Homework 2
601.419/619 Cloud Computing

Jason Zhang jzhan127
Collaborators: Will Ye, Simon Zeng

April 19, 2020

1 We saw that the additive-increase/multiplicative-decrease (AIMD) algorithm, the simple distributed feedback control algorithm used in almost all congestion control algorithms today, has some key strengths such as simultaneously optimizing fairness and efficiency. Despite inheriting those strengths from AIMD, TCP's congestion control algorithm is not ideal for datacenters and we saw in the Jupiter paper that even for lightly loaded datacenters, managing congestion is still a big open research challenge.

(a) Give a few (at least three) reasons why TCP is not effective for managing congestion in datacenters.
Answer:
1. TCP is unable to distinguish between true congestion events and just one off packet drops. TCP will treat both scenarios the same and reduce the congestion window for any lost packets. This can lead to clear underutilization of network bandwidth, affecting datacenter applications adversely.
2. Queuing delay for latency sensitive flows. TCP is quite the buffer hog, so if suppose two flows, one elephant, one small, are sent to the same switch, the short flow (which is latency sensitive), will have its packets trapped in the queue behind the elephant flow packets. In the context of the data center, an occurrence like this could be detrimental for data center applications with a deadline IE delivering a webpage.
3. TCP timeouts. In cases of extreme congestion, the sender will have to wait an entire TCP timeout in order to discover that there was some form of congestion. This is especially detrimental in the data center since computations across data centers require responses on the order of microseconds. An example of this occurrence would be during a TCP incast collapse. When multiple synchronized senders(happens if a host requests data from multiple servers at once) send to an aggregation switch and overload the buffer, the hosts will have to wait a whole TCP timemout to get retransmissions, bad for time sensitive applications.

(b) Describe at least one remedy in the context of datacenters for improving the performance of TCP.
Answer:
We can use ECN (Explicit Congestion Notifications). This allows for earlier detection of possible buffer overflow to signal the sender to reduce its sending rate. A prominent solution employing this concept is DCTCP. ECN enabled switches upon noticing a near full buffer may mark packets in the IP header going to the receiver, to which the receiver will echo this congestion notification to the sender in the hopes of reducing the sending rate. This ideally tries to avoid the need to drop packets to detect congestion.

(c) Is this remedy consistent with, at odds with, or orthogonal to the end-to-end principle?
Answer:
This remedy is at odds with the end to end principle since the principle states that the end hosts should handle the heavy lifting and application features, the components in between the end hosts only serve as passthrough and shouldn't do any modifications or

calculations other than proper routing. ECN violates the principle since it relies on the switches between the source and destination to action on the packets(marking them) and constantly check their buffers to determine plausible congestion.

2 Suppose we build a cluster with a fat tree topology using 24-port switches, following the topology design of the Al-Fares paper (that you read and reviewed).

(a) How many distinct end-to-end shortest paths are there in the physical topology between a source server A and a destination server B in a different pod? (Here, shortest paths are those that have the minimum number of switches along the path. This question concerns the physical topology, not routing or forwarding.)
Answer:
There would be $(k/2)^2$ shortest paths between two servers in the topology, which computes to 144 shortest paths between A and B.

(b) Suppose the cluster runs BGP, with each switch having its own AS number and each edge switch announcing a distinct attached IP prefix covering all its servers. (No other prefixes are announced.) Switches run standard IP forwarding with ECMP across all shortest paths. How many forwarding rules are in each edge switch? How many are in each core switch? (Each forwarding rule specifics a single egress interface for a set of matching packets.)
Answer:

For each edge switch: First each edge switch will have 12 entries for each of the servers under it. Next, based on BGP, since each aggregate switch in each pod serves as a provider for each of the edge switches, each edge switch will announce to the aggregate switches which will subsequently announce to the other edge switches since they are direct clients. For 12 aggregate servers that means each edge switch has 12 * 11 entries each being a path to other edge switches in the pod. Adding to the 12 we already have giving us 12* 11 + 12 entries just learning routes within a pod subnet, giving us 144 entries. Now to learn routes to switches outside of a pod, we rely on the customer provider relationship between core and aggregate switches. An aggregate switch upon learning a customer route from an edge switch will announce to everyone that customer route. This means the core switch will now know of a customer route to an edge switch. Since other pods are customers of the core switch, the route it just learned will propagate down (because of BGP rules) until it reaches an edge switch. From this example we can see that due to the hierarchical structure of fat tree, routes from each edge switch to other edge switches in the data center will eventually be propagated to each edge server because of the BGP rules. Therefore an edge switch will learn 144 entries from every other pod in the network. Given that there are 23 other pods in our case we effectively have 144 (from own pod) + 23*144 (from other pods) = 3456 forwarding rules in each edge switch.

Core switches: Each core switch will have a forwarding rule for each of the k/2 edge switches in each of the pods. From the BGP logic above, the core switches will learn 12 edge switch routes from each pod. It only learns 12 since there is only 1 path from each core switch to each pod. This then limits each core switches's forwarding table rules to only 24*12 = 288 rules.

(c) Suppose the switch hardware is capable of performing MPLS forwarding, with IP-inMPLS encapsulation, and the hardware can also still perform IP forwarding. More specifically, a forwarding rule on a switch can either (1) perform IP/ECMP forwarding as above, (2) match a prefix of IP packets and direct it into an MPLS tunnel, with ECMP-style hashing if there are multiple matching rules; (3) perform MPLS label swap forwarding; or (4) match an MPLS label, pop the MPLS header and continue forwarding out an interface via IP. Can you use this hardware to deliver data along the same paths as ECMP, but with fewer forwarding rules? If so, describe your solution and how many forwarding rules it needs at each edge switch and at each core switch.
Answer:
Since MPLS does not route based on per hop decisions, we can drastically reduce the

number of forwarding rules at each edge switch. With MPLS we essentially have predetermined routes from each edge switch and upon packet entry into the edge switch, proper label headers are attached which allow receiving switches to properly and quickly identify the switch it needs to forward to (by looking at the labels). The network is initially informed how to forward based on the label it gets since MPLS relies on precomputed paths. For each edge switch since we have predetermined paths between edge switches, we only need edge switches to know and operate on one of the possible paths every time when trying to route to a specific edge switch. This means that each edge switch will only need to have $11 + 23(12)$ forwarding rules from each edge switch to all other edge switches in the network. We also have the 12 servers under us, which gives us a total of $11 + 23\ (12) + 12 = 299$ forwarding rules at the edge switches.

At the core switches we can now have only 24 stored table entries. Given that each core switch only has one connection to a pod, it must take that path to get to that subnet which means we must have at least 24 forwarding rules.

(d) Compare the resilience of your solution with that of ECMP. Specifically, suppose one core switch fails. What plausibly happens within about 1 millisecond, i.e., with only local reaction at each switch, in ECMP and in your solution? (Note: the goal isn't to produce a scheme which is necessarily better or worse than ECMP; the goal is just to compare.)

