DumbNet: A Smart Data Center Network Fabric with Dumb Switches

Yiran Li
Institute for Interdisciplinary Information Sciences
Tsinghua University
liyr14@mails.tsinghua.edu.cn

Da Wei
Institute for Interdisciplinary Information Sciences
Tsinghua University
xichuanglian@gmail.com

Xiaoqi Chen
Department of Computer Science
Princeton University
xiaoqic@cs.princeton.edu

Ziheng Song
School of Computer Science
Beijing University of Posts and Telecommunications
songziheng@bupt.edu.cn

Ruihan Wu
Institute for Interdisciplinary Information Sciences
Tsinghua University
wrh14@mails.tsinghua.edu.cn

Yuxing Li
Institute for Interdisciplinary Information Sciences
Tsinghua University
yx-li17@mails.tsinghua.edu.cn

Xin Jin
Department of Computer Science
Johns Hopkins University
xinjin@cs.jhu.edu

Wei Xu
Institute for Interdisciplinary Information Sciences
Tsinghua University
weixu@tsinghua.edu.cn

ABSTRACT

Today's data center networks have already pushed many functions to hosts. A fundamental question is how to divide functions between network and software. We present DumbNet, a new data center network architecture with no state in switches. DumbNet switches have no forwarding tables, no state, and thus require no configurations. Almost all control plane functions are pushed to hosts: they determine the entire path of a packet and then write the path as tags in the packet header. Switches only need to examine the tags to forward packets and monitor the port state. We design a set of host-based mechanisms to make the new architecture viable, from network bootstrapping and topology maintenance to network routing and failure handling. We build a prototype with 7 switches and 27 servers, as well as an FPGA-based switch. Extensive evaluations show that DumbNet achieves performance comparable to traditional networks, supports application-specific extensions like flowlet-based traffic engineering, and stays extremely simple and easy-to-manage.

CCS CONCEPTS

• **Networks → Data center networks**;

KEYWORDS

Data center networks

ACM Reference Format:
Yiran Li, Da Wei, Xiaoqi Chen, Ziheng Song, Ruihan Wu, Yuxing Li, Xin Jin, and Wei Xu. 2018. DumbNet: A Smart Data Center Network Fabric with Dumb Switches. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3190508.3190531

1 INTRODUCTION

The network infrastructure is an essential component of modern data centers. It directly impacts the availability and performance of data center applications. Compared to the Internet, data center networks (DCNs) have unique requirements: it needs to provide high bisection bandwidth and low latency to servers in a cost-effective manner. The switches of a DCN are typically placed in the same building with many redundant links and are managed by a single entity.

Despite the differences, many of today's data centers still use similar hardware and protocols as the Internet to build their networks. IP longest prefix matching (LPM) is still the dominant packet forwarding mechanism employed in the switches. For large DCNs, IP LPM has scalability problems in terms of forwarding table size in switches, as IP prefixes cannot always be easily aggregated. DCNs also adopt complex distributed control plane protocols, such as Border Gateway Protocol (BGP), to manage link and router failures and to perform traffic engineering, which unnecessarily complicates network management. All the complexities require separate operator teams to manage the network and system, increasing the management overhead. Given a data center network with hundreds to thousands of switches that may have different

---

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04...$15.00
https://doi.org/10.1145/3190508.3190531

hardware models, different switch OSes and different firmware versions, this management process is very complex and error-prone.

On the other hand, the networks in traditional supercomputers are much simpler and often treated as an integral part of the computation system [2]. They use tag-based switching and allow application-level control with source routing, i.e., letting senders determine each hop of a packet. Unfortunately, the supercomputing networks only offer fixed topologies and limited fault-tolerance, making them unsuitable for modern data centers.

Software-defined networking (SDN) is an important step forward to simplify network management. It provides a centralized controller to manage the network based on a global view. The controller eliminates many inefficiencies and complexities of distributed protocols. However, the switch data plane still maintains a lot of states for routing, network measurement, access control, etc. The state not only increases the cost and hurt the scalability of the switch hardware but is also hard to manage. Specifically, state consistency is the key for operations, but it is hard to achieve consistency cross the host-network boundary as network devices are based on entirely different software stacks. It is fundamentally difficult to ensure the consistency of the sheer amount of state distributed over hundreds to thousands of switches. Even worse, state updates are common in DCN, in order to get better performance or prevent attacks [19].

As a step towards using a coherent infrastructure to handle both network and host systems, we present DumbNet, a new data center network architecture with no data plane state while still support application-specific performance tuning. DumbNet is a layer-2 network. Switches in DumbNet are extremely simple: they have no forwarding tables, no state, and thus require no configuration. DumbNet uses source routing for packet forwarding: each packet contains a path in its packet header, which is a list of tags denoting the output port at each hop; each switch simply examines the packet header to find out the output port at the current hop and forwards the packet accordingly. Such a simple design eliminates all states and thus all configurations from the switch. All control functions are pushed to hosts. All the routing decisions can be made locally on a single host, and thus DumbNet avoids the distributed states management system.

The major technical challenge of DumbNet is the control plane design to support all necessary network functionalities with dumb and stateless switches. We design host-based mechanisms to accomplish these tasks. Specifically, the control plane supports (*i*) automatically discover the network topology without any active discovery logics in switches, (*ii*) update the network for failures and failover to alternative paths, and (*iii*) leveraging the host-only design, we allow software to easily extend DumbNet. For example, we implement a flowlet-based traffic engineering, a layer-3 router, and virtual networks as examples. Our mechanisms automatically exploit the network topology (e.g., mostly-regular topology like fat-tree) to reduce the overhead of topology discovery and network update.

We provide two physical DumbNet switch implementations: (*i*) on commodity Ethernet switches using Multiprotocol Label Switching (MPLS) and (*ii*) on a Field-Programmable Gate Array (FPGA) development board. We implement the client module with Intel DPDK [9]. We have built a prototype of DumbNet with 7 switches and 27 servers, and have conducted extensive evaluations on the prototype, from micro benchmarks to real-world big data applications. Results show that DumbNet offers comparable performance to traditional network architectures while providing a purely host-based network software that is easy to integrate with upper software layers, opening many new opportunities for future cross-layer optimizations.

In summary, we make the following contributions.

- We propose DumbNet, a new DCN architecture with no state in switches. DumbNet takes advantage of source routing and SDN to build an extremely simple yet high-performance network that is easy to integrate with the software layer.
- We design a set of host-based mechanisms for topology discovery and maintenance, routing, failure handling and flowlet-based traffic engineering.
- We rethink the division of labor between switches and hosts. DumbNet simplifies switches to a practical extreme and pushes all functionalities to hosts.
- We provide several implementations of DumbNet switches and a prototype of a DumbNet fabric, demonstrating its feasibility as a real DCN with real data center benchmarks.

2 RELATED WORK

Data center network architectures. Researchers have proposed new scalable network topologies to support a massive amount of servers, new addressing schemes to support free migration of virtual machines, and new routing mechanisms to guarantee no routing loops and to provide high bandwidth and low latency [1, 8, 11, 13, 14, 29]. Source routing is proposed to provide flexible routing and reduce switch state [8, 12, 20, 32]. Many of these solutions are built upon existing commodity switches or proposed new functionality extending capability of today's switches. Different from them, DumbNet aims to simplify the management aspect of data center networks with simple switches that do not contain any data plane state. Sourcey [18] proposes a data center network architecture with no switch state and outlines the design of the data and control planes. We build on Sourcey by developing a concrete system.

Network control plane. Traditional networks extensively rely on distributed network protocols for network management, e.g., OSPF [26] and IS-IS [4] for intra-domain routing and BGP [33] for inter-domain routing. Distributed network protocols bring many painful network problems, like routing oscillation due to distributed route computation and suboptimal decisions due to local routing information. SDN [24] aims to solve these problems by a centralized controller with a global view. However, current SDN design still assumes a stateful network: the controller manipulates switch flow tables to achieve management goals. This complicates the problem as we have to solve two distributed state management problems: states in switches, as well as states in hosts. DumbNet goes one step further by making the network data plane completely stateless, and the controller only needs to manage the state in hosts. Existing work like Fastpass [31] also attempts to reduce the workload of SDN controllers.

