PRIORITISING INTERACTIVE FLOWS IN DATA CENTER NETWORKS WITH CENTRAL CONTROL

S Mohana Prasad

Outline

- Motivation and Introduction
- Overview of the Problems
- Part 1: Scaling Up Fastpass for Large Data Centers
- Part 2: ECN based Congestion Control for a Software Defined Network

Motivation and Introduction

- Data Center Networks:
  - Requirements: high utilization, low median/tail latencies and fairness
  - Offers: Structured/known topologies, Lesser chaos, lower scale (as compared to Internet), easily manageable as under a single entity, maintained with bleeding edge technology
- Why prioritise Interactive flow?
  - Data centers should accommodate mix set of workloads
  - Coexistence of Bulk and interactive traffic is necessary
  - Facebook servers - Map reduce job running and user queries coexist
- Why central control?
  - Gives fine grained control
  - Can achieve global objectives
  - Works well for smaller scale of operations
  - Why not take packet transmission, path selection and congestion control decisions at the controller?

Problem Statements

- Fastpass - a data center network architecture based on zero queuing from MIT's lab
  - Prioritizes interactive flow and is based on central control
  - Central arbiter supports only 1.5Tbps of network traffic on 8 cores and doesn't scale beyond this
  - We build efficient multi-core software architectures to support upto 7.1Tbps on 12 cores and the earlier 1.5 Tbps on just 3 cores
- Re-thinking congestion control in SDN:
  - Controller assisted congestion control with global view
  - Using ECN bits. Requires no change to end hosts or SDN switches
  - 30x improvement over TCP cubic, 1.7x over TCP RED in flow completion time of interactive traffic without any compromize on throughput of bulk traffic

Part 1: Scaling up Fastpass for Large Data Centers

FastPass

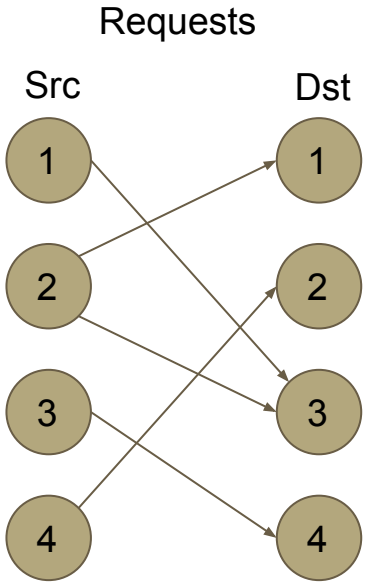
- **Idea**: fine-grained control over packet transmission times and network paths through a **central arbiter**
- Datacenter Network framework aiming at high utilization and **zero queuing**
- Provides low median and tail latencies for packets, high data rates between machines and flexible network resource allocation policies.
- Fastpass is based on two fast algorithms:
  - Determine the time at which each packet should be transmitted
  - Determines the path to use for that packet

Summary of Results	
§7.1	**(A)** Under a bulk transfer workload involving multiple machines, Fastpass reduces median switch queue length to 18 KB from 4351 KB, with a 1.6% throughput penalty. **(B)** Interactivity: under the same workload, Fastpass's median ping time is 0.23 ms vs. the baseline's 3.56 ms, 15.5× lower with Fastpass.
§7.2	**(C)** Fairness: Fastpass reduces standard deviations of per-sender throughput over 1 s intervals by over 5200× for 5 connections.
§7.3	**(D)** Each comm-core supports 130 Gbits/s of network traffic with 1 µs of NIC queueing. **(E)** Arbiter traffic imposes a 0.3% throughput overhead. **(F)** 8 alloc-cores support 2.2 Terabits/s of network traffic. **(G)** 10 pathsel-cores support >5 Terabits/s of network traffic.
§7.4	**(H)** In a real-world latency-sensitive service, Fastpass reduces TCP retransmissions by 2.5×.

Source: Fastpass paper

Timeslot Allocator

- Maximal Matching in a Bipartite Graph Problem
- Greedy Heuristic Solution



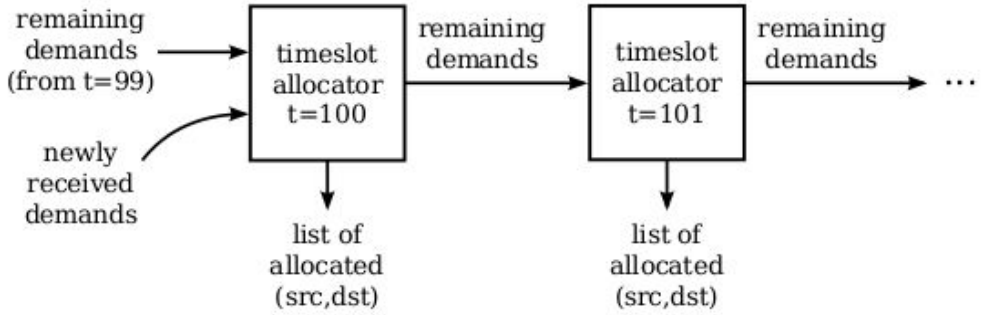
Src	Dst	Allocated Srcs and Dsts
1	3	
4	2	
2	3	
2	1	
3	4	

