Iroko

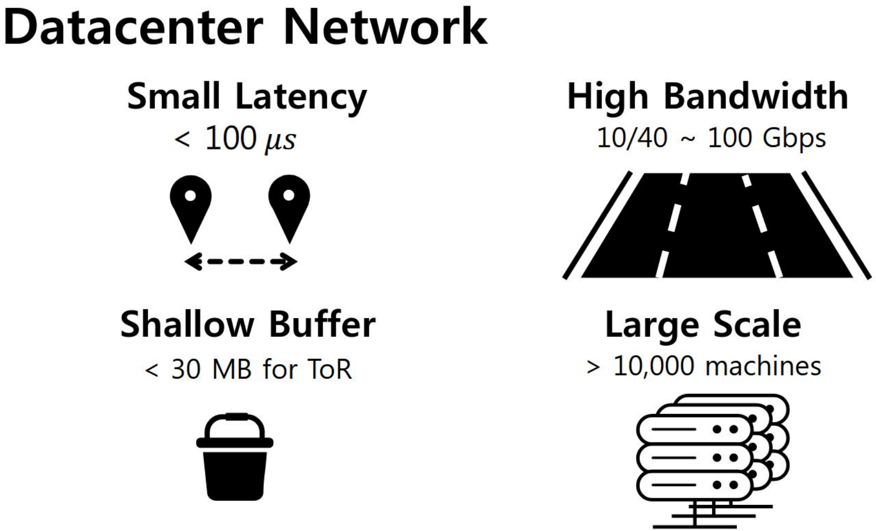
Predictive Datacenter Congestion Control

Congestion Control in Data Centers

- Ideal data center should have:
  - Low latency.
  - High utilization.
  - No packet loss or queuing delay.
  - Fairness.

- Most congestion control and flow scheduling variations are based on TCP.
  - Prioritizes **fairness** and **utilization**.
  - End hosts receive feedback based on round-trip latency and packet loss.
  - Hosts gradually adjust to “**fair**” network rate.

Problem



- Data centers are continuously improving

- TCP is a fundamentally **reactive** protocol.
  - When TCP's backoff kicks in, network conditions are already suboptimal.
  - TCP's convergence rate for small and medium sized flows can not keep up.

- A reactive protocol can not prevent queue buildup and bursts.
  - **Queueing latency** dominates.
  - Frequent retransmits reduce **goodput**.
  - Data center performance may be **unstable**.

Current research insights

- Admission control can have value over burst-and-backoff.
  - **ExpressPass** (2017): Network-enforced credit packet protocol.
  - **FastPass** (2014): “Zero-Queue” network managed by a central arbiter.

- Centralized traffic control solutions may be able to **scale** sufficiently.
  - Datacenter controllers have sufficient computing power at disposal (FastPass, 2014).
  - Benefit of **global traffic knowledge** amortizes RTT cost (Hedera, 2010).
  - **Compartmentalization** enables localized traffic management (Jupiter, 2015).

- **Preemptive** scheduling is possible in data centers.
  - Certain traffic patterns are **predictable** (MicroTE, 2011; Data Centers in the Wild, 2010).
  - Congestion resource management can be **trained** (Remy, 2014).

