Scalable Routing Information Base for the Global Data Plane

Jason Li XiangJun[1] and John D. Kubiatowicz[2]

Abstract

The number of edge devices has grown exponentially over the past few years and will continue to grow at an accelerated rate with the gradual adoption of 5G technology which brings even more bandwidth and connectivity to the world, and the normalization of smart devices and machines such as cars, refrigerators, and watches. Data, along with its transmission, storage, and eventual usage become an important issue for many. The Global Data Plane [1] project was created to address the challenge of collecting, accessing and managing data scalably with built-in security and good mobility for end devices by treating data as its first-class citizen and providing support for distributed storage, fast retrieval, and rule-based efficient routing.

Our work focuses on building a scalable and secure routing architecture for the top-level network that connects all the edge routers of different trust domains. Our proposed approach seeks to accommodate trillions of unique log hashes (or ids) and up to one million network devices which, in our view, should be more than sufficient to fulfill the promises of the GDP project. The objective of this paper is to propose a starting point or a sketch of how such a system should look like. Although we were not able to simulate the model and get results on 1 million nodes due to limited computing resources, our experimental results for the proposed architecture works reasonably well in a network of 400k top-level switches. We present our system architecture, design choices, and analysis of our simulation results in this paper.

Introduction

Our work revolves around building a routing backbone for trillions of connected network devices via location-independent identifiers for the Global Data Plane (GDP). The GDP project was first proposed by the Berkeley Swarm Lab and sought to deliver a higher level of abstraction for data that fits more with the requirements of modern edge devices. The GDP project proposes a "single-writer append-only log" model as the interface for its users. We present below a few important considerations in the conception of the GDP project and also our decision for the routing layer:

1. Security and Configurable Privacy: Data privacy has become a huge issue recently. As our lives become more digital, more of our private life is exposed. A smart home device like Amazon Echo is constantly recording the voices in our house including our conversations. Google's Nest thermometer is transmitting our room temperature to the cloud server around the clock. The GDP should provide an easy way for users to protect that data even when it is stored in cloud data centers. That is to say, encryption should come by default. No one but the owners of the data should have the rights to modify it. The owner should be able to give selective individuals READ permissions too. When the data is in transit from its source to the destination, the retriever should be able to specify regions through which he wants to avoid sending the data to prevent side-channel attacks. Modern attackers have

[1] Jason is an undergraduate student in Computer Science and Business, University of California, Berkeley Email: lxj@berkeley.edu

[2] John is a professor in the Department of Computer Science, University of California, Berkeley

grown increasingly sophisticated and side channels have become an important source for data leaks.

