Data Center Networks: Introduction and Challenges

Before we start

- HPC DC network design issues are different
  - Most works on interconnection topology
  - Communication patterns depend on application
    - Known for many types of applications
  - Goals are different

- We will talk about networking issues in Enterprise/Cloud DCs
  - Focus of almost all data center networking works in recent years
  - So DC will mean by default Enterprise/Cloud DCs in the rest of this presentation

Characteristics of DC Networks

- Interconnects very large number of servers
  - Results in a large network inside the DC, with thousands of switches
  - Topology is important
- Commodity servers and switches
- Different applications run on these servers
  - Applications may run directly on servers or on VMs
  - Applications may be of a single enterprise (Facebook, Google, etc.) and/or from a large no. of external clients who use the DC resources (clients of Microsoft Azure, Amazon AWS, Google Cloud etc.)
  - VMs may be moved to different physical servers dynamically without the user being aware of it

- "Fat" scale-out interconnection topologies make for small maximum number of hops and low oversubscription ratios
  - Regular topologies, unlike the arbitrary topology of a wide-area-network/internet
- Very low RTT within DC (within low 10s of microseconds)
- Physical resources are managed by a single organization
  - Amazon/Microsoft/Google/Facebook/ …..
  - Single administrative domain, limited geographical scope, less heterogeneity
    - Changing things is easier and faster – new protocols, new security policies,….
    - Unlike in the internet, which requires agreements between a large party of stakeholders….

Example Types of Applications

- Interactive applications
  - Web search, online shopping, mail,….
  - Many of them require processing large amounts of data in the background to generate results
  - Needs results in 1-2 seconds (including send and receive over the internet)
  - Implies large parallel processing, with each "worker" working on some parts of the data
- Non-interactive applications
  - Computation, updating bulk data (like indexes for search), file backups, VM migration,….
  - May involve large data transfers
  - Latency is less important, but throughput is

DC Traffic Characterization

(Validated by many actual studies in production DCs published in existing literature on data center networks)

- Intra-DC traffic is much larger than external traffic to-and-from outside
- Most of the DC traffic are short flows ("mice" flows, few 10s of KBs of data), with a few large flows ("elephant" flows, few MBs to few GBs)
- Short flows are usually caused by latency sensitive applications, large flows are usually caused by throughput sensitive applications
  - Extreme latency requirements
  - High bandwidth requirement
- However, most of the total bytes transferred are in this small no. of large flows
- There are no general patterns in the traffic matrix (who sends traffic to who) or in inter-arrival times of flows

What causes the short flows?

- Many applications involve query and search, that work on the "Partition/Aggregate" model
  - Query comes to a server
  - Server breaks up query to one or more "aggregators"
  - Each aggregator assigns multiple worker nodes to work on the data
  - Worker nodes send results to aggregators
  - Aggregators sends results to main query processor
  - Can be more than one level of aggregators in the tree
  - Based on the results, query processor may do more than one iteration of the above to refine the result
  - Total time available to query processor to produce the result is typically a few 100s of milliseconds

- Networking issues for partition/aggregate?
  - Total time is divided between computation and network delays
  - Typically each worker will have a deadline and if deadline is reached, aggregator may timeout (no result from worker) or worker may give partial/low-quality result back
  - If network delay is large, workers have less time, so more chance of failing deadline
  - Deadline miss means finally low-quality results going out from the query processor
  - Typically, worker nodes may have < 100 millisecond
- But why will network be slow? After all, the servers are connected by high speed links
  - Typical no. of hops is low (think Fat-Tree for example)
  - RTT inside a DC is only 10s of microseconds (uncongested case)
  - Most transfer operations are happening in parallel

Basic Model of a Switch

- A switch can be abstractly viewed as
  - An input queue for incoming packets on its ports
  - A processing engine
  - An output queue for outgoing packets to its ports
- Incoming packets are first put in the input queue, and then processed ,and then placed in output queue to be sent to appropriate outgoing link
- If packets arrive too fast on incoming links (faster than processing speed), queue builds up
  - Packets further back in the queue will have to wait to be processed (Queuing Delay)
  - If queue length becomes larger than buffer size, packets will be dropped
- TOR switches used are mostly commodity switches
  - "shallow-buffered" switches – (relatively) lower buffer size

