Totoro: A Scalable and Fault-Tolerant Data Center Network by Using Backup Port

Junjie Xie, Yuhui Deng, Ke Zhou

Totoro: A Scalable and Fault-Tolerant Data Center Network by Using Backup Port

Junjie Xie[1], Yuhui Deng[1,2], Ke Zhou[3]

[1] Department of Computer Science, Jinan University, Guangzhou, 510632, P.R. China
[2] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, P. R. China
[3] School of Computer Science & Technology, Huazhong University of Science & Technology, Key Laboratory of Data Storage Systems, Ministry of Education of China, P.R. China

[1] xiejunjiejnu@gmail.com; [2] tyhdeng@jnu.edu.cn

Abstract. Scalability and fault tolerance become a fundamental challenge of data center network structure due to the explosive growth of data. Both structures proposed in the area of parallel computing and structures based on tree hierarchy are not able to satisfy these two demands. In this paper, we propose Totoro, a scalable and fault-tolerant network to handle the challenges by using backup built-in Ethernet ports. We connect a bunch of servers to an intra-switch to form a basic partition. Then we utilize half of backup ports to connect those basic partitions with inter-switches to build a larger partition. Totoro is hierarchically and recursively defined and the high-level Totoro is constructed by many low-level Totoros. Totoro can scale to millions of nodes. We also design a fault-tolerant routing protocol. Its capability is very close to the performance bound. Our experiments show that Totoro is a viable interconnection structure for data centers.

Keywords: Data Center, Interconnection Network, Scalability, Fault Tolerance, Backup Port

1 Introduction

With the development of information digitization, large amounts of data is being created exponentially every day in various fields, such as industrial manufacturing, e-commerce, and social network, etc. For example, 72 hours of video are uploaded to YouTube every minute [1]. 1 billion active Facebook users upload 250 million photos every day. If these photos are printed and piled, the height would be as tall as 80 Eiffel Towers [2]. A report from IDC even shows that 1,800EB data has been created in 2011 with a 40-60% annual increase [3]. As data increases exponentially, the scale of data centers has been increased sharply.

In recent years, governments and multinational corporations are racing to invest amounts of money to build many large data centers. For instance, Google has already had 19 data centers where there are more than 1 million servers. Some corporations, such as Facebook, Microsoft, Amazon, eBay and so on, have hundreds of thousands

of servers in their own data centers. In this case, scalability becomes a necessary condition for data centers.

With the increasing scale of data centers, failures become quite common in the cloud environment. Failures from software, hardware, outage or even overheat will have a significant impact upon the running applications. For example, Amazon EC2 and RDS failed for several days, leading to the stoppage of some famous corporations [4]. Google also reports that 5 nodes will fail during a MapReduce job and 1 disk will fail every 6 hours in a 4,000-node cluster running MapReduce [5]. Hence, failures and their damages make fault tolerance a big challenge in the cloud environment.

In current practice, most of data centers are tree-based. At the top of its hierarchy, a tree-based data center provides the Internet services by core-routers or core-switches. However, there are three weaknesses about the top-level switches. Firstly, they can easily become the bandwidth bottleneck. Secondly, if one port fails, it will make its subtrees all isolated. In other words, they are the single points of failure. Thirdly, top-level switches are so expansive that updating will cause the steep rise in cost. In summary, tree-based structure lacks enough scalability and fault tolerance.

Fat-Tree [6] is an improved tree-based structure. It scales out with a large number of links and mini-switches. By using more redundant switches, Fat-Tree provides higher network capacity than traditional tree-based structures. But the scalability of Fat-Tree is still limited by the ports of switches fundamentally. DCell [7] is a level-based, recursively defined interconnection structure. It typically requires multiport (e.g., 3, 4 or 5) servers. DCell scales doubly exponentially with the server node degree. It is also fault tolerant and supports high network capacity. But the downside of DCell is that it trades-off the expensive core switches/routers with multiport NICs and higher wiring cost. FiConn [8] is also a new server-interconnection structure. It utilizes servers with two built-in ports and low-end commodity switches to form the structure. FiConn has a lower wiring cost than DCell. Routing in FiConn also makes a balanced use of links at different levels and is traffic-aware to better utilize the link capacities. However, the downside of FiConn is that it has lower aggregate network capacity.

