

DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers

Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi[†], Yongguang Zhang, Songwu Lu^{‡*}
Microsoft Research Asia, [†]Tsinghua University, [‡]UCLA
{chguo, hwu, kuntan, ygz}@microsoft.com,
[†]shijim@mails.thu.edu.cn, [‡]slu@cs.ucla.edu

ABSTRACT

A fundamental challenge in data center networking is how to efficiently interconnect an exponentially increasing number of servers. This paper presents *DCell*, a novel network structure that has many desirable features for data center networking. DCell is a recursively defined structure, in which a high-level DCell is constructed from many low-level DCells and DCells at the same level are fully connected with one another. DCell scales doubly exponentially as the node degree increases. DCell is fault tolerant since it does not have single point of failure and its distributed fault-tolerant routing protocol performs near shortest-path routing even in the presence of severe link or node failures. DCell also provides higher network capacity than the traditional tree-based structure for various types of services. Furthermore, DCell can be incrementally expanded and a partial DCell provides the same appealing features. Results from theoretical analysis, simulations, and experiments show that DCell is a viable interconnection structure for data centers.

Categories and Subject Descriptors

C.2.1 [Network Architecture and Design]: Network topology, Packet-switching networks

General Terms

Algorithms, Design

Keywords

Data center, Network topology, Throughput, Fault tolerance

1. INTRODUCTION

In recent years, many large data centers are being built to provide increasingly popular online application services, such as search, e-mails, IMs, web 2.0, and gaming, etc. In addition, these data centers also host infrastructure services

*This work was performed when Lei Shi was an intern and Songwu Lu was a visiting professor at Microsoft Research Asia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCOMM'08, August 17–22, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-60558-175-0/08/08 ...\$5.00.

such as distributed file systems (e.g., GFS [8]), structured storage (e.g., BigTable [7]), and distributed execution engine (e.g., MapReduce [5] and Dryad [11]). In this work, we focus on the networking infrastructure inside a data center, which connects a large number of servers via high-speed links and switches. We call it *data center networking* (DCN).

There are three design goals for DCN. First, the network infrastructure must be *scalable* to a large number of servers and allow for incremental expansion. Second, DCN must be *fault tolerant* against various types of server failures, link outages, or server-rack failures. Third, DCN must be able to provide *high network capacity* to better support bandwidth-hungry services.

Two observations motivate these goals. First, data center is growing large and the number of servers is increasing at an exponential rate. For example, Google has already had more than 450,000 servers in its thirty data centers by 2006 [2, 9], and Microsoft and Yahoo! have hundreds of thousands of servers in their data centers [4, 19]. Microsoft is even doubling the number of servers every 14 months, exceeding Moore's Law [22]. Second, many infrastructure services request for higher bandwidth due to operations such as file replications in GFS and all-to-all communications in MapReduce. Therefore, network bandwidth is often a scarce resource [5]. The current DCN practice is to connect all the servers using a tree hierarchy of switches, core-switches or core-routers. With this solution it is increasingly difficult to meet the above three design goals. It is thus desirable to have a new network structure that can fundamentally address these issues in both its physical network infrastructure and its protocol design.

To meet these goals we propose a novel network structure called *DCell*. DCell uses a recursively-defined structure to interconnect servers. Each server connects to different levels of DCells via multiple links. We build high-level DCells recursively from many low-level ones, in a way that the low-level DCells form a fully-connected graph. Due to its structure, DCell uses only mini-switches to scale out instead of using high-end switches to scale up, and it scales doubly exponentially with the server node degree. In practice, a DCell with a small degree (say, 4) can support as many as several millions of servers without using expensive core-switches or core-routers.

DCell is fault tolerant. There is no single point of failure in DCell, and DCell addresses various failures at link, server, and server-rack levels. Fault tolerance comes from both its rich physical connectivity and the distributed fault-tolerant routing protocol operating over the physical structure.

DCell supports high network capacity. Network traffic in DCell is distributed quite evenly among servers and across links at a server. High-level links in the hierarchy will not pose as the bottleneck, which is the case for a tree-based structure. Our experimental results on a 20-server DCell testbed further show that DCell provides 2 times throughput compared with the conventional tree-based structure for MapReduce traffic patterns.

In summary, we have proposed a new type of physical network infrastructure that possesses three desirable features for DCN. This is the main contribution of this work, and has been confirmed by simulations and experiments. A potential downside of our solution is that DCell trades-off the expensive core switches/routers with higher wiring cost, since it uses more and longer communication links compared with the tree-based structures. However, we believe this cost is well justified by its scaling and fault tolerance features.

The rest of the paper is organized as follows. Section 2 elaborates on design issues in DCN. Section 3 describes the DCell structure. Sections 4 and 5 present DCell routing and the solution to incremental expansion, respectively. Sections 6 and 7 use both simulations and implementations to evaluate DCell. Section 8 compares DCell with the related work. Section 9 concludes the paper.

2. DATA CENTER NETWORKING

Data centers today use commodity-class computers and switches instead of specially designed high-end servers and interconnects for better price-performance ratio [3]. The current DCN practice is to use the switch-based tree structure to interconnect the increasing number of servers. At the lowest level of the tree, servers are placed in a rack (typically 20-80 servers) and are connected to a rack switch. At the next higher level, server racks are connected using core switches, each of which connects up to a few hundred server racks. A two-level tree can thus support a few thousand servers. To sustain the exponential growth of server population, more high levels are added, which again use even more expensive, higher-speed switches.

The tree-based structure does not scale well for two reasons. First, the servers are typically in a single layer-2 broadcast domain. Second, core switches, as well as the rack switches, pose as the bandwidth bottlenecks. The tree structure is also vulnerable to “single-point-of-failure”: a core switch failure may tear down thousands of servers. A quick fix using redundant switches may alleviate the problem, but does not solve the problem because of inherently low connectivity.

To address DCN issues, it seems that we may simply reuse certain structures proposed in the area of parallel computing. These structures connect components such as memory and CPU of a super computer, and include Mesh, Torus, Hypercube, Fat Tree, Butterfly, and de Bruijn graph [12]. However, they addressed a different set of issues such as low-latency message passing in the parallel computing context and cannot meet the goals in DCN. We will provide detailed comparisons of these structures and our work in Section 8. We now elaborate on the three design goals we have briefly mentioned in the previous section.

Scaling: Scaling requirement in DCN has three aspects. First, the physical structure has to be scalable. It must physically interconnect hundreds of thousands or even millions of servers at small cost, such as a small number of links

at each node and no dependence on high-end switches to scale up. Second, it has to enable incremental expansion by adding more servers into the already operational structure. When new servers are added, the existing running servers should not be affected. Third, the protocol design such as routing also has to scale.

Fault tolerance: Failures are quite common in current data centers [3, 8]. There are various server, link, switch, rack failures due to hardware, software, and power outage problems. As the network size grows, individual server and switch failures may become the norm rather than exception. Fault tolerance in DCN requests for both redundancy in physical connectivity and robust mechanisms in protocol design.

High network capacity: Many online infrastructure services need large amount of network bandwidth to deliver satisfactory runtime performance. Using the distributed file system [8] as an example, a file is typically replicated several times to improve reliability. When a server disk fails, re-replication is performed. File replication and re-replication are two representative, bandwidth-demanding one-to-many and many-to-one operations. Another application example requiring high bandwidth is MapReduce [5]. In its Reduce operation phase, a Reduce worker needs to fetch intermediate files from many servers. The traffic generated by the Reduce workers forms an all-to-all communication pattern, thus requesting for high network capacity from DCN.

3. THE DCELL NETWORK STRUCTURE

The DCell-based DCN solution has four components that work in concert to address the three challenges. They are the DCell scalable network structure, efficient and distributed routing algorithm that exploits the DCell structure, fault-tolerant routing that addresses various types of failures such as link/server/rack failures, and an incremental upgrade scheme that allows for gradual expansion of the DCN size. Sections 3-5 describe these components in details.

3.1 DCell Physical Structure

DCell uses servers equipped with multiple network ports and mini-switches to construct its recursively defined architecture. In DCell, a server is connected to several other servers and a mini-switch via communication links, which are assumed to be bidirectional. A high-level DCell is constructed from low-level DCells. We use $DCell_k$ ($k \geq 0$) to denote a level- k DCell. The following example (in Figure 1) illustrates how DCells of different levels are constructed.

$DCell_0$ is the building block to construct larger DCells. It has n servers and a mini-switch ($n = 4$ for $DCell_0$ in Figure 1). All servers in $DCell_0$ are connected to the mini-switch. In our design, n is a small integer (say, $n \leq 8$). Therefore, a commodity 8-port switch with 1Gb/s or 10Gb/s per port could serve the purpose.