Reasons for Slow Network: Congestion at Switches

- The same parallelism of the workers also causes the "incast" problem
  - Most workers finish at about the same time (why?), and try to send the results back to the aggregators
    - "Incast" towards the aggregator
  - Can cause queuing delay in switches along the path towards an aggregator
- Short delay-sensitive flows have to compete with large delay-insensitive flows at switches
  - Packets of large flows can lie in switch buffers ahead of packets of short flows causing queueing delay
  - Normal switches do not treat such traffic separately

Can we use traditional TCP congestion control schemes?

- Using directly has problems
  - Broadly senses congestion by packet drop (TCP Tahoe/Reno/New Reno…) or increased RTT (TCP Vegas, FastTCP)
  - Packet drop causes retransmission, but typical min retransmission timeouts (RTO_min) is too high (200-300 millisecond)
    - Unacceptable delay for datacenter traffic
  - Difficult to distinguish increased RTT (from RTT in uncongested case) due to very small RTT
    - Factors like variable processing delay etc. can cause inaccuracies
  - Schemes do not prioritize
    - Ex. Process packets of flows with closer deadlines fast
- Ideally would like to know very fast before delay builds up and throttle flows appropriately
- Lots of work on reducing queueing delays for DC traffic

Other Issues: Load Balancing

- Scale-out topologies like Fat-Tree have given networks with a large number of alternate paths between a source and a destination
- How to load balance the flows along the different paths to ease congestion in a single path?
  - Possibilities
    - Per-flow basis: Route different flows on different paths (but each flow on one path)
    - Per-packet basis: Route different packets of a single flow on different paths (say round robin)
    - Anything else possible? We will see…
  - Can we use existing schemes like ECMP routing, MPTCP etc.?

Equal Cost Multi Path (ECMP) Routing

- Implemented at switch/routers
- Requires the underlying routing algorithm to compute multiple paths to a destination in the routing table
- Basic idea:
  - Assume all paths are of equal cost
  - Hash a flow onto a path based on the 5-tuple (source and destination IP and port, protocol)
  - All packets of a flow follow the same route
- Can work well for symmetric flows (all flows of similar size)
- For asymmetric traffic, may cause congestion and poor load balancing if multiple large flows hash to the same route due to hash collision

Multipath TCP (MPTCP)

- Sender and receiver side modification to extend basic TCP to use multiple paths
- For each Multipath TCP connection requested by user, sender creates more than one separate TCP connection underneath to destination, one for each path
  - User sees only one TCP connection, the one requested
- Each connection carries a subflow of the main flow (part of the main flow's data)
  - Ex. Packets sent through subflows in round-robin manner
- TCP options field used to ensure both sender and receiver are MPTCP-capable before using MPTCP
  - Drops back to normal TCP if receiver is not MPTCP-capable
- Receiver uses additional info sent in TCP options field to reassemble the subflow data into a single flow data before giving to application

Basic MPTCP Operation

- Subflow setup
  - For the first subflow, sender opens a TCP connection, sending MP_CAPABLE in TCP option field of SYN packet
  - If receiver is MPTCP capable, sends back the same option in SYN+ACK packet
    - If not, no option sent, so sender drops back to normal TCP
  - Sender responds back with the same option in ACK packet
  - Also exchanges random keys for future authentication
  - Future subflows are added with new TCP connections with MP_JOIN option in handshake packets to join as a subflow of the original connection
    - Connection id derived from hash of keys and MAC of keys used during handshake to ensure only correct subflows join (prevents attacks)

- Data send/receive
  - Each TCP connection for a subflow has its own sequence no. as in normal TCP
  - TCP option field used again to send mapping between data sequence in main TCP flow to sequence no. in the subflow packet sent
  - Receiver uses the mapping to create inorder sequence of bytes for the original flow before giving to application
    - One receiver side window only
  - If congestion or path break detected in one subflow, bytes of that subflow are sent through other subflow

