ScaledFastPass: A Scalable, Centralized Arbiter For Zero-Queue Data Centers, Beyond FastPass

Eugine Szeto  Eric Yu  Advisor: Richard Yang

Yale University

Abstract

Data centers are becoming more and more important in a cloud-connected, internet-based world. Facebook serves over 1 billion daily active users per day, and most of the photos, videos, and media content is hosted on internal servers in data centers all throughout the world. There have been many attempts to increase the efficiency of data center networks to improve performance. Hedera is an example of a dynamic flow scheduling system to improve load-balancing in multiple data center topologies [1].

In this paper, we first introduce Fastpass [2]. Fastpass is a data center network architecture that features a centralized arbiter, that uses fine-grained control to direct each packet through a proper path in the network at an assigned timeslot \( a \) to avoid queueing delays. However, there are a number of concerns over the centralized arbiter approach. Firstly, there are concerns over the ability to parallelize the pipeline process. Secondly, FastPass makes requirements about the topology of the data center. Lastly, its easy for a single arbiter to become a computational bottleneck in a network of hundreds of thousands of nodes, because FastPass uses an edge coloring algorithm to assign paths for all traffic within a data center. We propose a system for scaling Fastpass by distributing computational work between multiple arbiters. We partition the network so that local arbiters handle only a subset of network requests while a single global arbiter can coordinate traffic in between the different subsets. We show that FastPass is able to handle more demand and more endpoints under this system.

1 Introduction

1.1 The Problem

As the number of Internet users continues to increase, the demand for cloud service will also grow. A data center is the solution that many cloud providers use. However, data centers have high utilization, and they need to fairly allocate network resources among many users and applications. Currently, data centers use the methodologies of the Internet. Traffic is directed by routers and switches as requests and data packets arrive. As a result, queues can build up in switches that receive a high volume of traffic while other switches are under-utilized. These highly trafficked switches can become sources of bottlenecks, introducing latency and single points of failure. This is unacceptable, especially for services like Facebook or Twitter, that depend on serving real-time information to their users [3]. At the same time, large queues in a data center can even cause outages. In a recent report by the Uptime Institute, researchers found that network issues accounted for over a fourth of all services outages between 2016 and 2018 [4]. Clearly, there is a large need for more performant, fault-tolerant data center network architectures.

1.2 Fastpass

FastPass is a centralized control system where a single arbiter allocates timeslots and paths for all data packets within a data center. The goal of FastPass is to address the aforementioned problem by creating a network with the goals of (1) packets experience no queueing delays, (2) the network achieves high utilization, and (3) the network is able to support multiple resource allocation objectives between flows, applications or users.

Fastpass accomplishes the above by introducing the two algorithms for the single arbiter. It utilizes a timeslot allocation algorithm that can achieve max-min fairness or approximate minimum flow completion times. Furthermore, it uses an edge-coloring algorithm over a bipartite graph to assign paths for the data packets to take. The single arbiter is connected to all other endpoints in the network, and the endpoints kernel is modified such that each endpoint communicates with the arbiter before
sending any packets. This modification to the kernels must include an ability to queue packets that will be sent according to the timeslot and path that the arbiter determines.

The endpoint is very simple - before sending out any packet at the NIC, the packets are first sent over to the Fastpass arbiter as demands, indicating where the endpoint wants to send data to and how large the payload is. The endpoint then waits for the arbiter to respond with a precise timeslot to send the packet, and a viable path through the network it can take to avoid queueing delays at every switch. In the end, queueing at switches is inevitable because of inaccuracies in timing using programs, but it is minimal compared to the default TCP methods.

The Fastpass arbiter does three things - 1.) it communicates with endpoints and receives/sends demands back and forth, 2.) it processes demands and allocates precise times for certain demands to be started, and 3.) it computes a path through the entire network for each packet so packets do not run into the same switch and cause queueing delays. Each one of these actions can be done separately but requires information from previous steps, so they can be split up into different cores. Comm cores, alloc cores, and pathsel cores exist for these three purposes, respectively. The work done by these cores can be parallelized in certain conditions, an examples flow being shown in the figure below:

1.3 Issues with Fastpass

There are three problems that we have identified in Fastpass.

1. The first issue is the pipeline process of assigning timeslots that the authors claim will allow multiple cores to parallelize the workload. Since assigning timeslots require information from the previous allocation, it is not clear how the authors are able to parallelize this process. At the very least, it requires \( k \log N \) time to insert the \( k \) newly allocated timeslots from a previous allocation into the list of current timeslot allocation \( (N) \).