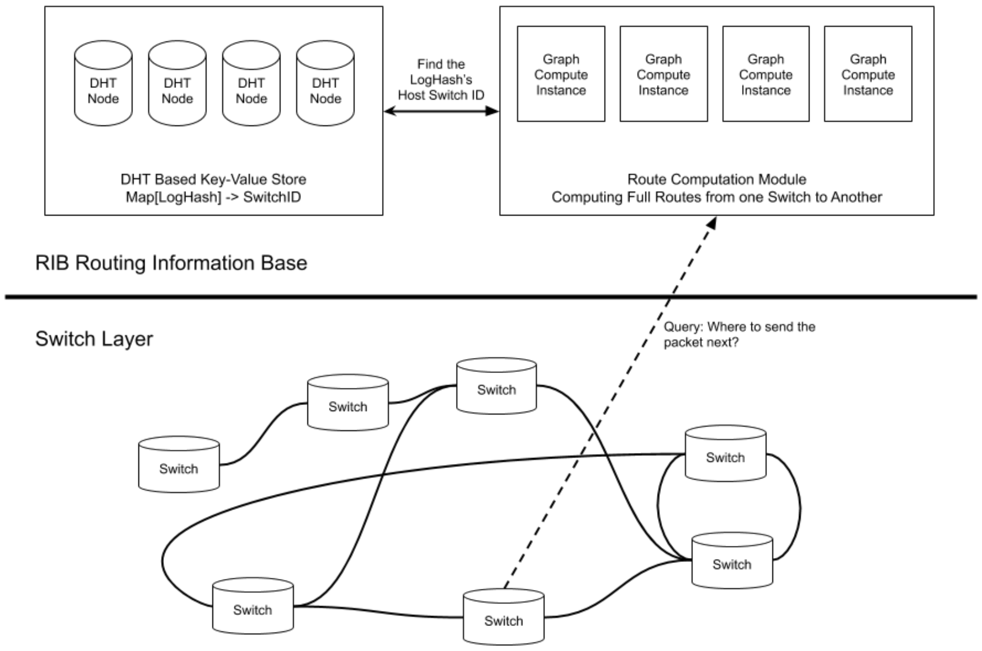
2. Scalability: According to the latest MOMO's report, it was estimated that over 2.5 quintillion bytes of data are created every single day [2]. And with the rise of the IoT devices, a recent estimate shows that we have more than 22 billion devices connected to the Internet. This number is projected to rise to 50 billion by 2030 [3]. The GDP thus has to accommodate potential data containers (a.k.a logs) on such a massive scale.
3. Durability of Data: Data loss happens and it is essential for backup ability to be built into the GDP system – the system should allow data containers to be hosted by multiple network endpoints. During retrieval, the routing layers should make smart decisions about where to get the data from, taking into account latency and user's explicit preferences, if any.

One of the most important characteristics of the GDP is its naming. Rather than using IP to communicate or an overlay network that is powered by IP, the global data plane is envisioned to have a true flat namespace – they are unique, self-certifying (via the use of public and private key infrastructure), and location independent. The traditional architecture uses location-dependent identifiers (e.g. IPs) along with a hierarchical search path from the local router's routing table all the way to tier 1 providers. Since IP addresses are typically distributed in blocks to providers in different countries and then further divided into different regions, the prefix of IP addresses can be used as a cue to make efficient decisions for routing. For example, if an IP address 216.190.0.0/16 belongs to the United States. If a user in Singapore wants to send traffic to any address covered by this range, his ISP can quickly figure out where the next hop would be. However, this approach has its own shortcomings. Many a time, routing and route computation, especially at the BGP layer, is not based on the shortest path possible but rather a set of complicated rules set by the ISPs which has more to do with business than efficiency. And we need to trust all the tier 1 providers in order for the system to function properly. Any misconfiguration by a top tier ISP, be it intentional or accidental, can disrupt traffic for customers beyond his service. As a quintessential example, a recent route misconfiguration by one of the top tier ISPs from China resulted in the traffic to Google from the US east coast being sent all the way to China and then Russia before going back to the US. In addition, the network does best-effort delivery. While this model works as it delivers the data, security is a big concern even when the data is transmitted over encrypted channels. Side-channel attacks are possible and an honest but curious on-path attacker can extract a significant amount of meta-information by looking at the communication pattern. Our solution to this issue is to allow clients to specify a subset of trusted top tier switches or a set of switches to be excluded.

While using a flat namespace can be an elegant solution for the problems faced by IP-based routing, there are a few challenges, especially if our network seeks to accommodate trillions of data containers. One of the biggest challenges is how to do lookup efficiently given that we might have such a huge amount of data and also how to route efficiently. This issue has implications regarding how we should construct our top-level topology. Various research has been done in the past. The first group of research on flat-name routing build on top of DHT technologies and led to the creation of Kademlia [4] and Chord [5] which essential store the hashes of a node's information in a distributed fashion. What distinguishes it from the traditional distributed storage system is that each storage node in these network does not need to know everyone else that is in the system, they just need to know a subset of the machines. Regarding efficiency, lookup can be on in O(log n) with n being the number of storage nodes in the system. There are also other schemes that built on these ideas such as the "Canon in G Major" [6] which extends a planer DHT into a hierarchical structure which would allow different trust domains to operate. However, all these solutions presuppose an overlay network and uses the native routing on the overlay network which is IP. This is a major limitation if we were to choose them for our design. There are also approaches such as ROFEL [7] and VRR [8] which aimed to build an id-based routing protocol by introducing a node id space upon which path discovery happens dynamically and automatically. VIR [9] furthered the work of ROFEL and VRR by proposing an intermediate virtual id space to dynamically optimize the connection between hosts to lower the stretch. However, even VIR's performance is not good enough for a large-scale network because of the following:

1. The overhead of nodes joining and leaving is significant.
2. Each node potentially has to store a lot of routing information. It is hard to impose such storage requirements on nodes if one does not have control over them as it is indeed the case in the top-level switch network in GDP.
3. Stretch is still significant. With about 200 nodes in the network, the stretch can go up to 3.5.
4. Security remains a top concern if all the nodes are not controlled by a single entity. A dishonest node in the network can potentially screw up the whole routing mechanism and affect routes that do not event pass through itself, much like the vulnerability in the GDP case.

For this reason, we have chosen to use a centralized RIB with path computation done via an enhanced Dijkstra algorithm which offers both speed guarantee and can allow the injection of user preference into the computation of routes such as the set of nodes to avoid. Dijkstra algorithm can be very fast on graphs with millions of nodes offering about 1 second latency [10]. Some C implementation of the graph engine such as graph-tool have a latency of 0.5 seconds on real-world road dataset that consists of 1.4 million nodes. Various techniques are also available to further speed up the shortest path search such as contraction hierarchies. By centralizing the management of the top-level topology and route computation, it offers a lot of speed-up potential and flexibility for shortest path computation which always yields a stretch of 1.



Overview

The figure above offered a good overview of our top-level architecture. There are two important elements in the system: switches and data containers(logs). Switches are responsible for forwarding the traffic and the containers are the actual data. While there might be trillions of containers, there would be at most a few million switches at the top level.

Each switch in the figure above represents an edge switch of a trust domain. A trust domain could be a cooperation, a local IPS or a university, inside which there could be thousands of nodes and millions of data containers. Each data container is identified by the hashes of its metadata which includes the public key of the owner of the data container. This design will make the container self-certifying because anyone who is given the hash of the container can verify if the network indeed returned him genuine data. Any data temperament by malicious on-path switches or corruption due to an unreliable link will be detected because the hash value does not match.

Our architecture does not assume any point-to-point connectivity at the switch layer. Nor is there a native routing capability such as the IP-based prefix routing capability. All switches are assumed to be L2 connected and the connection does not have to be a fully connected graph. However, one simplifying assumption that we made is that each node can reach the RIB directly without going through another node. This is a necessary condition for a switch to qualify as the top-level switch. Top-level switches can peer with each other and communicate via dTLS.

The routing information base is a huge cluster of nodes that mainly serve 2 distinct functions. The route computation node is responsible for maintaining an up-to-date diagram of the top-level switches and perform route computation. Assuming that we have 1 million nodes and each node connects with 5 other top-level switches, the total memory required to store this topology information would be (10 ** 6) * 5 * 256 / 8 / 10 ** 9 < 2 GB. Switches actively advertise their neighbor information to the RIB. The route computation modules are horizontally scalable because one computation instance is independent of another. The DHT based key-value store is used for efficient look up from the log hash request by the end user to the top-level switch that exports the container's information.

Edge switches, upon receiving a data retrieval request from its internal nodes will then contact the RIB for routes. The RIB computes the shortest path from the edge switch to the destination switch that holds the log id and then sends that information back to the switch which then sends the packet out according to the next hop switch id returned by the RIB.