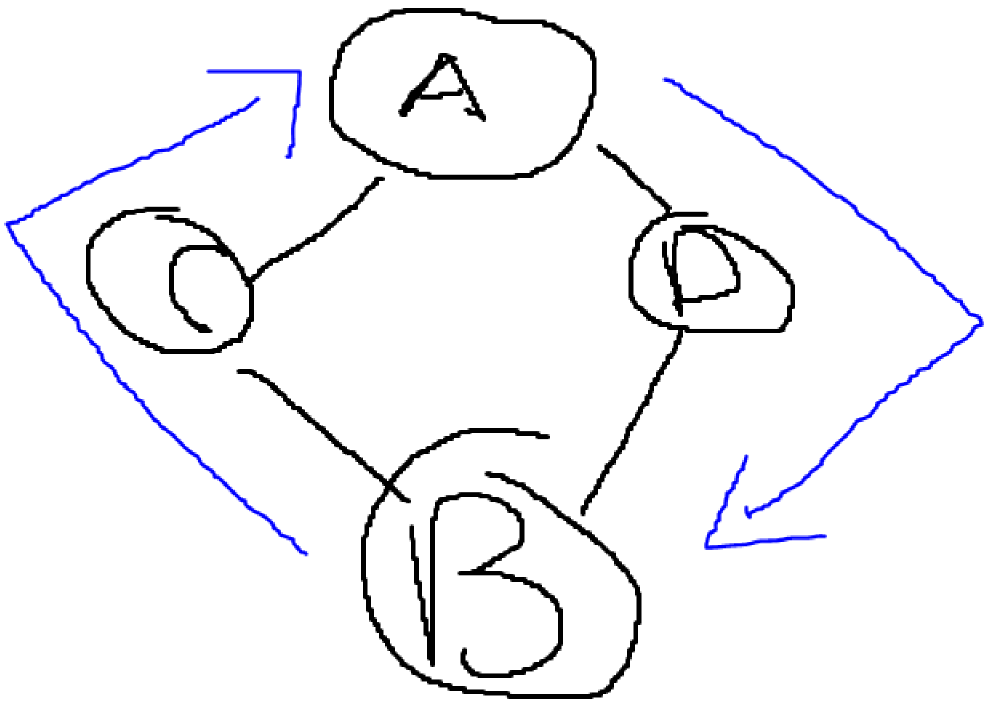
Answer:

In comparison to ECMP my solution is significantly less fault tolerant. In the case of a core switch failure, an aggregate switch will not have a forwarding rule due to its pre determined route being incapacitated. In comparison to a per hop routing, per hop routing would have been able to immediately decide to choose another ECMP vetted path without much strain. With this MPLS solution however, we would have to completely update the network with new predetermined paths which is a lengthy recovery time. We could install backup rules for the MPLS solution, in which in case of failure we resort to an alternate routing, but it does add more forwarding rules for switches to store in memory.

3 Routes between A and B are asymmetric when the A B path is not the same as the B A path. How can asymmetric routing occur (a) inside a datacenter with a topology such as fat-tree that uses ECMP, and (b) between two datacenters even if all networks use BGP with common business relationship policies (prefer customer over peer over provider for route selection, and valley-free export)? Give two examples (one for (a) and one for (b)) of how this could happen.

Answer:

A. With fat tree, asymmetric routing can easily occur with ECMP due to the amount of interconnectivity in the core and aggregation layers due to the pod concept. With multiple good paths to take between two end hosts, ECMP could pick one of many equal cost paths from A to B, and then pick a completely different equal cost path from B to A, thus giving us asymmetric routing.



B.
In this case we can see that in the case of BGP given a scenario like this where there are two least cost paths with only one extra hop (between C or D), BGP will need to resort to tie breaker strategies, since either path is viable. If say AS A prefers the route A-D-B and AS B prefers the route B-C-A, then we have asymmetric routing occurring.

4 BGP routing can be formulated as a game where the selfish players are autonomous systems (ASes). It can be shown that this game has no Nash equilibrium (stable state) in some cases. That is, the control plane will keep switching routes even though the physical network is stable. In this problem, we'll see an example where there are two equilibria, and different sequences of events could lead to one or the other.

(a) Describe a sequence of events (BGP announcement messages and path selection decisions) that lead to one stable state
Answer:
After the initial announcement from 1, 4 will know of a route to 1, and by the rules, it will export all customer rules. From the previous step, 3 will now know of a route to 1 which is 3 -> 4 -> 1. Since 2 is 3's customer, 2 will know of the route to 1 through 3 which is 2 -> 3 -> 4 -> 1. Since 2 prefers going through 3 and there is a valid path through to 1, we will always choose to route to 1 via 2 -> 3 -> 4 -> 1, thus achieving our stable state.

