *core*, *aggregation* and *access* layers. n_{acc} servers are connected to an access switch (no redundancy for access layer) which uses their uplinks to connect all aggregation switches. Each aggregation switch also has n_{agg} downlinks to all access switches. ThreeTier structure contains N_{agg} aggregation modules and each module consists of two aggregation switches (one for redundancy). In the core layer, two core switches (one for redundancy) can access each aggregation switches respectively. For simplicity, We do not consider any inter-switch link. Table 5 summarizes the topological comparison results.

Number of servers: DCell and FiConn grow double-exponentially with the level k. Totoro and BCube are exponentially incremental. The scale of ThreeTier and FatTree is limited by switches' ports and thus they lack scalability.

Number of switches: In practice, ThreeTier uses less switches than other structures. But such switches are high-density and more expensive. FatTree needs most switches

4. For Totoro, we found that the most effective method to disparate the network is to remove some high-level inter-switches.

For Manuscript Only

TABLE 5: Comparison of Topological Property.

Structure	# of Servers (T)	# of Switches	# of Wires	Degree	Diameter
ThreeTier	$n_{acc}n_{agg}$	$n_{agg}+2N_{agg}+2$	$T+2N_{agg}n_{agg}+2N_{agg}$	1	6
FatTree	$\frac{n^3}{4}$	$5\cdot\frac{T}{n}$	$3\cdot T$	1	6
DCell	$>(n+\frac{1}{2})^{2^k}-\frac{1}{2}$	$\frac{T}{n}$	$(1+\frac{k}{2})\cdot T$	$k+1$	$2^{k+1}-1$
BCube	n^{k+1}	$(1+k)\cdot\frac{T}{n}$	$(1+k)\cdot T$	$k+1$	$k+1$
FiConn	$>2^{k+2}\times(\frac{n}{4})^{2^k}$	$\frac{T}{n}$	$(\frac{3}{2}-\frac{1}{2^{k+1}})\cdot T$	$2-\frac{1}{2^k}$	$2^{k+1}-1$
Totoro	n^{k+1}	$(2-\frac{1}{2^k})\cdot\frac{T}{n}$	$(2-\frac{1}{2^k})\cdot T$	$2-\frac{1}{2^k}$	2^k

TABLE 6: Comparison of Bandwidth

Metric	BiW	AR_{o2o}	$AR_{o2m/m2o}$	BoD
ThreeTier	$\frac{n_{core}N_{core}}{2}$	1	1	$\frac{T^2}{N_{core}}$
FatTree	$\frac{T}{2}$	1	1	T
DCell	$\frac{T}{4\log_n T}$	$k+1$	$k+1$	$T\cdot 2^k$
BCube	$\frac{T}{2}$	$k+1$	$k+1$	$T\cdot 2^k$
FiConn	$\frac{T}{4\times 2^k}$	2	2	$T\cdot 2^{k+1}$
Totoro	$\frac{T}{2^{k+1}}$	2	2	$T\cdot 2^k$

For Manuscript Only

to be non-oversubscribed. Compared to DCell, BCube and FiConn, Totoro uses considerable amount of switches.

Number of wires: The number of wires reflects the density of available paths among nodes as well as the overhead of deployment and maintenance. As the structure level k increases, FiConn has the sparsest wiring density. The number of wires in Totoro will be almost twice as large as server nodes and this is a moderate situation to balance the performance and the deployment overhead.

Degree: The server degree of Totoro and FiConn approaches to 2 as k grows, but will never reach 2. They all achieve a smaller node degree than DCell and BCube, which means a lower overhead for deployment and maintenance. Furthermore, Totoro and FiConn are always incomplete and highly scalable by using available backup ports.

Diameter: It is known that the smaller the diameter is, the more efficient the routing mechanism will be. Both ThreeTier and FatTree have a fix diameter of 6. BCube achieves the smallest diameter among all structures. The diameters of Totoro, DCell and FiConn increase exponentially as the structural level grows. Due to the fact that a low-level Totoro can hold a large number of servers (e.g., a $Totoro_2$ with $n = 32$ has $32,768$ servers and a $Totoro_3$ with $n = 16$ has $65,536$ servers), the diameters of $Totoro_2$ and $Totoro_3$ are only 10 and 18, respectively. In addition, even though the diameters of ThreeTier and FatTree are both small, they cannot be comparable to Totoro since their scalability is limited by the number of switch ports.