Pipelined Allocator

Input tuples are of the form (Src,Dst, Num_Packets_Requested, Priority)

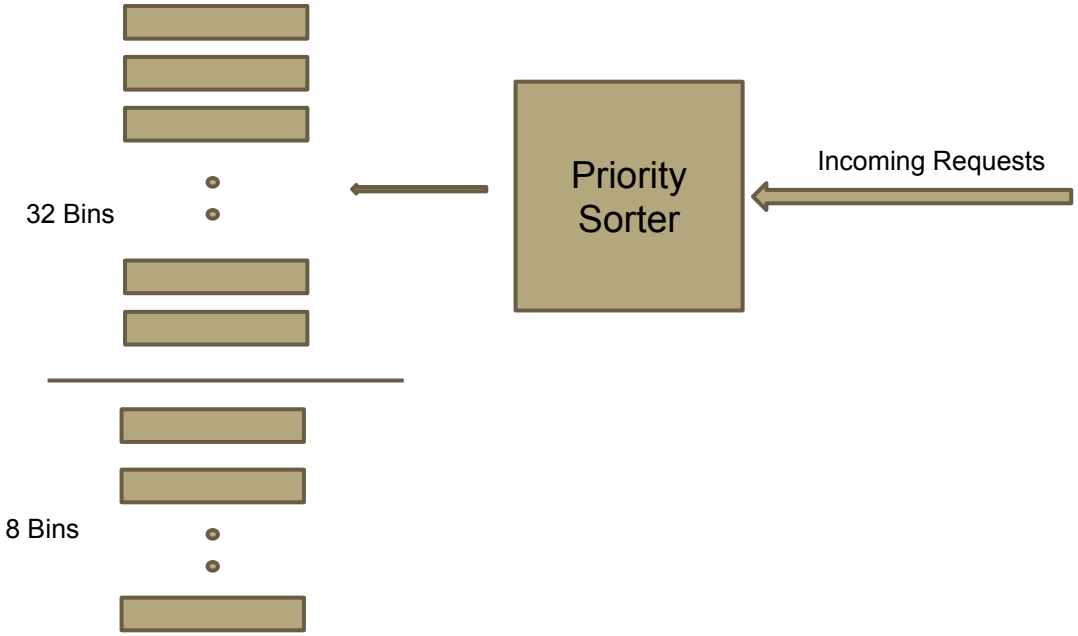
Head of the Pipeline alone reads new demands

To reduce the overhead of communication, we allocate in batches in each core. (Allocate for 8 timeslots in single core)



How to ensure Fairness?

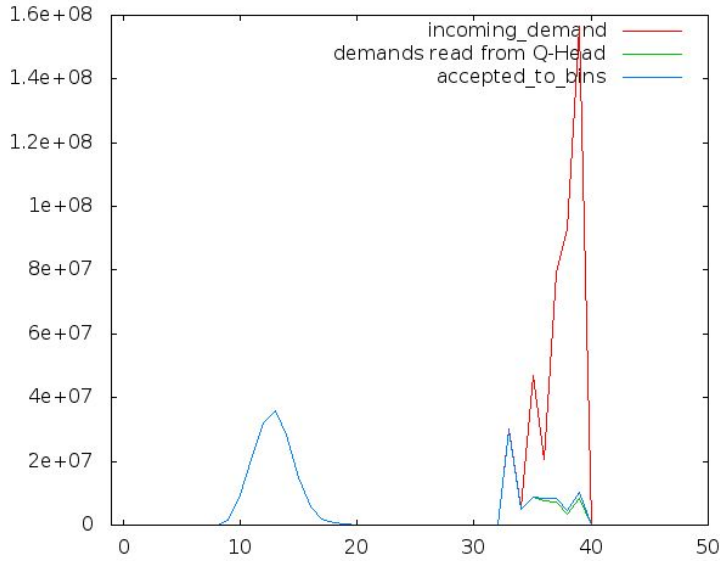
Assume 32 Priority Bins and the core allocates for 8 timeslots.

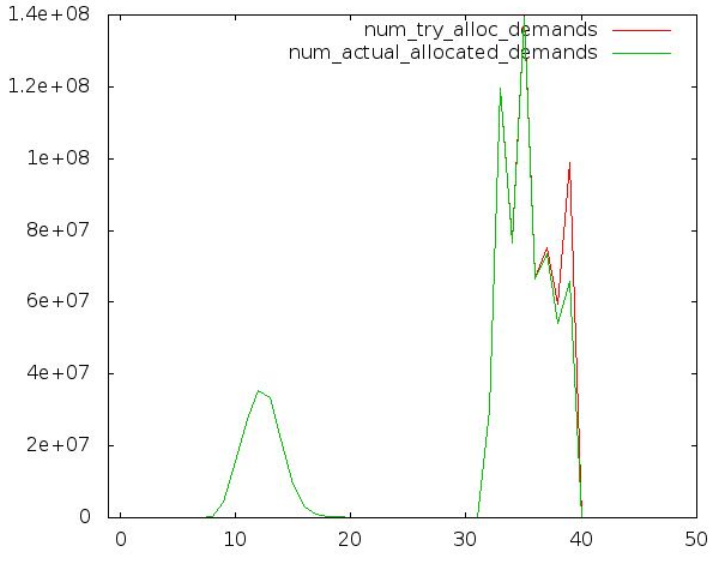


Analysis of the Pipeline

- Logged variables and analysed. Why not linear Scaling???
- How to test? Ans: Stress Test Core!!!!!