A level-1 $DCell_1$ is constructed using $n + 1$ $DCell_0$ s. In $DCell_1$, each $DCell_0$ is connected to all the other $DCell_0$ s with one link. In the example of Figure 1, $DCell_1$ has $n+1 = 5$ $DCell_0$ s. DCell connects the 5 $DCell_0$ s as follows. Assign each server a 2-tuple $[a_1, a_0]$, where a_1 and a_0 are the level-1 and level-0 IDs, respectively. Thus a_1 and a_0 take values from $[0,5)$ and $[0,4)$, respectively. Then two servers with 2-tuples $[i, j - 1]$ and $[j, i]$ are connected with a link for every i and every $j > i$. The linking result for $DCell_1$ is shown

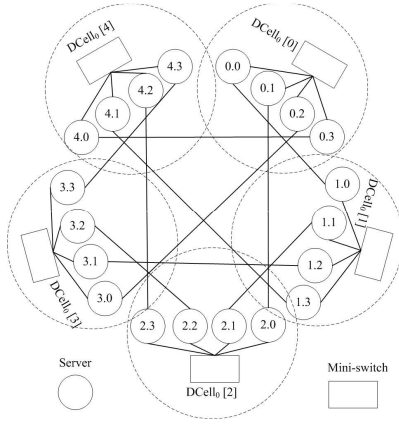


Figure 1: A $DCell_1$ network when $n=4$. It is composed of 5 $DCell_0$ networks. When we consider each $DCell_0$ as a virtual node, these virtual nodes then form a complete graph.

in Figure 1. Each server has two links in $DCell_1$. One connects to its mini-switch, hence to other nodes within its own $DCell_0$. The other connects to a server in another $DCell_0$.

In $DCell_1$, each $DCell_0$, if treated as a virtual node, is fully connected with every other virtual node to form a complete graph. Moreover, since each $DCell_0$ has n inter- $DCell_0$ links, a $DCell_1$ can only have $n + 1$ $DCell_0$ s, as illustrated in Figure 1.

For a level-2 or higher $DCell_k$, it is constructed in the same way to the above $DCell_1$ construction. If we have built $DCell_{k-1}$ and each $DCell_{k-1}$ has t_{k-1} servers, then we can create a maximum $t_{k-1} + 1$ of $DCell_{k-1}$ s. Again we treat each $DCell_{k-1}$ as a virtual node and fully connect these virtual nodes to form a complete graph. Consequently, the number of $DCell_{k-1}$ s in a $DCell_k$, denoted by g_k , is $t_{k-1} + 1$. The number of servers in a $DCell_k$, denoted by t_k , is $t_{k-1}(t_{k-1} + 1)$. The number of $DCell_{k-1}$ s in a $DCell_k$, (i.e., g_k), and the total number of servers in a $DCell_k$ (i.e., t_k) are

$$\begin{aligned} g_k &= t_{k-1} + 1 \\ t_k &= g_k \times t_{k-1} \end{aligned}$$

for $k > 0$. $DCell_0$ is a special case when $g_0 = 1$ and $t_0 = n$, with n being the number of servers in a $DCell_0$.

3.2 BuildDCells: the Procedure

To facilitate the DCell construction, each server in a $DCell_k$ is assigned a $(k + 1)$ -tuple $[a_k, a_{k-1}, \dots, a_1, a_0]$, where $a_i < g_i$ ($0 < i \leq k$) indicates which $DCell_{i-1}$ this server is located at and $a_0 < n$ indicates the index of the server in that $DCell_0$. We further denote $[a_k, a_{k-1}, \dots, a_{i+1}]$ ($i > 0$) as the *prefix* to indicate the $DCell_i$ this node belongs to. Each server can be equivalently identified by a unique ID uid_k , taking a value from $[0, t_k)$. The mapping between a unique ID and its $(k + 1)$ -tuple is a bijection. The ID uid_k can be calculated from the $(k + 1)$ -tuple using $uid_k = a_0 + \sum_{j=1}^k \{a_j \times t_{j-1}\}$, and the $(k + 1)$ -tuple can also be derived from its unique ID.

A server in $DCell_k$ is denoted as $[a_k, uid_{k-1}]$, where a_k is the $DCell_{k-1}$ this server belongs to and uid_{k-1} is the unique ID of the server inside this $DCell_{k-1}$.

The recursive DCell construction procedure *BuildDCells*

```

/* pref is the network prefix of DCell_l
l stands for the level of DCell_l
n is the number of nodes in a DCell_0*/
BuildDCells(pref, n, l)
Part.I:
  if (l == 0) /*build DCell_0*/
    for (int i = 0; i < n; i++)
      connect node [pref, i] to its switch;
    return;
Part.II:
  for (int i = 0, i < g_l; i++) /*build the DCell_{l-1}s*/
    BuildDCells([pref, i], n, l - 1);
Part.III:
  for (int i = 0, i < t_{l-1}; i++) /*connect the DCell_{l-1}s*/
    for (int j = i + 1, j < g_l; j++)
      uid_1 = j - 1; uid_2 = i;
      n_1 = [pref, i, uid_1]; n_2 = [pref, j, uid_2];
      connect n_1 and n_2;
  return;

```

Figure 2: The procedure to build a $DCell_l$ network.

is shown in Figure 2. It has three parts. Part I checks whether it constructs a $DCell_0$. If so, it connects all the n nodes to a corresponding switch and ends the recursion. Part II recursively constructs g_l number of $DCell_{l-1}$ s. Part III interconnects these $DCell_{l-1}$ s, where any two $DCell_{l-1}$ s are connected with one link. Recall that in Figure 1, nodes $[i, j - 1]$ and $[j, i]$ use a link to fully interconnect $DCell_0$ s inside a $DCell_1$. Similar procedure is used to interconnect the $DCell_{l-1}$ s in a $DCell_l$ as illustrated in Part III of BuildDCells.

Each server in a $DCell_k$ network has $k + 1$ links (i.e., the node degree of a server is $k + 1$). The first link, called a level-0 link, connects to a switch that interconnects $DCell_0$. The second link, a level-1 link, connects to a node in the same $DCell_1$ but in a different $DCell_0$. Similarly, the level- i link connects to a different $DCell_{i-1}$ within the same $DCell_i$.

3.3 Properties of DCell

The following theorem describes and bounds t_k , the number of servers in a $DCell_k$. Its proof is in Appendix A.

THEOREM 1.

$$t_k = g_k g_{k-1} \cdots g_0 t_0 = t_0 \prod_{i=0}^k g_i,$$

and

$$\left(n + \frac{1}{2}\right)^{2^k} - \frac{1}{2} < t_k < (n + 1)^{2^k} - 1$$

for $k > 0$, where n is the number of servers in a $DCell_0$.

Theorem 1 shows that, the number of servers in a DCell scales doubly exponentially as the node degree increases. A small node degree can lead to a large network size. For example, when $k = 3$ and $n = 6$, a DCell can have as many as 3.26-million servers!

Bisection width denotes the minimal number of links to be removed to partition a network into two parts of equal size. A large bisection width implies high network capacity and a more resilient structure against failures. DCell has the following lower bound to its bisection width.

THEOREM 2. *The bisection width of a $DCell_k$ is larger than $\frac{t_k}{4 \log_n t_k}$ for $k > 0$.*

The theorem can be proven by showing the bisection of DCell is larger than $\frac{1}{2^{t_k \log_n t_k}}$ times of the bisection of its embedding directed complete graph. The proof simply follows the techniques in Section 1.9 of [12].

4. ROUTING IN A DCELL

Routing in a DCell-based DCN cannot use a global link-state routing scheme since DCell’s goal is to interconnect up to millions of servers. The hierarchical OSPF [15] is also not suitable since it needs a backbone area to interconnect all the other areas. This creates both bandwidth bottleneck and single point failure.

In this section, we propose our DCell Fault-tolerant Routing protocol (DFR), a near-optimal, decentralized routing solution that effectively exploits the DCell structure and can effectively handle various failures (due to hardware, software, and power problems), which are common in data centers [3, 8]. We start with a routing scheme without failures and a broadcast scheme, on which DFR is built.

