Responsive Multipath TCP in SDN-based Datacenters and Dynamic Scaling of Virtual Network Functions

Duan Jingpu

University of Hong Kong

jpduan@cs.hku.hk

Feb. 2, 2015

Overview

1. First Section
   - Responsive Multipath TCP in SDN-based Datacenters
2. Second Section
   - Dynamic Scaling of Virtual Network Functions

- Section 1: Responsive Multipath TCP in SDN-based Datacenters.

Background and Motivation

- In datacenter network, content replication, virtual machine migration, and data shuffling in MapReduce tasks generate many large flows.
- These large flows constitute the majority of datacenter traffic.
- Efficient transfer of these large flows is crucial to the performance of a datacenter network.
- But current design of both transport layer protocol and routing scheme is not efficient enough in supporting the high throughput of large flows.

Routing in Datacenter Network

- Datacenter network has redundant paths that connect servers from different racks.

- How to route?

Routing in Datacenter Network

- Rely on ECMP to balance the traffic on different paths.
- But ECMP is not intelligent.
- Two large flows may be routed on the same paths, so none of them can achieve optimal throughput.

Routing in Datacenter Network

- Calculate route using SDN controller, like Hedera.
- More intelligent, can avoid paths collision, constantly re-balance entire traffic.
- But controller needs to constantly polls traffic statistics from all switches. Bad scalability and responsiveness.
- Existing flows may be re-routed to another path. Possible packet reordering and packet loss.

Datacenter Transportation Protocol

- Traditional TCP has been shown to be inefficient in datacenter network.
- People design TCP variants that target the needs of datacenter network, such as DCTCP and D2TCP.
- Their limitation is that they are not multi-path protocol.
- Failed to fully utilize path diversity in datacenter network.

Datacenter Transportation Protocol

- Use multi-path based transportation protocol to exploit available bandwidth.
- Split one TCP flow into multiple subflows.
- So that available bandwidth is more efficiently used.
- How to split?

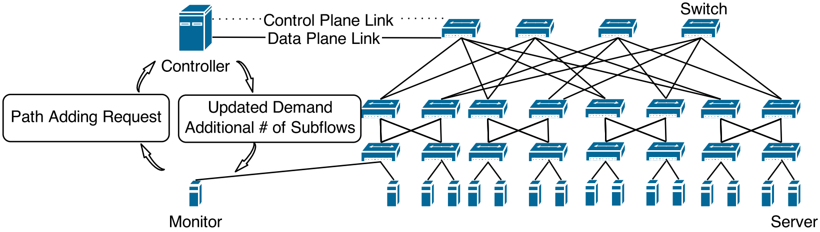
Datacenter Transportation Protocol

- Split at the switch.
- Reply on advanced switch features.
- Hard to deploy in a large scale.

Datacenter Transportation Protocol

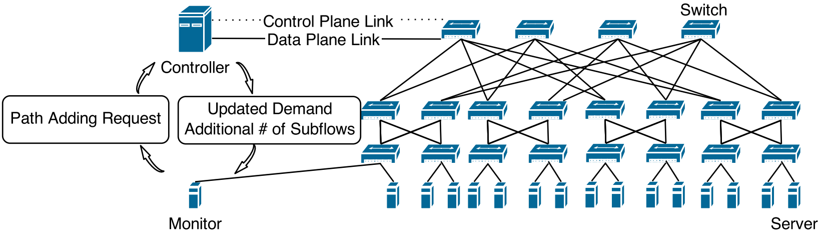
- Split at server, using MPTCP.
- Then flows are still routed using ECMP. We still have all mentioned potential problems.
- Number of subflows used by each flow is fixed. Add unnecessary overhead, can't react to traffic conditions.

Our Work



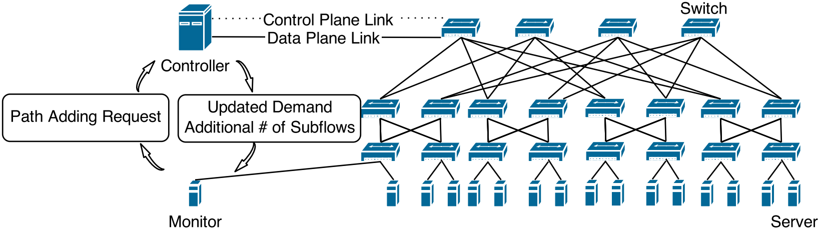
- Responsive MPTCP system for SDN-based datacenters.
- Controller + Monitor design.