Besides the structures mentioned above, there are various interconnection solutions presented in recent years, such as Portland [9], VL2 [10], CamCube [11] and so forth. These structures have their advantages in some aspects, yet they still have some deficiencies. In contrast to the existing work, we propose a new interconnection structure called Totoro. It utilizes commodity server machines with two ports as FiConn does. When constructing a high-level Totoro, the low-level Totoros use half of their available backup ports for interconnections as well. Totoro and FiConn share the similar principle to place the interconnection intelligence onto servers. In FiConn, all communication between two partitions flows through a unique link. This brings severe forwarding load to the servers at each end of this link. Unlike FiConn, there is no direct link between any two servers in Totoro. All servers communicate with each other through switches. There are multiple links connecting two partitions directly. All the data that flows from one partition to another partition can be distributed to these links. This lowers the forwarding load and makes the transmission more efficient. Totoro scales exponentially with the hierarchical level. A 3-level Totoro can hold more than 1-billion servers by using 32-port switches. Totoro is also fault tolerant, benefiting from multi-redundant links, which provides high network capacity. We

also design a fault-tolerant routing mechanism, whose capacity is very close to that of shortest path algorithm (SP) with lower traffic and computation overhead.

2 Totoro Interconnection Network

2.1 Totoro Architecture

Totoro is recursively defined. It consists of a series of commodity servers with two ports and low-end switches. We connect N servers to an N-port switch to form the basic partition of Totoro, denoted by $Totoro_0$. We call this N-port switch intra-switch. Each server in $Totoro_0$ is connected to an intra-switch by using one port and the rest of ports are called available ports. If consider a $Totoro_0$ as a virtual server, we denote the number of available ports in a $Totoro_0$ as c. Obviously, there is c = N. Next, we connect each $Totoro_0$ to n-port switches by using c/2 ports. Each $Totoro_0$ is connected to c/2 switches and each switch is connected to n $Totoro_0$s. Now we get a larger partition, which is denoted by $Totoro_1$ (e.g., in Figure 1). By analogy, we connect n $Totoro_{i-1}$s to n-port switches to build a $Totoro_i$ in the similar way. Note that, we will never connect switches with switches. We call a switch connecting different partitions an inter-switch. In a $Totoro_i$, switches and links connecting different $Totoro_{i-1}$s are called level-i switches and level-i links respectively. Peculiarly, the level of intra-switch is 0.

If the inter-switch has n ports, we note that the number of child partitions in a parent partition is also n. If we denote the number of available ports in a $Totoro_i$ as c_i, there is $c_i = c_{i-1} * n/2$. This is because a $Totoro_i$ has n $Totoro_{i-1}$s and we connect each

Table 1. Useful denotations and their meanings in the following text.

Denotation	Meaning
N	The number of ports on an intra-switch.
n	The number of ports on an inter-switch.
K	The top level in a Totoro.
$Totoro_i$	An ith level Totoro.
c_i	The number of available ports in a $Totoro_i$.
$[a_K, a_{K-1}, \ldots, a_i, \ldots, a_1, a_0]$	A (K+1)-tuple to denote a server, $a_i < n$ ($0 < i \leqslant K$) indicates which $Totoro_{i-1}$ this server is located at and $a_0 < N$ indicates the index of this server in that $Totoro_0$.
$(u - b_{K-u}, b_{K-u+1}, \ldots, b_i, \ldots, b_1, b_0)$	A combination of an integer and a (K-u+1)-tuple to denote a switch, $u \leqslant K$ indicates that it is a level-u switch, $b_i < n$ ($0 < i \leqslant K-u$) indicates which $Totoro_{u+i-1}$ this switch is located at and b_0 indicates the index of this switch among level-u switches in that $Totoro_u$.