Various solutions have been proposed in the past on topology discovery, network monitoring, and fault localization [5, 7, 16]. Since DumbNet switches only provide simple label-based forwarding and maintain no state, we design new host-based mechanisms for DumbNet to accomplish these tasks.

Multi-path routing. In order to support data-intensive applications and user-facing web services, data center networks provide many parallel paths to provide high bandwidth and fault-tolerance. There are many solutions on how to best construct network topologies, leverage the available paths to increase network throughput, reduce end-to-end latency and improve network robustness [3, 22, 27, 31, 34]. Specifically, [22] showed a need for multi-path routing in an irregular topology, with possibly different lengths for each path, to achieve optimal performance. In DumbNet, we use host-based mechanisms to collect path information, reliably store the information in a central controller, and perform multi-path routing in hosts.

3 DUMBNET OVERVIEW

DumbNet aims to build a fast, flexible and resilient network fabric with extremely simple switches. Specifically, we remove all data plane states and most processing logic from switches and implement all control plane functions except for failure notification with host-only software. As a result, DumbNet switches require no configurations. Of course, the simplified data plane brings challenges for initialization, maintenance, and routing. In this section, we provide an overview of these challenges and solutions.

3.1 Design goals and choices

Stateless and configuration-free switches. In existing networks, switches make forwarding decisions for each packet. Thus, we need to keep and manage a lot of states in the network in a distributed manner. Although SDN uses a centralized controller to simplify state management, the data plane states such as forwarding tables remains in switches. Coordinating updates on the state distributed over switches is a challenging problem and often requires sophisticated algorithms [19] that do not usually run on network equipment.

Our key design goal is to remove all state and complex logic from switches. DumbNet switches only do three things: forwarding packets based on tags in packet headers without table lookup, monitoring and broadcasting the port state, and reply a fixed unique ID. Such a simplification has three advantages. (*i*) It simplifies switch hardware. DumbNet does not need TCAMs or complex parsers to support many protocols, and thus saves chip area for more ports or larger packet buffers. (*ii*) It simplifies switch software, which is often an arcane and proprietary embedded system and is hard to debug. (*iii*) It eliminates switch configuration, making it easy to extend the consistency-based configuration management tool [6] to cover the network. Yet, a stateless switch still provides essential functionality it should offer, such as physical link state monitoring for its own ports, and fast link failure notification. As we will point out later, we keep the logic on the switch extremely simple using a two-stage protocol.

We think the DumbNet design strikes the right balance to simplify the switches while keeping them practical.

Packet routing with the centralized controller and host agents. We push routing computation to hosts, and it has three benefits. (*i*) It provides flexibility to make routing decisions in software without complicated routing protocols. (*ii*) The routing computation runs as normal server software and can reuse existing distributed system code, such as Paxos, for consensus. (*iii*) The computation resources on hosts are more abundant than switches and thus easy to support advanced features such as per-flow state tracking.

As it is expensive for each host to keep an up-to-date network topology, we divide host software into two parts, i.e., a centralized controller and agents on each host. The controller is responsible for maintaining the topology information and performing routing computation. The agents get available paths from the controller, choose one path for each packet, and encode the path information into packet headers. To reduce the overhead of frequently contacting the controller, host agents cache the relevant part of topology and use the cache to serve most packets.

Striking the right division of functionality between software and hardware. We try to keep DumbNet hardware switch simple. However, putting everything into software results in an over-complicated host agent. The hardware can support functions like multi-queue, priority and ECN much more easily and efficiently than software. Adding those functions will not change the stateless and configuration-free nature of DumbNet switches. In this paper, we focus on the core networking features only, and leave the advanced hardware features as future work.

Incremental deployment in existing networks. DumbNet allows incremental deployment. It can be deployed on existing commodity switches with MPLS. It is transparent to applications, which use the same TCP/IP interface for network communication. Underneath, the host agents of DumbNet push paths to packet headers for packet delivery.

Easy to extend in software. While DumbNet provides only core functionality as a network fabric, it provides a software interface that reveals the network state to the applications, making it easy to implement different functionalities. The information sharing with applications is carefully controlled to ensure fairness and security. We add flowlet-based traffic engineering, layer-3 routers, and network virtualization as extensions with little effort (Section 6). We can easily support existing source-routing based optimizations such as pHost [10] on to DumbNet too.

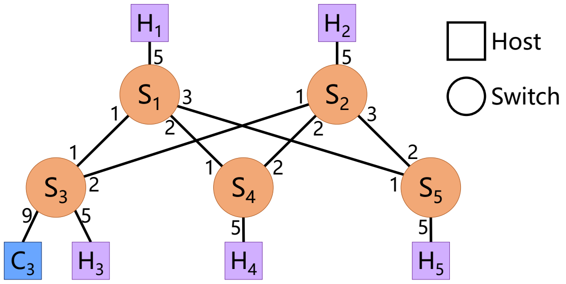


Figure 1: A sample DumbNet topology used in examples in Section 3.2 and 4.1.

3.2 End-to-end packet forwarding

We provide a concrete example on how DumbNet forwards packets through the dumb switches. Figure 1 illustrates a sample topology. Circles $\mathbf{S_1}$-$\mathbf{S_5}$ represent switches, and squares represent hosts. Host $\mathbf{C_3}$ is special as it serves as the controller. Hosts $\mathbf{H_1}$ to $\mathbf{H_5}$ are five hosts we use in this example. Edges in the figure represent network links, and the numbers at the ends of edges represent switch ports, link $\mathbf{S_2}$-$\mathbf{S_3}$ connects port 1 of $\mathbf{S_2}$ (denoted by $\mathbf{S_2}$-**1**) and port 2 of $\mathbf{S_3}$ (denoted by $\mathbf{S_3}$-**2**).

We consider a packet from $\mathbf{H_4}$ to $\mathbf{H_5}$ as an example. $\mathbf{H_4}$ gets available paths to $\mathbf{H_5}$ from the controller $\mathbf{C_3}$. One such path is $\mathbf{H_4}$-$\mathbf{S_4}$-$\mathbf{S_2}$-$\mathbf{S_5}$-$\mathbf{H_5}$. Since each hop only needs to know its outgoing port, we can encode the path as **2**-**3**-**5**-ø, where 2, 3, 5 are the outgoing ports on $\mathbf{S_4}$, $\mathbf{S_2}$ and $\mathbf{S_5}$, respectively. "ø" is a marker for the end of a path, and we set it to `0xFF`. With the path encoded in the packet header, each switch only examines the first tag, removes the tag, and forwards the packet to the port specified by the tag. For example, $\mathbf{S_4}$ examines the first tag and forwards the packet to port 2. Then the path in the packet header becomes **3**-**5**-ø when it reaches $\mathbf{S_2}$. When $\mathbf{H_5}$ receives the packet, it removes the ø tag and passes the packet up in the OS network stack, just as a normal Ethernet packet. The switches process packets based on the header tag only, requiring no table lookup.

3.3 Overview of challenges and solutions

Dumb switches bring challenges to the control plane design, especially at the scale of large data center networks. Here we summarize the key challenges and solutions.

Challenge 1: Automatic topology discovery. Because switches have no tables or address resolution logics, we rely on host-based mechanisms to discover the topology.

We use hosts to send probing packets to discover the topology. A probing packet can end up with three scenarios: (*i*) the packet is lost, which indicates no host on the path; (*ii*) the packet bounces back, which discovers the links in the path; (*iii*) the packet is received and replied by another host, which discovers a new host and possibly the controller if the new host knows. By probing all possible links with breadth-first search, we can discover the entire network topology efficiently.

Although theoretically any host can probe the entire network, for performance reasons, we only allow a single controller to do so, while other hosts just probe until they learn the location of the controller. We provide more details on topology discovery in Section 4.1.

Challenge 2: Fault tolerance. Traditional switches monitor link state by exchanging packets using protocols like Link Layer Discovery Protocol (LLDP). In DumbNet, we combine switch-based link state monitoring with a two-stage link failure notification protocol to provide simple yet efficient topology management. The switch provides hardware-based, zero-overhead and continuous link monitoring. The switch suppresses repeated port state notifications to deal with flapping links. Section 4.2 provides detailed discussion.