Route & Log Advertising

RIB will receive route advertisement from the top-level switches and also verify that those advertisements are indeed valid in that the advertisement has the consent of the data owner though a challenge-response mechanism via the use of public key cryptography. Each route advertising request from the top-level switch to the RIB will get a nonce which is a randomly generated 512-bit string. The nonce could potentially be signed by a publicly trusted certificate so that when the data owner received the challenge, he can validate that it is indeed from the real RIB. This way, man-in-the-middle attack can be prevented and we can make sure that the RIB, assuming that it is an honest player, will only advertise logs with the log owner's consent. After receiving the nonce from the RIB, the client needs to send back the following information via the top-level switch:

- Full metadata of the container information
- Timestamp
- Valid Window
- Signature

The full metadata will be used to compute the log hash. Timestamp and valid window will be set by the user to prevent purposefully delayed submission of a log advertising response by any on-path attackers. The RIB, upon receiving a request from the client will validate the time constraint by first checking if the timestamp exceeds its maximum age value. The RIB will then compute (Timestamp + Valid Window) and compare it with the current time. If the time is smaller than the currently time, this response has expired. The client will have to compute the following signature, using his private key to prove that he indeed owns the data that is about to be advertised:

```
Signature = PubKey_Sign(private_key, SHA3_512(Full metadata || Timestamp || Valid Window || nonce))
```

The computed signature will be verified by the server. Requests containing invalid data or invalid signature will be rejected. After the request is processed, a success message will be returned to the client. The RIB will then put a new entry into the distributed hash table cluster with log hash as the key and as the switch ID as the value.

For switches, there are two major types of requests made to the RIB:

- SWITCH_HELLO: when a new switch wants to register with the RIB
- SWITCH_ADVERTISE: when the switch wants to update its peer table to the RIB

SWITCH_HELLO has a very similar process as log advertising. Each switch also has its own public and private key pair. When a switch joins a network, it first sends a join request to the RIB. Within the same session, the RIB will reply with a nonce. The switch will then send the following information back to the RIB as the reply:

- Public Key of the Switch which is also used as the ID
- Timestamp
- Valid Window
- Signature

The signature will be generated as follows:

```
Signature = PubKey_Sign(switch_private_key, SHA3_512(Public Key of the Switch ||Timestamp || Valid Window || "SWITCH_HELLO" || nonce))
```

SWITCH_ADVERTISE is sent to the RIB whenever there is a change in its peer table. In this request, not only does the switch have to send his peers' ids over, he also has to obtain proof from his peers that they are indeed connected to prevent malicious top-switch operators from impersonating other switches. A possible attack to maliciously intercept the traffic going to Switch X would be to claim to the RIB that the malicious switch is directly connected to X. The RIB may then forward all traffic going to switch X along a path that contains the malicious switch. Having a proof mechanism would prevent such attacks. The request follows the same format as SWITCH_HELLO. The response is as follows:

- List of Proofs: (Peer ID, Timestamp, Valid Window, Proof)
- Timestamp
- Valid Window
- Signature

Proof must be returned by the peer and can be computed as follows:

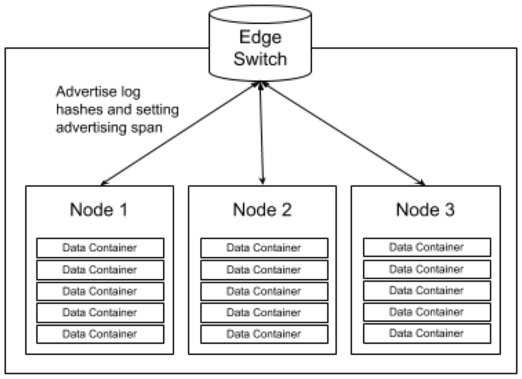
```
Proof = PubKey_Sign(peer_switch_private_key, SHA3_512(Public Key of the Peer || Public Key of this Requesting Switch || Timestamp || Valid Window || "SWITCH_ADVERTISE_PEER_PROOF" || nonce))
```