(b) Describe a different sequence of events that lead to a different stable state.
Answer:
Since 2 knows of a connection to 1, it announces it to 3. 3 learning of a path from its customer, exports it to everyone, thus 4 knows of this path. This now gives us the routing 4 -> 3 -> 2 -> 1. Now suppose 1 announces to 4, now 4 knows of a path to its customer which it exports to 3 as well. 3 already has a path through its customer which it prefers over going through 4 so it will route through the original path 3 -> 2 -> 1. 4 on the other hand will now route through 1, preferring its customer. Now 2 knowing of the new connection will prefer to go through 3. This leaves us with the equilibrium: 3 -> 2 -> 1, 4 -> 1, 2 -> 3 -> 4 -> 1 (each signifies the preferred path from each AS starting point).

(c) Suppose the network is now stabilized in state (a). A link fails; the BGP routers reconverge; the link recovers; BGP reconverges again; but now the network is in state (b) instead of state (a)! (This problem is sometimes known as a "BGP wedgie" because the system has gotten stuck in a bad state.) What sequence of events causes this story to happen? That is, which link failed, and which messages get sent?

Answer:

The sequence of events would be if the link between 3 and 4 failed. Starting from state (a), if 3 - 4 fails, then during the first reconvergence, 2 will announce its path to 1 to 3, and 3 and 2 will now prefer the path through 2 effectively giving us two separate paths to 1: 3 -> 2 -> 1, 4 -> 1. Now if the link recovers, 4 will announce its path to 3 and 3 will announce it to 2. 3 will still prefer going to its customer first which means it prefers 3 -> 2 -> 1. 2 however will prefer going through 3 so it prefers path 2 -> 3 -> 4 -> 1. 4 is unchanged with 4 -> 1, effectively depicting the state (b).

5 (a) In the original OpenFlow paper that we read, a centralized controller receives the first packet of each flow and installs forwarding rules to handle that flow. However, a significant concern for any centralized design is scalability. Describe a feature of Google's Firepath SDN design that allows it to scale to large datacenters

Answer:

A feature of the Firepath SDN design was that allowed for good scalability was distributed forwarding table computation based on link state being sent to switches from a master controller. Initially, each switch would start out with a static topology discovery and Neighbor discovery to learn local link states. Then it propagates this information through its Firepath client to a Firepath controller which keeps track of the link states acting as a sort of "database". It then sends out the received link states to all other switches, to which the switches compute their own forwarding table. This leads to a scalable design due to the fact that unlike how OpenFlow uses a centralized controller to even decide forwarding rules, the controller in Firepath only keeps track of link states, the switches themselves compute forwarding tables on their own, which creates a more distributed and independent system allowing for an alleviation of a potential centralization bottleneck in the controller. The Firepath SDN design also allowed for ease in configuration discovery, meaning configurations changes through addition/removal of servers was relatively painless due to the local link state discovery and the Firepath master updating everyone when new link states have been have been noted.

(b) Consider a cluster of database servers. The time taken for any server to respond to any request is 10 milliseconds 99.8% of the time, and 200 milliseconds 0.2% of the time. Assume these times are sampled independently at random for each request. We have a front end web server that, when a user request arrives, queries 100 database servers in parallel. When all responses are received, it can reply to the client. The web server itself is incredibly fast, so all its local processing always takes less than 1 millisecond. What is the probability that the web server needs $\geq$ 200 milliseconds to be ready to reply to the client? (As with all answers, briefly show how you calculated the answer.)

Answer:

We know that as we get a request we query 100 servers in parallel. Each one has a 99.8% chance of finishing in 10 milliseconds. Thus we can start with an easier computation of finding the probability that the web server needs $<$ 200 milliseconds. The probability for 1 machine to be less than 200 milliseconds run time is 0.998 this is because of our initial condition in which we only have two outcomes with associated probabilities: 99.8% for 10 ms and 0.02% for 200ms, and the only way to be under 200 ms is to not have a single server run for 200ms. Since we have 100 independently computational machines, the probability for 100 machines to be less than 200 milliseconds is 0.998^{100}. Now to get the probability of a web server being $\geq$ 200 milliseconds we simply do 1 - probability of being less than 200 ms which is $1 - 0.998^{100} = 0.18$