We use replication to tolerate controller failures. The controller replicas use Apache Zookeeper [17] to keep a consistency view of the network topology and serve host requests in the same way.

Challenge 3: Resource consumption and routing flexibility tradeoff in host agents. Ideally, if a host knows the entire topology, it can choose the best path for each packet. However, due to the resource consumption to maintain up-to-date topology information at every host, DumbNet only stores the global view in the controller. Each host caches the paths it needs for communication. While path caching reduces resource consumption, it also decreases routing flexibility and reliability. It is too slow if every link failure triggers hosts to contact the controller. We design an efficient path caching algorithm, called *path graph*, to allow hosts to configure how much information they cache vs. how much routing flexibility they get (detail in Section 4.3).

Challenge 4: Compatibility with existing networks. To make DumbNet practical, it needs to be compatible with existing Ethernet and OS networking stack. Theoretically, DumbNet is a layer-2 network protocol and any existing IP network can run on top of it. In practice, we implement a packet rewriting module in the host agent to allow unmodified applications to send DumbNet packets. We also design an Ethernet-friendly header format to encode routing tags. With these considerations, we can run DumbNet traffic on any MPLS-enabled Ethernet fabric together with normal Ethernet traffic, and on our custom FPGA-based stateless switches. We provide the details on backward compatibility in Section 5.

4 CONTROL PLANE

The host-based control plane in DumbNet handles the network topology discovery and maintenance, including failure handling.

An important goal of the control plane design is to improve the scalability of the network, and obviously, the scalability of the controller is the bottleneck. We solve the problem with two types of optimizations: (*i*) we use multiple controllers wherever possible, to perform tasks such as topology discovery and handling topologies queries from clients; and (*ii*) we aggressively cache paths a host *may* need, especially those links helpful for failure recovery, on the hosts to reduce the number of queries to the controller.

4.1 Topology discovery

While we can ask on administrators to manually enter topology configuration, it is costly and error-prone, especially for large and irregular topologies. A fully automated topology discovery process makes bootstrapping the network easier, and can tolerate mis-configurations in the underlying physical network. We design a topology discovery protocol using only the dumb switches and hosts. Essentially, we use a breadth-first search (BFS) originating from one host to reach all other hosts. However, with some prior knowledge about the topology, during bootstrapping the hosts can quickly verify (instead of discover) all links, and thus make the bootstrapping process faster while still maintain the tolerance to mis-configurations.

The goal of topology discovery is to use a single host to discover all switches, links, and hosts in the network. Since the switches do not have topology discovery logic, we rely on hosts to discover the network. Each host runs a *topology discovery* service that sends out and listens to probing messages. A probing message (PM) is a regular DumbNet packet. Its *payload* contains (*i*) a marker identifying it is a probing message, (*ii*) the source of the message, and (*iii*) the entire path to the destination (same as the tag sequence in the header). If a host receives a PM, it would reply to the sender using the reverse path contained in the payload, together with its identity, such as its MAC or IP addresses.

The topology discovery algorithm uses breadth-first search (BFS). A host first discovers the switch it attaches to (depth=0), and then gradually discovers switches that are one, two and more hops away (depth=$1, 2, \ldots$), together with their attached hosts. To keep track of the search state, the host maintains a queue Q for the newly discovered switches. Once the host finishes scanning all neighbors of one switch, it dequeues that switch and begins to process the next switch until Q is empty. To illustrate this process, we describe how the host $\mathbf{C_3}$ discovers the topology in Figure 1. We use $\mathbf{C_3}$ only as an example. In fact, a host does not have to be a controller to perform the discovery.

Discover switches, links, and hosts. When $\mathbf{C_3}$ starts, it first finds the port number on the switch it attaches to. To do so, it tries to send the following PMs and see which one bounces back to itself: $\mathbf{\underline{1}\text{-}}\varnothing$, $\mathbf{\underline{2}\text{-}}\varnothing$, $\mathbf{\underline{3}\text{-}}\varnothing$ and so on. Note that in this section, we underline the hop(s) in the path that the host is actively probing. As the PM $\mathbf{\underline{9}\text{-}}\varnothing$ bounces back, $\mathbf{C_3}$ learns that it connects to port 9 of a switch. Optionally, we can pass the maximum number of ports to discovery process as an argument to prevent sending too much probing packets.

$\mathbf{C_3}$ then queries the switch ID by adding tag **0** in the sequence to specify ID query packet. When switch receives packet with tag $\mathbf{0\text{-}9\text{-}}\varnothing$, it replies with its unique ID in packet tagged $\mathbf{9\text{-}}\varnothing$. $\mathbf{C_3}$ then receives and remembers the ID of this switch, say, $\mathbf{S_3}$. It is obvious that we can combine port number probing and switch ID query by directly sending $\mathbf{\underline{0}\text{-}\underline{1}\text{-}}\varnothing$, $\mathbf{\underline{0}\text{-}\underline{2}\text{-}}\varnothing$, $\mathbf{\underline{0}\text{-}\underline{3}\text{-}}\varnothing$ and so on.

Following the breadth-first search algorithm, the next step is to discover the neighboring switches of $\mathbf{S_3}$. Observe that if there is a link between two switches (e.g., $\mathbf{S_3\text{-}1}$ and $\mathbf{S_1\text{-}1}$), $\mathbf{C_3}$'s PM (e.g., $\mathbf{\underline{1}\text{-}\underline{1}\text{-}9\text{-}}\varnothing$) will bounce back. Thus, $\mathbf{C_3}$ enumerates all possible combinations of port pairs, e.g., $\mathbf{\underline{1}\text{-}\underline{0}\text{-}\underline{1}\text{-}9\text{-}}\varnothing$, $\mathbf{\underline{1}\text{-}\underline{0}\text{-}\underline{2}\text{-}9\text{-}}\varnothing$, etc. Remember the tag **0** here let the probed switch return its ID. Once $\mathbf{C_3}$ receives $\mathbf{\underline{1}\text{-}\underline{0}\text{-}\underline{1}\text{-}9\text{-}}\varnothing$ back, it discovers $\mathbf{S_1}$ and learns that $\mathbf{S_3\text{-}1}$ connects to $\mathbf{S_1\text{-}1}$. To continue finding switches further away from $\mathbf{S_1}$, $\mathbf{C_3}$ starts probing $\mathbf{1\text{-}\underline{2}\text{-}\underline{0}\text{-}\underline{1}\text{-}1\text{-}9\text{-}}\varnothing$, $\mathbf{1\text{-}\underline{2}\text{-}\underline{0}\text{-}\underline{2}\text{-}1\text{-}9\text{-}}\varnothing$, $\mathbf{1\text{-}\underline{2}\text{-}\underline{0}\text{-}\underline{3}\text{-}1\text{-}9\text{-}}\varnothing$, and so on.

In addition to the bouncing PMs, $\mathbf{C_3}$ may also to receive responses from other hosts and learn the identities and locations of the hosts at each depth. For example, $\mathbf{C_3}$ will receive a response from $\mathbf{H_3}$ for PM $\mathbf{\underline{5}\text{-}9\text{-}}\varnothing$ and a response from $\mathbf{H_1}$ for $\mathbf{1\text{-}\underline{5}\text{-}1\text{-}9\text{-}}\varnothing$.

Notice that to discover the neighbors and hosts for a switch, a host needs to send $O(P^2)$ PMs, where P is the number of ports per switch. Thus the overall discovery algorithm complexity, in terms of PMs sent, is $O(N * P^2)$, where N is the number of switches in the fabric. The PMs are sent out in parallel to improve performance.

Ambiguity in switch identity and our solution. There is a special case causing trouble for the algorithm above. For example, $\mathbf{S_1}$ and $\mathbf{S_2}$ have exactly the same return path to $\mathbf{C_3}$ (both are $\mathbf{1\text{-}9\text{-}}\varnothing$). Thus, $\mathbf{1\text{-}\underline{2}\text{-}\underline{0}\text{-}\underline{1}\text{-}1\text{-}9\text{-}}\varnothing$ (via $\mathbf{S_1}$) and $\mathbf{1\text{-}\underline{2}\text{-}\underline{0}\text{-}\underline{2}\text{-}1\text{-}9\text{-}}\varnothing$ (via $\mathbf{S_2}$) both bounce back to $\mathbf{C_3}$ during discovery. In this case, $\mathbf{C_3}$ never knows whether there is a direct link between $\mathbf{S_1\text{-}2}$ and $\mathbf{S_4\text{-}2}$ or not.