4.1 Routing without Failure

4.1.1 DCellRouting

DCell uses a simple and efficient single-path routing algorithm for unicast by exploiting the recursive structure of DCell. The routing algorithm, called *DCellRouting*, is shown in Figure 3. The design of *DCellRouting* follows a divide-and-conquer approach. Consider two nodes *src* and *dst* that are in the same $DCell_k$ but in two different $DCell_{k-1}$ s. When computing the path from *src* to *dst* in a $DCell_k$, we first calculate the intermediate link (n_1, n_2) that interconnects the two $DCell_{k-1}$ s. Routing is then divided into how to find the two sub-paths from *src* to n_1 and from n_2 to *dst*. The final path of *DCellRouting* is the combination of the two sub-paths and (n_1, n_2) .

```

/* src and dst are denoted using the  $(k + 1)$ -tuples
   src = [ $s_k, s_{k-1}, \dots, s_{k-m+1}, s_{k-m}, \dots, s_0$ ]
   dst = [ $d_k, d_{k-1}, \dots, d_{k-m+1}, d_{k-m}, \dots, d_0$ ]/
DCellRouting(src, dst)
  pref = GetCommPrefix(src, dst);
  m = len(pref);
  if (m == k) /*in the same  $DCell_0$ */
    return (src, dst);
  (n1, n2) = GetLink(pref,  $s_{k-m}$ ,  $d_{k-m}$ );
  path1 = DCellRouting(src, n1);
  path2 = DCellRouting(n2, dst);
  return path1 + (n1, n2) + path2;

```

Figure 3: Pseudocode for routing in a $DCell_k$ when there is no failure.

In Figure 3, *GetCommPrefix* returns the common prefix of *src* and *dst* and *GetLink* calculates the link that interconnects the two sub-DCells. The link can be directly derived from the indices of the two sub-DCells. Let s_{k-m} and d_{k-m} ($s_{k-m} < d_{k-m}$) be the indices of the two sub-DCells. Based on *BuildDCells* (shown in Fig 2), the link that interconnects these two sub-DCells is $([s_{k-m}, d_{k-m} - 1], [d_{k-m}, s_{k-m}])$.

<i>n</i>	<i>k</i>	t_k	Shortest-path		DCellRouting	
			mean	stdev	mean	stdev
4	2	420	4.87	1.27	5.16	1.42
5	2	930	5.22	1.23	5.50	1.33
6	2	1,806	5.48	1.18	5.73	1.25
4	3	176,820	9.96	1.64	11.29	2.05
5	3	865,830	10.74	1.59	11.98	1.91
6	3	3,263,442	11.31	1.55	12.46	1.79

Table 1: The mean value and standard deviation of path length in shortest-path routing and DCellRouting.

The following Theorem 3 gives the upper bound on the maximum path length of *DCellRouting*. It can be readily proved by induction.

THEOREM 3. *The maximum path length in DCellRouting is at most $2^{k+1} - 1$.*

Theorem 3 shows that the recursion occurs for at most $2^{k+1} - 1$ times. Therefore, *DCellRouting* can be performed quickly since a small *k* (say, $k \leq 3$) is sufficient to build large networks in practice. If we need to know only the next hop instead of the whole routing path, the time complexity can be reduced to $O(k)$ instead of $O(2^k)$, since we do not need to calculate the sub-paths that do not contain the next hop in this case.

The following theorem gives an upper bound on the diameter of the DCell network, i.e., the maximum path length among all the shortest-paths. It can be shown from Theorem 3.

THEOREM 4. *The diameter of a $DCell_k$ network is at most $2^{k+1} - 1$.*

The DCell diameter is small in practice. For $k = 3$, the diameter is at most 15. Given the total number of servers in $DCell_k$ as t_k . The diameter of a $DCell_k$ is less than $2 \log_n t_k - 1$, since we have $t_k > n^{2^k}$ from Theorem 1. One might conjecture that $2^{k+1} - 1$ is the exact diameter of DCell. This, however, is not true. We have two counter-intuitive facts on *DCellRouting* and DCell diameter:

- *DCellRouting* is not a shortest-path routing scheme. It can be shown by the following example. For a $DCell_2$ with $n = 2$ and $k = 2$, the shortest path between nodes $[0, 2, 1]$ and $[1, 2, 1]$ is $([0, 2, 1], [6, 0, 0], [6, 0, 1], [1, 2, 1])$ with length 3. The path using *DCellRouting* is $([0, 2, 1], [0, 2, 0], [1, 0, 0], [0, 0, 0], [1, 0, 0], [1, 0, 1], [1, 2, 0], [1, 2, 1])$ with length 7.
- $2^{k+1} - 1$ is not a tight bound for the diameter of DCell. For a DCell with $n = 2$ and $k = 4$, the diameter should be 27 rather than $2^{4+1} - 1 = 31$. However, the diameter of a DCell is $2^{k+1} - 1$ when $k < 4$.

Nonetheless, the performance of *DCellRouting* is quite close to that of shortest-path routing. Table 1 computes the average path lengths and the standard deviations under *DCellRouting* and shortest-path routing for DCells with different *n* and *k*. We observe that the differences are small. Since *DCellRouting* is much simpler than shortest-path routing, we use *DCellRouting* to build DFR.

4.1.2 Traffic Distribution in DCellRouting

We now show that the traffic is distributed quite evenly in *DCellRouting* under the all-to-all communication model.

High bandwidth can be achieved under both many-to-one and one-to-many communication models. DCell can thus provide high bandwidth for services such as MapReduce and GFS.

All-to-all communication model: In this model, all the nodes in a DCell communicate with all the other nodes. We have the following theorem, which is proved in Appendix B.

THEOREM 5. *Consider an all-to-all communication model for $DCell_k$ where any two servers have one flow between them. The number of flows carried on a level- i link is less than $t_k 2^{k-i}$ when using DCellRouting.*

Theorem 5 shows that, links at different levels carry a different number of flows. The bottleneck link, if any, is at the lowest-level links rather than at the highest-level links. Since k is small (say, $k = 3$), the difference among the number of flows at different levels is bounded by a small constant (e.g., $2^3 = 8$). Theorem 5 also shows that DCell can well support MapReduce [5], whose Reduce phase generates an all-to-all traffic pattern when each Reduce worker fetches intermediate files from many other servers.

One-to-Many and Many-to-One communication models: In both cases, a DCell server can utilize its multiple links to achieve high throughput. This property is useful for services such as GFS [8], which can utilize multiple links at a server to accelerate file replication and recovery.

We now introduce how a node selects its destinations to fully utilize its multiple links. Given a node src , the other nodes in a $DCell_k$ can be classified into groups based on which link node src uses to reach them. The nodes reached by the level- i link of src belong to $Group_i$. The number of nodes in $Group_i$, denoted as σ_i , is given by Theorem 6, which is proved in Appendix C.

THEOREM 6. $\sigma_i = (\prod_{j=i+1}^k g_j) t_{i-1}$ for $k > i > 0$ and $\sigma_k = t_{k-1}$.

Theorem 6 implies that, the number of nodes in $Group_0$ is $\sigma_0 = t_k - \sum_{i=1}^k \sigma_k$. The proof of Theorem 6 also shows how to compute the set of nodes for each $Group_i$ ($0 \leq i \leq k$). When src communicates with m other nodes, it can pick a node from each of $Group_0$, $Group_1$, etc. This way, the maximum aggregate bandwidth at src is $\min(m, k+1)$ when assuming the bandwidth of each link as one.

When multi-path routing is used, we can show that the maximum bandwidth between any two nodes in a $DCell_k$ is $k+1$. It also shows that DCell provides high network capacity though we do not use multi-path routing in this work.

4.2 Broadcast

A straightforward approach to broadcast is not fault tolerant: From src to all the other nodes, the approach constructs a spanning tree and then propagates broadcast messages along the tree. The scheme minimizes the number of forwarding messages, but is vulnerable to failures. When one intermediate node fails, the sub-tree under that node will not receive the broadcast message.

To address the above issue, we introduce *DCellBroadcast*, a simple yet robust broadcast scheme. In *DCellBroadcast*, a sender delivers the broadcast packet to all its $k+1$ neighbors when broadcasting a packet in a $DCell_k$. Upon receiving a broadcast packet, a receiver first checks whether this

packet has been received before. The receiver drops a duplicate packet but broadcasts a new packet to its other k links. *DCellBroadcast* is fault-tolerant in that a broadcast packet can reach all the receivers as long as the network is connected.

In *DCellBroadcast*, we limit the broadcast scope by encoding a scope value k into each broadcast message. The message is broadcasted only within the $DCell_k$ network that contains the source node. Since the diameter of $DCell_k$ is at most $2^{k+1} - 1$, a broadcast message needs at most $2^{k+1} - 1$ steps to reach all the nodes in $DCell_k$.

4.3 Fault-tolerant Routing

DFR is a distributed, fault-tolerant routing protocol for DCell networks without global link state. It uses *DCellRouting* and *DCellBroadcast* as building blocks. DFR handles three types of failures: server failure, rack failure, and link failure. Rack failure occurs when all the machines in a rack fail (e.g., due to switch or power outage). Link failure is a basic one since all the failures result in link failure. Hence, link failure handling is a basic part of DFR. DFR uses three techniques of *local reroute*, *local link-state*, and *jump-up* to address link failure, server failure, and rack failure, respectively. We now present the three techniques and then describe DFR.