6.3 Bandwidth

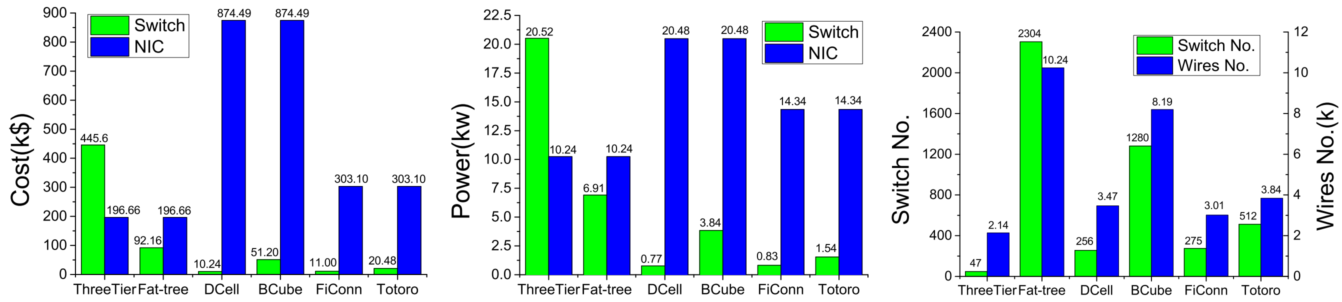
We evaluate four metrics about bandwidth to compare Totoro with state-of-the-art structures:

Bisection Width (BiW): The bisection width of ThreeTier is related to its core layer design. n_{core} denotes the ports of a core switch. N_{core} represents the number of core switches. To separate a ThreeTier structure, we only need to unplug half of its core links. This implies that the bandwidth of ThreeTier is totally limited by the core layer. FatTree uses more redundant switches in each layer. It has a large bisection width of $T/2$. DCell has a large bisection width of $T/(4\log_n T)$ since there are more ports on a server. The BCube structure is considered closely related to the generalized Hypercube [11]. It also achieves a large bisection width of $T/2$, which inherits the good characteristics of Hypercube. The bisection width of FiConn and Totoro are similar. As aforementioned, a low-level Totoro can hold a large number of servers. When we take a small number of k, the bisection width is large, e.g., $BiW = T/4$, $T/8$, $T/16$ when $k = 1, 2, 3$, respectively. A large bisection width means a fault-tolerant and resilient structure. In addition, a relative large bisection width also leads to the higher network capacity.

Acceleration Ratio under One-to-One communication models (AR_{o2o}): Parallel paths between two nodes help accelerating the communication in One-to-One model. If these paths are node-disjoint, the acceleration will be more significant. There exist more than one links from a certain server leaf to the access layer in ThreeTier. But only one is usually active and others are blocked for backup. Thus no acceleration is supported under One-to-One communication. FatTree suffers the same problem. For DCell, BCube, FiConn and Totoro, it can be proved that the acceleration ratio under One-to-One communication model, i.e., the number of parallel paths which are node-disjoint, is the port number of NIC. Note that in Totoro and FiConn, no parallel paths exist if the source or destination node is one-degree. This is a worthwhile trade-off to provide incremental scalability.

Acceleration Ratio under One-to-Many and Many-to-One communication models (AR_{o2m} and AR_{m2o}): In some distributed file systems like GFS and HDFS, One-to-Many and Many-to-One communication models are common and can be accelerated by using multiple paths. Take One-to-Many model for example, the source node distributes several unique data blocks through different paths. Then the destination nodes share their own blocks with each other to finish the whole process. Thus the One-to-Many communication is accelerated. It is required that these parallel paths are edge-disjoint, i.e., no edges appear on two paths simultaneously. Edge-disjoint complete graphs can be built to determine whether a structure can speed up One-to-Many communication [11]. For DCell, BCube, FiConn and Totoro, the acceleration ratio under O2M or M2O model equals the port number of NIC.