Our Idea

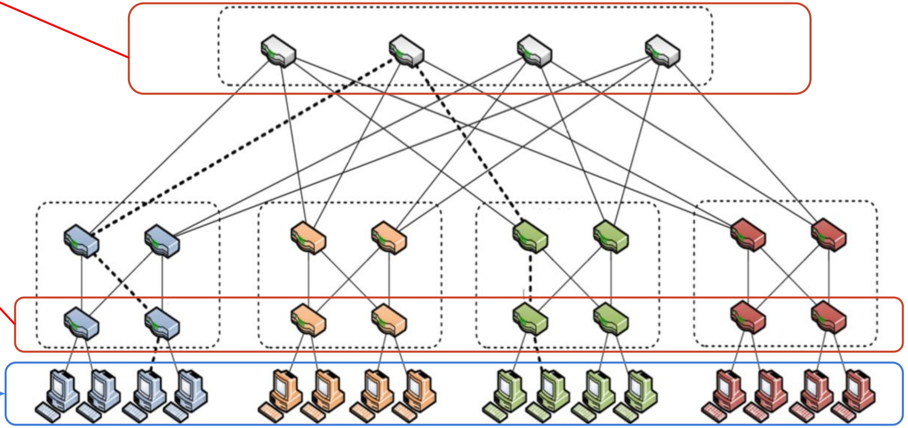
- What if we build an “**intelligent**” traffic arbiter?
  - **We** control the sending rate of hosts.
  - Arbiter **decides** bandwidth allocation per host.
  - Continuously **learns** on traffic feedback.
  - Tries to **predict** sending behaviour.
  - Schedules **future** host tx rate.
- Three research questions:
  - How would such a controller look like?
  - Is it really possible to analyze and predict data center traffic?
  - How does this design compare to more conservative solutions?

A traffic controller.

Iroko

Architecture

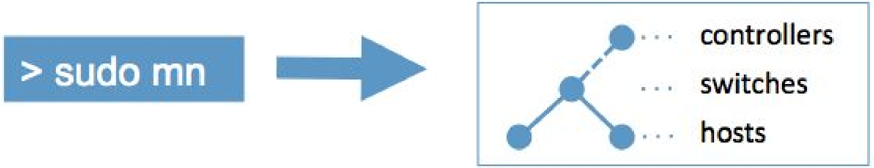
receive
information

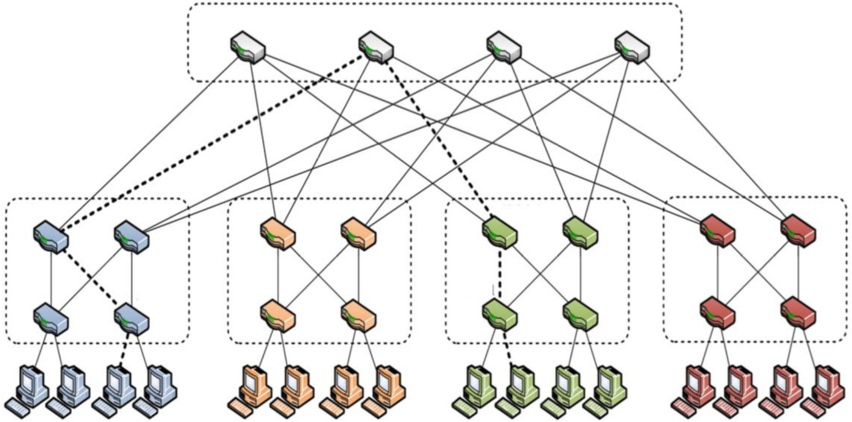


Send packet to update
bandwidth

Implementation: Topology Simulation

- Mininet:
  - Instant Virtual Network
- Fat Tree Topology:
  - 20 switches
  - 16 hosts
- Traffic Simulation:
  - Use Hedera benchmarks