4.3.1 Local-reroute and Proxy

As shown in Section 4.1, *DCellRouting* is simple and its performance is close to the shortest-path routing. This motivated us to introduce *local-reroute* to bypass failed links in *DCellRouting*. Effectively, *local-reroute* makes only *local* decisions to reroute packets.

Local-reroute is best illustrated by the following example. Let nodes src and dst be in the same $DCell_k$. We compute a path from src to dst using *DCellRouting*. Now assume an intermediate link (n_1, n_2) has failed. *Local-reroute* is performed at n_1 as follows. It first calculates the level of (n_1, n_2) , denoted by l . Then n_1 and n_2 are known to be in the same $DCell_l$ but in two different $DCell_{l-1}$ s. Since there are g_l $DCell_{l-1}$ subnetworks inside this $DCell_l$, it can always choose a $DCell_{l-1}$ (e.g., the one nearest to n_1 but different from the one n_2 is in). There must exist a link, denoted as (p_1, p_2) , that connects this $DCell_{l-1}$ and the one where n_1 resides. *Local-reroute* then chooses p_2 as its *proxy* and re-routes packets from n_1 to the selected proxy p_2 . Upon receiving the packet, p_2 simply uses *DCellRouting* to route the packet to dst . In *DCellRouting*, the new and the original paths converge when they both reach $DCell_l$ or above.

Local-route is efficient in handling link failures. This is because most links in a path are low-level links using *DCellRouting*. When local re-route is used to bypass a failed level- l link, the path length increases on average by the average path length in a $DCell_{l-1}$. This is because local re-route needs to route the packet into the proxy $DCell_{l-1}$ to bypass the failed link.

Local-reroute is not loop free. But loops can happen only when there are multiple link failures and the re-routes form a ring, hence is of very low probability. We discuss how to remove looped packets from DCell in Section 4.3.3. *Local-reroute* alone cannot completely address node failures. This is because it is purely based on DCell topology and does not utilize any kind of link or node states. We illustrate

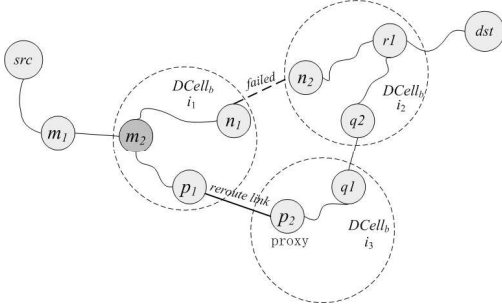


Figure 4: DFR: Fault tolerant routing in DCell.

the problem via an example. Consider from *src* to *dst* there is sub *DCellRouting* path $\{(q_1, q_2), (q_2, q_3)\}$. The level of (q_1, q_2) is 1 and the level of (q_2, q_3) is 3. Now q_1 finds that (q_1, q_2) is down (while actually q_2 failed). Then, no matter how we re-route inside this *DCell*₂, we will be routed back to the failed node q_2 ! In the extreme case, when the last hop to *dst* is broken, the node before *dst* is trapped in a dilemma: if *dst* fails, it should not perform local-reroute; if it is a link failure, it should perform local-reroute. To solve the problem faced by pure local-reroute, we next introduce local link-state.

4.3.2 Local Link-state

With local link-state, we use link-state routing (with Dijkstra algorithm) for intra-*DCell*_{*b*} routing and *DCellRouting* and local reroute for inter-*DCell*_{*b*} routing. In a *DCell*_{*b*}, each node uses *DCellBroadcast* to broadcast the status of all its $(k + 1)$ links periodically or when it detects link failure. A node thus knows the status of all the outgoing/incoming links in its *DCell*_{*b*}. *b* is a small number specifying the size of *DCell*_{*b*}. For example, a *DCell*_{*b*} has 42 or 1806 servers when *b* is 1 or 2 and $n = 6$.

Figure 4 illustrates how local link-state routing works together with local re-route. Use node m_2 as an example. Upon receiving a packet, m_2 uses *DCellRouting* to calculate the route to the destination node *dst*. It then obtains the first link that reaches out its own *DCell*_{*b*} (i.e., (n_1, n_2) in the figure). m_2 then uses intra-*DCell*_{*b*} routing, a local link-state based Dijkstra routing scheme, to decide how to reach n_1 . Upon detecting that (n_1, n_2) is broken, m_2 invokes local-reroute to choose a proxy. It chooses a link (p_1, p_2) with the same level as (n_1, n_2) and sets p_2 as the proxy. After that, m_2 routes the packet to p_2 . When p_2 receives the packet, it routes the packet to *dst*. Note that we handle the failure of (n_1, n_2) successfully, regardless of a link failure or a node failure at n_2 .

4.3.3 Jump-up for Rack Failure

We now introduce *jump-up* to address rack failure. Assume the whole *DCell*_{*b*}, *i*₂, fails in Figure 4. Then the packet will be re-routed endlessly around *i*₂, since all the re-routed paths need to go through r_1 . The idea of *jump-up* can also be illustrated in Figure 4. Upon receiving the rerouted packet (implying (n_1, n_2) has failed), p_2 checks whether (q_1, q_2) has failed or not. If (q_1, q_2) also fails, it is a good indication that the whole *i*₂ failed. p_2 then chooses a proxy from *DCells* with higher level (i.e., it *jumps up*). Therefore, with *jump-up*, the failed *DCell* *i*₂ can be bypassed. Note that when *dst*

is in the failed *i*₂, a packet will not be able to reach *dst* no matter how we local-reroute or jump-up. To remove packets from the network, we introduce two mechanisms as our final defense. First, a retry count is added in the packet header. It decrements by one when a local-reroute is performed. A packet is dropped when its retry count reaches zero. Second, each packet has a time-to-live (TTL) field, which is decreased by one at each intermediate node. The packet is discarded when its TTL reaches zero. When a packet is dropped, a destination unreachable message is sent back to the sender so that no more packets to the destination will be injected.

4.3.4 DFR: DCell Fault-tolerant Routing

Our DFR protocol uses all three techniques for fault-tolerant routing. The detailed procedure of DFR is shown in Figure 5. Denote the receiver node as *self.uid*. Upon receiving a packet, a node first checks whether it is the destination. If so, it delivers the packet to upper layer and returns (line 1). Otherwise, it checks the proxy field of the packet. If the proxy value of the packet matches the node, implying that the packet has arrived at the proxy, we then clear the proxy field (line 2). Let *dcn_dst* denote our *DCellRouting* destination. When the proxy field of the packet is empty, *dcn_dst* is the destination of the packet; otherwise, it is the proxy of the packet (lines 3-4). *DCellRouting* is then used to compute a path from the current node to *dcn_dst* (line 5), and to find the first link with level $> b$ from the path (with *FirstLink*, line 6). If we cannot find such a link (indicating that *dcn_dst* and the receiver are in the same *DCell*_{*b*}), we set *dij_dst*, which is the destination to be used in Dijkstra routing within this *DCell*_{*b*}, to *dcn_dst*. Once found, such a link is denoted as (n_1, n_2) . We know that n_1 and n_2 are in two different *DCell*_{*l-1*}s but in the same *DCell*_{*l*}, where *l* is the level of (n_1, n_2) . We then check the status of (n_1, n_2) . If (n_1, n_2) failed, we perform local-rerouting; otherwise, we set *dij_dst* to n_2 , the last hop in our DijkstraRouting (line 10). Once *dij_dst* is chosen, we use *DijkstraRouting* to perform intra-*DCell* routing and obtain the next hop. If the next hop is found, we forward the packet to it and return (line 13). However, If we cannot find a route to *dij_dst* and the destination of the packet and the receiver are in the same *DCell*_{*b*}, we drop the packet and return (lines 14-15); otherwise, we local-reroute the packet.

When we need to reroute a packet, we use *SelectProxy* to select a link to replace the failed link (n_1, n_2) . In case we cannot find a route to n_1 inside *DCell*_{*b*}, we treat it as equivalent to (n_1, n_2) failure. The idea of *SelectProxy* is simple. Our preferred choice is to find a link that has the same level as (n_1, n_2) . When rack failure occurs, we increase the link level by 1 to ‘jump-up’. Once we determine the level for proxy selection, we use a greedy algorithm to choose the proxy. We choose the node that has an alive link with our preferred level and is the closest one to *self.uid*. In the example of Figure 4, link (p_1, p_2) is chosen, node p_1 is in the same *DCell*_{*b*} with the current receiver m_2 and p_2 is chosen as the proxy.