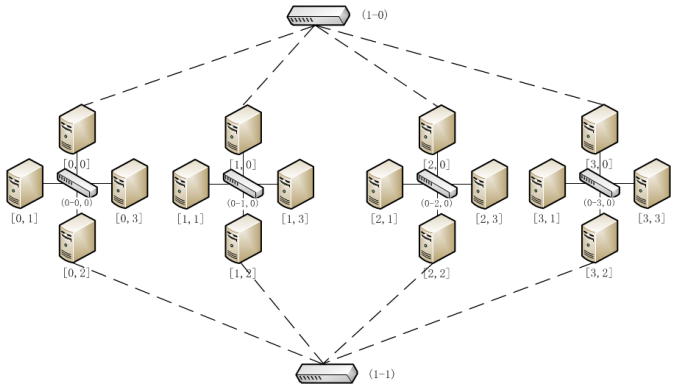

Fig. 1. A $Totoro_1$ structure with N = 4, n = 4. It is composed of 4 $Totoro_0$. Each $Totoro_0$ has 4 servers and an intra-switch with 4 ports. 4 $Totoro_0$ connect through 2 inter-switches.

$Totoro_{i-1}$ to $c_i/2$ inter-switches (level-i) by using half of its available ports ($c_{i-1}/2$). It implies that the number of paths among $Totoro_i$s is n/2 times of the number of paths among $Totoro_{i-1}$s. Multiple paths make the routing protocol robust and fault tolerant. Totoro can communicate in the presence of failures with redundant links. Furthermore, the number of high-level links is n/2 times of the number of low-level links means that high-level links will not be the bottleneck of the system and Totoro has high network capacity.

At the structure of Totoro, there are several inter-switches between two partitions. Servers in a $Totoro_i$ ($0 \leqslant i < K$) can access servers in another $Totoro_i$ directly by multiple paths without any other $Totoro_i$. For instance, in Figure 1, server [0, 1] wants to access [1, 1]. Under normal circumstances, we can choose the path [0, 1], (0-0, 0), [0, 0], (1-0), [1, 0], (0-1, 0), [1, 1]. Assume that the link between [0, 0] and (1-0) fails, this path is unavailable now. In this case, we can choose another path [0, 1], (0-0, 0), [0, 2], (1-1), [1, 2], (0-1, 0), [1, 1]. It is still the communication between two $Totoro_0$s without any other ones. Therefore, the structure of Totoro reduces the accessing distance between servers.

Observing the structure of Totoro, we find that not all servers are connected to inter-switches. In our design philosophy, we retain a large number of available ports for expanding. Thus, our Totoro is open and convenient to be expanded. We propose expanding Totoro by increasing the hierarchical level rather than updating the switches. This helps reduce the costs of devices and management in data centers. Without high-end devices, our Totoro also scales exponentially and we will discuss about this in Section 2.2.

2.2 Totoro Building Algorithm

Algorithm. 1. Totoro Building Algorithm.

```
0   TotoroBuild(N, n, K) {
1     Define t_K = N * n^K
2     Define server = [a_K, a_K-1, …, a_i, …, a_1, a_0]
3     For tid = 0 to (t_K - 1)
4       For i = 0 to (K - 1)
5         a_i+1 = (tid / (N * n^i)) mod n
6       a_0 = tid mod N
7       Define intra-switch = (0 - a_K, a_K-1, …, a_1, a_0)
8       Connect(server, intra-switch)
9       For i = 1 to K
10        If ((tid - 2^i-1 + 1) mod 2^i == 0)
11          Define inter-switch (u - b_K-u, …, b_i, …, b_0)
12          u = i
13          For j = i to (K - 1)
14            b_j = (tid / (N * n^j-1)) mod n
15          b_0 = (tid / 2^u) mod (N / n * (n/2)^u)
16          Connect(server, inter-switch)
17    }
```

Table 2. Total number of servers in $Totoro_u$ with different N, n, u.