8 Cores Batch Size 8





Attempts to Fix the Pipeline

Problem: Cores are not Fully Utilized. They are Idle most times.

- Not too Many contentions
- Is the allowed mask making demands queue up till the end? Ans: No!!
- The demands coming into the core are by themselves bursty
- Read from q_head when idle (somewhere in the middle) - 10% ⬆
- Hard Testing - 4Tbps
- Implemented Batch Processing - 1.5x improvement

Pipeline Inferences

- Longer pipelines are bad
- Shorter pipelines are unhealthy.
- **Pipeline Inequality**: At any point in time, if I take a snapshot of the pipeline, Input to the i^{th} core comes from the $(i-1)^{th}$ core. So, throughput of i^{th} core is less than $(i-1)^{th}$ core

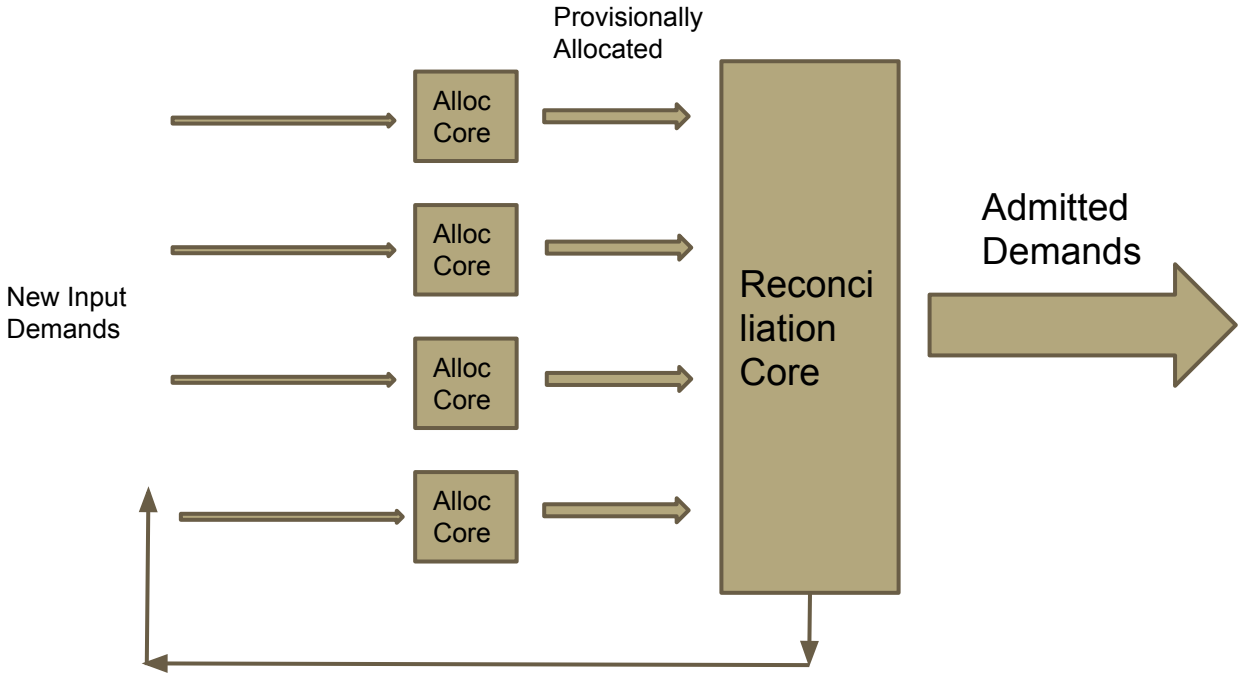
Throughput increases and then decreases with the number of cores. (Maxima at 8 cores for batch size 8)

Break the Rules!! Go Parallel ....

- Multiple cores allocate for same timeslot. Allocate demands in parallel for the same timeslot. **Earlier Pipeline Approach**: Allocate different Timeslots in Parallel
  - Shared Memory
  - Separate Memory
- Need for Reconciliation Core

Quick check using Pipeline - Read from multi-consumer queue. Read in round robin. Why it failed?

Parallel Architecture



Results with Parallel architecture

Hurdles/ variants tried: Shared Memory optimization , prefetching. Multi-consumer multi-producer queues to multiple single-consumer queues. Round robin reads.

- Was scaling linearly till 4 Cores. For 8 and 16 cores things broke down.