Our Work



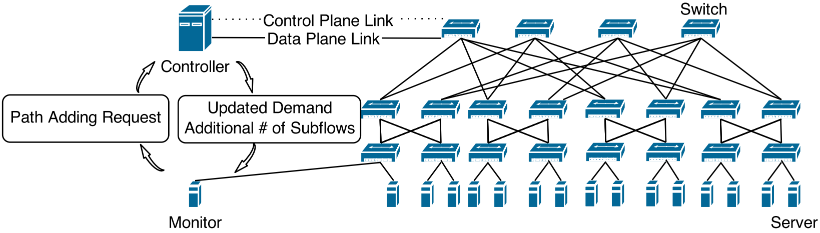
- Controller estimates the demand for each MPTCP flow.
- Controller sends updated demand to the monitor.

Our Work



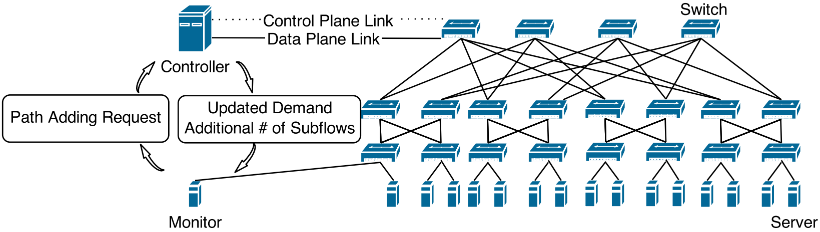
- Monitor records updated demand and current throughput.
- Monitor issues path adding request when necessary.

Our Work



- Controller calculates needed additional subflows.
- Controller sends this information back to monitor.

Our Work



- Dynamically decide the number of subflows to be used by each MPTCP flow and the best subflow path.
- With this system, better throughput, smaller overhead.

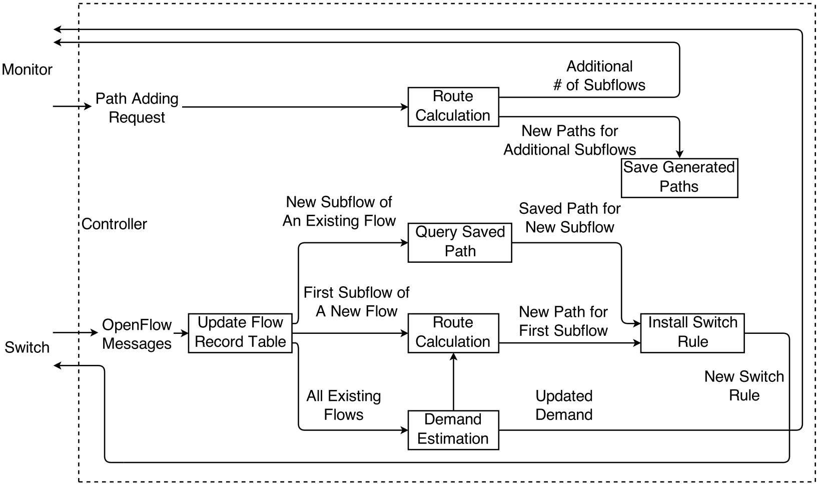


Figure: Architecture of Controller

Controller: Demand Estimation

- Fat-tree datacenter network is a fully non-blocking network.
- Bandwidth bottleneck along a flows path is either senders access link or receivers access link.
- We employ the demand estimation algorithm in Hedera to calculate flow's demand.
    - Proportionally increase the sending rate at sender's access link.
    - Proportionally decrease excessive sending rate at receiver's access link.
    - Stop until all sending rates stablize.

Controller: Demand Estimation

Algorithm: Demand Estimation and Dispatching

Input: demand record *d_t*, new flow *f_n* or expired flow *f_e*

```
old_d_t ← d_t;
update d_t to include f_n or exclude f_e;
run Hedera demand estimation algorithm with d_t as input;
for each flow f in d_t do
    if f is not in old_d_t then
        dispatch d_t[f] to the monitor at sender of f;
    else
        if | d_t[f] − old_d_t[f] | > δ_DDT ∗ old_d_t[f] then
            dispatch d_t[f] to the monitor at sender of f;
        end if
    end if
end for
```

