Improving Routing Performance via Dynamic Programming in Large-scale Data Centers

Junjie Xie, Lijun Lyu, Yuhui Deng, Member, IEEE, and Laurence T. Yang, Member, IEEE

Abstract—The Internet of Things has become a spotlight for a long period of time and generates massive amounts of sensor data. Thus data centers play more and more crucial roles in processing and analyzing the explosively increasing data. To remedy the shortcomings of traditional tree-based structure, many novel server-centric network structures have been proposed in recent years. Their original routing mechanisms based on Divide and Conquer (DC) are not able to work out the shortest paths. So there is still promotion room for communication delay reduction. Since Dynamic Programming (DP) is a classical strategy to obtain optimal solution, this paper proposes a routing mechanism based on DP and applies it to data center for better solving the weakness occurred by DC. Experiments firmly support the conclusion that adopting DP in routing calculation achieves appealing performance of short latency, great fault-tolerance and reasonable resource consumption. Theoretical analysis also proves that it is applicable to most popular structures.

Index Terms—Dynamic Programming, Internet of Things, Data Center, Routing.

I. INTRODUCTION

T

HE concept of Internet of things (IoT) was first proposed in 1999 by Auto-ID Center [1]. It refers to the interconnection of uniquely identifiable embedded computing-like devices within the existing Internet infrastructure. The things in the IoT cover a wide variety of devices such as heart monitoring implants, biochip transponders on farm animals, automobiles with built-in sensors and so on. Undertaking the task of ushering in automation in nearly all fields, the IoT has become a spotlight this years. However, due to the ubiquitous nature of connected objects in the IoT, an unprecedented number of devices are expected to be connected to the Internet. According to Gartner, there will be nearly 26 billion devices on the IoT by 2020. ABI Research estimates that more than 30 billion devices will be wirelessly connected to the IoT by 2020 [2]. Meanwhile, the enormous data generated by devices need to be processed in real time, which brings many challenges and workloads for IoT data centers. That is why large-scale data center networks with satisfactory performance are crucial to process and analyze the massive amounts of data. In purpose of promoting efficiency of the IoT, optimizing the performance of data center networks is the essential point.

Copyright (c) 2012 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.

J. Xie, L. Lyu and Y. Deng (corresponding author) are with the Department of Computer Science, Jinan University, Guangzhou, 510632, China. E-mail: tyhdeng@email.jnu.edu.cn.

L. T. Yang is with the Department of Computer Science, St. Francis Xavier University, Antigonish, NS B2G 2W5, Canada. E-mail: ltyang@stfx.ca

A desirable data center networking should meet three requirements. First, the network infrastructure must be scalable and enable incremental expansion [3]. Second, the data center networking must be fault tolerant against various types of failures [4] [5], since failures from software, hardware or even outage become quite ubiquitous [6] [7]. Last, in purpose of better supporting real-time services, the high network capacity and short latency are also requested. The efficiency of a data center network depends on a reasonable infrastructure and the corresponding routing algorithms. Due to the inherent vulnerabilities of traditional tree-based structure, Fat-tree [8], Portland [9] and VL2 [10] are proposed to enhance the performance of tree-based structures. However, such type of structures which put their linking and routing intelligence on switches is beyond the scope of this paper. Additionally, many scholars have proposed plentiful feasible schemes with economic servers and commodity switches to construct novel data center interconnections in recent years, such as DCell [11], Ficonn [12], Totoro [13] and so forth. These structures are all recursively defined, server-centric which put linking and routing intelligence on servers instead of switches. Thus the first aforementioned requirement is fulfilled. As for the fault tolerant capacity and latency, both of which should take consideration of the designation of routing algorithms.

For IoT data center networks which manipulate massive amounts of small message sensor data, the latency reduction comes to the priority. Apart from the construction of physical structure, routing algorithms in data center networks also play a significant role in decreasing data latency. An effective routing algorithm is able to figure out the shortest paths between nodes so as to accelerate data forwarding. As we know, the routing algorithms of most existing data center structure take advantage of the structural properties of symmetry and homogeneity and exploit Divide-and-Conquer approach to calculate the forwarding path. However, this kind of routing mechanism is simple but not truly effective since they can not assure to get an optimal solution to reduce forwarding overhead and communication delay. Take aforementioned DCell and Totoro for example, they both adopt a Divide-and-Conquer approach to work out the routing path. Servers in the same domain preserve link states of each other. All link states information needs to be updated at short intervals in case of any change or failure. Since the failures in data centers occur commonly, frequent data exchanges in network structure may be conducted, resulting to unnecessary consumption of bandwidth. During the routing process, every node executes routing algorithm to find a next hop to the final destination. This might also cause a relatively high CPU usage for all
servers comprising the path. For instance, the CPU utilizations are all over 40% for sender, receiver and forwarder in DCell [11]. Therefore, this broadcast policy is not very satisfactory. Actually, we can figure out the completed paths by only one time computation and release intermediate servers from repetitive computation.