To resolve the ambiguity between return path $\mathbf{S_4\text{-}S_1\text{-}S_3}$ and $\mathbf{S_4\text{-}S_2\text{-}S_3}$, $\mathbf{C_3}$ will further verify the path using $\mathbf{1\text{-}2\text{-}\underline{1}\text{-}\underline{0}\text{-}1\text{-}9\text{-}}\varnothing$ and $\mathbf{1\text{-}2\text{-}\underline{2}\text{-}\underline{0}\text{-}1\text{-}9\text{-}}\varnothing$. Since $\mathbf{C_3}$ already know $\mathbf{1\text{-}1\text{-}9\text{-}}\varnothing$ passes $\mathbf{S_3}$, $\mathbf{S_1}$ and $\mathbf{S_3}$ in order. The verify packet should return the ID of $\mathbf{S_1}$. The response of $\mathbf{1\text{-}2\text{-}\underline{1}\text{-}\underline{0}\text{-}1\text{-}9\text{-}}\varnothing$ shows that the return path is indeed $\mathbf{S_4\text{-}S_1\text{-}S_3}$, thus confirmed there is a direct link between $\mathbf{S_1\text{-}2}$ and $\mathbf{S_4\text{-}2}$.

Multiple controllers. We have multiple controllers in the network for fault tolerance and query performance. During initialization, all controllers start to probe the network without knowing each other. Theoretically, we can run the discovery algorithm at each controller and merge their results, but it does not worth the complexity given that the topology discovery is fast (see Section 7.2.1). In our current implementation, we only allow a single controller to complete the discovery, and other controllers become replicas for topology information maintenance. We keep the replicas consistent using Apache ZooKeeper [17] to store the topology changes.

4.2 Failure handling

As the switches are dumb, we have to handle failures with the help of host-based mechanisms. Although hosts are able to probe and discover the potential failures, a fast failure detection unavoidably involves a lot of unnecessary probing. Our goal is to allow the client to quickly failover to another path upon a link failure. Two mechanisms help us achieve the goal: (*i*) switch-hardware-based local port state monitoring, and (*ii*) two-stage failure handling with a combination of switch hardware and host-based broadcast to disseminate the message to all hosts.

Port state monitoring. Like in traditional Ethernet, DumbNet switches detect port up/down using the physical signal. The detection is easy even for the dumb switches. The tricky part is suppressing duplicate alarms (e.g. from a flapping link). The switches suppress alarms for 1 second, i.e. a switch will send out one alarm per second per port. We rely on host-based mechanisms to suppress alarms if the link flapping lasts longer (details below).

Link Failure Handling Stage 1: Failure notification.

(On switch.) On a port state change, the switch broadcasts *port-up/down* notification messages with a fixed number of hops. The limited hop number prevents loops without using spanning trees. As modern data center topologies often have small diameters, a max of 5 hops is often enough.

(On hosts.) Once a host receives a *port-down* notification, it knows failure position by checking the switch ID and port number included in the packet. It checks its local state for duplicate alarm suppression. If not duplicate, it starts to notify other hosts. A naive way is to let the host notify the controller first, let the controller update the topology and then notify others. However, it is slow and unreliable as the controller is a single point of failure. Thus, we let the host to directly *flood* the notification to the entire network. The message starts from the hosts on the same switch, then goes to hosts on the neighboring switches, like the way many peer-to-peer networks do flooding. As soon as a host receives the notification message, it can immediately route around the affected link, because it is likely to have alternative path cached (Section 4.3). Stage 1 happens without any controller involvement.

Note that we use a local broadcast on switches and host-based flooding on servers. This is because the broadcast requires little logic with no state on switches, while on host we can run more efficient flood algorithms.

Link Failure Handling Stage 2: Topology patch. As long as the network is still fully connected, the controller will eventually learn about the failure during the flooding. At this time, the controller updates the global topology and floods a *topology patch* message to all the hosts to guarantee connectivity. The message may help some host find a better path.

With this two-stage optimization, we remove the controller from the critical path of the failover and instead use it as an asynchronous book-keeper and performance optimizer. Our evaluation shows that this mechanism significantly reduces failover delay (Section 7.2.2).

If a DumbNet switch fails, all the neighbor switches will report link failure, and the switch will be removed from the topology. Meanwhile, upon receiving link-up notifications, the controller will probe the ports to discover and verify the newly added links and switches. This way, we guaranteed rapid reaction for link failures and a reasonable reaction time for topology updates and link addition.

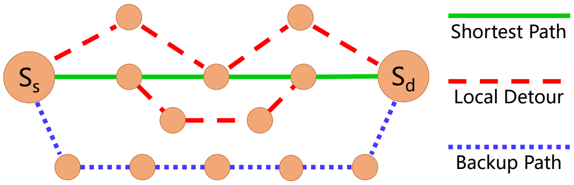


Figure 2: An example of a path graph.

4.3 Caching multiple paths

When a host asks the controller for a path to a specific destination, to one extreme, the controller can return the entire network topology and let the host figure out the best path itself; to another extreme, it can return a single path only. As the host caches the results, returning a larger portion of the topology means fewer queries to the controller and faster recovery. The benefit, however, comes at the cost of more resource utilization. We designed a mechanism, called *path graph*, as a tradeoff.

A path graph is a subgraph of the network topology containing three parts: (*i*) a *primary path*, which is one of the shortest paths from the source to the destination; (*ii*) some *local detours* by replacing a few switches or links on the primary path; and (*iii*) a *backup path*, which is a relatively short path that shares as few common switches or links as possible with the primary path. Figure 2 illustrates an example of a path graph.

While the primary path is optimal, hosts can use the local detours to quickly handle single link failures, and the backup path is designed to provide an alternative when many links on the primary path fail in a correlated way.

We use a subgraph instead of separate paths for efficiency reasons. The size of path graph is only proportional to the path length, which is much smaller compared to the entire topology, especially in a large network. Using path graph, we can merge multiple paths into a single subgraph instead of using separate paths. Besides saving space, the subgraph is also easy to scale up and patch failure. K-shortest paths only capture a very limited set of all possible paths, while subgraph represents many more possible paths with less memory overhead. When the network grows larger, the cost for computing K-shortest paths grows faster than that for computing the path graph. When link failure happens, it is easier to patch a subgraph than update a set of separate paths.

Algorithm for path graph generation. The controller can generate a path graph efficiently.

First, we compute the primary path with a common shortest path algorithm. It also randomizes the choice for equal cost links, so it generates different shortest paths, useful for load balancing.

Algorithm 1 Algorithm to find vertices in local detours

```
Input: The topology $G$; primary path $[p_1, p_2, \ldots, p_l]$; constant factors $\varepsilon, s$ (see Section 4.3 text).
Output: path graph $D$, including local detours for primary path
$D \leftarrow \emptyset$, $i \leftarrow 0$
while $i < l$ do
    $a \leftarrow p_i$
    $b \leftarrow p_{i+s}$                    ▷ or $b \leftarrow p_l$ if $i + s > l$
    $\mathcal{D}_i \leftarrow \{\forall x \in G, dist(a, x) + dist(x, b) \le s + \varepsilon\}$
    $D \leftarrow D \cup \mathcal{D}_i$
    $i \leftarrow i + s/2$
return $D$
```

Secondly, we increase the cost for all links on the primary path and run the shortest path algorithm again to generate the backup path. The increased cost makes it unlikely to reuse links and switches in the primary path, unless it is unavoidable (e.g. no redundancy for the link).

Finally, we add local detours to the primary path. We define an ε-good-detour for a primary path with length L as a path shorter than $L + \varepsilon$. Of course, there are many good detours. We prefer *local detours*, i.e. those deviate from the primary path as little as possible, in order to avoid affecting the global traffic pattern, should a large flow get detoured. Thus, we limit the local detours to alter at most s consecutive hops on the primary path, with detour length no more than $s + \varepsilon$. Letting the $P[1 \ldots L]$ be the primary path, Algorithm 1 provides a procedure to generate "s-steps, ε-good" local detours. We can prove that the union of primary path and all local detours form a connected subgraph, whose size grows linearly w.r.t. the length of the primary path. Precisely, the size of this subgraph is $O(L * P^{s+\varepsilon}/s)$, where P is the maximum number of ports per switch.