- Advantages
  - Can use multiple paths through multiple interfaces simultaneously
    - Ex. use a wireless and cellular interface at the same time for faster download
- Disadvantages
  - Middleboxes (Routers, ADCs, UTMs, IPS,…) that the packets pass through may not pass unknown options if not MPTCP-enabled, change the packets (ex. NAT) or behave in other unpredictable manner
  - Packet of subflows sent through different paths may encounter different delays and arrive out of order

Other Issues: Multi-tenancy Support

- DC resources will be shared by many "tenants" or users.
- Need to ensure that each user gets its allocated share of resources
- Performance Isolation
  - Allow multiple "tenants" to exist on the DC, each unaware of the others
    - Maintain minimum network bandwidth guarantee given to each tenant irrespective of network use by others
    - Allow VMs of one tenant to be moved without having the tenant changing anything like IP address, virtual switch configuration etc. (tenant should be unaware of its VM's move)
      - Tenant thinks its resources just as servers connected to a single switch!

Summary: Major Goals in DC Network Design

- Congestion Control/Queue Management
  - Maintain latency guarantees of short flows in the presence of large flows
  - Maintain high throughput for large flows
- Performance Isolation/Multi-tenancy Support
  - Allow multiple "tenants" to exist on the DC, each unaware of the others
    - Maintain minimum network bandwidth guarantee given to each tenant irrespective of network use by others
- Load Balancing
  - Take advantage of multiple paths to route different packets of a flow to reduce latency

Some Interesting Papers

- Interconnection network
  - VL2[Greenberg2009]
- Congestion Control
  - DCTCP[Alizadeh2010], D2TCP[Vamanan2012], TIMELY [Mittal2015]
- Load Balancing
  - CONGA[Alizadeh2014], Presto[He2015]
- Multi-tenancy Support
  - ElasticSwitch[Popa2013]

A brief summary of each paper follows. You should read on your own if you are more interested. There are also many more recent works.

VL2

- High level goal: Give each service the illusion that all the servers assigned to it, and only those servers, are connected by a single non-interfering Ethernet switch
  - Virtual Layer-2 switch semantics
- Switches interconnected with Clos topology
  - Large number of paths between servers
- Network switches are layer-3 switches running OSPF
  - Enables use of multiple paths
- Specifies a topology, a routing method that decouples application IPs from switch IP (allowing easy migration of servers the application runs on), and a load balancing scheme for utilizing different paths for different flows

- Address Resolution and Packet Forwarding
  - Two separate IP address spaces
    - Application specific address (AA) used by user application
    - Location specific address (LA) to identify a TOR switch
  - OSPF builds routing table over LAs
    - Switch level topology known to all switches
  - Application sends a packet with destination AA. Broadcast ARP is issued. VL2 agent s/w on application server traps it, and sends a query to a Directory Service for corresponding LA (switch to which the server with destination AA is connected)
  - Directory Service gives corresponding LA. The AA-to-LA mapping is cached (similar to ARP cache) for further sends
  - Packet is routed normally to the destination LA with an encapsulating header with LA address
  - LA decapsulates to find AA and gives to server with that AA

- Directory Service
  - Set of servers for AA-to-LA mapping
    - Small no. (5-10) write-optimized servers for updates, with strong consistency guarantee
    - A large no. (50-100) servers per 100K servers of read-optimized servers for answering queries, with eventual consistency guarantee
  - AA-to-LA mapping updated when a application is assigned a server initially or migrated to a different server
- More details in Section 4.2.1 in paper

- Load Balancing
  - Consider a TOR switch. It has multiple paths to a destination, through different intermediate switches
  - For each flow, choose one intermediate switch randomly
    - All packets of that flow goes through same path, so no reordering problem
  - Problem with this simple approach: If all intermediate switches have different LA address, needs VL2 agents at all TOR switches to update if an intermediate switch goes down
  - Solution:
    - Give all intermediate switches the same LA
    - Use ECMP and IP anycast to choose an intermediate switch
      - ECMP automatically handles failure
- More details in Section 4.2.2 in paper