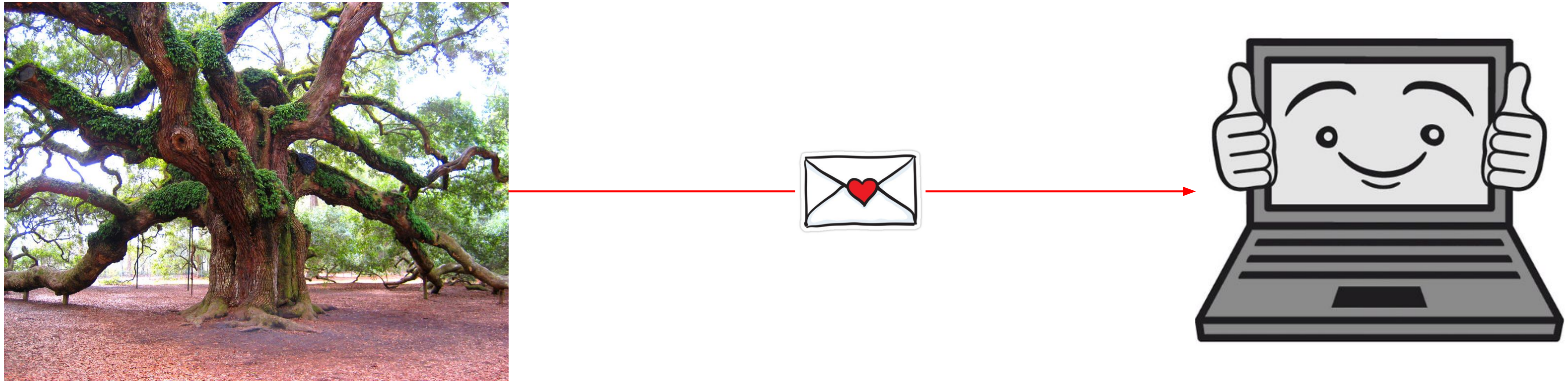
Implementation: Data Acquisition

- Decide to use data being collected from the connections between interfaces
  - Interfaces known in advance
- Use linux commands to get information directly:
  - tc qdisk & awk
- Allows us to collect usable information of network:
  - bandwidth, free bandwidth, drops, overlimits, and queues

```
cmd = "awk \"/^ *%s: / \"\' { if ($1 ~ /.*:[0-9][0-9]*/) { sub(/^.*:/, \"\") ; print $1 } else { print $2 } }\' /proc/net/dev" % (
    iface)
try:
    output = subprocess.check_output(cmd, shell=True)
except:
    print("Empty Request")
    output = 0
```

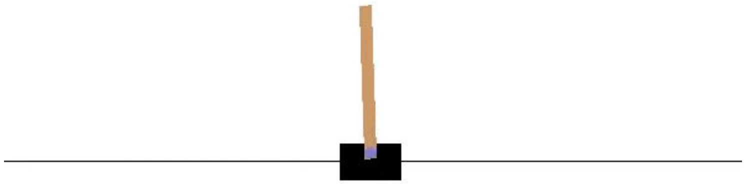
Implementation: Communication with hosts

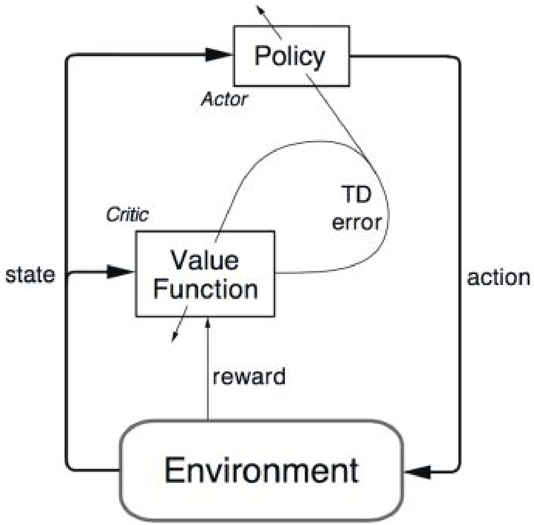
- Arbiter needs to be able to control hosts bandwidth
  - Otherwise it isn't all powerful
- Logic is implemented in application layer
  - C++ communicating via python sockets
- High Level View:
  - Controller sends packet ⇒ packet tells host how much traffic can send out



Implementation: Learning Model

- Prototype idea is simple temporal difference update:
  - $V(s) = V(s) + A_{learningRate} * (R + V(s') - V(s))$
  - R = modifier based on loss
- Goal Implementation:
  - Actor-Critic Model
  - Use loss rate & utilization as indication of success