N	n	u	t_u
16	16	2	4096
24	24	2	13824
32	32	2	32768
16	16	3	65536
24	24	3	331776
32	32	3	1048576

In Totoro, we can indicate a server in two ways: Totoro tuple or Totoro ID. Totoro tuple is a (K+1)-tuple [a_K, a_{K-1}, …, a_i, …, a_1, a_0]. It indicates where this server is located clearly and it will help calculate the common partition of two servers. For example, servers [0, 0] and [0, 1] in Figure 1, we know that these two servers are in the same $Totoro_0$[0] due to their common prefix (i.e., [0]). Totoro ID is an unsigned integer, taking a value from t_K (t_K is the total number of servers in a $Totoro_K$). Totoro ID will be used to identify a server uniquely. Note that, the mapping between Totoro tuple and Totoro ID is a bijection. In addition, we denote a switch as a combination of an integer and a (K-u+1)-tuple, (u - b_{K-u}, b_{K-u+1}, …, b_i,…, b_1, b_0). Algorithm 1 gives the Totoro building algorithm, which follows the principle in Section 2.1. The key in Algorithm 1 is to work out the level of the outgoing link of this server (Line 10).

Theorem 1 describes the total number of servers in $Totoro_u$:

Theorem 1:

$$t_u = N * n^u$$

Proof: A $Totoro_0$ has $t_0 = N$ servers. n $Totoro_0$s are connected to n-port inter-switches to form a $Totoro_1$. Hence, there are $t_1 = n * t_0$ servers in a $Totoro_1$. By analogy, a $Totoro_i$ ($i \leqslant u$) consists of n $Totoro_{i-1}$s and has $t_i = n * t_{i-1}$ servers. Finally, the total number of servers in a $Totoro_u$ is $t_u = N * n^u$.

3 Totoro Routing

3.1 Totoro Routing Algorithm (TRA)

Totoro routing algorithm (TRA) is simple but efficient by using Divide and Conquer algorithm. Assume that we want to work out the path from src to dst: Suppose src and dst are in the same $Totoro_i$ but two different $Totoro_{i-1}$s. Firstly, there must be a level-i path between these two $Totoro_{i-1}$s. We denote this path as P(m, n), which implies that m and src are in the same $Totoro_{i-1}$, while n and dst are in the same $Totoro_{i-1}$. Then, we work out P(src, m) and P(n, dst) respectively with the same technique. In this pro-

Algorithm. 2. Totoro Building Algorithm.

```
0  TotoroRoute(src, dst) {
1    If (src == dst)
2      Return NULL
3    Define k = lowestCommonLevel(src, dst)
4    If (k == 0)     // in the same Totoro0
5      Return P(src, dst)
6    Else
7      Define P(m, n) = getNearestPath(src, k)
8    Return TotoroRoute(src,m)+P(m,n)+TotoroRoute (n,dst)
9  }
```

Table 3. The mean value and standard deviation of path length in TRA and Shortest Path Algorithm in $Totoro_u$ of different sizes. M_u is the maximum distance between any two servers in $Totoro_u$.

N	n	u	t_u	M_u	TRA		Shortest Path Algorithm	
					Mean	StdDev	Mean	StdDev
24	24	1	576	6	4.36	1.03	4.36	1.03
32	32	1	1024	6	4.40	1.00	4.39	1.00
48	48	1	2304	6	4.43	0.96	4.43	0.96
24	24	2	13824	10	7.61	1.56	7.39	1.32
32	32	2	32768	10	7.68	1.50	7.45	1.26

cess, if we find src and dst are both in the same $Totoro_0$, we just return the directed path between them. Finally, we join the P(src, m), P(m, n) and P(n, dst) for a full path. Algorithm 2 follows the whole process mentioned above. The function getNearestPath just returns a nearest level-k path to the source host.

The SP that we use is Floyd-Warshall [13] algorithm. From Table 3, we observe that the performance of TRA is close to the SP under the conditions of different sizes. Although the SP is globally optimal, its computation complexity is as high as $O(n^3)$. It is not suitable for routing in data center. Our TRA is efficient enough and much simpler than the SP. Thus, we will build Totoro Fault-tolerant Algorithm based on TRA.