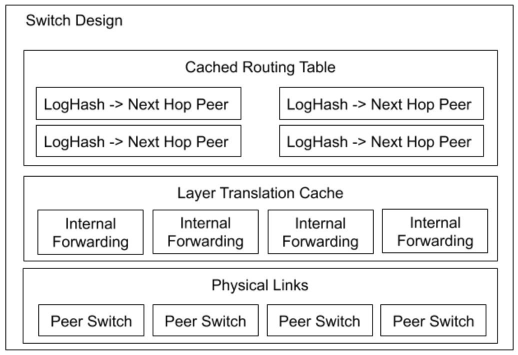
The inclusion of timestamp and valid window will prevent attackers from using the same proof even after some time. Although by that time the nonce might have expired, adding timestamp and valid window would still give the peer switch greater control.

The requesting switch will then generate the signature as follows:

```
Signature = PubKey_Sign(switch_private_key, SHA3_512(List of Proofs || Timestamp || Valid Window || "SWITCH_ADVERTISE" || nonce))
```

When Switch A added a correct proof for Switch B it means that Switch B is going to accept A's data, so the RIB will add a directed edge from Switch A to Switch B. Equivalently, A is saying to the RIB that I am reachable via Switch B. Notice that this has not changed anything about Switch B's neighbor map. Traffic directed to Switch B will still never pass through Switch A because Switch B has not advertised Switch A as its neighbor yet. For there to be edges in both directions, both Switch A and Switch B will have to advertise each other to the RIB.





Switch Design

Each edge switch will have to perform many roles. First and foremost, it has to establish and maintain connections with other peer switches via L2 links. Upon each new peer connection, the switch should send a HELLO message to the other switches and exchange public keys via a similar challenge and response mechanism. Whenever there is an update in the peer table, the switch should then send an update message of type SWITCH_ADVERTISE to the RIB.

The figure on the right outlines the internal structure of a trust domain. There might be many nodes under the edge switch which acts as an egress gateway. If a log owner wants to advertise his log form somewhere within the trust domain, he has to advertise it and the message will first go to the edge switch and then to the RIB. Similarly, when he wants to access a log stored in other domains, he needs to contact the RIB via the edge switch of the trust domain. In a sense, the edge switch needs perform the functions of Network Address Translations typically found in a router because it needs to provide a proxy capability for nodes within the trust domain to reach the outside and for an incoming request looking for a particular data container to be directed to the right internal node. This function is performed by the Layer Translation Cache in our design. As this paper is not focusing on the tunneling between layers across the hierarchy, we will not go into the details of the architecture here.

Cache Routing Table is a Least Recently Used (LRU) Cache to accelerate packet forwarding. It acts very much like the routing table in a switch. However, instead of doing prefix matching, this table maintains the destination switch ID and the next hop ID to speed up communication after the first packet passes through it. Since the very first time when a node P tries to retrieve content Q, routing resolution has to happen at each hop (switch), this layer can be used to efficiently forward packets after the very first trip because the next hop ID is cached in each switch along the route and no query to the central RIB is necessary.

Finding a Route

When a user sends a request to retrieve some data container by its ID. The request first hits a resolution service within the trust domain. If the data does not reside within the trust domain, the request will arrive at the edge router which will then make a ROUTE_COMPUTE request to the RIB. This message will include the following:

- Log ID
- (Not implemented) Trusted Switches to Include
- (Not implemented) Untrusted Switches to Exclude

The RIB will them perform route computation based on these parameters and return to the router the next hop ID. At the same time, the RIB will cache the computed route because very likely the RIB will be consulted again by the next hop router along the path. By putting the computed results in memory, the amount of computation can be reduced significantly.

Note that we assume the channel through which such communication happens is already established and is secure. Identities should have been proven for both parties using the public key infrastructure. Procedure to achieve this is very similar to that described in the Route & Log Advertising Section.