2. The second issue is that FastPass requires that tiers in the topology are rearrangeably non-blocking. This requires that endpoints are interconnected in such a way that any unused input-output pair can be connected by a path through unused nodes, regardless of what other paths exist at the time. However, this is difficult to accomplish, because flat topologies can be very difficult to scale with a centralized traffic engineering approach, and link failures can occur, requiring constant upkeep. [5]

3. The third issue, which is the focus of our paper, is scalability.

Because there is only one arbiter that centralizes the control of traffic within a data center, there is a question of what happens when there is a bottleneck? The bottleneck can be caused by computational demands as a result of allocating paths and timeslots for a growing network. Another possible bottleneck is from network demands; requests can fill up at the arbiter more quickly than the arbiter can process them. The original authors briefly discuss two strategies for overcoming this challenge. One suggestion is to use specialized hardware like an FPGA or ASIC. The other suggestion is to use a hierarchical solution involving local and core arbiters. We chose to explore the hierarchical solution, because a hierarchical solution has the potential to continue scaling, which allows for larger data centers or more trafficked data centers. In contrast, a specialized hardware implementation will still have an upper limit for how large the data center can become and how many requests can be processed at once by a single arbiter.

1.4 Our Goal

We aim to scale FastPass so that the system can be deployed in larger data centers where the bottlenecks would otherwise prevent the system from processing larger demands. In a large data center with many endpoints and a complex network topology, many things could become a bottleneck. If the arbiter itself is barraged by requests from all the endpoints, network could be a bottleneck and there could be significant queuing at the arbiter itself. Otherwise, computation can be a bottleneck. If there are many demands, the timeslot allocation and path selection algorithms could take a very long time. In this paper, we focus on the computational bottleneck that could occur when the arbiter is exposed to many demands and endpoints. We believe that multiple arbiters must be used to reduce the burden on a single arbiter. By using multiple arbiters, we try to distribute the work amongst the arbiters equally so that they can finish faster than in a single arbiter scenario. In the present work, we present a proof of concept that this centralized system can indeed scale as described by the original FastPass paper.

2 Architecture

Our architecture for scaling Fastpass focuses on the arbiter. To isolate the arbiter and run simulations that tested the arbiters ability to handle large amounts of demands from many endpoints, we implemented our own arbiter in Python, using the original authors code as an inspiration. The simulation consists of two parts: our custom
arbiter and a generator that randomly generates demands for the arbiter to handle.

In order to run the simulation, we needed to decide on a sample data center topology. Again, we wanted to focus on the computational scalability of the arbiter, so we chose a simple topology - endpoints are connected via switches, which connect every endpoint to all the others. It resembles the top layer of fat tree network architectures, which is typically found in data centers. Our simulation shows that this topology works well.

### 2.1 Timeslot Allocation

One of the points we want to validate in the original paper is regarding the pipelining problem during timeslot allocation. In our base implementation, we don't split the arbiter into multiple cores, and so there is no parallelization of the timeslot allocation cores. However, we can show that some amount of parallelized pipelining of the allocation cores is possible. If our arbiter is optimizing max-min fairness, we want to sort all demands by last allocated timeslot, and allocate demands that have not been allocated in a long time. Imagine allocator for timeslot 57 receives demands with last allocated timeslots, 44, 45, 46, 51, 53, 57 (not necessarily in order). It must sort these demands, but it can go ahead and pass all the demands to the next allocator in the pipeline, for timeslot 58. The allocator for timeslot 58 can go ahead and sort the demands, but it must realize that some demands in the previous allocator will have their last allocated timeslot value changed to 57, so it cannot process the demands with last allocated timeslots equal to 57 just yet - it must wait until the previous allocator finishes its demands to know the whole picture. However, it can process (sort and parse) all demands with last allocated timeslots less than 57, which accounts for the majority of demands. In this way, the processing of demands by the timeslot allocation cores can be parallelized.

Like the original FastPass paper, we sort the requests based on previous allocations to achieve max-min fairness. Each source-dest endpoint pair has a previously recorded timeslot allocation time, and we sort demands from oldest to most recent allocation, so we can allocate the older ones first. Request pairs (start point and end point) that were given priority in an earlier timeslot receive lower priority in future timeslots; they are processed last.

### 2.2 Path Selection

Like the original Fastpass paper, we also perform path selection for allocated packets in timeslots using a bipartite graph edge coloring scheme. The vertices represent the ToR switches, edges represent the allocated packets, and the colors represent the paths. We used the path selection code given in the Fastpass source code as inspiration for our own version in Python. We use the Hopcroft-Karp algorithm for bipartite graph matching to repeatedly compute colorings for the path selection graph.

### 2.3 Multiple Arbiters

The biggest difference between our architecture and the original Fastpass architecture is that ours uses multiple arbiters. We do this to distribute the workload of one arbiter to several, so that we can scale the network as the number of endpoints becomes large or the number of demands increases rapidly. In our architecture, there are two types of arbiters: local and global.