For a Batch size of 8 (tested on Ben)

1 core - 492 Gpbs

2 core - 886 Gbps

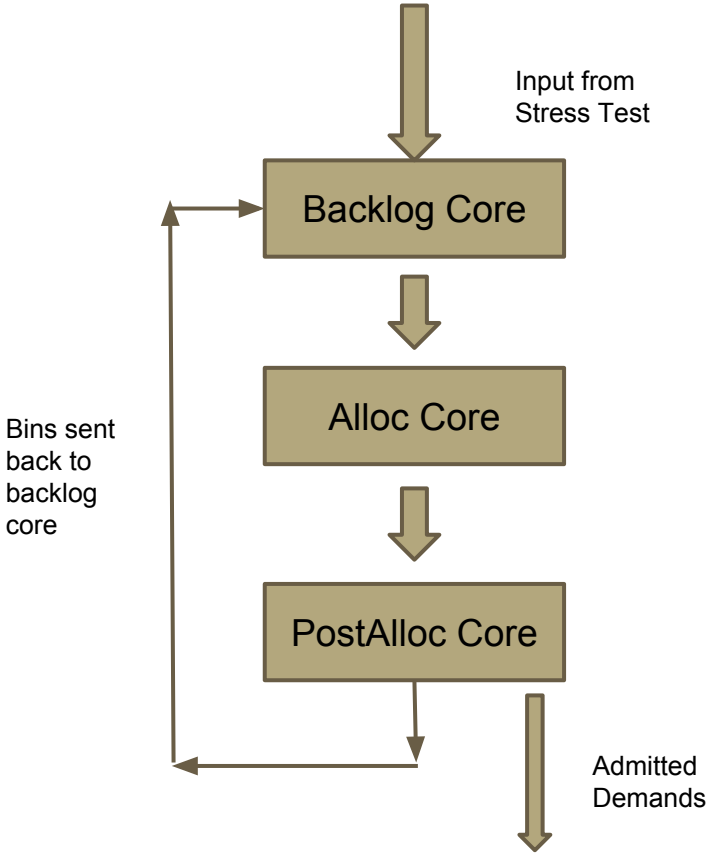
4 cores - 1614 Gpbs

Queues were Costly!!! A core wasn't able to read from more than 4 queues in that designated time.
After a lot of optimization, (bulk enqueues) we scaled to 8 cores but not beyond.
We got to 2.4Tbps for 8 cores.
Vtunes Profiler Helped!!!!

Share the load

Assume we are allocating for a single timeslot.

- Pipeline the work to multiple cores
- Make each core simple
- Assure high throughput
- Set of 3 cores allocate for a timeslot
- Circulate Bins

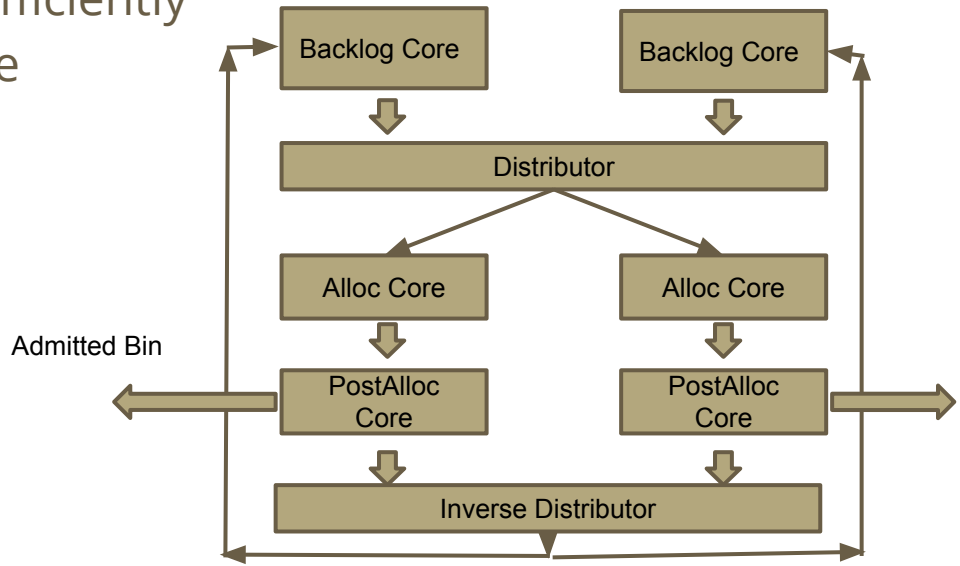


Combining Parallel such sets of cores

Distributor data structure.

