TAALK: Server-Load Aware Network Load Balancing

Alaleh Azhir, Tony Yang, Lucy Zhang, Kristin Yim, and Angelo Olcese

***Abstract*—TAALK is a system designed to improve server load balancing by considering the endpoint servers' load when making initial routing decisions. Previous efforts in this space, such as consistent hashing methods, do not consider server load when making these decisions. In order to implement our server-load aware solution, we implemented a sampling algorithm which uses two hash functions to choose two different endpoint servers. The rest of the packets in the flow are routed to the endpoint server that responds back first. By utilizing this algorithm, we are able to better distribute the load between endpoints. To test the effectiveness of TAALK, we created a Cypherpath topology and contrasted it with Maglev, Google's architecture. We evaluate both implementations using individual flow and job completion time. Our results suggest that while Maglev does better on average when measuring individual flow's completion times, for applications that require looking at all flows' completion times to measure the job completion time, our application is up to 30 times faster under high loads. We suggest further exploration using larger scale topologies and higher traffic to look at the scalability of our solution. We believe that in larger topologies, our solution would show higher speedups.**

I. Introduction

With the immense growth of cloud computing, the demand for large scale multi-tenant cloud environments has soared. Popular services such as Gmail receive millions of queries a second from all over the world which puts an incredible stress on the underlying datacenter infrastructure [1]. To deal with this magnitude of load, these services are hosted across many servers in different clusters throughout the globe [1]. It is important that within these clusters, traffic is spread evenly to endpoint servers such that no single server is overloaded. The key to achieving this is the multi-tenant load balancers which decide how traffic is routed. From current estimates, these load balancers are involved with nearly all external traffic and half of internal datacenter traffic [2], so their importance is evident.

Network load balancers typically sit between routers and endpoint servers, and route each packet to an endpoint server and then forward it accordingly, as shown in Figure 1. These load balancers are usually dedicated hardware devices [3] [4]. However, this approach has some downsides. Namely, dedicated hardware devices make scaling extremely difficult; as a service grows, new devices need to be installed. A better approach, is to implement these load balancers as a distributed software system running on commodity nodes in a datacenter as done by Google's Maglev. This allows for fast scaling as we can introduce new nodes very easily and for a low cost, and we have full control over the system to edit and test configurations [1]. Along with the great potential that we are afforded by software based load balancers comes a great complexity of implementation. One of the biggest challenges to address in the configuration is the idea of connection consistency [1], which means that packets from the same flow must always be routed to the same endpoint server.

Alaleh Azhir is with Biomedical Engineering, Johns Hopkins University
Tony Yang, Lucy Zhang, Kristin Yim, Angelo Olcese are with Computer Science, Johns Hopkins University

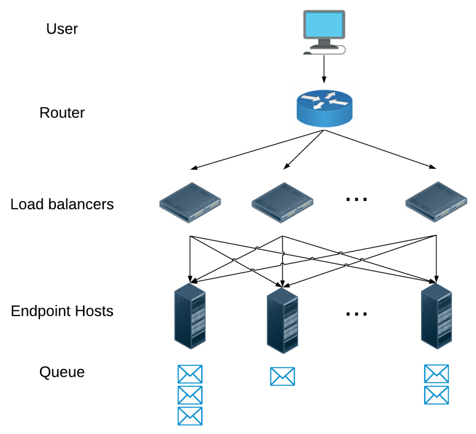

Fig. 1. Datacenter packet flow: Some endpoint hosts may have higher load than others.

Previously, load balancing solutions have worked to spread traffic equally to all available endpoint servers using hash functions and ECMP without measuring each server's load. While this helps to ensure that each endpoint server receives new flows with relatively similar frequency, it does not account for the magnitude of load associated with a flow. Some flows are longer than others, requiring more processing from the endpoint server. Or simply, some flows will require more work from the endpoint server depending on the nature of a request.