Controller: Route Calculation

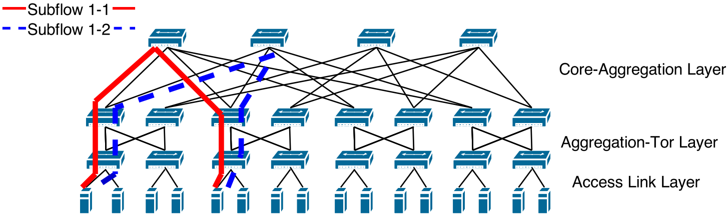
- Route calculation algorithm computes a set of new paths for the flow.
- It uses demand gap as input.
- The goal of route calculation algorithm is to cover as much the demand gap as possible.

Controller: Route Calculation

- Property 1: Aggregate MPTCP Flows Fairly Share Link Bandwidth.
  - When multiple subflows of a MPTCP flow traverse the same link, they are viewed as one aggregate flow.
  - Aggregate MPTCP flows fairly share the link bandwidth, because of MPTCP congestion control protocol.
  - So when n aggregate flows are contending for a link with bandwidth B, the expected throughput of each aggregate flow can be calculated as B/n.

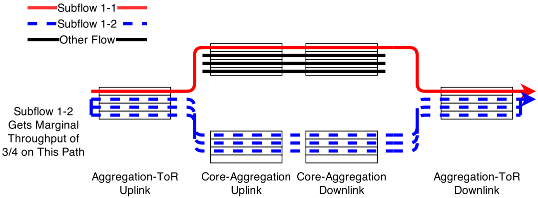
Controller: Route Calculation

- Property 2: Marginal Throughput.
  - Subflows at most Share Links in the Aggregation-ToR Layer, as illustrated in the following figure.



Controller: Route Calculation

- Property 2: Marginal Throughput.
  - Subflows at most Share Links in the Aggregation-ToR Layer.
  - Marginal throughput quantifies the achievable throughput of a subflow on a new path.
  - Especially when new subflow and existing subflows share links on this new path.



Controller: Route Calculation

- We apply the following two rules to calculate the expected throughput of a new subflow *sf* of flow *f* on a given path *p*.
- **Rule 1**: If there is no other subflow of *f* that uses *p*'s aggregation-ToR uplink, find out the largest number of aggregate flows sharing the same link on *p*, n_{max}. The expected throughput of *sf* on *p* is B/n_{max}.
- **Rule 2**: If there are *m* existing subflows of *f* that use *p*'s aggregation-ToR uplink, calculate *p*'s marginal throughput m_t, as well as the upper bound of *sf*'s expected throughput B/n_{max}.

Controller: Demand Estimation

Algorithm: Route Calculation

Input: flow f, demand gap d_g, number of existing subflows n_sf

```
set number of additional subflows n_new_sf = 0;
while n_new_sf + n_sf < M do
    find out a subflow path p with the largest expected throughput e_t;
    if f is a new flow then
        install switch rules on p;
        return;
    end if
    if e_t == 0 then
        return;
    end if
    save path p;
    n_new_sf += 1;
    if e_t > d_g then
        notify the sender monitor of f of n_new_sf;
        return;
    else
        d_g = d_g - e_t;
    end if
end while
notify the sender monitor of f of n_new_sf;
return;
```

Monitor

- Monitor is a daemon program running on each server.
- It keeps track of current throughput of each active flow.
- It constantly receives updated demand from controller.
- It issues path-adding request when it detects a significant gap between current throughput and updated demand.
- Monitor daemon is periodically executed every t_m seconds.

Monitor

Algorithm: Monitor Loop

Input: monitored flows $f[n]$, previous flow rates $p\_r[n]$, counters $count[n]$, flow demand $demand[n]$

for $i = 1 : n$ **do**
    obtain instant throughput i_r for flow $f[i]$;
    $p\_r[i] = 0.2 * p\_r[i] + 0.8 * i\_r$;
    if $| i\_r - p\_r[i] | < \delta_{RVT} * p\_r[i]$ **then**
        $count[i] = count[i] + 1$;
        if $count[i] == R$ **then**
            $count[i] = 0$;
            if $p\_r[i] < (1 - \delta_{DGT}) * demand[i]$ **then**
                $gap = demand[i] - p\_r[i]$;
                issue path adding request for $f[i]$ with gap;
                return;
            end if
        else
            return;
        end if
    else
        $count[i] = 0$;
        return;
    end if
end for

Performance Evaluation

- We evaluate our responsive MPTCP system using a NS3 simulator.
- We simulate a datacenter network with a 8-Ary fat-tree topology which contains 128 servers connected using 1Gbps links.
- Up to 4 subflows per MPTCP connection is allowed.
- Performance under different parameter settings is evaluated.
- Comparison against ECMP and Hedera is conducted.