- No more queues. We exchange using cache lines.
- Enables you to permute the bins efficiently
- Prefetching improved performance

	Ind 1	Ind 2	Ind 3	Ind 4
Dst 1	Src 3	Src 1	Src 4	
Dst 2	Src 4	Src 2	Src 3	
Dst 3	Src 1	Src 4		
Dst 4	Src 2	Src 3	Src 1	



Implementation Details

- Multiple Stress Test Cores
  - Each responsible for a subset of end nodes
- Multiple Cores in Parallel
- Tackling uneven load across different stress test cores:
  - In backlog core: Fill bins for atleast a predefined amount of time (0.2 microseconds)
  - Fill bins to at max upto (avg number of requests per bin)*1.3 or untill another bin is waiting to be processed
- Empirically find out parameters like - Max Bin Size, etc

Working in sync with Time

- TimeslotDetails structure:
  - Stress test cores sets (or unsets) a bit whenever it is time for resetting the corresponding alloc core
  - Backlog core checks the corresponding bit before sending a bin to a destination alloc core
  - Implemented using atomic bit_set_and_test operations to avoid race conditions
- Identity Distributor between postalloc core and stress test core to send back admitted bins.
- Rewrote Postalloc core with SIMD instructions to process in parallel.
- Better way to pass timeslot information:
  - Alloc cores maintain a local variable with the timeslot number that it is processing for. The backlog cores writes the timeslot number for which the bin has to be processed depending on the destination alloc core.

Fairness within Bins

- Interactive requests may get stalled in bins behind contending requests with huge backlogs

- Send out entries to bins with a maximum of 8 requests. This can ensure intra-bin fairness at a granularity of 8 timeslots.

- Can be done using the backlog structure at the backlog core.

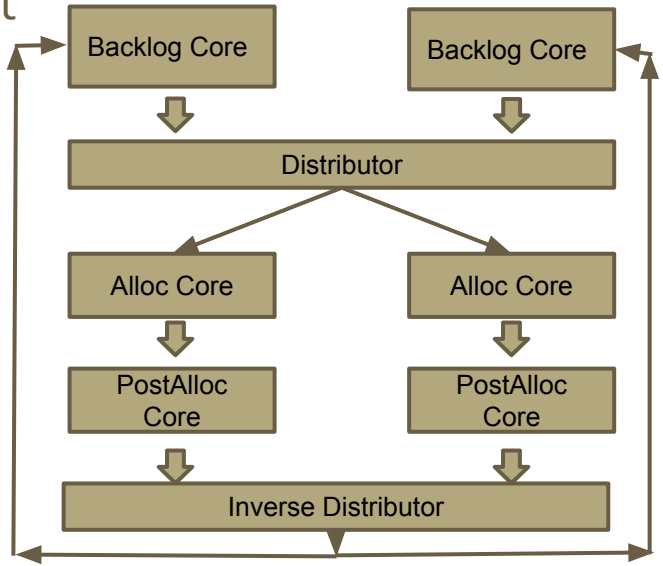
Results

- 2.15 Tbps on 3 cores (Stress test core + Backlog core + Alloc core + postalloc core)
- 4 Tbps with 2 such sets of 3 cores (6 cores)
- 7.1 Tbps with 4 such sets of cores (12 cores)

- From 6 cores to 12 cores, we do not get linear scaling. 12 Cores would span across 2 NUMA nodes on Ben and the inter-node-communication through QPI slows us down

Benchmark for distributor

- 3 cores - backlog and postalloc cores just read and write the bins without working on then
- Alloc cores do some work for a small amount of time - so that prefetching works properly
- Conducted experiments across 3 variables:
  - Number of alloc cores
  - Number of bins in circulation
  - Number of entries per bin



Benchmark Results

- Existing distributor results:
  - 1 alloc core: 0.12 microseconds per bin
  - 2 alloc cores: 0.2 microseconds per bin
  - 4 alloc cores: 0.5 microseconds per bin
  - 8 alloc cores: 0.964 microseconds per bin
  - 16 alloc cores: 1.468 microseconds per bin
- After optimization
  - 1 alloc core: 0.056 microseconds per bin
  - 2 alloc cores: 0.188 microseconds per bin
  - 4 alloc cores: 0.373 microseconds per bin
  - 8 alloc cores: 0.920 microseconds per bin
  - 16 alloc cores: 1.282 microseconds per bin