TAALK presents a novel solution to software network load balancing which is server load-aware. By building on previous solutions Maglev [1] and Ananta [2], we have implemented a load balancer which significantly decreases the chance of routing flows to endpoint servers which is already under heavy load. Our solution seeks to satisfy four properties: per-flow consistency, low tail latency, low network overhead, and low load on balancers. We utilize two hash functions to route Syn packets to two endpoint servers, and then based on whichever server responds first, the rest of the packets in that flow are routed to that server. This ensures per-flow consistency as all future packets of the flow are sent to the same endpoint server. By minimizing utilization of slow, over-utilized endpoints and picking the faster endpoint server, we seek to minimize tail latency. Additionally, the proposed solution increases the network overhead by adding few additional syn+ack packets from the rejected endpoint server to the user. Once a load balancer has decided which endpoint to send a flow to, we minimize the load on the balancers by bypassing the balancers when sending packets from endpoint servers to the user by routing directly from the endpoint server to the router (an optimization adopted from Maglev as shown in Figure 2). Each load balancer will have the same two hash functions and has the ability to compute to which endpoint server the flows should be routed. Therefore, adding or removing a load balancer does not affect the rest of the load balancers.

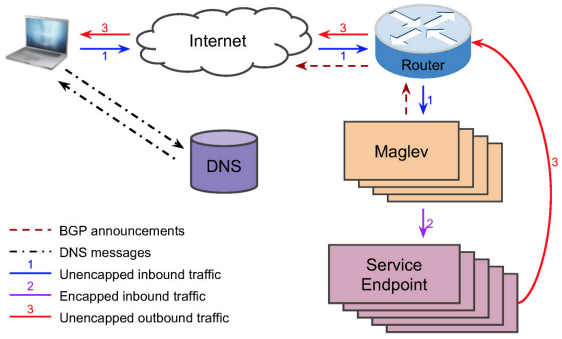

Fig. 2. Maglev packet flow with balancer bypass [1]

We simulated this design and compared it to a Maglev implementation on the same network topology. We believe that, as the system is scaled up, the benefit of server-load aware balancing will greatly outweigh the latency introduced by the extra load on the network. Furthermore, we observed that the job completion time (measured by the maximum flow completion time under each load) was up to 8 times faster in our solution compared to Maglev.

II. Related Works

Current load balancers tend to use consistent hashing, and fail to take into account a server's current load before sending packets. In Maglev [1], endpoints calculate a list of the hash table entries they would prefer, and each endpoint takes turns picking their top choices until the hash table is full. This method allows a datacenter to have multiple active load balancers, instead of an active-passive pair. In addition, this system makes it easy to add, remove, and restart balancers.

In Ananta's architecture, there are three main components, Ananta Manager, Multiplexers, and Host Agents, as shown in Figure 3 [2] . The Ananta Manager coordinates state across Agents and Multiplexers. The Multiplexers are a scalable set of dedicated servers for load balancing. Host Agents are co-located within destination servers, and they provide stateful NAT functionality. Like Maglev, Multiplexers in Ananta also use consistent hashing to forward incoming packets to servers. Multiplexers also keep track of their top-talkers - VIPs with the highest rate of packets. Once a Multiplexer detects that there is packet drop due to overload, it informs the Ananta Manager about both the packet drop and the current top-talkers (endpoints with the highest packet rates). The Ananta Manager then withdraws the top-talker VIP from all Multiplexers, creating a black hole for the VIP. This ensures there is minimal collateral damage due to Multiplexer overload.

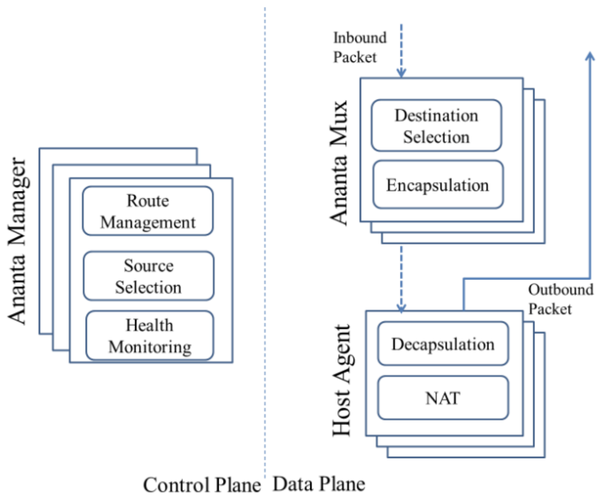

Fig. 3. The Ananta Architecture [2]