Bottleneck Degree (BoD): All-to-All communication model is widely used in parallel data processing framework, e.g., MapReduce. BoD is a metric that denotes the maximum



(a) Cost (b) Power (c) Switches and Wires

Fig. 6: Comparison of Cost and Power

For Manuscript Only

number of flows traveling through a single link under the All-to-All communication model. The bigger BoD means the heavier traffic workloads. For ThreeTier, N_{core} represents the number of core switches. ThreeTier has a large BoD since a large number of flows burden the core layer much. FatTree has the smallest BoD owing to the non-oversubscribed design. The BoD of Totoro is related to the structural level k. It is much smaller than ThreeTier, which means that the All-to-All flows are spread out over all links.

6.4 Cost and Power

Cost and power consumption are two significant issues for DCNs. Costs in DCN contain four major components: servers, infrastructure, power draw and network [6]. The existing studies have shown that datacenter computers rarely operate at full utilization [28] [29]. Some studies aimed to reduce the rapidly growing energy (or power) consumption [30] [31]. In this paper, we only discuss cost and energy consumption about static deployment [5], leaving energy-saving routing as our future work.

We build a container with 2048 servers as our comparison model. The price and power consumptions of switches and NICs are from Cisco, eBay and [11]. We build the same ThreeTier as that in Section 6.2 according to [7], including 43 access (WS-C4948-10G-E, $48 \times 1GE + 2 \times 10GE$), 2 aggregation (WS-C6506-E, $48 \times 10GE + 4 \times 40GE$) and 2 core (WS-C6504-E, $4 \times 40GE$) switches. The oversubscription ratios are 4.8 : 1 for access layer and 6 : 1 for aggregation layer. For FatTree, DCell, BCube, FiConn and Totoro, commodity 8-port switches (D-Link DGS-1008D) are used. ThreeTier[6] and FatTree require 1-port NICs (Intel EXPI9400PT). Servers in DCell and BCube are equipped with 4-port NICs (Intel EXPI9404PT). FiConn and Totoro need 2-port NICs (Intel EXPI9402PT). Fig. 6 depicts the comparison of cost, power and the numbers of switches and wires.

For ThreeTier, switches are much more expensive than those of other structures and consume the most power, even thought they are few in number. FatTree has the largest number of switches while they are commodity ones and thus the cost and power consumption are moderate. DCell and BCube have a higher cost and power consumption on NIC due to their usage of 4-port NICs. Totoro and FiConn control the switch cost successfully and achieve a graceful power saving. The numbers of switches and wires of Totoro also imply that their deployment overhead are acceptable and uncomplicated, which is consistent with the analysis of Degree property in Section 6.2.

In addition, ThreeTier and FatTree are usually over-subscripted in real data center environments. Their cost, power and wiring complexity can come down significantly as the over-subscription increases. However, the performance will degrade if the over-subscription is too large, as we discuss in the introduction. Moreover, the scaling of ThreeTier and FatTree are still limited to the number of switch ports and they are also not incrementally scalable.

In conclusion, Totoro is comparable with other structure in costs, power and deployment complexity. The superiority of Totoro is that it is incrementally scalable.

6.5 Price-performance Ratio

From the above analysis, Totoro and FiConn both achieve a relatively higher cost-efficiency. Totoro has a higher bisection width double that of FiConn while FiConn shows some advantages over Totoro in less cost, power and wires. Generally speaking, there always are some trade-offs in system design. Structures with a lower price/performance ratio are more desirable. Here we exploit **Price/BisectionWidth** ratio to evaluate the trade-off.

Suppose that T is the total number of hosts, n is the number of ports on a switch, k is the structural level, P_h and P_s are the prices of host and switch, respectively. The cost to build a $Totoro_k$ structure is:

$$C_t = T \times P_h + \frac{T}{n} \times (2 - \frac{1}{2^k}) \times P_s. \tag{9}$$

Similarly, the cost of $FiConn_k$ structure is:

$$C_f = T \times P_h + \frac{T}{n} \times P_s. \tag{10}$$

In combination with the bisection widths, the difference between their Price/BisectionWidth ratio is:

$$D = \frac{C_t}{T/2^{k+1}} - \frac{C_f}{T/(4 \times 2^k)}. \tag{11}$$

According to Eqs. (9) and (10), Eq. (11) can be transformed to:

$$D = 2^{k+1} \times (\frac{P_s}{n \times 2^k} - P_h). \tag{12}$$

5. We only consider the NICs, switches and wires since CPU, main memory and disk are not directly relevant to the topology structure.