Part 2: ECN Based Congestion Control for a Software Defined Network

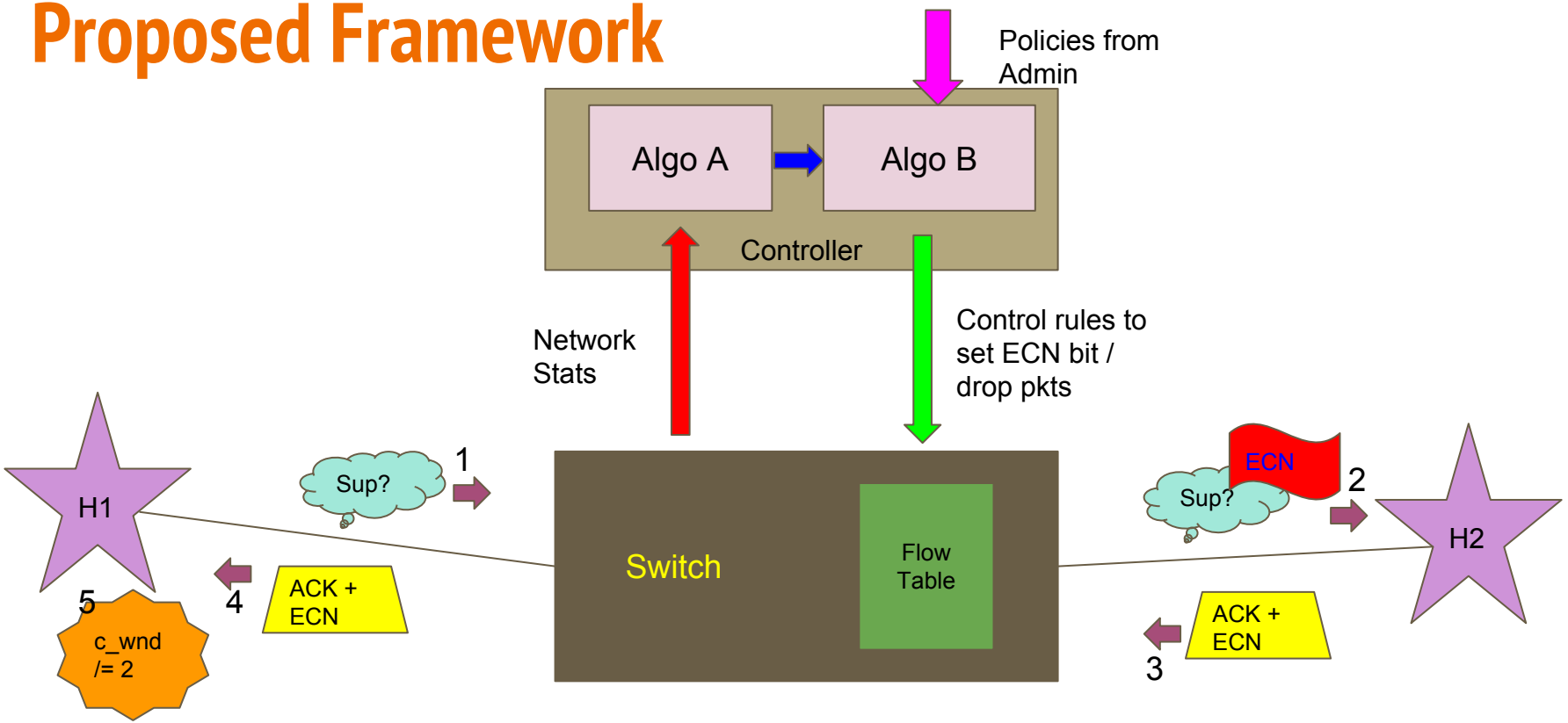
What is Congestion Control?

- When should an end node send a packet to ensure efficient and fair sharing of network resources
- Most algorithms developed are end to end.
- Some algorithms: In-network elements (like routers) assist end nodes in congestion control decisions

Problems with TCP

- Use packet drops or packet delays to identify congestion
- Each end node greedily tries to optimize its objective
- TCP is designed to operate on a wide variety of networks - “Jack of all trades, Master of None”

Proposed Framework



Salient Features of our Framework

- Global View: Predict congestion better
- Prioritization in Congestion Control: Penalize some flows more
- Maximize global objective rather than local greedy objectives
- No change to end nodes
- No change to switches
- Ensure fairness
- Easily pluggable congestion control algorithms
- Congestion control policies by Admin
- Controller assisted end to end system
- Apt for SDN

Proposed Algorithms

- Algorithm A
  - To detect the congestion in the network
  - Collect port stats every 2 seconds
  - if link utilization > 75%, inform Algorithm B

- Algorithm B
  - If link utilization > 75%, start penalizing the top “T%” flows
  - We linearly increase “T” with link utilization
  - T=50% at 100% link utilization

Advantages: Prevents saw-tooth type behaviour, prioritize interactive traffic

Implementation Details

- Mininet
- Floodlight SDN controller with OpenFlow V1.3
- Implemented as an application in Floodlight
- Evaluated against TCP CUBIC of Ubuntu 14.04 Kernel and TCP RED, TCP ECN of mininet
- All the links are 100 Mbps links

Result Summary

- 10% better throughput than TCP Cubic for bulk flows. Throughputs at par with ECN, RED
- Throughputs of bulk flows are closer to fair share values
- 30x better flow completion time to TCP Cubic on interactive flows
- In 2 evaluated scenarios of single hop topologies, consistently performed better than RED and ECN with respect to flow completion times
- In multi hop scenario, performed 1.5x, 1.7x better than ECN,RED

Evaluations - Throughput with bulk flows

- Proposed algorithm does not compromise on throughput of bulk flows

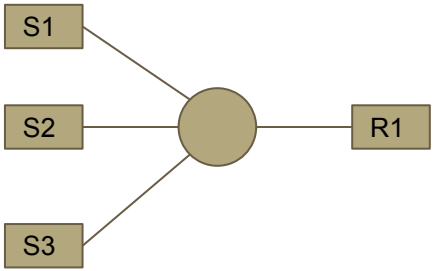


TABLE I
THROUGHPUT OF BULK FLOWS FOR TOPOLOGY 1 (MBPS)

Flow	TCP Cubic	ECN	RED	Proposed
S1	32.5	29.6	30.9	32.7
S2	25.8	29.4	28.1	29.7
S3	26.0	30.2	32.2	28.7
Total	84.3	89.2	91.2	91.1