Dean et al. discusses the idea of tail latency and motivates load-aware load balancing [5]. Tail latency occurs when a job runs in a data center with low latency most of the time, but with high latency a rare percentage of the time. Despite happening at low probability, this latency impacts the usability of interactive tasks. One of the solutions that the paper proposes is "hedge requests," and this is similar to the design we propose in this paper in that it sends multiples requests. A client sends a request to what it believes to be the best replica, and then falls back on a second request after some delay. Then, the client can cancel requests after the first result is received. Rather than sending the second request after some delay, we send multiple requests to replicas, and choose one when establishing the connection.

III. Design and Methods

A. Initial Design: Δt

Our first design idea was to use packet processing time to estimate the load of each endpoint server. At initialization, we assume all servers have Δt = 0. Then, for each new packet that comes to the load balancer, we first check our match/action table to see if we have already seen other packets of the same flow. If the packet is the first of a new flow, we pick the server with the smallest Δt value (or randomly between the servers sharing the smaller Δt value) to send the packet to. As soon as a server is designated for the first packet of a flow, we send the information regarding the packet header and its designated server to the control plane. We require the packets to be returned from servers back to load balancers before traveling to the user. When a packet gets returned from server, we update the Δt value of that particular server by subtracting the sent time from the received time of that packet. Moreover, the load balancers will not be sharing the Δt values as that can introduce latency, however, we expect that the Δt values will be similar across all load balancers.

Also, Δt may not be an accurate measurement of each server's load. In Figure 4, we see that this technique does not accurately represent the load. For example, consider the 100th packet to enter an endpoint's queue and no packets come afterwards. The endpoint would tell the load balancer that it has high load, which was true at the time when the packet came in, but is no longer true once the packet is processed. The Δt model does not use consistent hashing, hence introducing overhead as a centralized control layer needs to keep track of active flows.

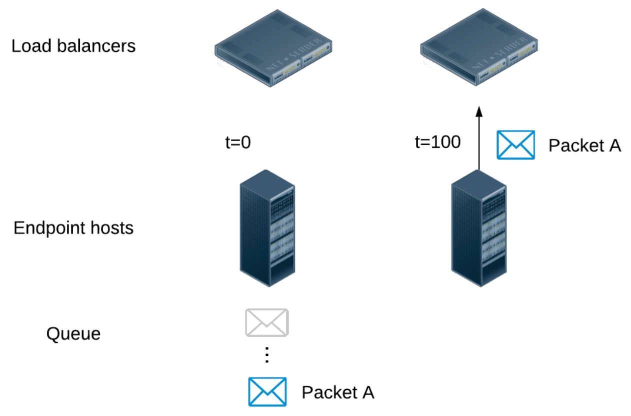

Fig. 4. Inaccurate representation of server load using Δt. The load balancer will think that this endpoint server has high load based on packet A's Δt even though its queue is empty.

B. Final Design: Sampling

Instead of implementing the Δt method, we used an alternative sampling method. We used servers instead of switches for our load balancers. Instead of consistent hashing in load balancers as proposed by Maglev, our load balancers are each equipped with 2 different hash functions. Upon receiving the first packet of a flow, our load balancer duplicates the packet and sends it to both endpoint servers (each corresponding to 1 hash function). Each server is also equipped with the 2 different hash functions. Upon receiving the first packet each endpoint server hashes the header to figure out whether it is corresponding to the first or second hash function. Upon finishing the computation, we could use a reserved bit [6] in the packet header which will get overwritten for the first packet by the endpoint server to indicate whether to use the first or second hash function for the future packets of the same flow (this choice was changed to using timestamp option of TCP during implementation). This ensures consistency. Although this method will have an overhead (by duplicating all syn packets and sending the additional Syn+Ack from the slower endpoint server), we predict that this congestion will be negligible.