5. INCREMENTAL EXPANSION

It is unlikely that a full *DCell*-based DCN is constructed at one time. Servers need to be added into data centers incrementally. When new machines are added, it is desirable that existing applications should not be interrupted, thus

```

DFR(pkt) /*pkt is the received packet*/
1  if (pkt.dst == self.uid) { deliver(pkt); return; }
2  if (self.uid == pkt.proxy) pkt.proxy = NULL;
3  if (pkt.proxy != NULL) dcn_dst = pkt.proxy;
4  else dcn_dst = pkt.dst;
5  path = DCellRouting(self.uid, dcn_dst);
6  (n1, n2) = FirstLink(path, b);
7  if ((n1, n2) == NULL) dij_dst = dcn_dst;
8  else
9    if ((n1, n2) fails) goto local-reroute;
10   else dij_dst = n2;
11  next_hop = DijkstraRouting(pkt, dij_dst);
12  if (next_hop != NULL)
13    forward pkt to next_hop and return;
14  else if (self.uid and pkt.dst are in a same DCell_b)
15    drop pkt and return;
local-reroute:
16  pkt.retry --;
17  if (pkt.retry == 0) {drop(pkt); return;}
18  pkt.proxy = SelectProxy(uid, (n1, n2))
19  return DFR(pkt);

```

Figure 5: Pseudocode for DFR.

requiring that: (1) re-wiring should not be allowed, and (2) addresses of existing machines should not change. A direct consequence of these requirements is that, the number of servers in a $DCell_0$, denoted as n , should be fixed.

A straightforward way to gradually build DCell is the bottom-up approach. When a $DCell_0$ is full, we start to build a $DCell_1$. When a $DCell_{k-1}$ is full, we start to build a $DCell_k$. This way, neither re-addressing nor re-wiring is needed when new machines are added. The system grows incrementally. However, this approach may generate interim structure that is not fault-tolerant. For example, when the number of nodes in the system is $2 \times t_{i-1}$, it will form two full $DCell_{i-1}$ s connected by a single link. If this link fails, the network is partitioned into two parts.

In this work, we propose a *top-down* approach to incrementally build a DCell. When constructing a $DCell_k$, we start from building many incomplete $DCell_{k-1}$ s and make them fully connected. Hence, even interim structure is fault tolerant. In our approach, we require that the minimal quantum of machines added at one time be much larger than one. In this paper, we use $DCell_1$ as the basic adding unit. This does not pose any difficulty in reality since servers are added in racks in data centers. A $DCell_1$ has 20, 30, 42 servers when $n = 4, 5, 6$. It can be readily placed into a rack, which typically has 20-80 servers. We also fix k at the planning stage of a data center. A choice of $k = 3$ accommodates millions of servers.

The *AddDCell* procedure is shown in Figure 6. It adds a $DCell_1$ and runs recursively. When adding a new level-1 DCell d_1 , *AddDCell* starts from $DCell_k$ and recursively finds the right $DCell_2$ for the new $DCell_1$. The right sub-DCell for d_1 is found via calling sub-routine *GetIndex*, which simply checks the number of its $DCell_{l-1}$. If the number is less than $(t_1 + 1)$, it allocates a new empty $DCell_{l-1}$. If all the $DCell_{l-1}$ s are full, a new $DCell_{l-1}$ is created; Otherwise, it finds the first none-full $DCell_{l-1}$. Then d_1 is added into the sub-DCell. After d_1 is added, all the nodes in d_1 are connected to their existing neighbors (not shown in Figure 6).

```

AddDCell(pref, l, d1) /*d1 is the DCell_l to add*/
if (l == 2)
  i is the largest index of the existing DCell_1s;
  assign prefix [pref, i + 1] to d1;
  return;
id = GetIndex(pref, l);
AddDCell([pref, id], l - 1, d1);
return;

GetIndex(pref, l) { /*get the DCell_{l-1} to add d1*/
m = the number of DCell_{l-1} subnetworks;
if (m < (t1 + 1)) return m;
if (all these m DCell_{l-1} are full)
  return m;
m2 = the smallest index of the non-full DCell_{l-1}s;
return m2;
}

```

Figure 6: Add a $DCell_1$ subnetwork into a $DCell_l$ network.

The operations of *GetIndex* show that, when the first $t_1 + 1$ of $DCell_1$ s are added into a $DCell_k$, $t_1 + 1$ of $DCell_{k-1}$ subnetworks are created and each contains only one $DCell_1$. Note that there are only t_1 machines in a $DCell_1$. Therefore, $t_1 + 1$ is the maximum number of $DCell_{k-1}$ s to form a full-connected graph. Compared with the bottom-up scheme, this top-down approach generates much more fault-tolerant interim structures, as it seeks to build complete graph at each level of $DCell$. A $DCell_i$ forms a complete graph at level i if all its $DCell_{i-1}$ are fully connected. The connectivity of our incrementally-built DCell is formalized by Theorem 7. The proof can be derived from the *AddDCell* procedure.

THEOREM 7. *A $DCell_i (i > 2)$ built by *AddDCell* either is a complete graph at level i , or becomes a complete graph at level i after we remove its $DCell_{i-1}$ subnetwork that has the largest prefix.*

An incrementally expanding DCell is highly fault tolerant because of good connectivity. When building a $DCell_k (k > 1)$ using *AddDCell*, the number of $DCell_{k-1}$ networks is at least $\min(t_1 + 1, \theta)$, where θ is the number of added $DCell_1$ s. This result, together with Theorem 7, demonstrates the good connectivity achieved by *AddDCell*.

AddDCell also fits naturally with our DFR routing algorithm. When a server cannot forward a packet to a sub-DCell, it just treats the next link as a failed one (no matter whether it is due to failure or an incomplete DCell). Using $DCell_1$ as the adding unit also ensures that the intra-DCell link-state routing of DFR works well since DCells at levels 1 and 2 are always fully connected.

6. SIMULATIONS

In this section, we use simulations to evaluate the performance of DFR under server node, rack, and link failures. We compare DFR with the Shortest-Path Routing (SPF), which offers a performance bound. The results are obtained by averaging over 20 simulation runs.

6.1 DFR in a Full DCell

In our simulation, different types of failures are randomly generated. A randomly selected node routes packets to all the other nodes. We study both the path failure ratio and the average path length for the found paths. In all the simulations, we set the intra-DCell routing level $b = 1$ and each

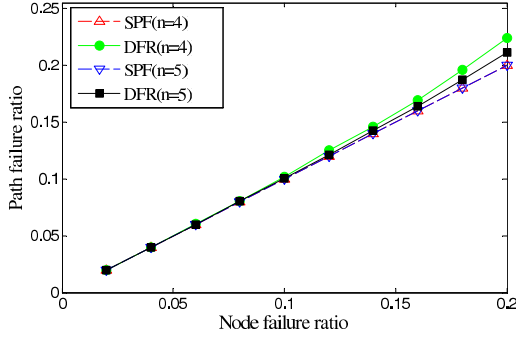


Figure 7: Path failure ratio vs. node failure ratio.

$DCell_1$ is a rack. We vary the (node/rack/link) failure ratios from 2% to 20%. The networks we use are a $DCell_3$ with $n = 4$ (176,820 nodes) and a $DCell_3$ with $n = 5$ (865,830 nodes).

Figure 7 plots the path failure ratio versus the node failure ratio under node failures. We observe that, DFR achieves results very close to SPF. Even when the node failure ratio is as high as 20%, DFR achieves 22.3% path failure ratio for $n = 4$ while the bound is 20%. When the node failure ratio is lower than 10%, DFR performs almost identical to SPF. Moreover, DFR performs even better as n gets larger.

Since DFR uses local reroute to bypass failed links, one concern is that local reroute might increase the path length. Table 2 shows that, the difference in path length between DFR and SPF increases very slowly as the node failure ratio increases. We also have studied the standard deviations of the path lengths under DFR and SPF. The standard deviation of DFR also increases very slowly as the node failure ratio increases. When the node failure rate is as high as 20%, the standard deviation is still less than 5.

We have also studied the effect of rack failure. We randomly select $DCell_1$ s and let all the nodes and links in those $DCell_1$ fail. Table 2 shows that, the impact of rack failure on the path length is smaller than that of node failure. This is because when a rack fails, SPF also needs to find alternative paths from higher-level DCells. Our jump-up strategy is very close to SPF routing in this case. The path failure ratio is not shown here since it is very similar to the node failure case in Figure 7.