Evaluations - Throughput with bulk flows

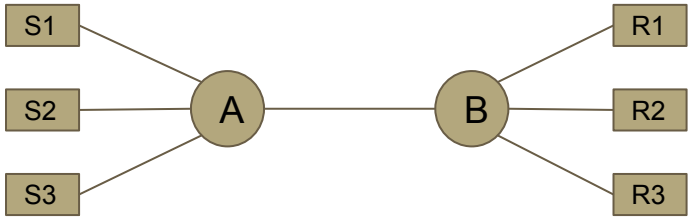


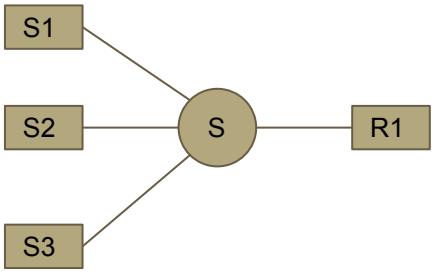
TABLE II
THROUGHPUT OF BULK FLOWS FOR TOPOLOGY 2 (MBPS)

Flow	TCP Cubic	ECN	RED	Proposed
S1	27.5	31.3	32.9	31.3
S2	33.3	29.9	30.9	31.9
S3	23.3	32.9	32.1	32.5
Total (Mbps)	84.1	94.1	95.9	95.7

Fairness of the proposed algorithm is better than Cubic and is at par with RED, ECN

Evaluations - Coexistence of interactive and bulk traffic

- Proposed algorithm improves flow completion time of interactive traffic in presence of bulk traffic
- (S1,R1), (S2,R1) are bulk flows using *iperf* . (S3,R1) - 2 MB interactive traffic

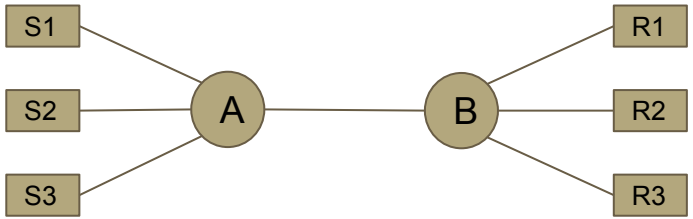


TCP Cubic	ECN	RED	Proposed
10.64	0.36	0.41	0.34

- 30x improvement over TCP Cubic, at par with ECN and 1.2x improvement over RED

Evaluations - Coexistence of interactive and bulk traffic

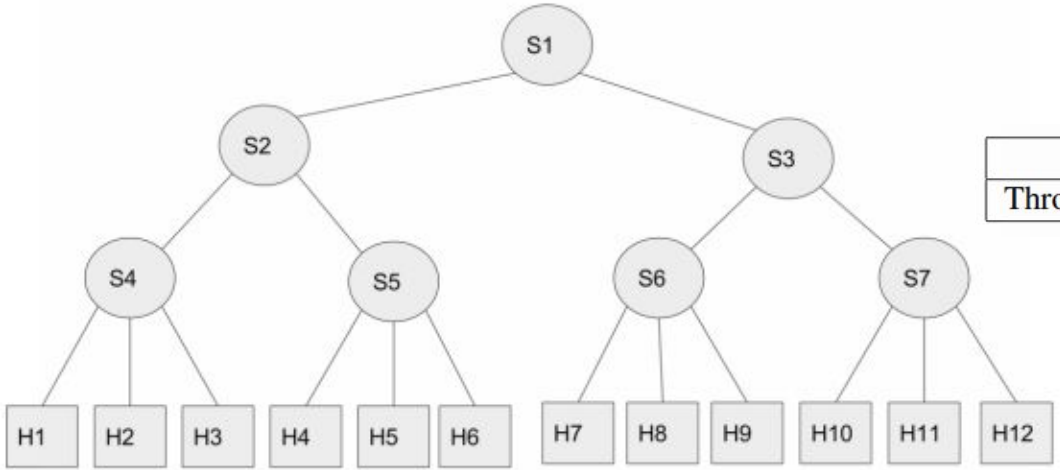
- (S1,R1) and (S2,R2) are the bulk flows generated through *iperf*
- (S3,R3) is a 2 MB interactive flow



TCP Cubic	ECN	RED	Proposed
10.41	0.46	0.37	0.37

- 28x better than TCP cubic and 1.25x better than ECN
- Outperforms TCP cubic and performs best in both scenarios