DCTCP (Data Center TCP)

- Goal: Maintain low switch buffer occupancy while maintaining high throughput
- Identifies three main reasons for buffer buildup at switches
  - Incast problem – large no. of flows sending packets towards the switch at the same time, as we have seen for partition/aggregate type of computation
  - Queue buildup at one port due to large flows
  - Buffer pressure: short flows on one port suffering loss/delay due to large flows on other ports
- DCTCP main idea
  - React to congestion in proportion to the extent of congestion
    - Different way of changing congestion window due to congestion avoidance to avoid packet loss as much as possible
- Needs change in sender and receiver
  - But not large change
- Needs ECN capable switches (available in commodity switches)

- How to know extent of congestion?
  - Use ECN-like method
  - When a packet passes a switch, switch marks the packet with "Congestion Experienced" code point (in traffic class field of IP header) if queue length > a threshold K
  - Receiver sets the ECN-Echo flag in TCP header in ACK if it receives a packet with CE code point set
  - Sender knows exactly which packets experienced congestion looking at the ACK
    - Easy to see if receiver sends one ACK for every packet received, but it does not! TCP sends cumulative acks
    - Needs a slightly modified ECN Echo setting scheme at receiver (see Figure 10 and associated discussion in paper)

- Sender keeps track of fraction ∂ of bytes (in some past window) that encountered congestion (see exact formula in paper)
  - ∂ estimates the probability of congestion
- On receiving an ACK with ECN bit set,
  - cwnd = cwnd $\times$ (1 - ∂/2)
- All other features of TCP congestion control like slow start etc. remain the same

D^2TCP (Deadline Aware Data Center TCP)

- DCTCP uses congestion signal, but is deadline-agnostic
  - Throttles all flows equally, irrespective of their deadline
  - Ideally, should throttle flows with closer deadlines less than flows with far off deadlines
- Goal of D^2TCP: Keep advantage of DCTCP + deadline awareness, without needing custom switches
- User application also passes deadline info while sending data to network stack
- D^2TCP computes fraction of packets encountering congestion similar to DCTCP

- Computes a penalty function $p = \partial^d$ (larger d means closer deadline)
- Congestion window W change in congestion avoidance phase

$$W = W \times (1 - p/2) \text{ if } p > 0$$
$$= W + 1 \text{ if } p = 0$$

  - For no congestion, $\partial = 0$, so $p = 0$, so window increases by 1 as in normal TCP
  - For full congestion (all packets marked with ECN), $\partial = 1$, so $p - 1$, so window is halved
  - Window reduced by extent of congestion and closeness of deadline in between
    - Larger d (closer deadline) makes p smaller (for same ∂), so less throttling
- d =1 is just DCTCP, For long flows with no deadline, d is set to 1
- See Section 3.2.3 in paper on how to set d for short flows

TIMELY (Transport Informed by MEasurement of LatencY)

- Unlike DCTCP and others, proposes using RTT measurement as congestion signal
  - If it can be measured accurately, a single high RTT can immediately signal congestion
- Not specific to TCP, can be applicable to any reliable transport protocol
- General problem of using RTT as a congestion signal in data centers
  - Given very low RTT in a DC, RTT is hard to measure accurately due to variations introduced by host processing delay etc.
- TIMELY's premise: recent NIC advances provide h/w support for high quality packet timestamping as well as h/w generated acks
  - Can be used to remove unpredictable host response delays

- Three modules
  - RTT measurement
  - Rate Computation: Using RTT measurement to compute a target sending rate
    - Rate based protocol instead of window based
  - Rate Control Engine: Inserting delays between segments to achieve the target sending rate
- RTT measurement
  - Packet timestamping + ack generation at NIC
  - Typical components of a RTT
    - Serialization delay to transmit all packets in the segment
    - Wire propagation delay for the packet and its ack
    - Host response delay for ack generation
    - Queueing delay in the network