Figure 8 plots the path failure ratio under link failures, which would occur when wiring is broken. We see that the path failure ratio of DFR increases with the link failure ratio. However, the path failure ratio of SPF is almost 0. This is because very few nodes are disconnected from the graph (indicating the robustness of our DCell structure). However, DFR cannot achieve such performance since it is not globally optimal. When the failure ratio is small (say, less than 5%), the performance of DFR is still very close to SPF. As shown in Table 2, the average path length under link failure is similar to that under node failure.

6.2 DFR in a Partial DCell

We have also evaluated DFR when a DCell is incrementally upgraded using our incremental expansion procedure of Section 5. To this end, we consider a large $DCell_3$ network with $n=6$ (that supports up to 3.26-million servers).

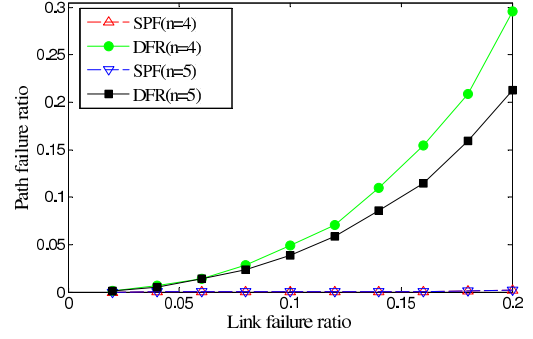


Figure 8: Path failure ratio vs. link failure ratio.

Failure ratio	Node failure		Rack failure		Link failure	
	DFR	SPF	DFR	SPF	DFR	SPF
0.02	11.60	10.00	11.37	10.00	11.72	10.14
0.04	12.00	10.16	11.55	10.01	12.40	10.26
0.08	12.78	10.32	11.74	10.09	13.73	10.55
0.12	13.60	10.50	11.96	10.14	14.97	10.91
0.2	16.05	11.01	12.50	10.32	17.90	11.55

Table 2: Average path lengths for DFR and SPF in a $DCell_3$ with $n = 4$.

We show that DFR achieves stable performance with various ratio of deployed nodes.

Figure 9 plots the performance of DFR with the ratio of deployed nodes varying from 10% to 100%. The failure rate is set to 5% for all three failure models. The results of SPF are stable at 5% under nodes and rack failures, but close to 0 under link failures. DFR has its path failure ratios smaller than 0.9% under link failures, and smaller than 6% under node and rack failures, respectively. The result demonstrates that DCell is fault tolerant even when it is partially deployed.

7. IMPLEMENTATION

In this section, we design and implement a DCN protocol suite for DCell. We also report experimental results from an operational DCell testbed with over twenty server nodes.

7.1 DCN Protocol Suite

The DCN protocol suite serves as a network layer for DCell-based data centers. It includes DCN addressing, DCN header format, and protocols for neighborhood and link-state management. It provides functionalities similar to IP over the Internet [18].

DCN Addressing: We use a 32-bit uid to identify a server. The most significant bit (bit-0) is used to identify the address type. If it is 0, the address is the uid of a server; otherwise, the address is a multicast address. For a multicast address, the 1~3 bits are used to identify the scope of the DCell, within which the message can propagate. The remaining 28 bits are used to indicate the multicast group. Currently, only one concrete multicast address is defined: when these 28 bits are all ones, it defines a broadcast address.

DCN Header: Figure 10 shows the format of the DCN header. The header size is 20 or 24 bytes depending on the existence of the *Proxy DCN Address* (see Section 4.3 for de-

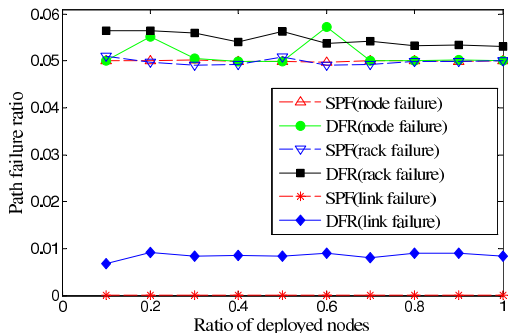


Figure 9: Path failure ratios when the node/rack/link failure probabilities are 5%.

0		4		8		16		31	
Version		DHL		Protocol		Payload Length			
Identification									
Retry		Flags		TTL		Header Checksum			
Source DCN Address									
Destination DCN Address									
Proxy DCN Address									

Figure 10: The DCN protocol header.

tails). The design of the DCN header borrows heavily from IP, herein we highlight only the fields specific to DCN. *Identification* is 32, rather than 16 bits, as in IP. This is because the link bandwidth in data centers is quite high. A 16-bit field will incur identification recycles within a very short period of time. *Retry* is a 4-bit field to record the maximum number of local-reroutes allowed. It decrements by one for each local-reroute performed. A packet is dropped once its retry count becomes 0. Our current implementation sets the initial retry count as 5. *Flags* is a 4-bit field, and only bit-3 of the flag, PF (Proxy Flag), is currently defined. When PF is set, the Proxy DCN Address is valid; otherwise, the data payload starts right after the Destination DCN Address.

Neighbor Maintenance: To detect various failures, we introduce two mechanisms for neighbor maintenance. First, a DCN node transmits heart-beat messages over all its outbound links periodically (1 second by default). A link is considered down if no heart-beat message is received before timeout (5 seconds by default). Second, we use link-layer medium sensing to detect neighbor states. When a cable becomes disconnected or re-connected, a notification message is generated by the link-layer drivers. Upon receiving notification, the cache entry at the corresponding neighbor is updated immediately.

Link-state Management: DFR uses link-state routing inside $DCell_b$ subnetworks. Therefore, each node needs to broadcast its link states to all other nodes inside its $DCell_b$. This is done by using DCellBroadcast (see Section 4.2). A node performs link-state updates whenever the status of its outbound neighboring links changes. It also broadcasts its link states periodically. Links that have not been refreshed before timeout are considered as down and thus removed from the link-state cache.

7.2 Layer-2.5 DCN Prototyping

Conceptually, the DCN protocol suite works at layer 3 and is a network-layer protocol. However, replacing IP with the DCN layer requires application changes, as almost all current network applications are based on TCP/IP. To address this issue, we implement the DCN suite at an intermediate layer between IP and the link layer, which we call *Layer-2.5 DCN*. In DCN, IP address is used for end-host identification without participating in routing, and current applications are supported without any modification. In our design, we choose to have a fixed one-to-one mapping between IP and DCN addresses. This design choice greatly simplifies the address resolution between IP and DCN.

We have implemented a software prototype of Layer-2.5 DCN on Windows Server 2003. Our implementation contains more than 13000 lines of C code. The DCN protocol suite is implemented as a kernel-mode driver, which offers a virtual Ethernet interface to the IP layer and manages several underlying physical Ethernet interfaces. In our current implementation, operations of routing and packet forwarding are handled by CPU. A fast forwarding module is developed to receive packets from all the physical network ports and decide whether to accept packets locally or forward them to other servers. The forwarding module maintains a *forwarding table*. Upon receiving a packet, we first check whether its next hop can be found in the forwarding table. When the next hop of a packet is not in the forwarding table, DFR routing will be used to calculate the next hop, which is subsequently cached in the forwarding table. When any link state changes, the forwarding table is invalidated and then recalculated by DFR.

7.3 Experimental Results

We have an operational testbed of a $DCell_1$ with over 20 server nodes. This $DCell_1$ is composed of 5 $DCell_0$ s, each of which has 4 servers (see Figure 1 for the topology). Each server is a DELL 755DT desktop with Intel 2.33GHz dual-core CPU, 2GB DRAM, and 160GB hard disk. Each server also installs an Intel PRO/1000 PT Quad Port Ethernet adapter. The Ethernet switches used to form the $DCell_0$ s are D-Link 8-port Gigabit switches DGS-1008D (with each costing about \$50). Each server uses only two ports of the quad-port adapter. Twisted-pair lines are used to interconnect the DCN testbed. Two experiments are carried out to study the fault-tolerance and network capacity of DCell:

Fault-tolerance: In this experiment, we set up a TCP connection between servers [0,0] and [4,3] in the topology of Figure 1. The path between the two nodes is [0,0], [0,3], [4,0], [4,3] initially. To study the performance under link failures, we manually unplugged the link ([0,3], [4,0]) at time 34s and then re-plugged it in at time 42s. We then shutdown the server [0,3] at time 104s to assess the impact of node failures. After both failures, the routing path is changed to [0,0], [1,0], [1,3], [4,1], [4,3]. And after re-plug event, the path returns to the original one. The TCP throughput is plotted in Figure 11. The CPU utilizations are about 40%, 45%, and 40% for sender, receiver, and forwarder, respectively.