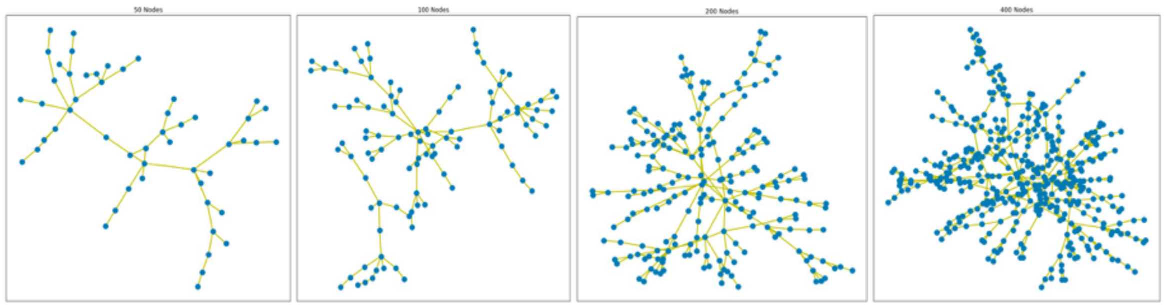
Evaluation

Our implementation has about 2,500 lines of code and is done in Golang because its friendliness with I/O intensive programming. For languages such as Java and Python, without using special packages every TCP connection established will create a new thread. Running simulations on the millions scale is simply impossible because the sheer amount of threads and context switch overhead will paralyze the operating system. Go, on the other hand, employs light-weight threads called go routines which are not real operating system threads. Go has its own user space runtime scheduler to coordinate the different virtual threads which are much faster than using the thread management of the Linux operating system. Another way to deal with the Linux limitation is to use kernel bypass techniques via packages such as DPDK and move the network stack into the user space along with the use of C/C++ which gives even more control and efficiency to the programmer. However, given the limited time we have, Golang seems to be our best choice as it strikes a balance between verbosity and efficiency.

Due to limited time and resources, our prototype of the system did not include the security dimension of our outline architecture. All participants in the system are assumed to be trustworthy. We don't think adding the security dimension would greatly impact the experimental results as most of the cryptography burden is one-time and only happens at the set-up time. Once a router enters its stable mode and starts passing packets, it has to do very little cryptography.

Graph Generation

To evaluate the scalability of our implementation, we tried to find publicly available datasets on the topology of tier 1 ISPs but we only found limited and dated information with about three thousand nodes which does not meet our needs so we need a way to generate large and density-constraint network graphs of different sizes and they have to resemble real-world network topologies. We used a very simple algorithm. Starting from the second node, each new switch is equally likely to be connected with switches that are already in the network. To show the reader how the produced graphs look like, we have included below visualizations for n = 50, 100, 200, 400. Visually, they do have the topology of a real network and theoretically, we feel it is a sensible model because when new switches connect, it is likely to peer with someone who's already connected in the network.