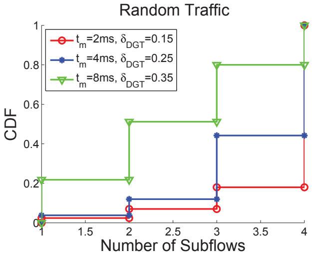
Performance Evaluation

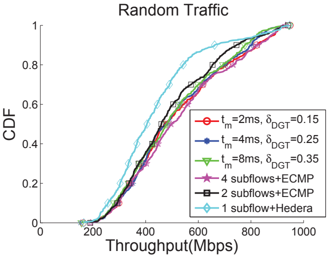
TABLE I: Summary of Experiment Results

	Random Traffic		Permutation Traffic		Shuffling Traffic	
	NS	TP(Mbps)	NS	TP(Mbps)	JCT(ms)	NS
$t_m = 2ms, \delta_{DGT} = 0.15$	3.72	521	3.88	766	1402	1.92
$t_m = 4ms, \delta_{DGT} = 0.25$	3.40	513	3.82	767	1441	1.28
$t_m = 8ms, \delta_{DGT} = 0.35$	2.47	506	3.75	760	1454	1.03
4 subflows+ECMP	4	530	4	734	1332	4
2 subflows+ECMP	2	493	2	643	1394	2
1 subflow+Hedera	1	438	1	747	1652	1

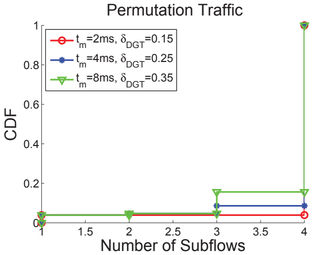
NS: Average number of subflows per flow.
TP: Average throughput per flow.
JCT: Average shuffle completion time per MapReduce job.

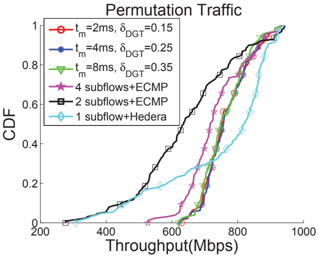
Performance Evaluation



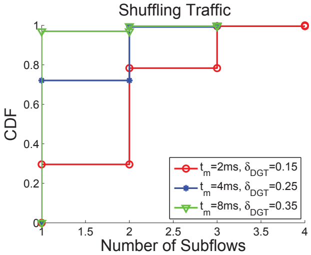


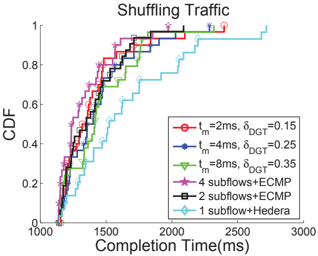
Performance Evaluation





Performance Evaluation





- Section 2: Dynamic Scaling of Virtual Network Functions.

Background

- Network middleboxes, i.e. NAT, Firewall, Proxy, are running everywhere.
- Many of them are usually implemented as proprietary hardware, making them notoriously hard to upgrade and scale.
- With cloud and virtualization technology, network middleboxes with a software-implementation can be run on virtual machines.

Background

- Efforts have been made on improving the performance of NFV system:
  - Increase the processing speed of software network middlebox running on virtual machine.
  - Dynamically scale network middleboxes in face of traffic change.

IMS System

- We start our research by studying a specific network middlebox system: Ip Multi-media Subsystem (IMS).
- We use an open-source implementation of IMS system, Project Clearwater.