Evaluations - Multi hop scenario

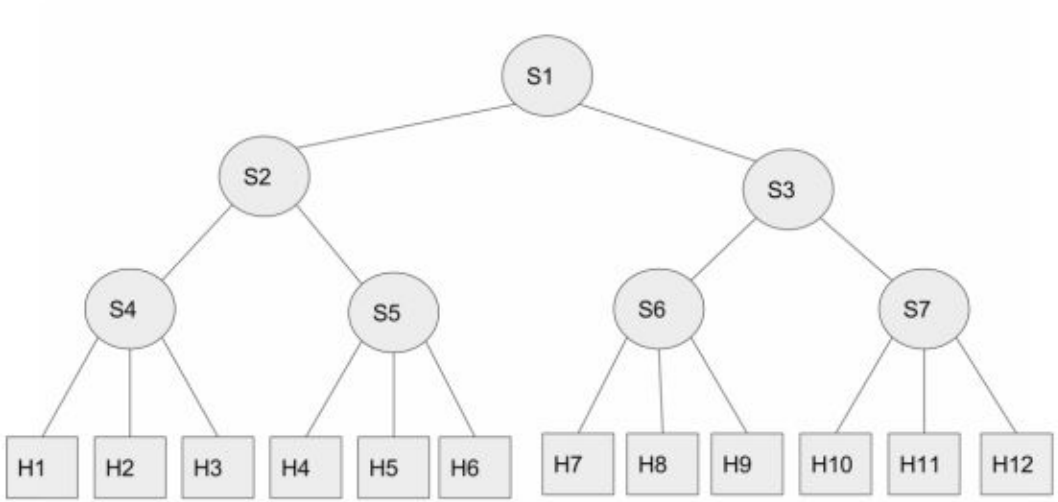


- (H1,H7), (H2,H8), (H4,H10), (H5,H11) are the bulk flows.

	TCP Cubic	ECN	RED	Proposed
Throughput	76.7	83.9	85.4	83.8

- Bulk flow throughputs are not compromised

Evaluations - Multi hop scenario



- (H1,H7), (H2,H8), (H4,H10), (H5,H11) are the bulk flows.
- (H3,H9) and (H6,H12) are 2 MB interactive flows.

TCP Cubic	ECN	RED	Proposed
19.46	1.01	1.1	0.64

- 30x better than TCP Cubic, 1.5x better than ECN and 1.7x better than TCP RED

Thank You

Logging and Debugging

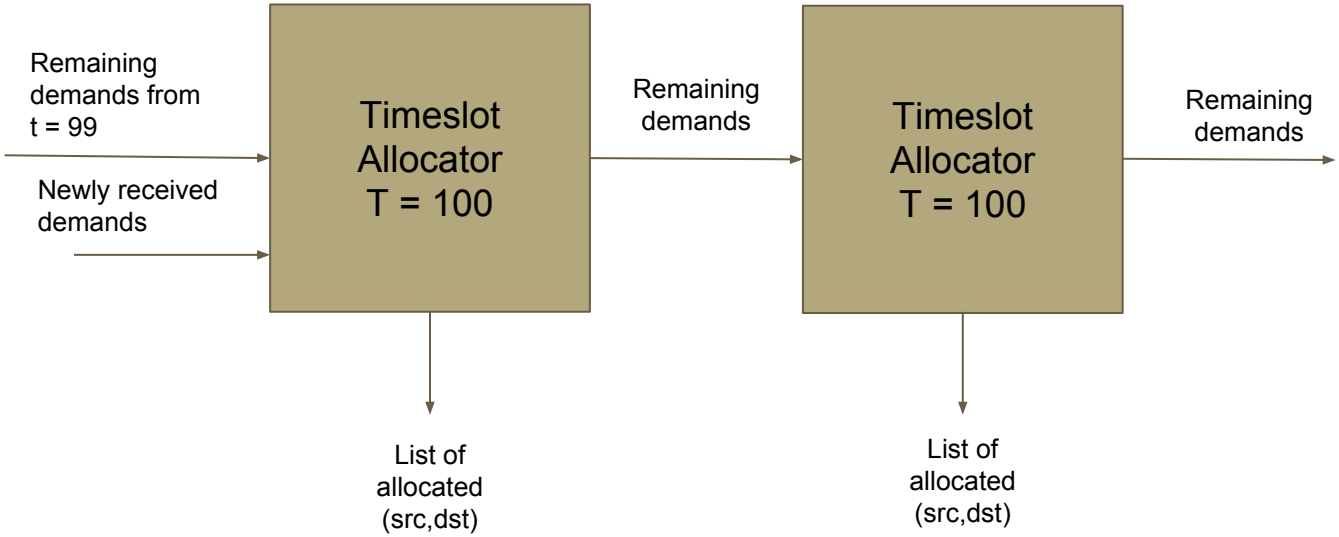
- As our system is performance critical, we shouldn’t add the the burden of printing logs to our working cores

- Separate *log cores* that log and print different program variables

- wait_time_counters to detect deadlocks and bottlenecks

Another attempt for a better distributor

- Efficient single producer single consumer buffer queue available
  - Prefetches next entries based on time of access
- Implemented the distributor with these buffer queues
- A similar Benchmark for these buffer queues
  - works better than our earlier distributor for small bin sizes (4 entries)
  - Not efficient when bins are larger than one cache line (order of microseconds for 256 entry bin)
- Reason:
  - The buffer queue involves copying of the bin entries which slowed it down
  - The time of access can be random as we do a random permutation in the distributor and the prefetcher may not work well


Remaining demands from t = 99
Newly received demands
Timeslot Allocator T = 100
Remaining demands
Timeslot Allocator T = 100
Remaining demands
List of allocated (src,dst)
List of allocated (src,dst)