Local arbiters control their own partition of the network. Given our topology defined above, we split the network into two halves if there are two arbiters, three if there are three, etc. All the endpoints in the local arbiters subnetwork are connected directly to the arbiter, and they send all their traffic to their local arbiter. The local arbiter is responsible for doing timeslot allocation and path selection for packets that stay within its subnetwork (intra-domain traffic). For packets that must traverse across subnetworks, the local arbiter cedes control to the global arbiter.

To this end, we designed and implemented two systems of multi-arbiters, and tested the performance of these two systems with our simulation of the original Fastpass system with a single arbiter. Our first system is a naive multi-arbiter, which uses a naive method of having multiple arbiters take in separate demands from separate partitions of the overall network, computing timeslots and paths, and then resolving conflicting allocations using a global arbiter. Our second system avoids the problem of resolving conflicting allocations by separating batches of time slots in global and local sections.

### 2.4 naive Multi-arbiter System

Our naive system is composed of two components - local arbiter and global arbiter. First, we partition the network. In this paper, for simplicity, we consider only a single-layer network where switches connect every endpoint to each other. The local arbiters, corresponding to different partitions, take in demands that originate from their partitions. Each local arbiter has complete knowledge of the network topology, and computes in parallel time slot allocations and path selections for each of their demands. However, this naive approach creates an important issue - that each local arbiter may assign time slots that conflict or do not meet bandwidth requirements. Thus, there must be an arbiter capable of resolving each time slot, choosing a set of allocations and path selections from the
ones the local arbiters have produced. This is the global arbiter, which is connected to each of the local arbiters. The global arbiter takes in the allocations processed by the local arbiters, and greedily chooses a set that works globally for a single time slot. It then broadcasts the assignment to all the endpoints. The global arbiter continues to resolve allocations and broadcasts it to the entire network.

2.5 Batched Multi-arbiter System

In the batched multi-arbiter system, we still have the concept of local and global arbiters. Similar to the previous method, the network is partitioned, and each local arbiter is responsible for a partition of the network. This time, each local arbiter is completely responsible for all the traffic within its partition. Given a set of demands, it knows which ones complete within its domain and which ones must cross into the other network partitions. It allocates time slots and performs path selection for all of the demands within its partition. But what about the traffic that goes across domains? Each time it encounters a demand that is inter-domain, it will send it to the global arbiter. The global arbiter is responsible for receiving all inter-domain demands and processing them. However, another problem arises - how can the local arbiter that handles intra-domain demands and global arbiter that handles inter-domain demands avoid conflicts? We solve this problem by creating the concept of batches. A batch is a chunk of time slots, perhaps 50 or 100. Local arbiters schedule intra-domain packets for one batch (time slots 1-50), while the global arbiters schedule inter-domain packets for the next batch (51-100). By doing this, we can do parallel computation of demands without conflict. The local arbiters broadcast their batch assignments directly to their partition endpoints, and the global arbiter broadcasts its batch assignments to the entire network.

2.6 Inter-domain Traffic

When a local arbiter encounters a request pair where the destination is outside the local domain, the local arbiter forwards the request to the global arbiter, which processes that request in the subsequent timeslot. Once the global arbiter creates a path assignment for all the traffic in its queue, the path assignments are passed onto the respective local arbiters. The local arbiters take into account the path assignments made by the global arbiter as they continue to allocate paths for the intra-domain traffic in the next timeslot that occurs after the global arbiter finishes up with its assignments. The cycle begins anew when the local arbiters begin assignment once again, with inter-domain traffic being forwarded to the global arbiter for subsequent assignment.

3 Implementation

In our implementation using Python 3.6, the requests in a data center are represented by randomly generated pairs of numbers. The first number in the pair represents the source endpoint, and the second number represents the destination endpoint. Each request is assumed to be the same size. We simulated the arbiters inside a 2017 MacBook. Each local arbiter is run on its own thread with a list of demands that originate in its own domain that were generated before the start of the arbiter thread. The global arbiter is also on its own thread. Instead of simulating network communication, we use a queue. One queue is for local arbiters to forward requests to the global arbiter. The other queue is for the global arbiter to return allocated paths to the local arbiters.

Within an arbiter, the list of demands is processed. First, the demands are sorted as previously described to accomplish timeslot allocation. Second, each demand is assigned to the current timeslot unless it can no longer be assigned due to conflicting resource demands at an endpoint. Such demands that cannot be assigned are left for future allocations along with any demands that were unable to be processed due to the limiting number of endpoints. The demands that are allocated in the current timeslot are assigned a path as described above.

3.1 Local Arbiter

All demands are initially processed in the local arbiter. A local arbiter will only receive demands that originate within its domain.