When s and ε in Algorithm 1 becomes large enough, the path graph grows to cover the entire network topology, and the host's routing decision degenerates to the traditional ECMP. In small topologies, the degenerated case is likely to happen if hosts communicates with each other. In large topology, not all pair of hosts communicates and thus the subgraph size is much smaller.

The path graph not only provides a configurable tradeoff between cache size and resilience but also help avoid overloading the controller during a link failure. Section 7.2.2 shows the performance improvements.

5 DATA PLANE

The data plane of DumbNet is simple: the host agent makes the routing decisions, including traffic engineering optimizations, while the switches just forward packets using tags. We first introduce the host agent design, focusing on efficiency and compatibility with existing networks. Then we briefly introduce our two physical switch implementations.

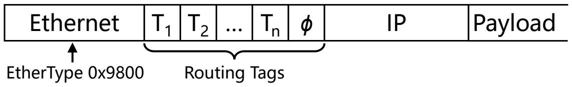


Figure 3: Packet format.

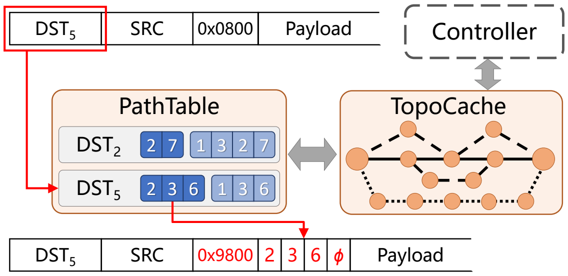


Figure 4: Host cache.

5.1 Ethernet-compatible packet header

To coexist with Ethernet traffic, we keep the original Ethernet header intact and insert our path tags between the Ethernet and the IP header. To distinguish from normal Ethernet packets, we set a separate `EtherType`, `0x9800`, in the Ethernet header. Figure 3 illustrates the packet format.

As each switch consumes one tag, the destination host agent needs to check if the remaining tag is ø. If so, it removes the tag and passes the packet up the normal network stack directly, as the packet is exactly an IP packet. Otherwise, the agent drops the packet. Note that we regenerate the Ethernet checksum once we remove the tag.

5.2 Host agent

The host agent handles most logics of DumbNet, and thus we want it to be efficient yet feature-rich. The host agent contains a kernel module that handles packet sending and receiving, as well as several service daemons. We have introduced some of these services previously, such as the topology discovery service. Now we focus on the kernel module and a path cache service.

The kernel module. The kernel module sits between the NIC driver and the kernel network stack and intercepts all incoming and outgoing messages. For incoming packets, the kernel module checks the `EtherType` to filter DumbNet packets, checks and removes the ø tag for validation, and passes it to the normal IP stack for processing. For outgoing packets, the kernel module queries the path cache service for the routing tag sequence, inserts the sequence into the packet header, and pushes the packet further down the network stack to the NIC. We implement the kernel module with the kernel NIC interface (KNI) in the Intel DPDK library [9].

The path cache service. We need to compute the routing tag sequence for packet headers. To accelerate the computation, we adopt a two-level cache architecture: the *TopoCache* and the *PathTable*.

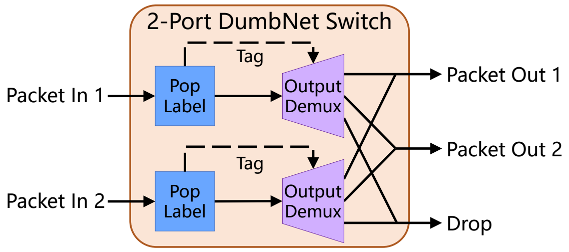


Figure 5: FPGA-based switch diagram.

TopoCache interacts with the controller and aggregates all path graphs (Section 4.3) from the controller. To find a path between a (`src`, `dst`) pair, the TopoCache first checks if it has the location of `dst` locally. If not found, it queries the controller and integrates the returned path graph into its cache. Otherwise, it computes the k shortest paths from `src` to `dst` and randomly chooses one as the path.

As a host usually sends many packets to the same destination, we cache the results from TopoCache in a PathTable. The PathTable is indexed by hosts, i.e., destination MAC address. It caches both the shortest path and backup paths. Note that for each destination, TopoCache computes k shortest paths and PathTable caches them all. The k choices are useful for load balancing. The PathTable remembers the previously used choice for each flow, and binds a flow to a particular path, except when a customized routing function (see Section 6) tells it to do otherwise. In addition to the k shortest paths, PathTable also caches the backup path discussed in Section 4.3. Caching backup paths allows the hosts to failover fast, significantly reducing the end-to-end downtime. The flows will automatically choose a new path when the older path is invalidated. Figure 4 summarizes the path cache service.

5.3 Physical switch implementation

We provide two DumbNet physical switch implementations.

First, we implement DumbNet in legacy Ethernet switches using MPLS to emulate the push-label routing. We implement the switch on commodity Arista switches with MPLS enabled, by inserting static rules that statically map the MPLS labels to the physical port numbers on the switch. The switch ID query packet is converted to a UDP packet and handled by the switch's CPU. We configure the host agent to use MPLS labels. We set the host MTU to 1450 to make packet shorter, and this leaves space for the MPLS labels in the header. We provide the MPLS-based implementation as a way to stay compatible with commodity Ethernet, and a way to build a real-world large-scale testbed for performance evaluation. The switch natively supports both port monitoring and broadcast that we need.

Then, we implement a DumbNet switch prototype using FPGA to show the switch hardware simplicity. We use an ONetSwitch45 FPGA development board [25] that has a Xilinx Zynq TM-7000 FPGA and four 1GE ports. We use a very simple, unoptimized two-stage processing architecture as Figure 5 shows. In stage 1, a pop-label module parses and removes the first routing tag, and in stage two, a set of demultiplexers send the packet to the correct port. Section 7.1 shows that even the unoptimized design reduces the FPGA resources utilization by almost 90% while achieving good performance.

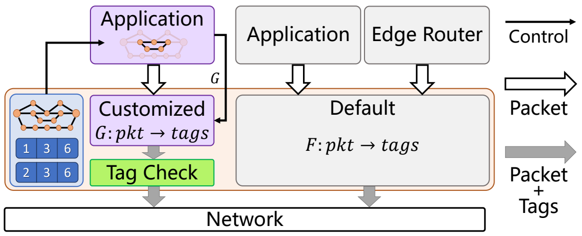


Figure 6: DumbNet interface.

6 EXTENDING DUMBNET FUNCTIONS

In conventional SDN, to implement a new function, one needs to coordinate state updates on hosts, controller, as well as multiple switches. In contrast, DumbNet maintains all network state on the host, mainly in the same topology service module that is closer to where the transfer originates. The collocation of network state management and the network users (applications) allows more convenient interfacing between the software and the network. Using well-known software configuration management tools, we also avoid update consistency problems.

In this section, we introduce the extension interface DumbNet provides, and then provide three examples of extensions: network virtualization, flowlet-based traffic engineering and layer-3 routing, showing that their implementations are both easy and efficient.

6.1 DumbNet extension support

Sophisticated and performance-sensitive applications often require better performance, be it higher throughput or lower latency. These applications often have information to do better network optimization too. For example, big data analysis applications often know the flow size in advance. Thus, it is essential to allow the applications to interact with network routing. The host-based design of DumbNet fits this requirement well.

DumbNet provides the following support for application-specific extensions, and Figure 6 illustrates the architecture.

1) Abstractly, some applications can provide a customized routing function, whose input is a packet and the network topology, and output is the corresponding route. Other applications without custom routing shares the same default routing function.

2) The TopoCache service offers an interface to reveal partial or entire network topology graph to applications that want to manage routing themselves. Each application may receive different topology based on its permission and priority.

The TopoCache service may offer different topologies also based on the overall workload.

3) The data plane forwarding agent supports priority-based packet forwarding and multiplexes packets from different applications on a single link. Also, the system provides a *path verifier* to check a route before adding it to the PathTable, to ensure that the application-generated routes do not violate security policies. As we will show in Section 7.2.2, the checking does not add too much overhead.