3.2 Totoro Broadcast Domain (TBD)

In order to send the packets with correct paths, servers need to detect and share the link states. Although global link states can help servers work out the optimal path, it is impossible to share the global link states in data center due to its large scale of nodes. Therefore, we introduce the definition of Totoro Broadcast Domain (TBD) to break up the network. We define a variable called bcLevel for broadcast domain, which means that a $Totoro_{bcLevel}$ is a TBD. The server in a TBD is called inner-server, while the server connected to a TBD with an outgoing link whose level is larger than bcLevel is called outer-server. Take Figure 1 for example, assume that bcLevel = 0. Then $Totoro_0[0]$ is a TBD. [1, 0], [2, 0], [3, 0], [1, 2], [2, 2], [3, 2] are the outer-servers of $Totoro_0[0]$.

Servers detect the states of links connected to them and broadcast the states to its intra-switch and inter-switch (if it has) periodically. If a server receives a link state packet, 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 so, broadcast it to the intra-switch. If not, broadcast it to the inter-switch if this server is connected to an inter-switch with a link whose level is smaller than bcLevel.

3.3 Totoro Fault-tolerant Routing (TFR)

In combination of TRA and TBD, we propose a distributed, fault-tolerant routing protocol for Totoro without global link states. Firstly, we give the constraint to bcLevel: bcLevel $\geqslant \log_n(2^K/N)$. It makes sure that every inner-server can find an arbitrary level link in its TBD as well as its link state.

We divide the Totoro network into several TBDs and they are connected by links with level $\geqslant$ bcLevel + 1. We use Dijkstra [12] algorithm for routing within TBD and TRA for routing between TBDs. Take Figure 1 for instance, assume that bcLevel is 0 and we want to find out the path from src[0, 2] to dst[1, 1]. Src and dst are in two different $Totoro_0$. By using TRA, we find that a level-1 link between these two $Totoro_0$ is required. We just get the nearest path P([0, 2], [1, 2]). In the real routing calculation, we just need to work out the next hop. Note that [1, 2] is an outer-server of this TBD. Hence, it can be simplified to work out the path from [0, 2] to [1, 2]. Then by using Dijkstra algorithm, we find that the next hop is [1, 2]. Furthermore, we add a proxy field to the packet header, which means a temporary destination. If this field is

Algorithm. 3. Totoro Fault-tolerant Routing Algorithm.

```
0  TotoroRoute(this, pkt) {
1    If (this == pkt.dst) deliver(this, pkt) and Return
2    ElseIf (pkt.proxy == this) pkt.proxy = NULL
3    If (pkt.ttl <= 0) drop(pkt) and Return
4    pkt.ttl -= 1
5    Define next = dijkstraRouting(pkt.dst)
6    If (next == NULL)
7      If (pkt.proxy == NULL)
8        Define k = lowestCommonLevel(this, pkt.dst)
9        Define pathSet = getLocalPaths(this, k)
10       Foreach (m, n) in pathSet
11         next = dijkstraRouting(n)
12         If (next != NULL)
13           pkt.proxy = n
14           Break
15     Else
16       next = dijkstraRouting(pkt.proxy)
17   If (next != NULL) deliver(next, pkt) and Return
18   drop(pkt) and Return
19 }
```

not empty, servers just need to find out the next hop to the proxy by using Dijkstra without TRA. After the packet arrives at the proxy, TRA will be used again to find out the next proxy.

If the proxy is unreachable (e.g., P([0, 2], [1, 2]) fails), we can set the proxy as [1, 0]. Then the failure will be bypassed successfully. In conclusion, TRA will be used to find out the proxy on the nearest path firstly. If it fails, TFR will reroute the packet to another proxy by using local redundant links. Moreover, if there exist several available links, TFR can choose one according to a random algorithm or the link load. Algorithm 3 shows the detailed procedure of TFR.