One design change we made during implementation was the use of the reserved bit. We initially wanted to overwrite the reserved bit in the packet header to indicate which hash function to use for future packets of the same flow. However, in order for our design to scale, we would have to rely on the assumption that the user does not ever modify the reserved bit. We realized that this assumption is not always realistic, and we cannot guarantee what the user does. Therefore, we decided to use the Timestamp option [6], [7] in the TCP header instead. In our final design, we chose two prime numbers, 65537 and 65539, and each corresponds to one of our hash functions. After an endpoint server computes which hash function was used, it will increase the Timestamp on the packet header to a multiple of whichever prime number corresponds to the hash function used. We chose these two prime numbers because the first number that is a multiple of both is too large to fit in the Timestamp option.

IV. Implementation

Code repository: https://github.com/cloud-sp19/Project/

A. Initial Implementation

Initially, we looked at a few different options for implementation of a simulation: Mininet [8], Omnet++ [9], and the INET [10] framework. Mininet was the most intuitive to use, but because it is geared towards SDN, we would have to create a controller for the network and implement the load balancers as OpenFlow switches instead of servers, which is different from Maglev and our design.

Using Omnet++ and INET, we were able to implement a fattree structure, as shown in Figure 5, that could be parameterized to different scale. We were also able to establish a TCP connection between a user and an endpoint and send packets. However, as we started trying to create our own TCP applications, we started having more trouble with connecting to the INET code and being able to access the packets' information. We found that the documentation for Omnet++ was much more clear than the documentation for INET. We were unable to implement Equal Cost Multi Path (ECMP) and Maglev load balancing. At the end we decided to use CypherPath [11] instead, even though we would not be able to simulate our architecture at a very larger scale.

B. Final Implementation

For our final implementation, we used CypherPath to implement both TAALK and Maglev. We used the same topology for both TAALK and Maglev, as shown in Figure 6. Our implementation contains 6 different machines: 1 User, 1 Router, 2 Load Balancers, and 2 Endpoint Hosts.

1) User: The user is responsible for sending traffic to our endpoint servers. It uses multithreading to simultaneously send between 1 to 100 flows. Each flow is identical in size and about 6 packets long. Furthermore, the user replicates this process by repeating it 10 times. From here on we refer to x simultaneous flows as load x (used for the figures). The user is also responsible for calculating flow completion time

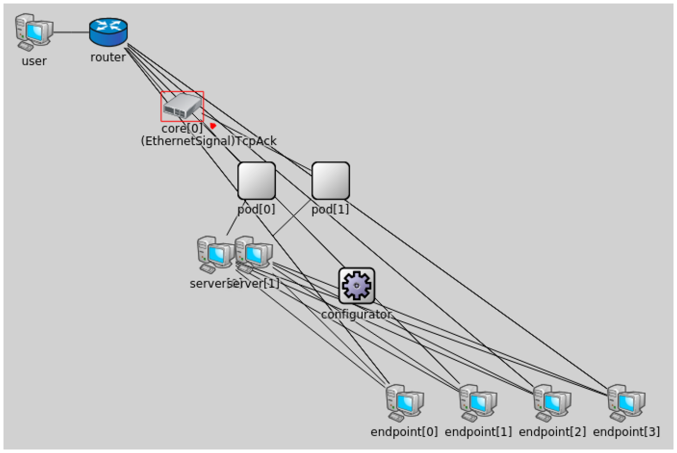

Fig. 5. Fattree implementation in Omnet++ with k=2 and 4 endpoints. The red circle shows a TCP ACK packet.