With the above two mechanisms, we can trivially implement network virtualization: we only need to provide different topologies for applications on different virtual network. Of course, we need to verify the paths to prevent malicious applications from violating the separation. More sophisticated virtualization layers can implement their own packet scheduling mechanism based on the topology information too.

6.2 Traffic engineering

As an example extension, we show how we implement flow-level traffic engineering on DumbNet. Traditional SDN makes the task hard as rerouting a flow involves updating state on many elements and thus require sophisticated coordination [19]. Enabling flowlet-based load balancing requires even more (often not existing) software/hardware support and configurations on each switch [21].

As described in Figure 4, the default routing function takes destination MAC address as input and determines path. To implement flowlet-based load balancing in DumbNet, the routing function uses flowlet ID instead of destination MAC address, taking the packet's destination IP address, port number, and a timestamp into consideration. The function can then deterministically choose one of the many k paths available in the PathTable, based on the flowlet ID, which will be bumped whenever flowlet timestamp expires. Thus, whenever flow ID changes, the extension will output a different path for packets with the same destination.

While per-flow tracking is expensive in traditional networks [21], it is doable on the host because the number of flows is limited at each host, and hosts already allocated the resources to track flow states in the operating system.

In summary, three features of DumbNet makes traffic engineering simple and efficient: 1) the source determines the entire path so we do not need any distributed updates; 2) each host maintains its own flow-level information; and 3) caching multiple alternative paths allows fast path switching.

In addition to Flowlet, we are implementing other typical traffic engineering approaches as future work, such as congestion-avoiding rerouting using based on early congestion notification (ECN), as well as approaches like pHost [10].

6.3 Layer-3 routing

In addition to network services, we can build network applications on top of DumbNet, and here we show how we can build a software-based router/gateway easily with less than 100 lines of code.

A router is simply a number of host agents running on the same node, one for each DumbNet (or other conventional) subnet. When it sends packet to a connecting DumbNet network, it adds tags to the outgoing packet as a normal host does. As all the incoming packets are Ethernet packet, the forwarding logic of router remains unchanged.

Optionally, we can extend source-routing to cross subnets, if both subnets run DumbNet, and there are direct short-cuts between switch ports of different subnets. This is common in data center networks because they are under a single administrative domain. For cross-subnet flows, the router can optionally tell the source host about the path in both subnets. The source host can then use the combined path directly without going through the router.

7 EVALUATION

We present our experimental evaluations to demonstrate the viability and performance of DumbNet. We performed evaluations of DumbNet with the following three implementations.

Testbed. Our testbed is a real data center network with 7 Arista 7050 64-port 10Gbps Ethernet switches and 27 servers. Each server has two 6-core Xeon E5-2620 (Ivy Bridge) CPUs running at 2.1GHz, 128 GB memory, and a Mellanox Connect X3 10GE NIC. We implement DumbNet with the DPDK + MPLS approach as described in Section 5.3. The testbed is organized as a leaf-spine topology with 2 switches in the spine layer and 5 switches in the leaf. Each leaf switch has 5 servers and a 10 GE uplink to each spine switch.

Emulation. To evaluate networks larger than our testbed, we create a software emulator similar to the architecture of Mininet [23]. We use separate threads to emulate DumbNet switches. The controller runs in a separate thread. To make the emulation efficient, we use one thread to emulate all hosts on the same switch. Our emulator runs on a server with 24 CPU cores.

FPGA. We use the FPGA-based switch we introduce in Section 5.3. We perform two evaluations with the FPGA switch. 1) We use the four ports to verify whether the hardware switch implementation works, and 2) we synthesize the forwarding logic module with more ports to show the simplicity of DumbNet switch design.

All the implementations are made with reasonable effort. We test it with typical Linux distribution and commodity switch in a data center running other jobs. With the simplicity of DumbNet, it is possible to get much better performance by paying more effort.

We use `iperf` for traffic generation in the micro-benchmarks, and use HiBench [15], a popular benchmarking suite, to generate real-world big data workload.

7.1 Implementation complexity

Figure 7 shows the FPGA resource utilization for DumbNet switches with different numbers of ports. We compare the resource utilization with the existing NetFPGA OpenFlow Ethernet switch [28] (ported to the ONetSwitch45 platform).

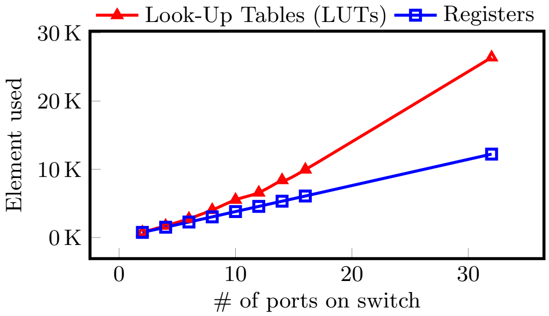


Figure 7: FPGA resource utilization v.s. # of ports.

For a 4-port switch, the OpenFlow switch uses 16, 070 look-up tables (LUTs) and 17, 193 registers, while we use 1, 713 and 1, 504, respectively. Note that we excluded the resources in input/output buffers and the Ethernet interfaces for the comparison. Plus, we do not use the embedded CPU cores on the FPGA while the OpenFlow switch uses it for its control agent. The switch implementation is short and straightforward too: only 1, 228 lines of Verilog code.

The comparison shows that we can dedicate most of the chip area to the switching fabric and packet buffers, instead of lookup tables and control logics, resulting in a resource efficient hardware switch design with high port density.

On the software side, DumbNet consists of about 7,500 lines of C/C++ code excluding extensions. Table 1 shows a breakdown of the code. It takes approximately 6 human-months to develop the core components of DumbNet, including the evaluation design, where about 1/4 of our engineering efforts dedicated to. Thus, DumbNet design does reduce network complexity.

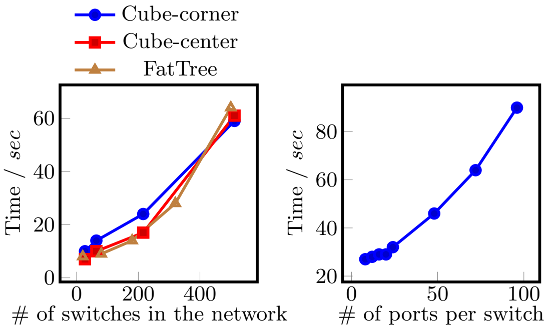
7.2 Micro benchmarks

7.2.1 Topology discovery. **Discover time w.r.t. the network size.** Using the emulator, we test the topology evaluation process on networks with different sizes (in terms of the number of switches) and various topologies. We use *fat tree* and *cube*, two most common topologies for data center networks. We test three controller placements: in the leaf of the fat-tree, in the center of the cube and at a corner of the cube. All switches in the experiment have 64 ports, and the controller needs to probe each possible port to find available links.

Figure 8(a) shows that the discovery is fast: even in a network with 500 switches with 64 ports each (translating to 5,000 hosts), we can discover the entire topology using a single controller within 70 seconds. As the bottleneck of topology discovery is the packet processing rate of the controller, the time is roughly proportional to the number of switches in the network, which grows linearly with the number of probe messages. Figure 8(a) also shows that the network size is the primary contributing factor to the discovery time, while the topology and the location of the controller both seem less important. This emulation result is worse than real-world performance. This is because when the topology size gets larger, the host processor becomes very busy and thus increase the discovery time.

Agent	Disc.	Maint.	Graph	Total	+Flowlet	+Router
5000	600	200	1700	7500	100	100

Table 1: Code breakdown in different modules



(a) Different topology, controller position and number of switches.

(b) Different port density on a 8-cube topology.

Figure 8: Topology discovery time with different settings.

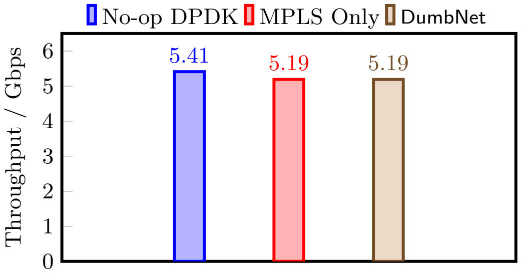


Figure 9: Testbed comparison of throughput between different network implementations.