In the naive multi-arbiter system, local arbiters process demands taking into account the entire network topology, essentially ignoring which demands are intra-domain and which ones are inter-domain. It passes these assignments via a Python thread-safe queue to the global arbiter (with a network delay simulated), which then resolves conflicts from the other local arbiters. The local arbiter must know which time slots were actually selected by the global ar-
biter, so it receives back that information in the form of a dictionary. The dictionary contains updated information regarding when each allocation was last assigned to a time slot, so it can use this data in its own processing in the future.

In the batched multi-arbiter system, local arbiters process demands ONLY that start and end inside its domain. Otherwise, it passes them via a Python thread-safe queue to the global arbiter. It schedules fifty timeslots at a time, then assumes that inter-domain traffic will take place in the next fifty time slots, and so it continues to schedule time slots after the fifty global ones.

The local arbiter and the global arbiter communicate via a Python thread-safe queue (since they run on different threads to simulate parallelism) that has a built-in delay to simulate network delay.

### 3.2 Global Arbiter

The global arbiter begins the process of assigning timeslots when it receives the start signal from all the local arbiters. Then, it will make timeslot allocations for the inter-domain demands that have been passed up from the local arbiters.

In the naive multi-arbiter system, the global arbiter receives time slot allocations from multiple local arbiters that attempt to schedule demands across the whole network. It iterates through the list of demands received and greedily chooses a maximal set of allocations that work in a single timeslot. It then forwards the information of which time slots it chose back to the local arbiters, which must then incorporate this knowledge during their scheduling.

In the batched multi-arbiter system, the global arbiter receives only inter-domain demands from the local arbiters. It assumes that every other batch of time slots is for inter-domain traffic, so it can process demands even while the local arbiters are processing the intra-domain demands.

### 4 Results

We used a late 2015 iMac with a 3.2 GHz i5 Intel core chip and 16 GB 1867 MHz DDR3 RAM to conduct our experiments. We find that overall, our system performs faster than a single arbiter, and it is able to handle more traffic and more endpoints and switches in our simulated data center environment.

We conducted tests by varying the number of endpoints, switches, and demands in our naive network topology. We created graphs depicting how each of our implementations scale with these variables:

As we demonstrate in the results, the naive implementation performed the worst across every possible metric. We believe this happens because most of the computational bottleneck of Fastpass occurs during the path selection process. The naive implementation does extra path selection work compared to the regular Fastpass implementation, explaining the huge decrease in performance. The scaled Fastpass implementation did perform better than the original implementation, showing that splitting up the computation is a possible scaling mechanism.

### 5 Discussion

As mentioned in an earlier section, we are concerned about the pipelining process of FastPass. Our requests are randomly generated and sorted before the arbiter individually assigns the timeslots. Since our data center was simulated on a single computer, we did not have
the physical equipment to adequately replicate problems with network latency issues.

6 Future Directions

In the future, we want to explore other ways to scale Fastpass to larger networks. In this paper, we focused on computational bottleneck. We want to simulate an actual network so that we can assess how big of a problem network bottleneck can be for an arbiter. Even if the topology is simple and the number of endpoints are low, if there are many demands, will the arbiter be able to receive all of them from its communication cores without queueing?

We would also like to examine faster algorithms for doing timeslot allocation and path selection. In our implementation, we did these sequentially in order to highlight computational overhead in our simulation. However, parallelizing these functions and finding new ways to reduce work might help point us to an ideal number of arbiters to use in a multi-arbiter system. Currently, it is a trial and error process, but we may be able to compute it one day.

Finally, we would like to investigate the effect of complicated network topologies on scaling Fastpass. In this paper, we used an extremely simple topology - endpoints connected by switches that connect every endpoint together. But, in a multi-layer or irregular topology it will be interesting to see how that affects path selection. Path selection algorithms tend to be computational expensive, so it may be the case that complicated topologies are ill-suited to Fastpass in the first place.

7 Conclusion

In this paper, we have found that our multi-arbiter system for scaling Fastpass does indeed improve the performance for arbiters to schedule traffic in a crowded network. We have seen that adding many endpoints and demands slows down a single arbiter's performance, creating a computational bottleneck. Our architecture addresses this problem, and manages to split up a single arbiter's work effectively across multiple arbiters to alleviate this computational overhead and reduce delay, enabling a data center using this system to grow in size and traffic.

8 Distribution of Work

In this project, we worked together to deconstruct the Fastpass algorithms and figure out a way to scale it for simple network topologies. Eugene worked mostly on the implementation in Python of the multi-arbiter system, while Eric worked on the initial, simple simulation for a single-arbiter system and validated some of the hypotheses that the Fastpass paper had. We worked together with our advisor to come up with a design for the multi-arbiter system.

9 References