Since Dynamic Programming (DP) is able to work out the optimal solution, we applied it to aforementioned data center structures instead of their original mechanisms. We name our work Athena Routing Mechanism (ARM) 1. The basic principle of ARM is transferring the responsibility of path calculation on source server, the intermediate nodes only take charge of transmitting data packets. The validation and capacity of a path are confirmed by path probing scheme, thus broadcast can be omitted for saving bandwidth. Besides, our routing algorithm is based on DP, hence the shortest paths can be worked out primarily in path calculation process. The selection of paths and load balancing are achieved by path probing procedure. In the following parts, we will present the fundamental theory and implementation of our ARM. We will also prove the high efficiency and superior fault tolerant capability of ARM at the support of extensive simulations. Moreover, our ARM is a propable option which can be generalized to most other server-centric data center structures mentioned before. In this paper, we choose Totoro as the physical network architecture to test the performance of ARM.

Our work has the following strong points:

1) Lower communication latency
2) Favorable fault-tolerant capability
3) Resources consumption saving

The rest of the paper is organized as follows. Section 2 introduces the related work predecessors have achieved. Section 3 elaborates on Athena Routing Mechanism. Section 4 describes the implementations and Section 5 use simulations to evaluate ARM. Lastly, Section 6 concludes the paper.

II. RELATED WORK

The study of improving routing performance have thrived for decades. Optimal routing algorithms are widely applied in the fields of both traditional network or emerging wireless sensor network (WSN). For example, Genetic algorithm (GA) is a classical heuristic that mimics the process of natural evolution for optimization or search problems. However, GA or other generic heuristic solutions do not guarantee to find out the optimal paths without enough iteration. They are more suitable for dynamic and unpredictable topology while current data centers are quite regular and symmetric. Thus we do not consider them as our available solution to data center network. In the following text, we will emphasize on the routing algorithms of server-centric structures, which follow regular linking philosophies.

Based on a Divide-and-Conquer approach, DCellRouting [11] firstly calculates the intermediate links which interconnects two substructures comprising the source and destination servers. Then a “left” sub-path and a “right” sub-path can be worked out recursively to form a completed path. In addition, DCell also adopts a broadcast scheme by dividing the whole network into broadcast domains. Hence the Dijkstra algorithm [15] can replace DCellRouting to find the next hop with shortest length in a broadcast domain. Moreover, there are Local-reroute and Jump-up policy assisting DCellRouting to achieve a high fault tolerant capacity. Namely, if an intermediate server fails to find a next hop by DCellRouting, data packets will be transmitted to a proxy server connects with the intermediate server through an equivalent or higher level of link. Totoro routing mechanism [12] shares the similar routing principle with DCell, so we will not repeat here.

BCubeRouting [16] finds a path from a source to a destination in a BCube structure by correcting one digit at one step to systematically build a series of intermediate servers. Since two neighboring servers which connect to the same level-i switch only differ at the ith digit in the address arrays. And the number of different digits of two address arrays is \( k+1 \) at most (\( k \) is the structural level), thus the longest shortest path length between any server pairs of a \( BCube_k \) is \( k+1 \). Nevertheless, this low-diameter feature is benefited from the corresponding structure with considerable wiring cost, which may not be very suitable to be generalized to other more economic structures.