- TIMELY defines RTT only as propagation delay + queuing delay
  - Transmit time can be found from segment size and line rate, and subtracted from measured total RTT
  - Host response time is negligible for NIC-generated ACK
- Propagation delay can be estimated from minimum RTT (RTT without congestion)
- Queueing delay is the only variable
  - Shows that variation in RTT gives direct feedback about queueing delay encountered

- Sending Rate Computation
  - Delay gradient based algorithm
    - Positive gradient means a rising queue and vice versa
    - Computes gradient as ratio of difference in rtt (computed as exponentially weighted moving average of rtt differences) and minimum RTT
    - Increase sending rate if gradient ≤ 0, decrease otherwise
      - Simple algorithm. More details of exact algorithm in Algorithm 1 in paper

- Rate Control Engine
  - Uses time of send of last segment and target rate given by rate computation module to schedule when the next segment is to be sent
  - When due, send to NIC for immediate transmission; hold in scheduler queue until then

CONGA

- Distributed congestion-aware load balancing scheme
- Instead of per-flow or per-packet load balancing, does per flowlet load balancing
- Flowlet – Burst of consecutive packets of the same flow until a threshold gap occurs
  - Each flowlet sent on the same path, different flowlets (of same or different flow) can go through different paths
  - Sending flowlets of same flow on two different paths can cause out-of-order arrival?
    - However, if threshold gap (in definition of flowlet) > max difference in latency between two paths, then reordering chance is very low
    - Latency of alternate paths in DC between same source-destination is usually comparable (if congestion is similar), so viable for DCs
    - Paper shows flowlets can be defined for DC traffic such that reordering does not happen if sent on different paths

- But how to keep track of congestion?
- CONGA solution – in-network leaf switch -to-leaf switch feedback
  - Overlay network formed over the leaf switches
    - Leaf switch gets a tunnel endpoint address
    - Each leaf knows the destination leaf for each packet
  - Packets from VM sent to CONGA fabric, sent to a leaf switch (source leaf). Source leaf finds destination leaf for packet
  - Each leaf maintains a Congestion-To-Leaf table (2-d table, with (p, q)-th entry giving congestion measure to leaf p through next hop uplink q of the source leaf switch)
  - Source leaf encapsulates the packet in a separate header, chooses an uplink based on congestion information in the table (minimizes the maximum of local congestion to uplink and congestion in path through uplink in table), and sends to that uplink

- Encapsulating header has two important fields
  - LBTag to identify the uplink through which the packet is sent (identifies a path partially)
  - CE field – for switches to add congestion information
  - As the packet passes through the switches in the overlay network, each switch updates the CE field with congestion metric value if the link congestion > congestion in CE field currently
    - So when it reaches the destination leaf switch, the CE field contains the maximum congestion of any link in the path followed by the packet
  - Each leaf also maintains a 2-d Congestion-from-Leaf table, with (p, q)-th entry giving congestion from source leaf p received with LBTag value q.

- When the packet reaches the destination leaf switch,
  - The Congestion-from-Leaf table is updated from the CE field
  - The packet is decapsulated and the data packet inside is sent to the application
- When a packet is sent in the reverse direction next, one entry (one LBTag value, chosen in some fashion like round-robin or last-changed) is sent back in another field in the encapsulating header
  - Gives the congestion feedback to source leaf
- On receiving this, source leaf updates the corresponding Congestion-To-Leaf table entry
  - This is used for next packet sending

- Load balancing to maintain similar congestions along the different paths

Presto

- Load balancing at the edge
  - Implemented as vSwitch modification in hyperviser
- Uses the intuition that ECMP works very well for balancing load if all flows are of similar size (symmetric traffic)
- Large flows are broken into smaller flowcells (64KB) at the vSwitches and sent through a ECMP like mechanism
  - Small flows are usually anyway < 64 KB size, so mostly one flowcell
  - Why not flowlets? Because flowlets can be quite large, causing asymmetry again…
- What about reordering between flowcells on different paths for large flows?
  - Handled by modifying GRO (Generic Receive Offload) handler in hyperviser to reorder there instead of pushing up to TCP stack
  - Sequentially increasing Flowcell id in each flowcell for reordering