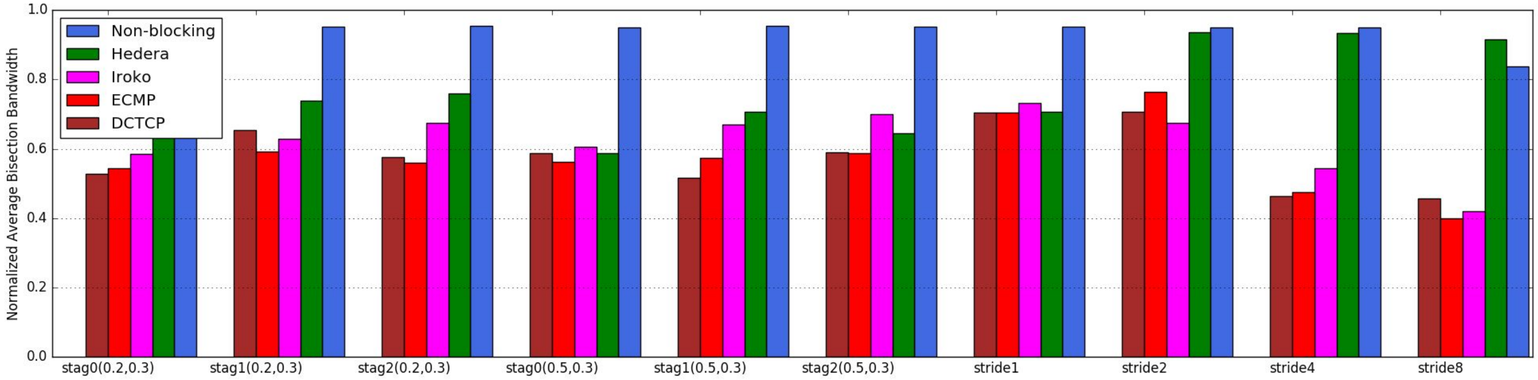
Evaluation & Methodology

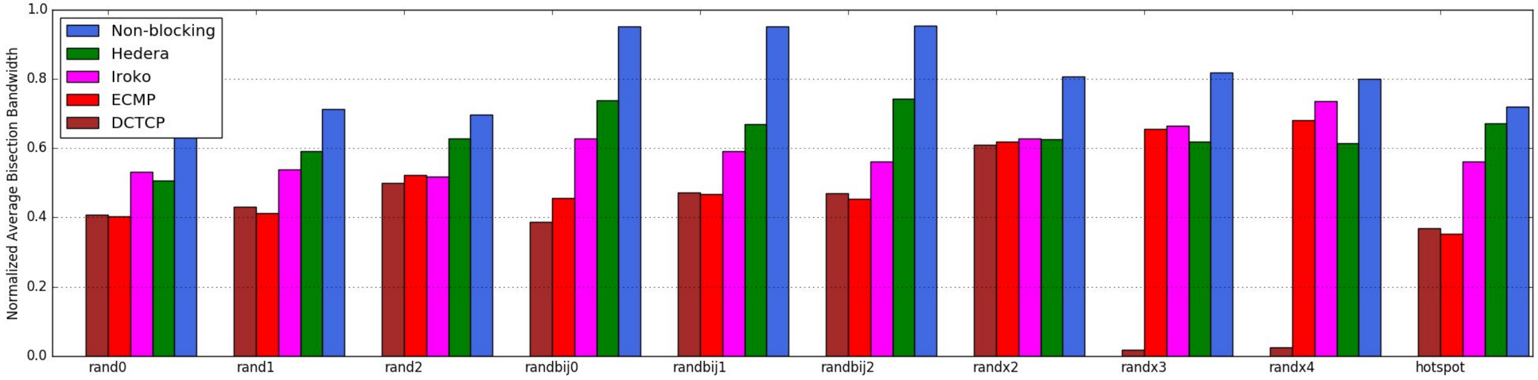
- Metrics we are mainly interested in
  - Utilization
  - Packet drop rate
  - Queuing delay
  - Fairness
  - Starvation
- Target environment
  - Mininet simulation of small data center of 16 hosts and 20 switches with FatTree topology
  - ECMP enabled switches for load balancing
  - Assume switches support OpenFlow
- Comparison with Hedera, DCTCP and ECMP
  - TCP Cubic for Hedera and ECMP
  - Using UDP traffic for Iroko. Rate limiting implemented at the application layer