The Traffic-Oblivious Routing (TOR) [12] of FiConn is also recursively defined to make use of the level-based feature. Each intermediate server runs TOR to find the next hop by lowest common level. TOR balances the use of different levels of links and servers. In order to further balance the traffic volume, Traffic-Aware Routing (TAR) is proposed. TAR utilizes a greedy approach to hop-by-hop setup of traffic-aware path on each intermediate server. That is, the intermediate servers always select the outgoing link with higher available bandwidth to forward the traffic. In a FiConn, if the outgoing link found by TOR has lower bandwidth than the other links, then it will be bypassed via randomly selecting a third FiConn in FiConn_{i+1} to relay the traffic. In order to avoid the considerable overhead caused by exchanging traffic states among servers, FiConn adopts a probing policy which is analogous to our ARM to establish routing entry for data flow. Even though, the path length of FiConn might be a weakness comparing with ARM.

In a word, except BCubeRouting, all routing algorithms above require intermediate servers involved in the routing process. The repetitive computing will undoubtedly cause unnecessary burden for intermediate servers. Moreover, there is still room to promote in path length for those routing algorithms. Therefore, our ARM can be a desirable solution to address the above problems for server-centric and cost-efficient data center networks.

III. THE ATHENA ROUTING MECHANISM (ARM)

In this paper, our Athena Routing Mechanism consists of routing algorithm and a path probing policy. As we know, the general routing algorithm is based on broadcasting link status in a broadcast domain, and the source and intermediate servers do repetitive computation to find a next hop continually.

---

1 A preliminary short version of this paper [14] appears in the 14th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP2014).
with the link status. This policy causes considerable waste of computation and network bandwidth inevitably. Hence we propose another routing mechanism called Athena Routing Mechanism (ARM) to address this problem. Since the physical structure is computable, the path to any destination host can be worked out by the source host solely. Intermediate nodes only take charge of forwarding data packets. Before moving to the detailed implementation of ARM, we first focus on presenting the basic algorithm, Athena Routing Algorithm (ARA), which is conducted by the source host.

A. Dynamic Programming (DP)

DP is a method for solving a complicated problem by breaking it down into several simpler subproblems, solving them and then combine the solutions of them to reach an overall solution for the whole problem. DP is usually used for optimization [17] [18] [19]. It has two crucial properties: optimal substructure and subproblem overlap. Optimal substructure requires that the solved substructures at each step is always optimal. Subproblem overlap means that the solutions of subproblems overlap and thus the space of subproblems become small. Subproblem overlap is the main difference between “Divide and Conquer” and “Dynamic Programming”.

To achieve DP, there are two ways: Top-down approach and Bottom-up approach. The former one only calculates the needed subproblems and memorize the solutions to the table. Whenever attempting to solve a new subproblems, we check the table firstly to get the solution if it exists. The later one solves every lower-level subproblems firstly and use their solutions to form the optimal solutions to the higher-level subproblems.

Taking advantage of optimal substructure and subproblem overlap, DP presents an optimal solution for the given problem with far less time. Some classical algorithm problems are solved by DP, including Fibonacci sequence, Sequence alignment, Tower of Hanoi puzzle, Egg dropping puzzle, knapsack problem and so forth.

In all aforementioned recursively defined structures, a high-level structure can be constructed by several low-level structures. Consequently, the path from a source to a destination can be divided to certain low-level paths. Besides, the shortest paths between two nodes are always existing, which means the requirement of optimal substructure is also met. Therefore, we can draw to conclusion that DP is able to applied to the routing computation process for data center networks.

Algorithm 1: Athena Routing Algorithm

```java
Function ARoute (src, dst, count)
1. if src == dst then
2. return NULL
3. u = getLCL (src, dst)
4. if u == 0 then
5. return P (src, dst)
6. Set topLinkSet = getTopLinks (src, dst, u)
7. Set result
8. for each link L ∈ topLinkSet do
9. leftPathSet = ARoute (src, L.left, count)
10. rightPathSet = ARoute (L.right, dst, count)
11. result.add (leftPathSet + L + rightPathSet)
12. SortByLength (result)
13. result = result.sublist (0, count)
14. return result
```

B. Athena routing Algorithm (ARA)

ARA is based on DP to obtain simplicity and efficiency. We present how we recursively work out constant number of shortest paths in Algorithm. 1. The function getLCL returns the lowest common level u of two nodes (Line. 4.). Then the function getTopLinks (Line. 7.) figures out all level-u links starting from the whole level-u substructure which the source is located. Afterwards, for each independent top link, we can recursively find a set of completed paths from the source server to the destination server. After each round of recursion completed, a sort function will be performed so that we can limit the needed number of shortest paths. What noteworthy is the total independent paths might share same nodes, even though we choose absolutely different links in every round of recursion. This is because different high-level paths may share the same low-level paths. So when one path fails, we can not assure all the other paths are unaffected. This is a tradeoff to facilitate the routing algorithm and we can alleviate this problem by detecting multiple paths simultaneously. As Fig. 1 indicates, a set of paths connecting the same source and destination contain the same overlap path which is already found out, thus repeated computation can be omitted.