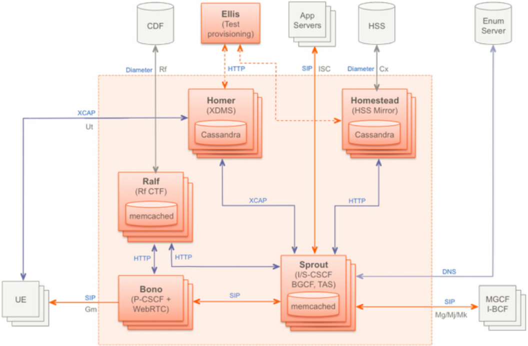
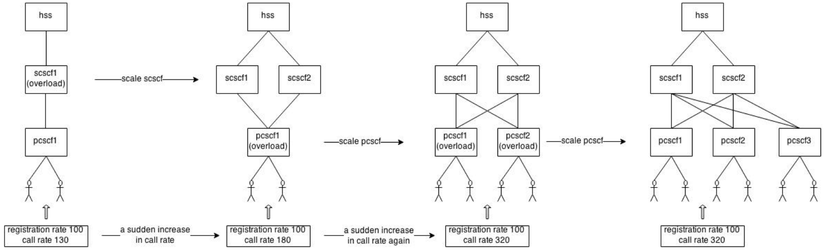


Figure: Architecture of IMS System, from Project Clearwater Website

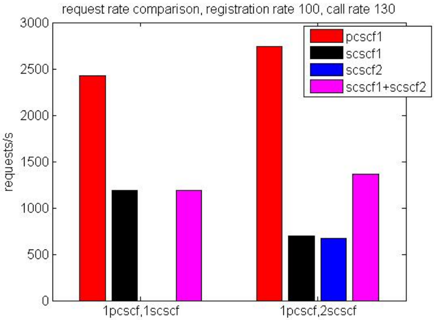
Identifying the Bottleneck

- In order to dynamically scale the system, we need to identify the bottleneck in the system.
- We design the following simple experiment to show the bottleneck of the system.



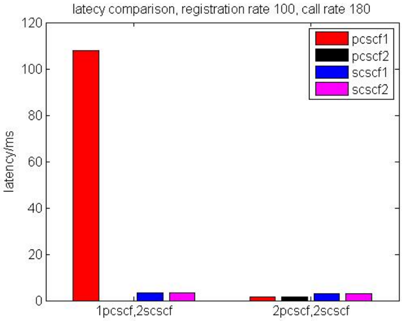
Identifying the Bottleneck

- With registration rate 100 and call rate 130, SCSCF overloads.
- SCSCF performs blocking operations that retrieve user information from database.
- When it overloads, it fails to serve some calls and for some calls to terminate.



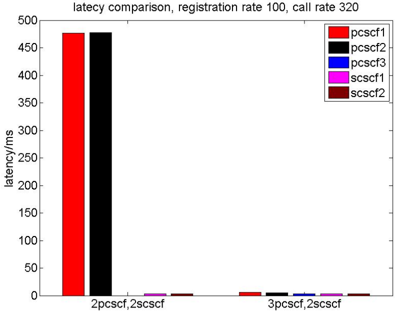
Identifying the Bottleneck

- With registration rate 100 and call rate 180, PCSCF overloads.
- PCSCF performs non-blocking operations, acting as a pure proxy.
- When it overloads, its request processing latency increases drastically.



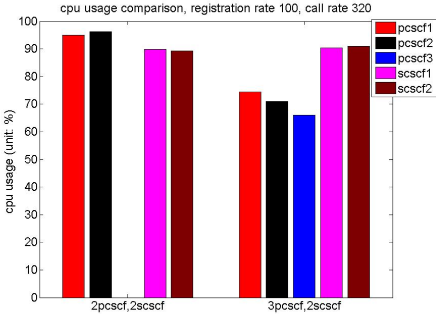
Identifying the Bottleneck

- With registration rate 100 and call rate 320, PCSCF overloads again.
- SCSCF will overload soon, if workload keep increasing.



Identifying the Bottleneck

- With registration rate 100 and call rate 320, PCSCF overloads again.
- SCSCF will overload soon, if workload keep increasing.



What We Learn

- Under different workload, different part of the system becomes bottleneck.
- It's hard to design an accurate mathematical model that takes workload as input and output the number of required resources.

What We Plan to Do

- Monitor the overloading signal. (Latency, CPU usage, Successfully Completed Requests)
- Scale the system in react to the overloading signal.
- After scaling, need to balance the load on each instance.
- Scale up vs scale down:
    - It's easier to scale up than to scale down.
    - Because it's even harder to determine whether you can scale down.

The End

- Future Work:
  - Along the direction of datacenter network, design more efficient datacenter transport protocols with control plane assistance and implement them in real system.
  - Along the direction of NFV, carry on this existing research and find out more interesting from it.

The End

- Thank you and Q & A!