We make two observations from the experiment. First, DCell is resilient to both failures. The TCP throughput is recovered to the best value after only a few seconds. Second, our implementation detects link failures much faster than node failures, because of using the medium sensing

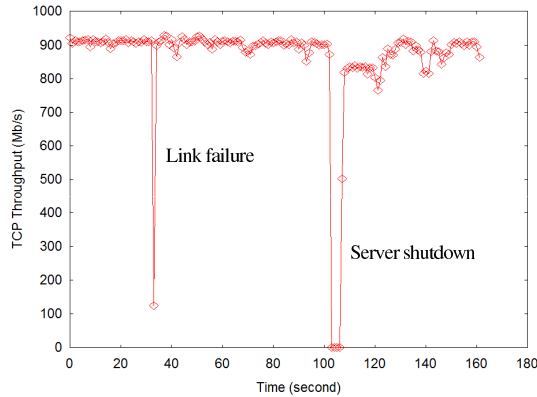


Figure 11: TCP throughput with node and link failures.

technique. Figure 11 shows that, the link failure incurs only 1-second throughput degradation, while the node failure incurs a 5-second throughput outage that corresponds to our link-state timeout value.

Network capacity: In this experiment, we compare the aggregate throughput of DCell and that of the tree structure. The target real-life application scenario is MapReduce [5]. In its Reduce-phase operations, each Reduce worker fetches data from all the other servers, and it results in an all-to-all traffic pattern. In our experiment, each server established a TCP connection to each of the remaining 19 servers, and each TCP connection sent 5GB data. The 380 TCP connections transmitted 1.9TB data in total. There was no disk access in this experiment. This is to separate network performance from that of disk IO. We study the aggregate throughput under DCell and a two-level tree structure. In the two-level tree, the switches of the 5 *DCell*₀s were connected to an 8-port Gigabit Ethernet switch.

Figure 12 plots both the aggregate throughput of DCell and that using the two-level tree. The transmission in DCell completed at 2270 seconds, but lasted for 4460 seconds in the tree structure. DCell was about 2 times faster than Tree. The maximum aggregate throughput in DCell was 9.6Gb/s, but it was only 3.6Gb/s in the tree structure.

DCell achieves higher throughput than the tree-based scheme. We observed that, the 20 one-hop TCP connections using the level-1 link had the highest throughput and completed first at the time of 350s. All the 380 TCP connections completed before 2270s. Since we currently use software for packet forwarding, CPU becomes the major bottleneck (with 100% CPU usage in this experiment), which prevents us from realizing all the potential capacity gains of DCell. In our future implementation when packet forwarding is off-loaded to hardware, we expect DCell to deliver much higher peak throughput of about 20Gb/s. Moreover, our current gain is achieved using a small-scale testbed. The merits of DCell will be more significant as the number of servers grows.

One might expect the tree structure to have much higher throughput than the measured 3.6Gb/s. Each pair of the servers sharing a single mini-switch should be able to send/receive at the line speed, and the aggregate throughput should be close to 20Gb/s in the beginning. However, this is not true. The top-level switch is the bottleneck and soon gets congested. Due to the link-layer flow control, this will eventu-

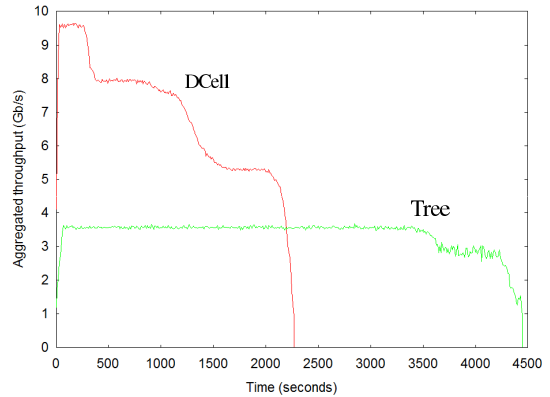


Figure 12: Aggregate TCP Throughput under DCell and Tree.

ally cause queue to build up at each sender’s buffer. All TCP connections at a server share the same sending buffer on a network port. Therefore, all TCP connections are slowed down, including those not traversing the root switch. This results in much smaller aggregate TCP throughput.

8. RELATED WORK

Interconnection networks have been extensively studied in parallel computing for two decades (see e.g., [21, 16, 6, 17, 13, 12]). In this section, we compare DCell with several representative interconnection structures in the DCN context. Our comparison does not imply, in any sense, that other structures are not suitable for their original design scenarios. Instead, we intend to show that DCell is a better structure for data centers, which require scaling, fault tolerance, high network capacity, and incremental expansion.

Table 3 shows the comparison results. We use N to denote the number of supported servers. The metrics used are: (1)Node degree: Small node degree means fewer links, and fewer links lead to smaller deployment overhead; (2)Network diameter: A small diameter typically results in efficient routing; (3)Bisection width (BiW): A large value shows good fault-tolerant property and better network capacity; (4)Bottleneck Degree (BoD): BoD is the maximum number of flows over a single link under an all-to-all communication model. A small BoD implies that the network traffic is spread out over all the links.

FullMesh has the smallest diameter, largest BiW, and smallest BoD, but its node degree is $N - 1$. Ring and 2D Torus use only local links. Ring has the smallest node degree, but a BiW of 2, large diameter and BoD. 2D Torus [17] uses only local links and has a constant degree of 4. But it has large diameter ($\sqrt{N} - 1$) and BoD (in proportion to $N\sqrt{N}$). These three structures are not practical even for small data centers with hundreds of servers.

In a Tree structure, servers are attached as leaves, and switches are used to build the hierarchy. When the switches are of the same degree d , the diameter of the Tree is $2 \log_{d-1} N$. But Tree has a bisection width of 1 and the bottleneck degree is in proportion to N^2 . Fat Tree [6] overcomes the bottleneck degree problem by introducing more bandwidth into the switches near the root. Specifically, the aggregated bandwidth between level- i and level- $(i + 1)$ is the same for all levels. Due to the problem of large node degree at the

Structure	Degree	Diameter	BiW	BoD
FullMesh	$N - 1$	1	$\frac{N^2}{4}$	1
Ring	2	$\frac{N}{2}$	2	$\frac{N^2}{8}$
2D Torus	4	$\sqrt{N} - 1$	$2\sqrt{N}$	$\frac{N\sqrt{N}}{8}$
Tree	-	$2 \log_{d-1} N$	1	$N^2 \binom{d-1}{d^2}$
FatTree	-	$2 \log_2 N$	$\frac{N}{2}$	N
Hypercube	$\log_2 N$	$\log_2 N$	$\frac{N}{2}$	$\frac{N}{2}$
Butterfly ⁺	4	$2l$	$\frac{N}{l+1}$	$O(Nl)$
de Bruijn	d	$\log_d N$	$\frac{2dN}{\log_d N}$	$O(N \log_d N)$
DCell	$k + 1$	$< 2 \log_n N - 1$	$\frac{N}{4 \log_n N}$	$< N \log_n N$

⁺For Butterfly, we have $N = (l + 1) \times 2^l$.

Table 3: Comparison of different network structures.

root node, alternatives were proposed to approximate a Fat Tree using multi-stage networks that are formed by small switches. In that case, the required number of switches scales as $O(N \log N)$, where N is the number of supported servers. Hence, the scalability of Fat Tree is not comparable with that of DCell.

Hypercube [12] is a widely used structure in high-performance computing. It has large bisection width $\frac{N}{2}$ and small bottleneck degree ($\frac{N}{2}$). The problem of hypercube is that its node degree is $\log_2 N$, which is not small when N becomes large. Hypercube hence does not scale well and is hardly applicable to large data centers.

Butterfly [12] has a constant node degree of 4. The number of supported nodes in a Butterfly network is $N = (l + 1)2^l$, where l is the dimensions of Butterfly. The diameter and bottleneck degree of Butterfly are reasonably good, considering that it has a node degree of four. Butterfly, however, does not scale as fast as our DCell. Furthermore, it is not fault-tolerant, since there is only one path between two nodes.

de Bruijn [14] achieves near-optimal tradeoff between node degree and network diameter. The number of supported nodes in a de Bruijn network is $N = d^\Delta$, where d is the node degree and Δ is the diameter. de Bruijn also has reasonably good bisection width and bottleneck degree. But de Bruijn is not suitable for data centers for two reasons: (1) de Bruijn is not incrementally deployable. If we increase Δ even by one, the whole network has to be re-wired; (2) the links in de Bruijn are asymmetric, thus doubling our wire deployment and maintenance effort.

The structures in Table 3 sample only a small representative portion of the interconnection networks. There are other structures such as k -ary, n -cubes [16], star-graph [1], cube-connected cycles [17]. There are also switch-based interconnects like Autonet, Myrinet, and Server-Net [6] that support networks with arbitrary topologies. When compared with the existing structures, to the best of our knowledge, DCell is the only one that scales doubly exponentially and targets the DCN scenario.