C. Path probing of ARM

The basic idea of Athena Routing Mechanism is figuring out all completed paths by ARA before sending probing packet (Actually we need not find all paths in practice. A threshold value can be utilized to limit the number of paths). If the source host src maintains a path cache to the destination host dst, this step can be omitted. Then a set of probing packets will be dispatched to detect the capacities of selected paths. The intermediate servers record the link capacities in the probing packets and forward them to the destinations by the pre-calculated paths. After the probing packets arrive, the destination servers reply these probing requests by sending the probing packets back to the source hosts according to the original paths. Among all the valid paths, we tend to choose
the one with higher bandwidth and/or shorter length, so that we can get sufficient resources every time and utilize links evenly.

**D. Properties of ARA**

Our ARM is not constrained to some certain structure, but can be generalized to most server-centric recursive structures mentioned before. Take Totoro as an exemplification, our ARM performs well according to various experiments. Primarily, our routing algorithm works out all paths directly connecting the substructures which src and dst are located respectively. With the feature of DP, we can ensure the shortest path length.

Besides, our ARA takes consideration of all top-level switches connecting the substructures which src and dst are located respectively. Thus we can get sufficient feasible paths by ARA to achieve a desirable fault tolerant capacity. Moreover, the path probing policy of ARM is able to save a great number of bandwidth for Totoro, since the size of probing packet is far less than broadcast data packet among all servers. The intermediate servers are also released from heavy routing computation loads because they only carry on forwarding data packets.

**Theorem 1:** \( k \) is the structural level. \( N \) represents the total number of servers in \( \text{Totoro}_k \). \( T_k \) is denoted as the time complexity of calculating a level-\( k \)-path. The time complexity of ARA is

\[
    T_k = O(N \frac{k}{2}).
\]

**Proof:** Suppose the source host and the destination host locate in different substructures, denoted as \( \text{Totoro}_{k-1}[\text{src}] \) and \( \text{Totoro}_{k-1}[\text{dst}] \) respectively. These two substructures are connected directly with level-\( k \) switches. Hence shortest paths from \( \text{Totoro}_{k-1}[\text{src}] \) to \( \text{Totoro}_{k-1}[\text{dst}] \) must go through these switches directly without circuitously traversing other \( \text{Totoro}_{k-1}s \). The number of level-\( k \)-switches is \( (n/2)^k \), i.e., the number of level-\( k \) links is also \( (n/2)^k \). For each level-\( k \) link \((m, n)\), suppose \( m \) and \( n \) are two end nodes of this link (See Fig. 2). ARA has to calculate two sub-paths: one is from src to \( m \), another is from \( n \) to dst. Calculation of sub-path can be conducted by following the similar steps recursively. Therefore, we can get \( T_k = 2 \times T_{k-1} \times (n/2)^k \). According to mathematical induction, we can finally figure out the following equation

\[
    T_k = \frac{N \frac{k}{2}}{2^{k-1}}.
\]

Omitting the denominator (since it is lager than 1), theorem 1 is rigidly proved. Note that \( k \) is a small integer, a low-level Totoro (e.g., \( n = 32, k = 3 \)) still can support more than one million servers \((32^{2^{3+1}} = 1048576)\). Thus the time complexity of ARA is relatively low when \( k \) is small.

Similarly, we can also get the time complexity of ARA in DCell and Ficonn respectively in following theorem 2 and theorem 3. The proofs will be elaborated in Appendix.