6. For common ThreeTier structures, a server may be equipped with a 2-port NIC, one for active traffic and another for backup. For simplicity, we assume a 1-port NIC is required.

Since $n \geq 4$, $k \geq 2$ and the price of commodity switch is much less than that of host, we can draw a conclusion that D **is smaller than** 0, i.e., Totoro achieves a lower price-performance ratio than FiConn.

7 Conclusion and Future Work

The existing structures of interconnection networks are hardly to meet the requirements of both incremental s-calability and cost-efficiency. This drives us to develop a new structure called Totoro. The theoretical analysis and extensive experiments all demonstrate that Totoro satisfies the design goals of scalability, cost-effectiveness, high network capacity and robustness. The proposed structure can significantly help the DCN builders rethink the present design and provides an alternative solution to the existing DCNs, especially in the scenario that incremental scalability and cost-effectiveness are vitally required.

Even though Totoro achieves relatively high bandwidth and has a low network diameter, there exist trade-offs between such network goodness and economy. Similar to DCell, BCube and FiConn, packet-forwarding in Totoro may experience delays. This is mainly because we regard servers as "routers" in the forwarding and the packet-handling ability of current NIC is still weaker than the professional chips on switches or routers. But this weakness will be overcome as the NIC becomes more and more powerful.

In the future work, we will focus on the problem of energy saving and be devoted to the design of elastic routing, which is self-adaptive to the traffic mode to lower the overall energy consumption. As a consequence, that Totoro structure is elastic enough to balance the performance and energy conservation due to the vast redundant paths.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and useful suggestions which help us to improve Totoro. This work is supported by the NSF of China under grant (no. 61272073, and no. 61572232), the NSF of Guangdong Province (no. S2013020012865).

References

[1] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[2] K. Wu, J. Xiao, and L. M. Ni, "Rethinking the architecture design of data center networks," *Frontiers of Computer Science*, vol. 6, no. 5, pp. 596–603, 2012.

[3] Y. Sverdlik, "Microsoft plans quincy data center expansion," http://www.datacenterdynamics.com/focus/archive/2013/12/microsoft-plans-quincy-data-center-expansion, 2013.

[4] CISCO, "Cisco global cloud index: Forecast and methodology, 2013c2018," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.html, 2015.

[5] J. Dean, "Experiences with mapreduce, an abstraction for large-scale computation," in *Proc. Parallel Architectures and Compilation Techniques (PACT)*, vol. 6, 2006, pp. 1–1.

[6] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *Proc. ACM Special Interst Group Data Commun. (SIGCOMM)*, vol. 39, no. 1, pp. 68–73, 2008.

[7] Cisco, "Cisco data center infrastructure 2.5 design guide," 2007.

[8] D. Li, C. Guo, H. Wu *et al.*, "Ficonn: Using backup port for server interconnection in data centers," in *Proc. IEEE Int. Conf. Comput. Commun. (INFOCOM)*. IEEE, 2009, pp. 2276–2285.

[9] CNET, "Google uncloaks once-secret server," http://www.cnet.com/news/google-uncloaks-once-secret-server-10209580, 2009.

[10] C. Guo, H. Wu, K. Tan *et al.*, "Dcell: a scalable and fault-tolerant network structure for data centers," in *Proc. ACM Special Interst Group Data Commun. (SIGCOMM)*, vol. 38, no. 4. ACM, 2008, pp. 75–86.

[11] C. Guo, G. Lu, D. Li *et al.*, "Bcube: a high performance, server-centric network architecture for modular data centers," in *Proc. ACM Special Interst Group Data Commun. (SIGCOMM)*, vol. 39, no. 4. ACM, 2009, pp. 63–74.

[12] J. Xie, Y. Deng, and K. Zhou, "Totoro: A scalable and fault-tolerant data center network by using backup port," in *Proc. 10th IFIP Int. Network and Parallel Computing (NPC)*. IFIP, 2013, pp. 94–105.

[13] Y. Cui, H. Wang, X. Cheng, D. Li, and A. Yla-Jaaski, "Dynamic scheduling for wireless data center networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2365–2374, 2013.

[14] K. Bilal, S. U. R. Malik, O. Khalid *et al.*, "A taxonomy and survey on green data center networks," *Future Generation Computer Systems*, vol. 36, pp. 189–208, 2014.