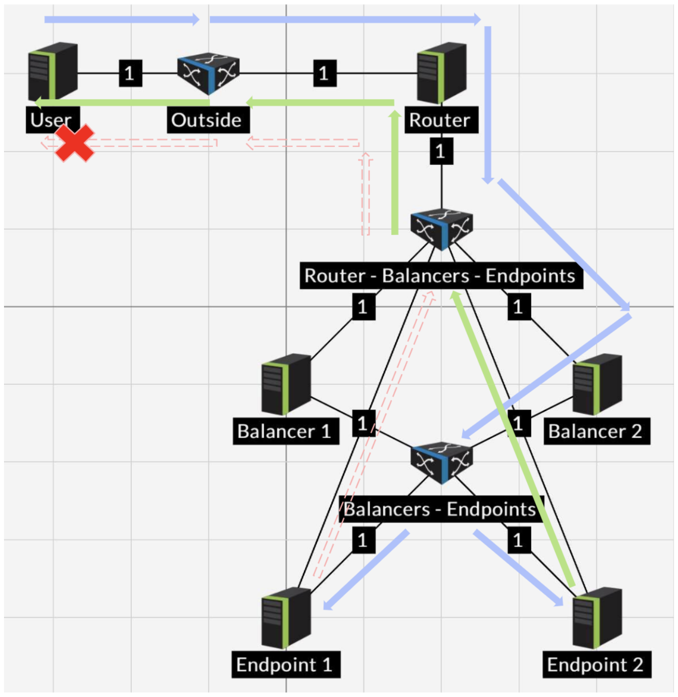

Fig. 6. Topology of our implementation in CypherPath.

for each flow. It computes the difference in time from when a TCP connection is first created and when it is closed. In our implementation, each simulation is repeated ten times.

2) Router: The router has two different parts. First, it must route incoming packets from the user to the load balancers. When it receives a packet from the user, it changes the destination IP of the packet to one of the load balancers randomly to partially simulate ECMP (but we do not use hashing). Second, the router must route packets from the endpoint servers to the user. When it receives these packets, it rewrites the source IP with the router IP to mask the endpoint IPs.

3) Load Balancer: The only part of our architecture that varies between TAALK and Maglev is the Load Balancer. In our Maglev simulation, when the load balancer receives a Syn packet (the first packet of a flow), it picks one endpoint host randomly. When it receives a packet from a flow it has already seen, it uses the timestamp option in TCP to see which endpoint server to send the packet to, ensuring consistency. In our TAALK simulation, when the load balancer receives a Syn packet, it sends the packet to both endpoint hosts. Similar to the Maglev, for all other packets besides the Syn it uses the timestamp option to see where to send the packet.

4) Endpoint Server: The two endpoint hosts in our implementation are DNS servers. When they receive a packet, normally they would resolve the domain name, but here they just send back refusals. Once the endpoint host receives a packet, it increases the timestamp option of TCP to reach a multiple of one of the prime numbers. In this simulation, Endpoint 1 and Endpoint 2 will increase the timestamps to 65537 and 65539 respectively.

V. Evaluation and Discussion

As you can see inFigure 7, the average completion time of our design is shorter for load size between 8 and 25 and longer for the rest. We predict this could be potentially caused by the network overhead due to the additional syn+ack packet sent from the other endpoint host to the user and the duplication of the syn packets. The peak could indicate the congestion due to handling too many flows simultaneously. Moreover, as we expected the flow completion time increases when the load is increased.

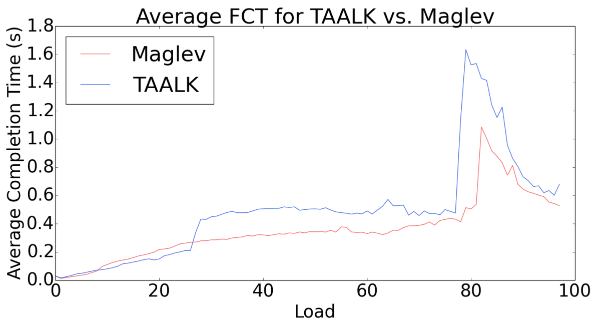

Fig. 7. Average individual flow completion time of our design compared to Maglev

In Figure 8, we can see that Maglev despite having smaller flow completion time has many more outliers than our design, leading to increased tail latency. This means that for applications that require all the flows of a job to be completed, the job completion time would be much higher in Maglev compared to our design. This graph also has its y-axis cut off at 5 in order to make plots more visible, however, Maglev had certain flows that took up to 32 seconds as seen in Figure 9. In Figure 9, we can see that TAALK drops job completion time by a factor of 30 for load 88. Furthermore, TAALK shows much less fluctuation for tail latency compared to Maglev, leading to better predictability.