**Theorem 2:** \( k \) is the structural level. \( N \) represents the total number of servers in \( \text{DCell}_k \). \( T_k \) is denoted as the time complexity of calculating a level-\( k \)-path. The time complexity of ARA is

\[
    T_k = O(N \log^2 N).
\]

**Theorem 3:** \( k \) is the structural level. \( N \) represents the total number of servers in \( \text{Ficonn}_k \). \( T_k \) is denoted as the time complexity of calculating a level-\( k \)-path. The time complexity of ARA is

\[
    T_k = O(N \log^2 N).
\]

Comparing with SPA (based on Floyd-Warshall algorithm and the time complexity is \( O(N^3) \)), our ARM is able to work out the shortest paths with more acceptable complexity. Hence all above structures will get a huge promotion of average path length. Besides, according to the physical properties of the architecture, the paths between arbitrary pair of nodes can be totally figured out by the source server solely. Especially for DCell, which adopts broadcasting link status among servers, our path probing mechanism will efficiently save network bandwidth as well as relieve computation load for intermediate servers. All in all, our proposal can promote the whole efficiency and fault tolerant capability for many structures.

**IV. ATHENA PROTOCOL IMPLEMENTATION**

**A. ARM Address**

Since most applications are based on TCP/IP, we then design ARM protocol as a 2.5-layer protocol. In adaptation of Totoro structure, we represent a specific server by a 32-bits tuple named ARM Address. Since it has the same length with IP address, we then utilize this tuple in place of IP address in the IP header, i.e., we set the IP address to the same value of ARM address. Thus we can use the source and destination address from IP header directly, rather than add two additional fields.

There are three fields in and ARM address: \( L_i \), \( \text{dir} \) and \( \text{vmid} \). \( L_i \) denotes the server position in the network. In this paper, we suppose \( i \) is no more than three so that \( L_i \) consists of tuples from \( L_0 \) to \( L_3 \) with 6-bits length each. Actually, \( i \) indicates the level of Totoro structure. Note that a 4-level Totoro structure \((k = 3, n = 48)\) can support as many as five millions servers. And we can simply complete the high-order position or adjusting length of each \( L_i \) field to apply to a smaller structure with less servers. The \( \text{dir} \) takes up one bit to indicate this port connects to an intra-switch or inter-switch. \( \text{vmid} \) means the index of virtual machine in a physical server. It occupies seven bits so that we can support 127 virtual machines at most (\( \text{vmid} = 0 \), represents the physical server itself). We set this field only for adapting the trend of cloud
B. Packet format

There are two types of packet: path-probing packet and data packet. Before dispatching a data packet, a set of path-probing packets will be delivered first to confirm the capacities of selected paths. It involves the source address and destination address, as well as the capacity of one path and so forth. The difference between path-probing packet and data packet is the former involves the fields of source and destination address as well as capacity. Fig. 3 shows the format of packet header of these two types of packets.

C. ARM protocol

As the fact that two adjacent servers in a path only differ at one “bit” (i.e., one item in Totoro tuple), we adopt the path transformation vector to preserve path information in Fig. 4. Vector is included in a data packet, and each item of this vector represents “one-bit” change. An intermediate server determines the next hop according to the value of item which current pointer is pointing at. This approach of preserving the path information in a vector has been adopted in former research [20]. If a server receives a packet from the upper layer, it first checks whether the destination address is a loop address or not. If so, then it returns the packet to the upper layer. Otherwise, the server checks if it maintains a cache of path information to the destination. If not, it then employs ARA to figure out a set of paths and update path cache. Then the probing packet of request is constructed, in which the path transformation vector is also initialized. Afterwards, the probing packet is dispatched, intermediate servers forward it to the next hop according to the vector. When it arrives at the destination host, the host will send back a reply message along the previous path. Then a data packets will be dispatched to the destination along the selected path.

V. EXPERIMENTS AND RESULTS

A. Totoro Structure

In order to evaluate the performance of our Athena Routing Mechanism, we simulate it on the physical framework of Totoro and compare with the original Totoro Fault Tolerant Algorithm (TFR) and Shortest Path Algorithm (SPA, based on Floyd-Warshall [21]). TFR is the original Totoro Fault Tolerant Algorithm, which broadcasts link status in a domain and calculates the routing path hop-by-hop. This section will briefly presents the physical structure of Totoro.

Totoro structure consists of a series of commodity servers with dual ports and low-end commodity switches. The basic partition of Totoro is denoted as Totoro_0, constructed by n servers connecting to an n-port switch. As mentioned before, Totoro is a server-centric structure with recursive definition. A Totoro_i (i > 0) is constructed from n Totoro_{i-1}s. Each round of construction consumes half of the total available ports, and the rest half are remained for expansion. As Fig. 5, a Totoro_1 structure with N = 4, n = 4 is composed of 4 Totoro_0s. Each Totoro_0 has 4 servers and an intra-switch with 4 ports. 4 Totoro_0s connect through 2 inter-switches. Unlike DCCell and FiConn, there are duple direct links between two equivalent substructures, thus the redundant links can be fully used for distributing data flows. Please refer to [13] for details.