Iroko vs ECMP vs Hedera vs DCTCP

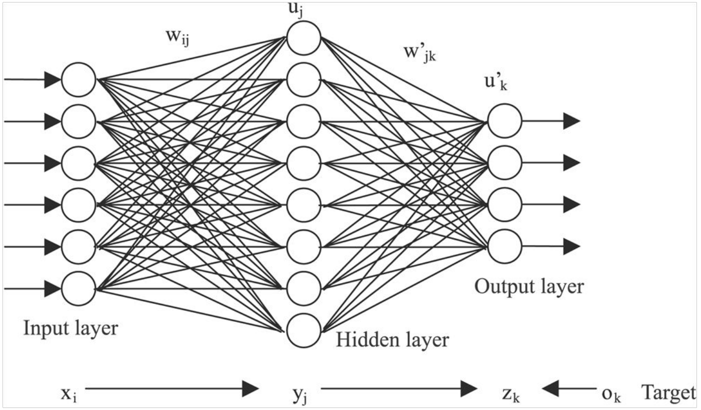
- ECMP
  - Switch hashes the <src ip, src port, dst ip, dst port, protocol> to a port
  - Packets belongs to same flow routes to same port
- Hedera
  - Central controller pulls traffic information from the switches
  - Identifies “elephant flows” and assigns dedicated path
- DCTCP
  - TCP optimized for data center
  - Congestion window size back off based on ECN
  - Marks packets passing through a link with utilization more than a threshold

Preliminary Results (Utilization)





Discussion



- ML is a tricky business:
  - What **learning model** to choose?
  - What **parameters** to choose?
  - **Convergence takes time**; need to learn for a long time to see effects that repeat over long time scales
  - Are we chasing our own tail?

- Is datacenter traffic actually **predictable**?
  - Elephant flows might not need to be predicted. Hosts can report.
  - Short lived flows might be very hard to predict and may need finer grained epochs.
  - Is datacenter traffic ergodic (statistical properties can be deduced from a sufficiently long random sample) ?

Discussion Continued

- Scaling?
  - **Simulation network is small**
  - How does learning scale?
  - Learning space is huge

- Eval Limitations:
  - What is the deciding factor in the performance (many variables)
  - Simulation is on local machine; **OS scheduling and utilization effects** come into play
  - Reproducibility and correctness of the implementation

- Topology and Traffic Matrix
  - Currently using a (small) fat-tree topology
  - **Predetermined traffic matrices**. May not be representative of real traffic
  - How do the techniques translate to other topologies and different traffic matrices

Future Work

- Add explicit **host feedback**
  - Hosts can actively request more data
  - Hosts can send **predictive requests** e.g. Map-Reduce job is expected to finish shortly
- Learning Improvements
  - Learn on t*epoch_length; predict and update hosts on (1-t)*epoch_length where t in (0,1)
  - Learn on individual flows rather than hosts
- Use a **token based system** to allocate bandwidth
  - Allows hosts to makes choices about when to use bandwidth
  - Can predict hosts anticipated future bandwidth needs by tokens kept in reserve by hosts
  - Host can cooperate - use techniques from game theory to organize
- Different routing strategy and traffic matrices
  - Currently relying on ECMP for routing; explore other routing options
  - Acquire actual datacenter traffic to test on