4 EXPERIMENT EVALUATION

4.1 Evaluating Path Failure

We use simulation to evaluate the performance of Totoro under four types of failures, including link, node, switch and rack failures. In the simulation, we also use SP to compare with our TFR. The SP that we used is based on Floyd-Warshall algorithm and it offers a performance upper bound under the structure of Totoro. The networks where we run TFR and SP are a $Totoro_1$ (N=48, n=48, K=1, t_K=2,304) and a $Totoro_2$ (N=16, n=16, K=2, t_K=4,096). Each $Totoro_0$ is a rack. Failures are generated randomly and their ratios vary from 2% to 20%. In our simulation, each node routes packets

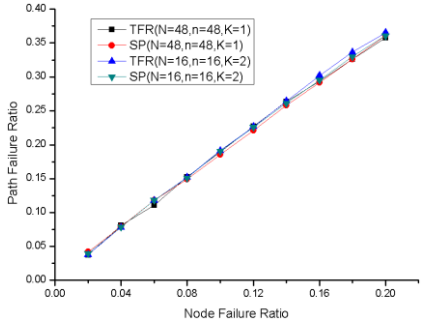

Fig. 2. Path failure ratio vs. node failure ratio.

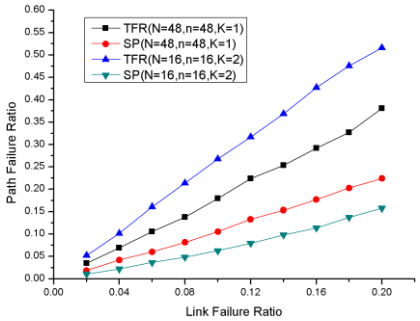

Fig. 3. Path failure ratio vs. link failure ratio.

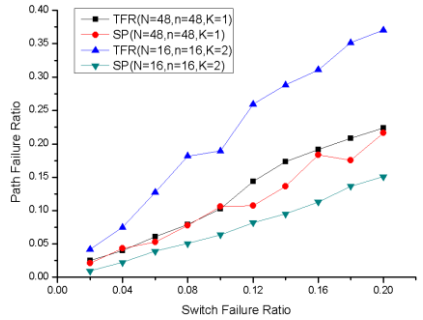

Fig. 4. Path failure ratio vs. switch failure ratio.

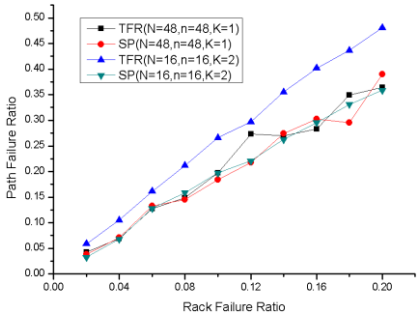

Fig. 5. Path failure ratio vs. rack failure ratio.

to all the other nodes 20 times. Therefore, each simulation result is an average of 20 running results.

Figure 2 plots the path failure ratio versus the node failure ratio. It shows that the performance of TFR is almost identical to that of SP, regardless of the number of servers. The server failure is quite common as we mention in Section 1 according to [5]. The remarkable capacity of TFR benefits from the technique of rerouting, which maximizes the usage of redundant links when a node failure occurs.

Figure 3 plots the path failure ratio versus the link failure ratio. We observe that the path failure ratio of TFR increases with the link failure ratio. Our TFR performs well when the link failure ratio is small (i.e., lower than 4%). But it can not perform as well as SP when the link failure ratio increases and the performance gap between them becomes larger and larger. For instance, in the same Totoro$_2$ (N=16, n=16, K=2), the gap is about 16% (0.21 - 0.05) when the link failure ratio is 8%. It rises to 32% (0.43 - 0.11) when the link failure ratio increases to 16%. This is because link failure just result in a very few nodes' being disconnected. The SP is global optimal and it always finds out a path from the source to the destination, if it exists. Thus, SP can achieve a good performance even when the link failure ratio is high. But our TFR is not global optimal and not guaranteed to find out an existing path. We also observe that Totoro which holds more servers has a lower path failure ratio. It implies that the integrated capacity of fault tolerance will be more obvious under the condition of a

large scale. This enlightens us to improve our TFR because of its huge performance improvement potential on handling link failure.