[15] F. P. Tso and D. Pezaros, "Improving data center network utilization using near-optimal traffic engineering," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1139–1148, 2013.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM Special Interst Group Data Commun. (SIGCOMM)*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[17] C. Kim, M. Caesar, and J. Rexford, "Seattle: A scalable ethernet architecture for large enterprises," *ACM Transactions on Computer Systems*, vol. 29, no. 1, p. 1, 2011.

[18] R. Niranjan Mysore, A. Pamboris, N. Farrington *et al.*, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *Proc. ACM Special Interst Group Data Commun. (SIGCOMM)*, vol. 39, no. 4. ACM, 2009, pp. 39–50.

[19] Y. Deng, "Risc: A resilient interconnection network for scalable cluster storage systems," *Journal of Systems Architecture*, vol. 54, no. 1, pp. 70–80, 2008.

[20] K. Chen, C. Guo, H. Wu *et al.*, "Dac: generic and automatic address configuration for data center networks," *IEEE/ACM Transactions on Networking*, vol. 20, no. 1, pp. 84–99, 2012.

[21] CTOCIO, "Ten worst cloud crashes in 2011," http://www.ctocio.com/hot news/2370.html, 2011.

[22] G. Lu, Y. Shi, C. Guo, and Y. Zhang, "Cafe: a configurable packet forwarding engine for data center networks," in *Proc. ACM Special Interst Group Data Commun. (SIGCOMM) on Programmable routers for extensible services of tomorrow*, 2009, pp. 25–30.

[23] G. Lu, C. Guo, Y. Li *et al.*, "Serverswitch: A programmable and high performance platform for data center networks," in *Proc. Symp. Network System Design and Implementation (NSDI)*, 2011, pp. 15–28.

[24] M. Al-Fares, S. Radhakrishnan, B. Raghavan *et al.*, "Hedera: Dynamic flow scheduling for data center networks." in *Proc. Symp. Network System Design and Implementation (NSDI)*, vol. 10, 2010, pp. 19–19.

[25] M. Alizadeh, A. Greenberg, D. A. Maltz *et al.*, "Data center tcp (d-ctcp)," *Proc. ACM Special Interst Group Data Commun. (SIGCOMM)*, vol. 41, no. 4, pp. 63–74, 2011.

[26] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM Special Interst Group Data Commun. (SIGCOMM) on Internet measurement*. ACM, 2010, pp. 267–280.

[27] K. Bilal, M. Manzano, S. U. Khan *et al.*, "On the characterization of the structural robustness of data center networks," *IEEE Transactions on Cloud Computing*, vol. 1, no. 1, pp. 1–1, 2013.

[28] D. Abts, M. R. Marty, P. M. Wells *et al.*, "Energy proportional datacenter networks," in *Proc. ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 338–347.

[29] D. Li, Y. Yu, W. He, K. Zheng, and B. He, "Willow: Saving data center network energy for network-limited flows," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 9, pp. 2610–2620, 2015.

For Manuscript Only

[30] B. Heller, S. Seetharaman, P. Mahadevan *et al.*, "Elastictree: Saving energy in data center networks." in *Proc. Symp. Network System Design and Implementation (NSDI)*, vol. 3, 2010, pp. 19–21.
[31] Y. Guo, Y. Gong, Y. Fang, P. Khargonekar, and X. Geng, "Energy and network aware workload management for sustainable data centers with thermal storage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2030–2042, 2014.

Junjie Xie is a research student at the Computer Science Department of Jinan University. His current research interests cover network interconnection, data center architecture and cloud computing.

Yuhui Deng is a Professor in the Department of Computer Science at Jinan University. His research interests cover green computing, cloud computing, information storage, computer architecture, performance evaluation, etc.

Geyong Min is a Professor of High Performance Computing and Networking in the Department of Mathematics and Computer Science within the College of Engineering, Mathematics and Physical Sciences at the University of Exeter, United Kingdom. His research interests include Next Generation Internet, Wireless Communications, Multimedia Systems, Information Security, Ubiquitous Computing, Modelling and Performance Engineering.

Yongtao Zhou is a research student at the Computer Science Department of Jinan University. His current research interests cover data deduplication and distrtributed system.

For Manuscript Only