Final
601.419/619 Cloud Computing

Jason Zhang jzhan127

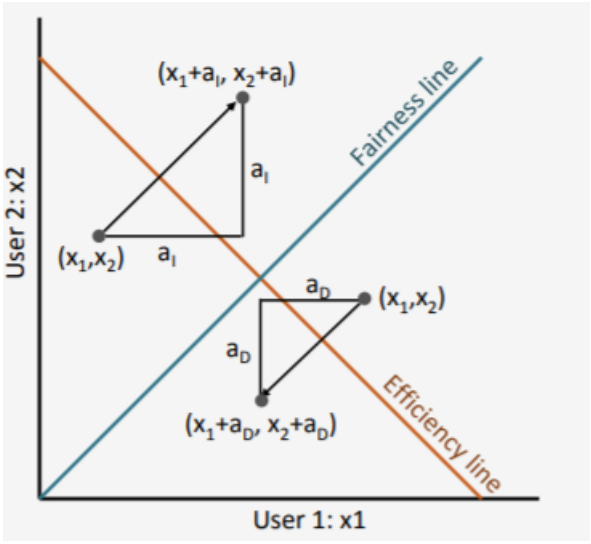
May 12, 2020

1 Two advantages of Clos topologies over Jellyfish is that in for one, it is easier in terms of physical construction as a Clos topology is hierarchical and organized, whereas Jellyfish, with its random connections is extremely difficult to implement in practice since it does not account for cabling difficulties, and second it is easier/better to do congestion control and load balancing in a Clos topology for two reasons, one due to Jellyfish's random nature we could encounter asymmetrical topologies which means that there could be extremely high path diversity or not enough path diversity and thus traditional load balancing strategies may not work well, two, which also ties into load balancing, is that ECMP is not enough for Jellyfish as it does not provide the necessary path diversity needed for Jellyfish (From the paper).

Two disadvantages of Clos topology vs Jellyfish is that we get this ability to have incremental expansion in Jellyfish. This is due to the random graph scheme we have which makes expansion much more simple (since we just randomly connect a new switch we want to add) as compared to the rigid structure of Clos topologies making expansion a hassle, and we can also get higher throughput with Jellyfish due to the fact that we have on average shorter path length in Jellyfish as compared to Clos topologies, which need to share links much of the time in order to get to a destination host, which can reduce total throughput in the data center.

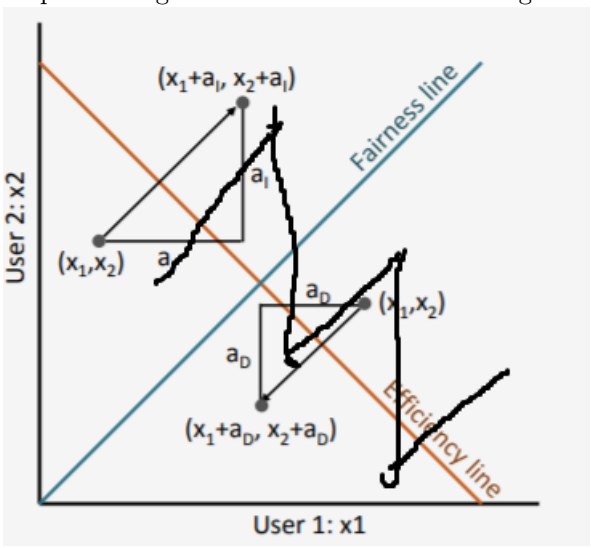
2 Based on the BGP routing policies, S5 can reach S9 via S5 -> S2 -> S3 -> S6 -> S9. This is due to the fact that S9 will message its provider S6, which will message its provider S3, who messages everyone its connected to messages from its clients (s6) thus S2 will know the route S2 -> S3 -> S6 -> S9. Since S5 is the client of S2, S2 will tell S5 this route thus S5 will take S5 -> S2 -> S3 -> S6 -> S9. It will not take S8 due to the fact that S8 and S9 are peers, so S9 will not tell S8 anything. The path stretch in this case is the Length of S5-> S2->S3->S6->S9 / Length of S5 -> S8 -> S9 = 2.

3 Neither converges to both fairness and efficiency. For AIAD the problem stems from the fact that we only make linear increases/decreases which means we can only stay on the line that we are on, for example if we are on a line that is parallel to the fairness line but not on it, as our initial allocation, we will never be able to converge to fairness since additive changes will only keep us on the line we are currently in.



As the diagram shows here, we only stay on the line forever, we never deviate from it thus we cannot converge to fairness and efficiency.

MIAD will not converge due to the nature of its changes as well. Its behavior can be seen in the below graph in black. The multiplicative increase will eventually keep going farther away from the fairness line and additive decrease is not adequate enough to direct the allocation towards the right direction in fairness. In terms of efficiency the MIAD strategy will just keep bouncing around thus unable to converge.



4 The bottleneck link here is the link from the Switch directly connected to E, all the other links can get all the bandwidth they need since there is enough bandwidth in the earlier links. But once we get to the last link, we need to essentially share a !0 Gbps link with a !0 Gbps flow from D and 6 Gbps flows from A, B, C together. Thus we need to only give D 4 Gbps in order not reduce the rate of the smaller flows. Thus our allocation should be A: 1 Gbps, B: 1 Gbps, C: 4Gbps, D: 4 Gbps.

5 Agility refers to the ability to use any server for any service at any time, and it is important because it increases the utilization of the servers and reduces overall costs in the data center. One technique used by cloud providers to maximize agility is to use VMs since it allows for rapid installation of service code when the time comes, making it a quick and efficient strategy to be able to deploy changes.

6 Scalablity is bottlenecked by the fact that SDNs require centralization due to the controllers needing global visiblity in the network. It is not very fault tolerant due to the fact that we have a central point of contact whenever a switch needs to make a forwarding decision, thus

if the controller goes down, the switches are essentially unable to forward properly(solution typically stems to replica Controllers or default rules in switches in case of failure), performance limitations are that we need to always contact the controller whenever we want to make a forwarding decision, this physically equates to an extra hop, introducing higher latency in the network, it also introduces more network traffic since we need to also send the data to the controller as well.
Two approaches to mitigate these limitations are to first, since the Controller can be the central failure point, we can have replica controllers such that if on goes down, another one can step in to take on responsiblities(done by an election process), Google's Firepath employs this type of strategy with their Firepath masters. Another approach (also employed by Google's Firepath) was to instead do distributed forwarding table computation at the switches with the link state sent by the controller. This reduces the amount of network traffic since switches do not have to send entire flows to controllers to compute a forwarding and also reduces the amount of computational strain on the controllers.

7 Practical cloud systems guarantee completeness. It is overall a much simpler guarantee since we have the leeway of not having to ensure timeliness or perfection since completeness only guarantees that we will eventually discover the faulty node. This grants more flexibility and less of a hassle for systems.

A Ring failure detector essentially orders nodes in a ring such that a node only has connections to adjacent nodes on the ring. Nodes will use heartbeats to detect whether or not a node has is faulty. It is not complete since only adjacent nodes will immediately know if a node is down. If the messages from adjacent nodes to other nodes get lost/are faulty and then the node fails when notifying the other nodes that a node is faulty, the other nodes in the network may not find out that the first node that failed was faulty, thus making Ring not complete.

Gossip based failure detectors are poor since we rely on others to tell us whether or not a node is faulty. There is the possibility that the network get bisected, but due to the fact that nodes rely on adjacent nodes to detect non adjacent node failures, the nodes may not be cognizant of a failure.