- Instead of per-hop load balancing like ECMP, Presto keeps full paths, and chooses one path among the paths to send a flowcell
- How are these paths found and kept?
  - Central controller creates path and sends LB info to all vSwitches (SDN-like)
    - Partitions the network into multiple spanning trees
      - Each spanning tree gives one path between any two vSwitches
      - Different spanning trees give different paths between same pairs
    - Each vSwitch gets a unique shadow MAC address in each spanning tree
    - Controller installs forwarding table for each spanning tree in all vSwitches
    - When the first packet of a flowcell comes to sender, the vSwitch software chooses one shadow MAC for the physical destination MAC (kept the same for all subsequent bytes of that flowcell).
    - Shadow MACs for each destination MAC are chosen in a round robin fashion for load balancing flowcells

- How to separate loss from reorder?
  - Important because response is different
    - For loss, GRO module should send up to TCP immediately for retransmission, congestion window change etc.
    - For reorder, GRO module wait for missing flowcel to come
  - Presto solution
    - Bytes within the same flowcell follow the same path, so should come in order. If not, flag as loss. Otherwise wait (for some time)
    - Some corner cases handled separately

ElasticSwitch

- Goal: Provide minimum b/w guarantee to each tenant while providing high network utilization
- Simple approach: static reservation to guarantee bandwidth
  - Does not utilize network well if some tenant is not using its minimum bandwidth and some other tenant needs to use more than its minimum guarantee
- ElasticSwitch's approach
  - Modify TCP's rate control algorithm dynamically based on minimum guarantee and actual use
    - Allows each VM to get its minimum guarantee share
    - Allows unutilized capacity to be shared by other VMs (beyond their minimum guarantee)

- Two layers – Guarantee Partitioning (GP) and Rate Allocation(RA)
- Guarantee Partitioning (GP)
  - Minimum guarantee given to VM X is broken up into minimum guarantee for each X-Y connection for each VM Y that X communicates with
    - Can be static or based on actual throughput for each pair seen in some past window
    - Assumes that there is separate (not part of this paper) admission control and VM placement algorithms that guarantee that minimum bandwidths assigned can be satisfied by the existing network (links can take the sum of the minimum bandwidths assigned to them)
  - GP gives this info to RA

- Rate Allocation (RA)
  - Uses any TCP rate control algorithm to compute sending rate for each X-Y communication
  - Chooses the current rate as max (guarantee given by GP, sending rate computed by TCP)
    - Ensures if there is spare capacity because some flow is not using its minimum guarantee, others who need it can
  - Simple implementation can lead to a scenario where many VMs sharing a link with unutilized capacity can detect this at about the same time, and try to send at increased rate, thereby clogging the network
    - Different solutions proposed
      - Don't assign full link capacity while assigning minimum bandwidths
      - After a congestion, hold rate increase for a period inversely proportional to the guarantee of that flow
      - Less aggressive increase as current rate goes over guarantee
- Uses Open vSwitch (SDN) to measure VM-to-VM flows
- Implemented in the hypervisor, with GA and RA invoked periodically to adjust rates

References

- [Greenberg2009] A. Greenberg et al., "VL2: A Scalable and Flexible Data Center Network," ACM SIGCOMM 2009
- [Alizadeh2010] M . Alizadeh et al., "Data Center TCP (DCTCP)," ACM SIGCOMM 2010
- [Vamanan2012] B. Vamanan et al., "Deadline-Aware Datacenter TCP (D2TCP)," ACM SIGCOMM 2012
- [Mittal2015] R. Mittal et al., ""TIMELY: RTT-based Congestion Control for the Datacenter," ACM SIGCOMM 2015
- [Alizadeh2014] M. Alizadeh et al., "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters," ACM SIGCOMM 2014
- [He2015] K. He et al. "Presto: Edge-based Load Balancing for Fast Datacenter Networks, " ACM SIGCOMM 2015
- [Popa2013] L. Popa et al., "ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing," ACM SIGCOMM 2013