We graphed the distribution of individual flow completion time for load 82 (Figure 10), 83 (Figure 11) and 93 (Figure 12). We see that for load 82, the distributions are separate indicating that most of the flows in Maglev took a much

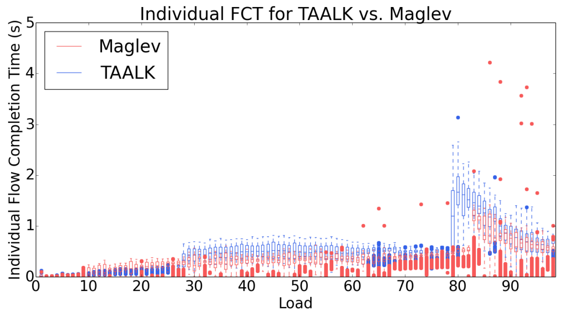

Fig. 8. This shows boxplots of the individual flow completion time at each load. The y-axis has been cut at 5s for better visibility. We see Maglev has many more outliers than TAALK.

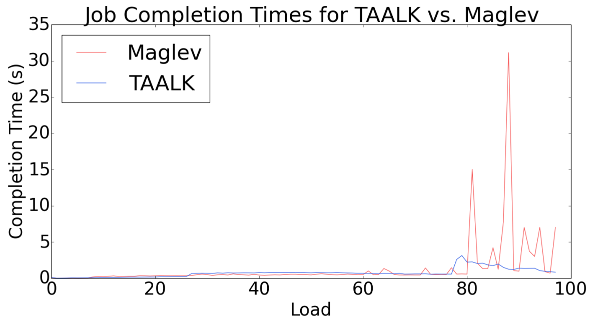

Fig. 9. Maximum individual flow completion time of our design compared to Maglev.

shorter time to complete than they did in TAALK, however, there is a Maglev flow that took ~15s. This shows that its corresponding job completion would still be ~8 times slower than it is in TAALK. Furthermore, since around load 80 we see a peak in average flow completion time of TAALK in Figure 7, we can conclude even when the network is congested, TAALK seems to function better for job completion time. In Figure 11, we see some overlap between TAALK and Maglev. This is an example where Maglev has better performance and completes both flows and jobs faster probably due to its lower network overhead. Lastly, in Figure 12, we see that the distributions overlap, although Maglev again shows high tail latency. At the same time we can see that TAALK has more individual flows that take a longer time to complete (~1s).

VI. Conclusion

We have introduced TAALK, a system where load balancers are aware of endpoint server loads. Load balancers in TAALK send syn packets to two servers (determined by two hash functions). A connection is then established with whichever server sends back an Ack packet first (aka. the faster server with presumably a lower load). We evaluated TAALK by implementing it in CypherPath and comparing flow completion time with Maglev (which picks endpoint servers randomly). Our results show that TAALK and Maglev have similar individual flow completion times for smaller loads. For larger loads, the average individual flow completion time for Maglev is shorter as it benefits from lower network overhead, however

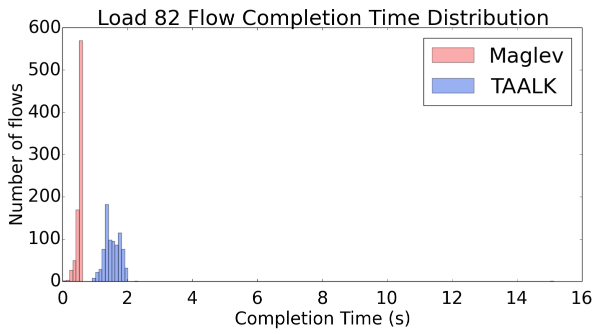

Fig. 10. The individual flow completion time distribution for Maglev and TAALK under load 82. There is a Maglev outlier at ~15s.