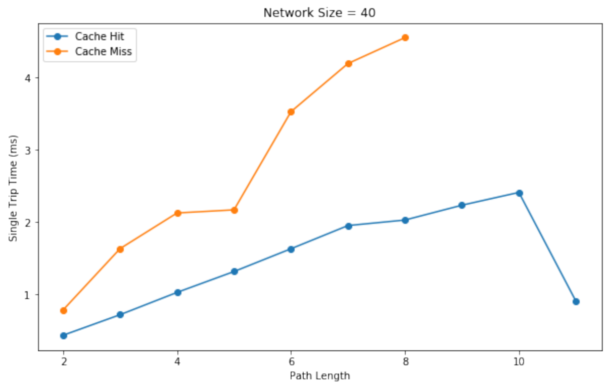
Experiment Set-Up

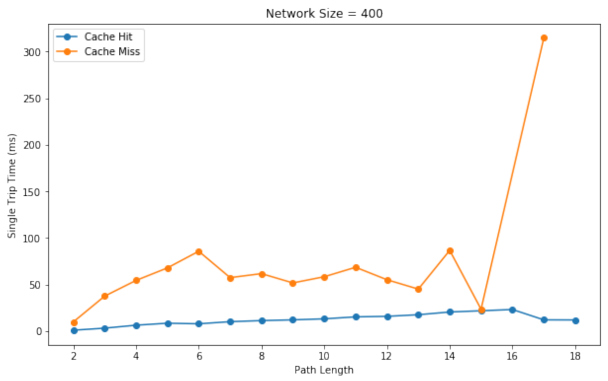
Each switch instance is given a forwarding delay of a normal distribution with a mean of 1 millisecond. We instruct the system to send 1 thousand packets between 1 thousand pairs of two randomly chosen nodes and collect the logs printed by each switch. We then instruct the system to send another 1 thousand packets to the same set of source-destinations to measure the trip time with route cache.

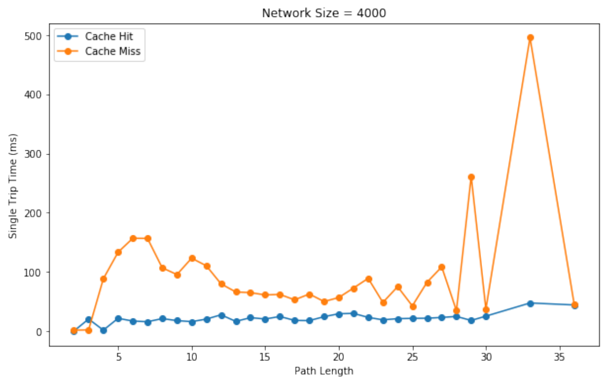
A vanilla Linux installation is not capable of having more than 1024 concurrent TCP connections per process due to system limitations. To accommodate a large number of concurrent connections, we have to make the following change:

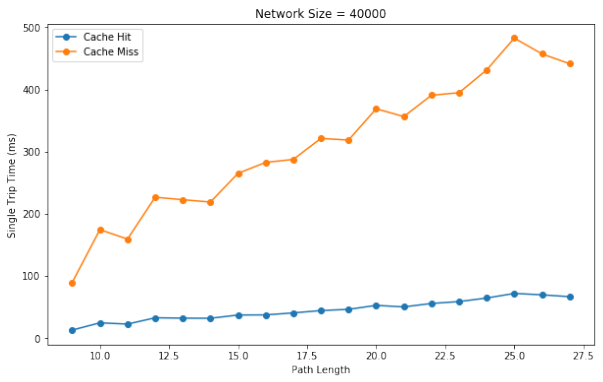
1. File Descriptors: We first increased the kernel max limit to 2 million by executing "echo 20000000 > /proc/sys/fs/nr_open" and then in the current shell, changed the per process file descriptor limit to the maximum value via "ulimit -n 20000000"
2. Maximum TCP pages: We increased the maximum number of pages that could be allocated for TCP connections. By default, it is around 3 GB, we had to increase this limit by executing "sysctl -w net.ipv4.tcp_mem="383865 511820 2303190"
3. TCP specific settings: We edited the system configuration file at /etc/sysctl.conf to have the following
    a. fs.file-max = 20000000
    b. net.ipv4.ip_conntrack_max = 20000000
    c. net.ipv4.netfilter.ip_conntrack_max = 20000000
4. Buffer Size: the following commands were executed to increase the TCP read and write buffer size
    a. sysctl -w net.ipv4.tcp_rmem="1024 4096 16384"
    b. sysctl -w net.ipv4.tcp_wmem="1024 4096 16384"
    c. sysctl -w net.ipv4.tcp_moderate_rcvbuf="0"

Our Golang programs are compiled into statically linked binary files. They are then executed with sudo privileges on two AWS r5.metal machines, each with 96 logical processors, 768 GB memory and 19,000 Mbps bandwidth.