Discovery time w.r.t. per-switch port count. Using the same emulator with a $8 \times 8 \times 8$ cube topology, we vary the number of ports per switch while holding the topology and number of links constant, and then evaluate the topology discovery time. Figure 8(b) shows that the time consumption w.r.t. the number of ports roughly follows a quadratic trend. Such trend is consistent with our analysis in Section 4.1 that the complexity in terms of PMs is $O(N * P^2)$ where P is the maximum per-switch port count.

Testbed results. In our testbed, it takes $3 \sim 5$ seconds for a single controller to discover the entire network topology with 7 switches, 10 links, and 27 hosts. The discovery on real testbed is faster as the probing runs in parallel on different machines, instead of the emulation using a single node.

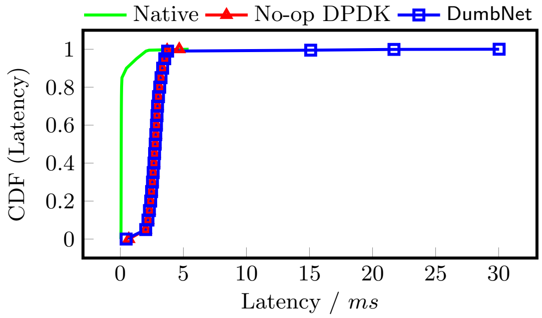


Figure 10: The round-trip latency distribution of DumbNet in the testbed, compared to native Ethernet and no-op DPDK.

7.2.2 Throughput and latency. **Single host throughput.** Under failure-free network conditions, DumbNet achieves similar performance both in terms of throughput and latency as normal DPDK-based network. Figure 9 shows the throughput comparison of no-op DPDK, adding empty MPLS with DPDK and DumbNet on our testbed.

Our servers with DPDK enabled (but performs no packet processing) can achieve a throughput of 5.4 Gbps, about half of the theoretical 10 Gbps throughput. The performance loss is because DPDK does lots of tasks in software instead of hardware, such as checksum and packet segmentation. Enabling MPLS header, even with only a single constant tag, adds an extra header-copy operation, causing about 4% additional overhead, making the throughput at around 5.19 Gbps. We consider these two the baseline for our evaluation on single host performance. DumbNet's source routing and tagging add only negligible overhead, and the throughput is still at 5.19 Gbps.

Aggregate throughput on leaf switches. In another experiment, we use two leaf switches in the testbed, each connecting 14 hosts, and send traffic between them. Recall that a leaf switch has a link to each spine switch, and thus the total uplink from a leaf switch is 20 Gbps.

The measured aggregated throughput reaches 18.5 Gbps. This throughput shows that our MPLS-based implementation enables the switches to run at wire speed, and our load balancing can fully utilize both paths.

Testbed Latency. For latency, DumbNet achieves almost the same latency with the no-op DPDK. On our testbed, we send 100 packets between every pair of hosts and measure the end-to-end round-trip time (RTT) of these packets. Figure 10 plots the cumulative distribution (CDF) of the RTT numbers. We can see that using software-based DPDK significantly increases the latency over the native Ethernet. However, the extra overhead of DumbNet is negligible, comparing to the no-op DPDK overhead.

Zooming into the long tail of Figure 10, we find about 0.5% of the packets have a latency as long as 20 to 30 milliseconds. This is consistent with our design that before two servers initiate their connection, the host agents need to ask the controller for the initial path (the path graph), and this query takes an extra round trip to the controller. Since the sender and receiver send their queries in tandem, the extra latency overhead is $2 * RTT$ plus two query response time at the controller. Since all hosts start to ping each other at the same time, long tail in packet latency CDF is the result of concurrent queries to the controller from all machines, which resembles the worst case tail latency distribution. In actual networks, path queries from different hosts will be more evenly distributed over time, which leads to shorter tail latency.

We believe several design choices contribute to the low latency overhead in DumbNet: efficient client cache, the client data path implementation as well as the simple switch design.

Latency overhead breakdown. Table 2 shows the breakdown of latencies on different functions in the DumbNet kernel module. We use a fat-tree topology with 5,120 switches and 131,072 links. To measure PathTable lookup time, we inserted 10K random entries into the Table. The path length we verify is 16, longer than most DCN paths. We run each test 1,000 times and take the average time. We can see that the latency overhead from DumbNet kernel module is actually very low.

FPGA Switch Latency. To test the forwarding performance of the FPGA implementation, we measure the latency by sending packets with fixed tags. We let each packet go through 3 hops in the FPGA switch, keeping the same number of hops as in the testbed. To focus on the switch latency, we exclude the software stack overhead. The average latency for the 3 hops is $100.6\mu s$, with a maximum of $152\mu s$. Considering our unoptimized FPGA implementation with 1GE ports, we believe the low latency shows the advantage of the dumb switch design.

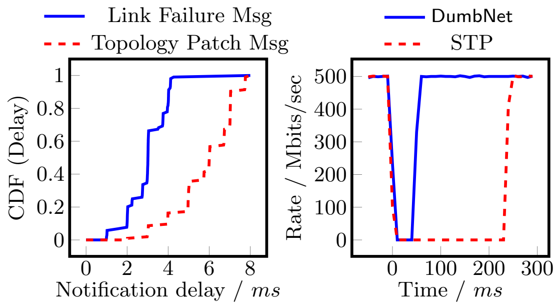
7.3 Failure recovery

Link Failure Notifications. The speed of hardware determines link failure detection time. The key is to reduce the notification delays. We focus on evaluating the delay using the real testbed by injecting link failures. As Section 4.2 describes, we notify hosts using two messages in two stages, the link failure message from a host and topology patch message from the controller.

We measure the time between the failure discovery and notification arrival at each host. It is tricky to measure delay in a distributed testbed as we do not have a synchronized clock. Instead, we let each host to ACK a measurement server once it receives the notification, and compute the receiving time by adjusting the RTT between the host and the measurement server.

PathTable Lookup	Path Verify	Find Path
0.37 μs	7.17 μs	1.50 μs

Table 2: Time of different kernel module functions.



(a) CDF of topo. change notification delays.

(b) Throughput change.

Figure 11: Network recovery statistics in the testbed (failure at time 0).

Figure 11(a) shows the cumulative distribution of the notification delay on all hosts. We can see that the majority of hosts receive the link failure notification within 4 milliseconds, and receive the patch message within 8 milliseconds; the entire process finishes within 10 milliseconds. When a host receives the notification, it is likely to find an alternative path in the local cache, and thus will recover the transmission without further delay.

Comparing to traditional STP. We evaluate how fast DumbNet recovers from a single link failure. We record the transfer rate between two hosts connecting to different leaf switches in the testbed. We inject a single failure (i.e. cut one of the two links between spine switch and leaf switch) while the network is at close to its full capacity (We limit the network bandwidth to 0.5 Gbps so we can saturate the link).

We run a script on Arista switch to monitor the port state and send the Link Failure Notification packets as described in Section 4.2. These packets can be sent even faster if it's done by hardware. The baseline we compare against is the off-the-shelf Ethernet Spanning Tree Protocol (STP) [30]. As there are redundant paths, STP can also automatically find alternative paths.

Figure 11(b) shows that the DumbNet approach described in Section 4.2 is almost 4.7× faster than STP. This is because with all states in hosts, instead of running a fully distributed, multi-round STP protocol on link failure, the DumbNet failover only needs to broadcast two notification messages and switch to the backup path in local PathTable immediately.

Storage overhead of path graph. The fast recovery is at the cost of caching a path graph instead of individual paths. Here we evaluate the number of switches cached with different ε choices (see Section 4.3). We emulate a path graph with a $10 \times 10 \times 10$ cube topology. We fix the parameter s at 2, meaning that we allow a maximum of 2 steps deviation from the shortest path. Then we randomly pick primary paths of different length in the network and see the size of their corresponding path graph with different ε. Figure 12 shows the result. We find that for longer paths, a larger ε results in lots of extra caching, as it allows detours at each hop. For shorter paths, even with a large ε, the cache size is still reasonable. In real data centers, the path is usually short, and thus our the path caching cost stays small.