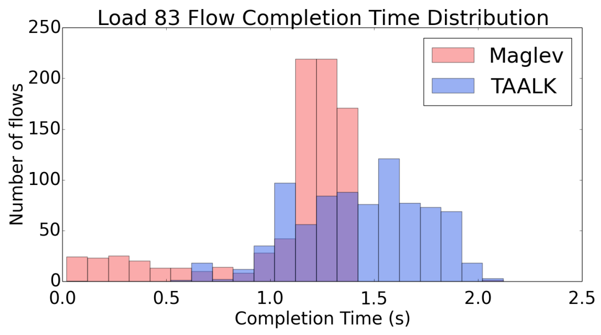

Fig. 11. The individual flow completion time distribution for Maglev and TAALK under load 83.

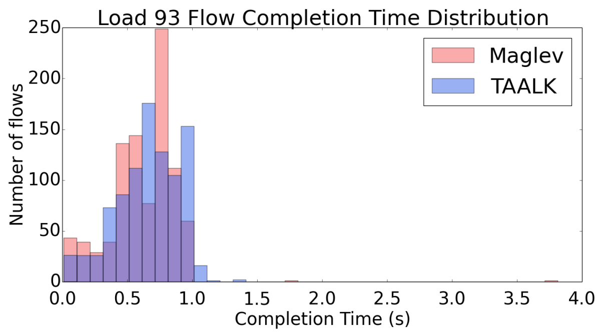

Fig. 12. The individual flow completion time distribution for Maglev and TAALK under load 93.

the tail latency is quite high for Maglev compared to our implementation. Therefore, our solution successfully decreases the job completion time by having up to 30 times smaller tail latency.

VII. Future Work

We suggest running more extensive simulations using CypherPath, with more endpoint servers and changing the traffic by sending flows one at a time as well as simultaneously. We may be able to see higher improvements for TAALK and Maglev on larger scale simulations. Another way we can evaluate our implementation is by looking at the load of endpoint hosts. We would like to graph the loads of each endpoint host to see if the load distribution is more even in TAALK compared to Maglev.

To further improve our design, we can have the two endpoint servers that are picked communicate with each other. For example, if the load balancer sends a syn packet to server i and server j, i and j can send each other their corresponding loads, as shown in Figure 13. The server with a higher load can simply drop the syn packet. In this design, the endpoint servers also have to know the two hash functions so that when they receive a packet, they can compute which other server received the same packet. This design increases the computation necessity of the endpoint servers, however may decrease the network load overhead as it will stop the other endpoint from continuously sending syn+ack to the user.

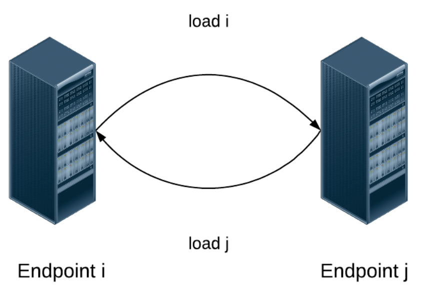

Fig. 13. Endpoint server i and endpoint server j sending their respective loads to each other.

References

[1] D. E. Eisenbud, *Maglev: A fast and reliable software network load balancer.*, 2016.
[2] P. Patel, "Ananta: Cloud scale load balancing." *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, 2013.
[3] "A10 Networks AX Series." [Online]. Available: http://www.a10networks.com.
[4] "Kemp." [Online]. Available: http://www.kemptechnologies.com/
[5] J. Dean and L. A. Barroso, "The tail at scale." *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.
[6] M. S. West, *TCP/IP Field Behavior*, 3 1748.
[7] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance." RFC Editor, 1992.
[8] "Mininet," 2019. [Online]. Available: http://mininet.org/
[9] "OMNeT++ Discrete Event Simulator," 2019. [Online]. Available: https://omnetpp.org/
[10] "INET Framework," 2019. [Online]. Available: https://inet.omnetpp.org/
[11] "Cypherpath," 2019. [Online]. Available: https://www.cypherpath.com/.