(c) In the setting of the previous question, suppose the web server needs to perform two phases of processing before replying to the client. Phase 1 queries 100 database servers as described above. But now, after receiving all responses from phase 1, it can begin the second phase, where it queries another 200 database servers in parallel. Finally, when all responses are received, it can reply to the client. What is the probability that the

web server needs to wait ≥ 200 milliseconds for its requests to complete?
Answer:
In this question we have essentially 100 + 200 independent database servers being queried. Using the same logic above, we can calculate the probability of being less than 200 ms which is 0.998^{300}. Now taking the inverse, the probability of being $\geq$ than 200 ms iw $1 - 0.998^{300} = 0.45$

6 In a regular Clos datacenter, among the four types of resources (compute, memory, storage, and network), which ones are likely to become scalability bottlenecks for MapReduce and Spark? You will need to explain the workflows of these systems to answer this question.
Answer:
The biggest bottleneck for both MapReduce and Spark will be network resources. Given the distributed computing paradigm both employ, the ability to be able to send data to and from a separate server is imperative. Each of the mapping or reducing jobs needed to be done will have to be farmed out to another server on the network. Data from the main server will have to be sent over somehow, and the post processed data needs to be sent back to the main process to be merged at some point as well. When the servers synchronize and send completed job responses back to the original server, we can easily see congestion over the network along with a prominent issue in datacenters, TCP incast. Compute for both MapReduce and Spark are unlikely an issue since they were both designed to run on commodity servers. Memory is mainly applicable to Spark due to the fact that Spark prides itself as being able to do computations faster than MapReduce, primarily by loading all the necessary information into memory. This could potentially create a strain since RAM is much more expensive than ordinary hard disk. Storage may only be an issue if data is obscenely large, but expanding storage capacity is not really too much of a hassle.

7 REVIEW

(a) Summary:
The project I am reviewing is Group 2: Jellyfish vs Fat-tree(ECMP). They are examining the efficacy of using ECMP in other network topologies other than Fat Tree. Specifically for this project they are focused on seeing if ECMP would work well in a Jellyfish topology, and maybe expanding to test other topologies. The importance of this work lies in the the fact that if ECMP could work for Jellyfish and produce optimal results then maybe network operates could re-evaluate their choices of network topologies and shift to other topologies other than fat tree. Their approach is to use mininet to create Fat tree and Jellyfish and then use a iPerf and Wireshark to potentially generate and analyze their flows through the network. They then will try to examine network characteristics through these events to judge their goodness. In terms of updates, they have run iPerf tests, did some packet analysis, including unique paths traversed, and outputted Bandwidth analysis for Fattree and Jellyfish.

(b) Comments:
I think its a good project, but I wonder why Jellyfish was chosen as a comparison between the two topologies. In terms of networks Jellyfish may theoretically have many performance gains over fat tree, but the hectic connections required makes it less practical as a topology of choice(from an implementation standpoint).

(c) Grading:
(a) Everything ran to completion and worked. The graphs matched the report.

8 One of the reasons that MapReduce is slow is the barrier between the Map and the Reduce phases, i.e., no Reduce task can start before all Map tasks finish. Suppose a programmer writes a variant of MapReduce without a barrier, i.e., where Reduces can start before all Maps are done computing.

(a) Demonstrate with an example that this Mapreduce run may be incorrect, e.g., it may generate an incorrect value.
Answer:
If for example we have 2 mappers and 1 reducer trying to count the number of words

in a passage. If mapper 1 gets half the passage and mapper 2 gets the second half of the passage and mapper 1 finishes first. Without a barrier, mapper 1 would send its results to the reducer which would output potentially only half the true count of the target word in the passage(since we didnt wait for the other mapper to finish).

(b) Show how you can resolve this correctness issue while (partially) preserving the performance improvement of this "barrier-less" MapReduce.

Answer:

We could potentially have the reducer immediately run each mapper's output, but doesn't output any results until all mapper's are finished running. The reduce upon seeing all mapper jobs finish, will constantly store its outputs as intermediary results until all the mapper jobs finish, to which it will know that there will be no more data fed to it and thus it can output. This method still has the performance gain of no barrier between mapper and reducer, but it does need to wait at the reducer step for all the mappers to finish.