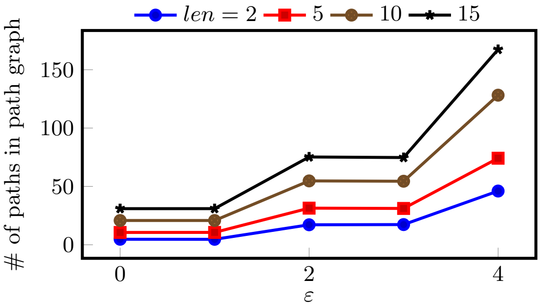


Figure 12: The size of path graph w.r.t. ε choices, under a 10-cube topology.

Even in a large data center with 2,000 switches and 100,000 hosts, saving both TopoCache and PathTable (defined in Section 5.2) will cost at most 10MB of memory, a negligible amount for modern servers. Nevertheless, caching only the relevant topology and paths may improve lookup performance.

7.4 Performance with real workload

As a real-world macro benchmark, we use Intel HiBench [15], a commonly used big data benchmark suite, to evaluate the performance of DumbNet on our testbed topology as described early in this section. As we do not have enough servers to saturate the network bandwidth in HiBench, we limit spine switch port speed to 500 Mbps. We have flowlet enabled (see Section 5.2) in the experiment. Note that we use HiBench to capture the flow dependencies in real-world applications, rather than to simulate a real-world heavy workload to stress test the prototype system.

Like the evaluation in Section 7.3, we compare performance among full DumbNet with TE, no-op DPDK, and a version of DumbNet without the path graph and flowlet-based traffic engineering.

Figure 13 shows that DumbNet outperforms conventional network in all the tasks. Flowlet TE plays an important role. As the hosts randomly generate paths, it is likely that each host uses a distinct path for each flowlet, leading to more evenly distributed traffic, therefore reduces the likelihood of link congestion. The comparison to the version without TE confirms that the performance becomes much worse in the single-path setting.

8 CONCLUSION AND FUTURE WORK

The development of SDN leads us to revisit the design philosophy question: how should we divide responsibilities between switches and hosts? How can we better integrate network functionalities with the software? Through the design and implementation of DumbNet, we achieve one extreme: making switches as dumb as possible. With switches that have no

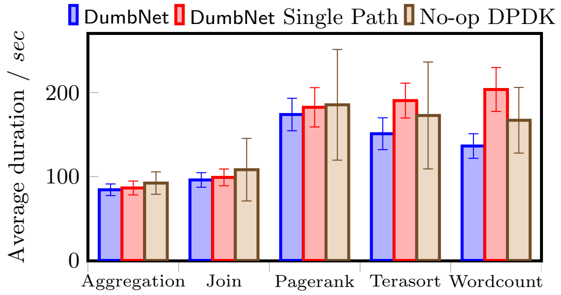


Figure 13: Testbed performance of HiBench tasks.

CPUs and no state, we show that we can efficiently support essential network functionalities such as bootstrapping, routing and fault tolerance. We can also support powerful software extensions and demonstrate traffic engineering, layer-3 routing and network virtualization with very small effort. Furthermore, we may achieve higher performance by porting the host agent into today's smart-NIC. In practice, keeping switches dumb not only reduces the switch hardware cost but also avoids the complicated distributed state update problem. Philosophically, building the extreme solution allows us to come back to a clean start to rethink about switch and DCN fabric design, which leads to our future work: what else we must add back to the dumb switches to make the DCN even more powerful? For example, we are adding mechanisms for packet statistics and ECN support to the switch. Note that these mechanisms either require no state, or only soft state, keeping the switches dumb.

ACKNOWLEDGEMENT

We would like to show great appreciation to our shepherd Edouard Bugnion and anonymous reviwers for their feedback. This work was supported in part by the National Natural Science fundation of China (NSFC) Grant 61532001, Tsinghua Initiative Research Program Grant 20151080475, MOE Online Education Research Center (Quantong Fund) Grant 2017ZD203, and gift funds from Huawei and Ant Financial.

REFERENCES

[1] Hussam Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. 2010. Symbiotic Routing in Future Data Centers. In *ACM SIGCOMM*.

[2] Eric A Brewer and Bradley C Kuszmaul. 1994. How to Get Good Performance from the CM-5 Data Network. In *IEEE International Parallel Processing Symposium (IPDPS)*.

[3] Matthew Caesar, Martin Casado, Teemu Koponen, Jennifer Rexford, and Scott Shenker. 2010. Dynamic Route Recomputation Considered Harmful. *ACM SIGCOMM Computer Communication Review* 40, 2 (2010), 66–71.

[4] R.W. Callon. 1990. Use of OSI IS-IS for Routing in TCP/IP and Dual Environments. RFC 1195 (Proposed Standard). (December 1990).

[5] Mark Coates, Rui Castro, Robert Nowak, Manik Gadhiok, Ryan King, and Yolanda Tsang. 2002. Maximum Likelihood Network Topology Identification from Edge-based Unicast Measurements. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 11–20.

[6] CoreOS. 2017. etcd Homepage. https://coreos.com/etcd/. (2017).

[7] Amogh Dhamdhere, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. 2007. NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM CoNEXT*.

[8] Luyuan Fang, Fabio Chiussi, Deepak Bansal, Vijay Gill, Tony Lin, Jeff Cox, and Gary Ratterree. 2015. Hierarchical SDN for the Hyper-Scale, Hyper-Elastic Data Center and Cloud. In *ACM SOSR*.

[9] The Linux Foundation. 2017. DPDK Homepage. https://dpdk.org/. (2017).

[10] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT*.

[11] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*.

[12] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. 2010. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *ACM CoNEXT*.

[13] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. 2008. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *ACM SIGCOMM*.

[14] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*.

[15] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *IEEE International Workshop on Information and Software as Services (WISS)*.

[16] Yiyi Huang, Nick Feamster, and Renata Teixeira. 2008. Practical Issues with Using Network Tomography for Fault Diagnosis. *ACM SIGCOMM Computer Communication Review* 38, 5 (2008), 53–58.

[17] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*.

[18] Xin Jin, Nathan Farrington, and Jennifer Rexford. 2016. Your Data Center Switch is Trying Too Hard. In *ACM SOSR*.

[19] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic Scheduling of Network Updates. In *ACM SIGCOMM*.

[20] Sangeetha Abdu Jyothi, Mo Dong, and P Godfrey. 2015. Towards a Flexible Data Center Fabric with Source Routing. In *ACM SOSR*.

[21] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. 2007. Dynamic Load Balancing Without Packet Reordering. *ACM SIGCOMM Computer Communication Review* 37, 2 (2007), 51–62.

[22] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. 2017. Beyond fat-trees without antennae, mirrors, and disco-balls. In *ACM SIGCOMM*.

[23] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *ACM SIGCOMM HotNets Workshop*.

[24] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.

[25] MeshSr. 2015. ONetSwitch45 Webpage. https://github.com/MeshSr/ONetSwitch/wiki/ONetSwitch45. (2015).

[26] J. Moy. 1998. OSPF Version 2. RFC 2328 (INTERNET STANDARD). (April 1998).

[27] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C Mogul. 2010. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *USENIX NSDI*.

[28] Jad Naous, David Erickson, G Adam Covington, Guido Appenzeller, and Nick McKeown. 2008. Implementing an OpenFlow Switch on the NetFPGA platform. In *ACM/IEEE ANCS*.

[29] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *ACM*

SIGCOMM.

[30] Institute of Electrical and Electronics Engineers. 2004. IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges. *IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998)* (June 2004), 1–277. https://doi.org/10.1109/IEEESTD.2004.94569

[31] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM*.

[32] Ramon Marques Ramos, Magnos Martinello, and Christian Esteve Rothenberg. 2013. SlickFlow: Resilient Source Routing in Data Center Networks Unlocked by OpenFlow. In *IEEE Conference on Local Computer Networks (LCN)*.

[33] Y. Rekhter, T. Li, and S. Hares. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard). (January 2006).

[34] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. 2011. Network Architecture for Joint Failure Recovery and Traffic Engineering. *ACM SIGMETRICS Performance Evaluation Review* 39, 1 (2011), 97–108.