B. Evaluating Failure

In Fig. 6, we evaluate the path failure ratio of Totoro using ARM under four types of failure, including link failure, server failure, switch failure and rack failure. We run ARA, TFR and SPA on a Totoro_2 (n = 16, k = 2, t_k = 4096) under those four types of failures. Meanwhile, the results are compared with TFR and SPA. A rack consists of a whole Totoro_0. Failures are generated randomly ranging from the ratio of 2% to 20%. Servers deliver packet to other nodes 20 times, Which means every final result is the average of 20 running results.

Before presenting our experiment results, we will introduce TFR and SPA briefly. By TFR, servers randomly choose a
nearest level-\(u\) (\(u\) denotes the lowest common level with destination) link to the next hop, proceeding this process recursively until finding the destination. If failure occurs, another level-\(u\) or even higher links will be adopted. In addition, Totoro breaks the whole network into broadcast domains (TBD). In a TBD, link status are exchanged by broadcast among all servers. Thus the Dijkstra algorithm can be performed in a TBD to shorten path length. SPA is based on Floyd-Warshall algorithm, which requires global link states information to find out the shortest path. Consequently, SPA is globally optimal whereas the time complexity of it is as high as \(O(N^3)\), where \(N\) denotes the total number of servers in a Totoro structure.

As Fig. 6a and Fig. 6b indicate, the path failure ratios of the all three algorithms are equivalent under server and rack failure respectively. That is, the fault tolerant capacity of ARM is almost optimal. It also proves that ARM makes full use of redundant links and switches between two substructures. Rack failure means all servers in a rack are invalid, so it is analogous to server failure. In Fig. 6c, under switch failure scenario, ARM performs much better than the original TFR. From Fig. 6d, we can see when a high link failure occurs, ARM achieves slightly better fault tolerant capacity than TFR. But compared with SPA, ARM is still a bit inferior. This makes sense since all top-level links calculated in Totoro are direct links between two substructures. On the contrary, SPA will traverse all feasible links in the whole structure until finding a valid path. Hence this is a tradeoff that ARM makes to facilitate algorithmic complexity and save computation resources.

ARM is much simpler than SPA, for the latter’s computation complexity is as high as \(O(N^3)\), and it requires frequent exchanges of link status, which may cause heavy loads for data center. In conclusion, the failure experiments prove the great fault tolerant capacity as well as high efficiency of our Athena Routing Mechanism.

D. Evaluating CPU Usage

In Fig. 7, we also append the CPU usage of committing routing algorithm in source server to evaluate the computing efficiency. The experiment environment is under a Lenovo T350 G7 server with quad-core processors and 8GB memory. We simulate a Totoro\(_2\) (\(n = 16, k = 2, N = 4096\)), and run ARM on the experimental server to work out 10 completed paths with any node in the same or neighbor 3 Totoro\(_1\) as the destination. Because of data locality, a server usually communicates with servers which are located in the same row or adjacent several rows. We set the initial nodes amount as 10 and increase by 10 per second until reaching the threshold value 500, and the total computing time is 3 minutes.

As Fig. 7 indicates, the CPU usage continuously increases until achieving the peak value of 28% at around the 20\(_{th}\) second. Afterwards, it dramatically drops to 0% and remains to the end. Thus we can get a conclusion that our ARM has great performance with rather low CPU usage. Besides, the dash line represents the number of nodes which the experimental server