In recent years, many P2P structures have been developed for scalable lookup service, with Chord [23] and CAN [20] being two representatives. P2P and DCN target different networking scenarios. In P2P, nodes use logical links to interact with each other, a node degree of $O(\log N)$ is considered quite reasonable. But DCN requires a much smaller node degree since links must be physically deployed. More-

over, DCN needs to increase its physical network capacity and to allow for incremental expansion. All these factors lead to a different set of challenges and hence different design choices.

One deployment problem faced by DCell, as well as other low-diameter structures (including Hypercube, Butterfly, de Bruijn), is the wiring problem. Though the degree of DCell is small (e.g., ≤ 4), the high-level links in a DCell may travel a relatively long distance. For example, for a DCell with $n = 6$ and $k = 3$, a $DCell_2$ has 1806 machines. The wires to interconnect the $DCell_2$ s may be quite long. At the current stage, we can at least use optical fibers for long-haul interconnections. The latest Ethernet optical transceivers support connections from 550 meters to 10 kilometers [10]. Another option is to aggregate the spatially adjacent wires in DCell using multiplexing/demultiplexing technologies like SDH (Synchronous Digital Hierarchy); this reduces the number of physical wires.

9. CONCLUSION

In this paper, we have presented the design, analysis, and implementation of *DCell*. DCell is a novel network infrastructure to interconnect servers. Each server is connected to a different level of DCells via its multiple links, but all the servers act equally. High-level DCells are built recursively from many low-level DCells. DCell uses only mini-switches to scale out, and it scales doubly exponentially with the server node degree. Therefore, a DCell with a small server node degree (say, 4) can support up to several millions of servers without using core switches/routers.

On top of its interconnection structure, DCell also runs its fault-tolerant routing protocol DFR. DFR performs distributed, fault-tolerant routing without using global states and has performance close to the optimal shortest-path routing. Moreover, DCell offers much higher network capacity compared with the tree-based, current practice. Traffic carried by DCell is distributed quite evenly across all links; there is no severe bottleneck.

The best application scenario for DCell is large data centers. In recent years, online services supported by data centers have gained increasing popularity. They include both end-user applications (e.g., Web search and IM) and distributed system operations (e.g., MapReduce and GFS). Given such a trend, data center networking (DCN) is likely to become an important research area. The ultimate goal for DCN research is to support various, all-to-all, many-to-one, one-to-many traffic patterns, all at high speed, in a scalable and fault-tolerant manner. DCN research thus calls for renovations in both its physical topology and efficient protocol designs. The DCell-based solution represents our effort along this direction.

10. ACKNOWLEDGEMENT

We thank James Hamilton, Parantap Lahiri, Michael Manos, and Junfeng Zhou for information on data centers, Wei Chen, Frans Kaashoek, Xiangyang Li, Robert Morris, Mike Schroeder, Zhong Zhang, the members of the Wireless and Networking Group of Microsoft Research Asia for their early feedback, Frans Kaashoek and the anonymous reviewers for their valuable suggestions that help to improve the presentation of the paper.

11. REFERENCES

- [1] S. Akers and B. Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE trans. Computers*, 1989.
- [2] S. Arnold. Google Version 2.0: The Calculating Predator, 2007. Infonortics Ltd.
- [3] L. Barroso, J. Dean, and U. Hözl. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, March-April 2003.
- [4] A. Carter. Do It Green: Media Interview with Michael Manos, 2007. <http://edge.technet.com/Media/Doing-IT-Green/>.
- [5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*, 2004.
- [6] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection networks: an engineering approach*. Morgan Kaufmann, 2003.
- [7] F. Chang et. al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI'06*, 2006.
- [8] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *ACM SOSP'03*, 2003.
- [9] T. Hoff. Google Architecture, July 2007. <http://highscalability.com/google-architecture>.
- [10] Intel. High-Performance 1000BASE-SX and 1000BASE-LX Gigabit Fiber Connections for Servers. http://www.intel.com/network/connectivity/resources/doc_library/data_sheets/pro1000mf-mf-lx.pdf.
- [11] M. Isard, M. Budiu, and Y. Yu. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *ACM EuroSys*, 2007.
- [12] F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays. Trees. Hypercubes*. Morgan Kaufmann, 1992.
- [13] K. Liszka, J. Antonio, and H. Siegel. Is an Alligator Better Than an Armadillo? *IEEE Concurrency*, Oct-Dec 1997.
- [14] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience. In *ACM SIGCOMM*, 2003.
- [15] J. Moy. OSPF Version 2, April 1998. RFC 2328.
- [16] L. Ni and P. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, Feb 1993.
- [17] B. Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic, 2002.
- [18] Jon Postel. Internet Protocol. RFC 791.
- [19] L. Rabbe. Powering the Yahoo! network, 2006. <http://yodel.yahoo.com/2006/11/27/powering-the-yahoo-network/>.
- [20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM'01*, 2001.
- [21] H. Jay Seigel, W. Nation, C. Kruskal, and L. Napolitano. Using the Multistage Cube Network Topology in Parallel Supercomputers. *Proceedings of the IEEE*, Dec 1989.
- [22] J. Snyder. Microsoft: Datacenter Growth Defies Moore's Law, 2007. <http://www.pcworld.com/article/id,130921/article.html>.
- [23] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM'01*, 2001.

APPENDIX

A. Proof of Theorem 1

The first equation can be directly derived from the definitions of g_k and t_k . From both definitions, we have

$$t_k = g_k \times t_{k-1} = (t_{k-1} + 1)t_{k-1} = (t_{k-1})^2 + t_{k-1}$$

Therefore, we have

$$\begin{aligned} t_k &= (t_{k-1} + \frac{1}{2})^2 - \frac{1}{4} > (t_{k-1} + \frac{1}{2})^2 - \frac{1}{2} \\ t_k + \frac{1}{2} &> (t_{k-1} + \frac{1}{2})^2 \end{aligned}$$

Similarly, we have

$$\begin{aligned} t_k &= (t_{k-1})^2 + t_{k-1} = (t_{k-1} + 1)^2 - t_{k-1} - 1 \\ &< (t_{k-1} + 1)^2 - 1 \\ t_k + 1 &< (t_{k-1} + 1)^2 \end{aligned}$$

Then, we have $t_k > (n + \frac{1}{2})^{2^k} - \frac{1}{2}$ and $t_k < (n + 1)^{2^k} - 1$, and conclude the proof of Theorem 1. \square

B. Proof of Theorem 5

The total number of flows in a $DCell_k$ is $t_k(t_k - 1)$. The number of level- i ($0 \leq i \leq k$) links is t_k .

We first consider the level- k links. Note that in a $DCell_k$, a flow can travel at most one level- k link once. This is because all the $DCell_{k-1}$ s are one-one connected. When node $n1$ in one $DCell_{k-1}$ to reach node $n2$ in another $DCell_{k-1}$, it needs only to traverse the link that connects these two $DCell_{k-1}$ s. Since the links are symmetric, the number of flows carried on all the level- k links should be identical. Then, the number of flows carried in a level- k link is smaller than $\frac{t_k(t_k-1)}{t_k} = t_k - 1 < t_k$.

For level- $(k-1)$ links, note that when a flow traverses a level- k link, it traverses at most two level- $k-1$ links. The number of flows carried over a level- $(k-1)$ link is thus smaller than $2t_k$.

Similarly, the number of flows that a level- i link carries is $2^{k-i}t_k$. \square

C: Proof of Theorem 6

src uses its level- k link to connect to a $DCell_{k-1}$. src uses its level- k link to reach all the nodes in that $DCell_{k-1}$. Hence we have $\sigma_k = t_{k-1}$.

For the level- i ($k > i > 0$) link of src , src uses it to connect to a $DCell_{i-1}$. Note that all the nodes in this $DCell_{i-1}$ need to go through this level- i link of src to reach src . The number of nodes in this $DCell_{i-1}$ is t_{i-1} . Each node in this $DCell_{i-1}$ can use its level- $i+1$ links to connect to a $DCell_i$. All the nodes in this $DCell_i$ also need to go through the level- i link of src . The number of total nodes is then $t_{i-1} + t_{i-1}t_i = t_{i-1}g_{i+1}$. All these nodes then use their $i+2$ links to connect to a set of $DCell_{i+1}$ DCells, to expand the total number of nodes to $t_{i-1}g_{i+1}g_{i+2}$. Similar procedures can be sequentially carried out to the level $i+3$, $i+4$, \dots , k links. Consequently, we get $\sigma_i = t_{i-1} \prod_{j=i+1}^k g_j$ for $k > i > 0$. \square