Figure 4 plots the path failure ratio versus the switch failure ratio. It shows that TFR performs almost as well as SP in $Totoro_1$ (N=48, n=48, K=1). But the performance gap between TFR and SP becomes larger and larger with the increase of switch failure ratio in the same $Totoro_2$ (N=16, n=16, K=2). We also observe that path failure ratio of SP is lower in a larger-level Totoro. It means that more redundant high-level switches help bypass the failure rather than become the single points of failure. Given this, our future work will be devoted to improving the performance of TFR under switch failure.

In our simulation, we also study the relationship between the rack failure and the path failure. We select a rack randomly and make all the nodes and links in this rack fail. Figure 5 plots the path failure ratio versus the rack failure ratio. It shows that in a low-level Totoro (e.g., $Totoro_1$), TFR achieves results very close to SP. But the capacity of TFR in a relative high-level Totoro (e.g., $Totoro_2$) could be improved. However, TFR performs still well enough when the rack failure ratio is lower than 10%.

4.2 Evaluating Network Structure

In this section, we compare Totoro with traditional Tree structure and several recent structures of Fat-Tree, DCell and FiConn to evaluate our network structure. Table 4 summarizes the topological property comparison result of different network structures. We denote the total number of servers as T.

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

As we all know, the smaller the diameter is, the more efficient the routing mechanism will be. The diameter of Tree is $2\log_{d-1}T$, where d is the number of switch ports. Fat-Tree has a diameter of $2\log_2 T$. The upper bound of diameter of DCell is $2\log_n T-1$. And the diameter of FiConn is O(logT). They all achieve a relative small diameter. It seems that Totoro has a large diameter. But it is not accurate in practice. A low-level Totoro can hold hundreds of thousands or even millions of servers, e.g., a $Totoro_2$ with N = n = 48 has 110,592 servers and a $Totoro_3$ with N = n = 32 has 1,048,576 servers. However, the diameters of $Totoro_2$ and $Totoro_3$ are only 10 and 18, respectively. In addition, even though the diameters of Tree and Fat-Tree are both small,

Table 4. Topological property comparison of different network structures.

Structure	Degree	Diameter	Bisection Width
Tree	--	$2\log_{d-1}T$	1
Fat-Tree	--	$2\log_2 T$	T/2
DCell	k + 1	$<2\log_n T-1$	$T/4\log_n T$
FiConn	$2 - 1/2^k$	O(logT)	O(T/logT)
Totoro	$2 - 1/2^k$	O(T)	$T/2^{k+1}$

they can not be comparable with Totoro since their scalability is limited by the number of switch ports.

Tree structure has a bisection width of 1 because each server or switch has only one path to the upper node. The failure of one path will make their subtrees all isolated. What's more, high-level links will become the bandwidth bottleneck. Fat-Tree overcomes these problems by using more redundant switches near the root in its hierarchy. It has a large bisection width of T/2. DCell has a large bisection width of $T/4\log_n T$ since it has more ports on a server. And the bisection width of FiConn is O(T/log(T)). Totoro also has a relative large bisection width of $T/2^{k+1}$. As we have mentioned above, 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. Thus, we can draw a conclusion that Totoro is fault-tolerant both in the topological analysis and from the path failure evaluating above. In addition, a relative large bisection width also leads to a higher network capacity.

In a word, Totoro gains the good scalability, fault tolerance and relative high network capacity. These attractive properties all meet the current requirement of data centers. Hence, our Totoro is a viable interconnection solution for data centers.