\begin{table}[h]
\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
Failure Ratio & 0.04 & 0.08 & 0.12 & 0.16 & 0.20 \\
\hline
Server Failure &  \\
TFR & 8.44 & 8.48 & 8.52 & 8.57 & 8.63 \\
ARM & 7.34 & 7.41 & 7.48 & 7.56 & 7.64 \\
SPA & 7.34 & 7.41 & 7.48 & 7.56 & 7.64 \\
\hline
Link Failure &  \\
TFR & 8.98 & 9.68 & 10.64 & 11.97 & 13.69 \\
ARM & 7.47 & 7.68 & 7.90 & 8.16 & 8.47 \\
SPA & 7.47 & 7.69 & 7.94 & 8.22 & 8.54 \\
\hline
Switch Failure &  \\
TFR & 8.43 & 8.46 & 8.50 & 8.53 & 8.58 \\
ARA & 7.39 & 7.53 & 7.67 & 7.82 & 7.97 \\
SPA & 7.40 & 7.55 & 7.70 & 7.86 & 8.03 \\
\hline
Rack Failure &  \\
ARM & 7.33 & 7.40 & 7.48 & 7.57 & 7.66 \\
SPA & 7.33 & 7.40 & 7.48 & 7.55 & 7.65 \\
\hline
\end{tabular}
\end{center}
\caption{Average Path Lengths in T\(_{16,2}\).}
\end{table}

C. Evaluating Path Length

The efficiency of routing algorithm can be directly evaluated by path length. A short average path length contributes to a short latency. Table I lists the values of average path length calculated by ARM, TFR and SPA respectively for a Totoro\(_2\) (\(n = 16, k = 2, N = 4096\)). We take SPA as the benchmark of routing performance because SPA is globally optimal. Comparing the results of ARM and SPA, it is easy to draw a conclusion that the differences are negligible. In some cases, such as server failure and rack failure, the average path lengths of both are equivalent. Whereas in link and switch failure, our ARM achieves shorter path lengths than SPA does, this is because the path failure ratio of ARM is a bit higher than that of SPA, thus our total path length is shorter. Besides,
calculates per second. From the graph, we can see the server figures out 10 paths of 500 nodes per second in the cost of 0% CPU usage, this is benefited from path cache constructed in the source server. Suppose a server maintains a cache of 10 completed paths to all of the rest nodes in the network. The maximal length of a path consists of 5 hops, and the vector of a path take up 6bits × 5hops = 30bits memory, that is 4B approximately. Then we can get the largest size of cache to preserve all path information on each host is 4B × 10 × 4096 ≈ 164KB at most. This also further proves our ARM is resource saving.

VI. CONCLUSION

In this paper, we applied Dynamic Programming to routing mechanisms of novelly proposed server-centric data center structures. Since communication latency is significant for Internet of Things, the conventional routing algorithms can not assure to get the optimal solutions. This motivates us to propose a universally applicable routing mechanism named ARM comprising of a routing algorithm called ARA and a path probing scheme. On basis of DP, ARA is able to find out the shortest paths with lower time complexity than SPA. ARM adopts source routing and path probing scheme, which means the source server performs computing a set of completed paths and dispatchs probing packets to detect the connectivity and capacities of paths. ARM is implemented by a 2.5-layer protocol and a 32-bits ARM address. In comparison with other conventional routing policies, our ARM simplifies the functionalities of intermediate servers as well as eliminates the extra bandwidth consumption caused by broadcasting link states among servers. Besides, our failure experiments on Totoro structure prove the satisfactory fault tolerant capacity of ARM comparing with TFR and SPA under different types of failures. We also demonstrate the relatively low CPU usage ratio of source server during the path computing process. Therefore, our ARM is a reliable and efficient mechanism which can be generalized to most server-centric structures to promote the overall performance of data center network. In the future work, we will focus on the implementation of ARM in DCell, FiConn and other structures to further verify the performance of our ARM.

APPENDIX A

PROOF OF THEOREM 2

According to the process of ARA, all level-k links out from $DCell_{k-1}$s where the source node locate will be fully utilized. Except one direct path (e.g., Path1 in Fig. 8) connecting two $DCell_{k-1}$s where src and dst locate respectively, most of level-k paths have to traverse intermediate $DCell_{k-1}$s before they reach the destination $DCell_{k-1}$. For each path, the traversing order is $DCell_{k-1}[src] → DCell_{k-1}[mid] → DCell_{k-1}[dst]$. Thus the path calculation of substructures is divided into 3 parts. DCell uses direct links to form a complete graph in each level and thus there are $N_{k-1} DCell_{k-1}$s in a $DCell_k$ (Note that, $N_{k-1}$ indicates the number of a $DCell_{k-1}$). Then the time complexity can be expressed as