Results and Analysis

The result shows that the route latency goes up with the path length. This is mainly due to the increase in path computation time. At 40,000 nodes, the single trip time still looks acceptable. It would be more informative to test how the system will perform when it reaches 1 million virtual switches. Another observation is that the cache hit time stayed almost flat with a small positive gradient for all network sizes which highlights that caching is an effective way of speeding up packet forwarding.

We also noticed the unusual spikes when n = 4000 and 400. Our conjecture is that the spikes could have been caused by Go's garbage collector or it could be a very corner case in the graph that demanded a lot of computation time which has happened a few times to us. While our Dijkstra's algorithm works well most of the time, it does take an unusually large amount of time to compute for some routes. It would definitely be interesting to continue investigating what is causing the performance degradation.

Future Work and Discussion

Discrete-time simulation
In our set-up, we used a simple delay for the network switches and measured the single-trip latency. This approach, however, can be very inaccurate. It might overstate the actual latency by a significant amount because context switch, go routine switch, driver, and network overhead will add to the delay. Using a discrete time step simulation tool will help mitigate this problem and produce better results.

Running for 1 million nodes using real-world topology

Due to limited time and resources, we were unable to run our experiment with 1 million nodes. The next step for us is definitely to deploy more machines and simulate the architecture on more nodes perhaps with real-world data. It would be interesting to see to what extent does our graph generation techniques reflect the real-world network topology and the associated performance differences.

Making tweaks to the shortest graph computation algorithm

While the go program did deliver satisfactory performance with 40 thousand nodes, it might not be able to achieve a computation time below 1 second when there are one million nodes. Our graph computation engine is not as fast as the state-of-the-art graph-tool and in our future research, we will build on the c-library graph-tool and investigate other techniques to speed up the shortest path search. For example, a real-world switch network might also have some unbalanced traffic patterns. There might be a few switch-to-switch highways that efficiently connect big groups of top-tier switches and the rest are just like the country-side roads which do not usually carry a lot of traffic. Contraction hierarchies may be applied here to drastically speed up the shortest path search.

References

1. Jing, Xiangpeng, and Dipankar Raychaudhuri. "Global control plane architecture for cognitive radio networks." 2007 IEEE International Conference on Communications. IEEE, 2007.
2. Internet Society Global Internet Report, https://future.internetsociety.org/2019/wp-content/uploads/sites/2/2019/04/InternetSociety-GlobalInternetReport-ConsolidationintheInternetEconomy.pdf
3. Global Connected and IoT Device Forecast Update, https://www.strategyanalytics.com/access-services/devices/connected-home/consumer-electronics/reports/report-detail/global-connected-and-iot-device-forecast-updateds
4. Maymounkov, Petar, and David Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric." International Workshop on Peer-to-Peer Systems. Springer, Berlin, Heidelberg, 2002.
5. Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.
6. Ganesan, Prasanna, Krishna Gummadi, and Hector Garcia-Molina. "Canon in G major: designing DHTs with hierarchical structure." 24th International Conference on Distributed Computing Systems, 2004. Proceedings.. IEEE, 2004.
7. Caesar, Matthew, et al. "ROFL: routing on flat labels." ACM SIGCOMM Computer Communication Review 36.4 (2006): 363-374.
8. Caesar, Matthew, et al. "Virtual ring routing: network routing inspired by DHTs." ACM SIGCOMM computer communication review. Vol. 36. No. 4. ACM, 2006.
9. Lu, Guor-Huar, et al. "Virtual id routing: a scalable routing framework with support for mobility and routing efficiency." Proceedings of the 3rd

international workshop on Mobility in the evolving internet architecture. ACM, 2008.
10. Aviram, Nimrod, and Yuval Shavitt. "Optimizing Dijkstra for real-world performance." arXiv preprint arXiv:1505.05033 (2015).