5 Conclusion

Structures neither proposed in the area of parallel computing nor based on tree hierarchy in current practice meet the requirements of scalability and fault tolerance. This drives us to present a new structure called Totoro. Then we detail the physical structure of Totoro. It is hierarchically and recursively defined. Through topological property analysis we know that Totoro scales exponentially and it is convenient to expand Totoro by using backup built-in Ethernet ports. In addition, we elaborate a distributed and fault-tolerant routing protocol called TFR, which is designed to handle various failures. The experiments show that its capability of handling fault tolerance is very close to that of the SP, especially in the presence of server failure. Furthermore, TFR significantly reduces the network traffic of sharing link states and the computation overhead. Lastly, we compare Totoro with other interconnection structures in some aspects, including node degree, diameter, and bisection width. It shows that Totoro is able to satisfy the demands of scalability and fault tolerance. In a word, topological analysis, experiments and comparison prove that Totoro is a viable interconnection solution for data centers. One problem faced by Totoro is the failure handling under some kinds of failures. Failure handling depends largely on fault-tolerant algorithm. We have seen the huge performance improvement potential from TFR. In the future work, we will be devoted to solving this problem by using more techniques, such as multiplexing and retouring based on remote proxy.

Acknowledgments We would like to thank the anonymous reviewers for helping us refine this paper. Their constructive comments and suggestions are very helpful. This work is supported by the National Natural Science Foundation (NSF) of China under grant (No.61272073, No. 61073064), the Scientific Research Foundation for the Returned Overseas Chinese Scholars (State Education Ministry), the Educational Commission of Guangdong Province (No. 2012KJCX0013), the Science and Technology

Planning Project of Guangdong Province (No.2012A080102002), the Science and Technology Planning Project of Guangzhou (No. 2012J4100109), Open Research Fund of Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCH201107). The corresponding author of this paper is Yuhui Deng.

References

1. Statistics-YouTube, http://www.youtube.com/t/press_statistics
2. A Typical Day In the Internet, http://www.mbaonline.com
3. Gantz, J. F., Chute, C.: The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC (2008)
4. Ten worst cloud crashes in 2011, http://www.ctocio.com/hotnews/2370.html
5. Dean, J.: Experiences with MapReduce, an abstraction for large-scale computation. In: PACT: 15th international conference on Parallel architectures and compilation techniques, vol. 16, no. 20, pp. 1-1. ACM (2006)
6. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: ACM SIGCOMM Computer Communication Review, vol. 38, no. 4, pp. 63-74. ACM (2008)
7. Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., Lu, S.: Dell: a scalable and fault-tolerant network structure for data centers. In: ACM SIGCOMM Computer Communication Review, vol. 38, no. 4, pp. 75-86. ACM (2008)
8. Li, D., Guo, C., Wu, H., Tan, K., Zhang, Y., Lu, S.: FiConn: Using backup port for server interconnection in data centers. In: INFOCOM 2009, IEEE, pp. 2276-2285. IEEE (2009)
9. Niranjan Mysore, R., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Vahdat, A.: PortLand: a scalable fault-tolerant layer 2 data center network fabric. In: ACM SIGCOMM Computer Communication Review, vol. 39, no. 4, pp. 39-50. ACM (2009)
10. Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Sengupta, S.: VL2: a scalable and flexible data center network. In: ACM SIGCOMM Computer Communication Review, vol. 39, no. 4, pp. 51-62. ACM (2009)
11. Costa, P., Donnelly, A., O'shea, G., Rowstron, A.: CamCube: a key-based data center. Technical Report MSR TR-2010-74, Microsoft Research (2010)
12. Dijkstra, E. W.: A note on two problems in connexion with graphs. In: Numerische mathematik, vol. 1, no. 1, pp. 269-271. Springer, Heidelberg (1959)
13. Floyd, R. W.: Algorithm 97: shortest path. In: Communications of the ACM, vol. 5, no. 6, pp. 345. ACM (1962)
14. Deng, Y.: RISC: A resilient interconnection network for scalable cluster storage systems. In: Journal of Systems Architecture, vol. 54, no. 1, pp. 70-80. Elsevier (2008)
15. Parhami, B.: Introduction to parallel processing: algorithms and architectures. In: Series in Computer Science. vol. 1. Springer, Heidelberg (2006)
16. Loguinov, D., Kumar, A., Rai, V., Ganesh, S.: Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 395-406. ACM (2003)
17. Barroso, L. A., Dean, J., Holzle, U.: Web search for a planet: The Google cluster architecture. In: Micro IEEE, vol. 23, no. 2, pp. 22-28. IEEE (2003)