$$T_k = 3 \times N_{k-1} \times T_{k-1}. \quad (5)$$

Since there is $N_k < (n + 1)^{2^k}$ (see Theorem 1 in [11]), we can further get

$$T_k < 3^k \times (n + 1)^{2^k - 1}. \quad (6)$$

By using the “Newton binomial theorem”, there is

$$3^k = (1 + 2)^k = 2^k + C_k^1 \cdot 2^{k-1} + C_k^2 \cdot 2^{k-2} + \ldots + C_k^{k-1} \cdot 2^1 + 1$$

$$< 2^k + 1 + 2^{k-1} \times (2^k - 2)$$

$$= 2^{2k-1} + 1$$

$$< (2^k)^2 + 1.$$ 

Based on the properties of DCell, there is $log N = O(2^k)$. Therefore, the complexity of $T_k$ is

$$T_k = O(3^k \times (n + 1)^{2^k - 1})$$

$$= O(((2^k)^2 + 1) \times (n + 1)^{2^k})$$

$$= O((log^2 N + 1) \times N)$$

$$= O(N log^2 N). \quad (8)$$

APPENDIX B

PROOF OF THEOREM 3

The linking philosophy of FiConn is similar to that of DCell. It forms a complete graph in each level by using direct links. The level-k path calculation is also divided into 3 parts. Hence, the time complexity can be expressed as

$$T_k = 3 \times T_{k-1} \times g_k. \quad (9)$$

Fig. 7: Resource Usage.

Fig. 8: Dividing level-k problem into level-k − 1 problems in complete-graph-based structure.
where $g_k$ indicates the number of $FiConn_{k-1}$ as well as links out going from the source $FiConn_{k-1}$. Since there is $g_k = N_k/N_{k-1}$, we can further get

$$T_k = 3 \times T_{k-1} \times N_k/N_{k-1}$$

$$= 3^2 \times T_{k-2} \times N_k/N_{k-1} \times N_{k-1}/N_{k-2} \quad (10)$$

Similarly, there are $\log N = O(2^k)$ and $3^k < (2^k)^2 + 1$. Since $n$, the number of ports on a switch, is a small integer, then finally we have

$$T_k = 3^k \times N/n$$

$$= O((2^k)^2 + 1) \times N/n)$$

$$= O((\log^2 N + 1) \times N/n)$$

$$= O(N\log^2 N). \quad (11)$$

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their useful suggestions and valuable feedback which help us to improve our work. This work is supported by the National Natural Science Foundation (NSF) of China under grant (No.61272073, No.61073064) and the key program of Natural Science Foundation of Guangdong Province (No. S2013020012865). The corresponding author of this paper is Yuhui Deng.

REFERENCES


Junjie Xie is a research student at the Computer Science Department of Jinan University. He received the B.E. degree in software engineering from Jinan University, Guangzhou, China, in 2013. His current research interests cover network interconnection, data center architecture and cloud computing.

Lijun Lyu is a research student at the Computer Science Department of Jinan University. Before joining Jinan University, Dr. Yuhui Deng worked at EMC Corporation as a senior research scientist from 2008 to 2009. He worked as a research officer at Cranfield University in the United Kingdom from 2005 to 2008. He received his Ph.D. degree in computer science from Huazhong University of Science and Technology in 2004. His current research interests cover green computing, cloud computing, information storage, computer architecture, performance evaluation, etc.

Yuhui Deng is a professor at the Computer Science Department of Jinan University. Before joining Jinan University, Dr. Yuhui Deng worked at EMC Corporation as a senior research scientist from 2008 to 2009. He worked as a research officer at Cranfield University in the United Kingdom from 2005 to 2008. He received his Ph.D. degree in computer science from Huazhong University of Science and Technology in 2004. His current research interests cover green computing, cloud computing, information storage, computer architecture, performance evaluation, etc.

Laurence T. Yang received the B.E. degree in computer science and technology from Tsinghua University, China, and the Ph.D. degree in computer science from University of Victoria, Canada. He is a professor with the School of Computer Science and Technology, Huazhong University of Science and Technology, China, as well as with the Department of Computer Science, St. Francis Xavier University, Canada. His research interests include parallel and distributed computing, embedded and ubiquitous/pervasive computing, big data. He has published more than 200 papers in various refereed journals (about 40%) in IEEE/ACM Transactions/Journals and the others mostly in Elsevier, Springer, and Wiley Journals). His research has been supported by the National Sciences and Engineering Research Council of Canada (NSERC) and the Canada Foundation for Innovation.