Cloud Computing
Jellyfish in Data Centers

Aditi Sen & Charissa Zou

Paper

- Paper: http://pbg.cs.illinois.edu/papers/jellyfish-nsdi12.pdf

Jellyfish: Networking Data Centers Randomly

Ankit Singla[†*], Chi-Yao Hong[†*], Lucian Popa[♯], P. Brighten Godfrey[†]

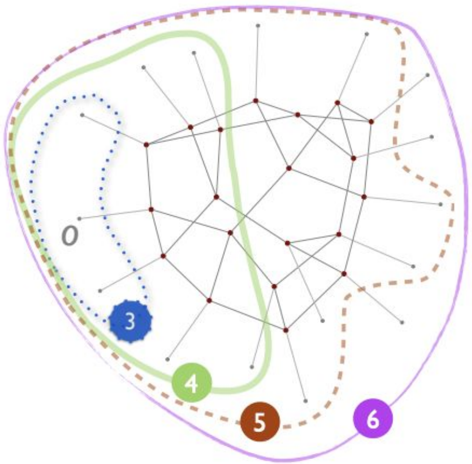
[†] University of Illinois at Urbana–Champaign

[♯] HP Labs

Our Project

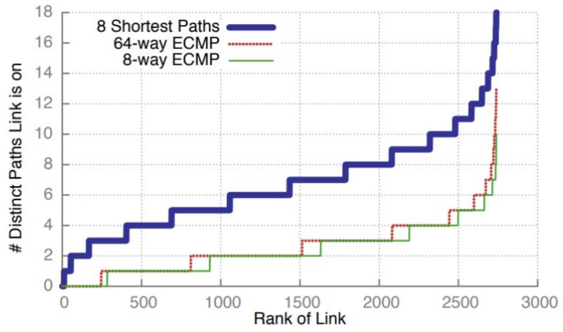
Goals

1. Implement a degree bounded random graph Jellyfish network to simulate data center traffic

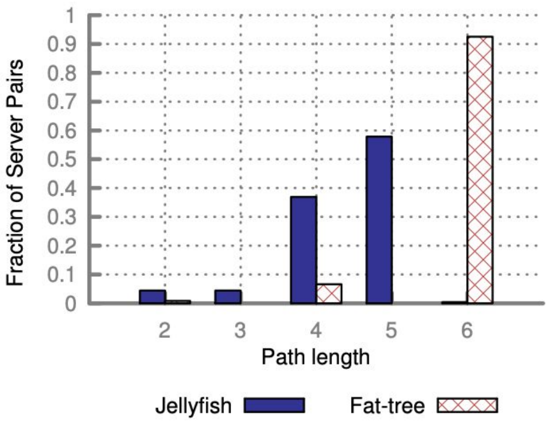


Goals

2. Reproduce Figure 9 in the paper
    - k-shortest path provides better path diversity than ECMP for Jellyfish



3. Reproduce Figure 1c in the paper
    - Jellyfish produces shorter paths on average



Reach Goal

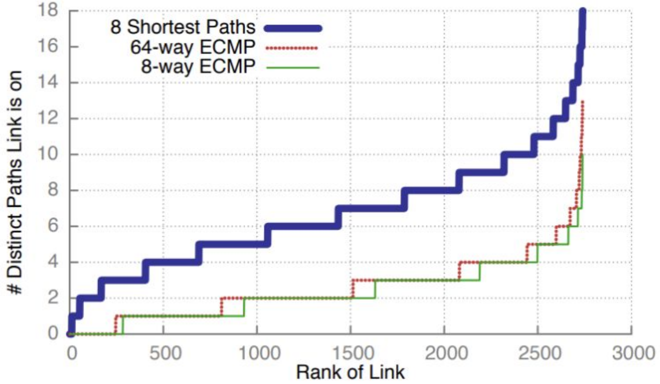
4. Reproduce Table 1 in the paper
   - Compare Jellyfish and Fat Tree throughput under different routing protocols
   - TCP vs. MPTCP
   - With the same switch equipment, Jellyfish supports 25% more servers than fat tree

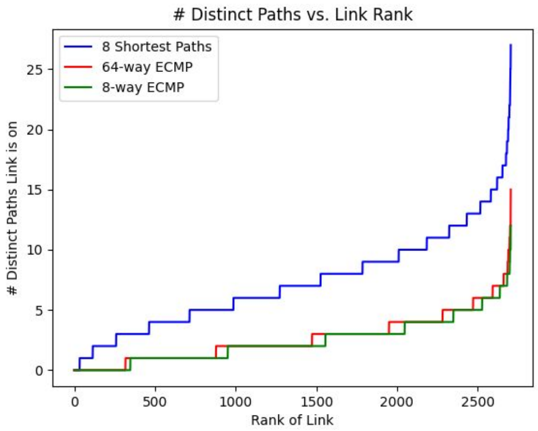
Congestion control	Fat-tree (686 svrs) ECMP	Jellyfish (780 svrs) ECMP	Jellyfish (780 svrs) 8-shortest paths
TCP 1 flow	48.0%	57.9%	48.3%
TCP 8 flows	92.2%	73.9%	92.3%
MPTCP 8 subflows	93.6%	76.4%	95.1%

Progress

Last Time

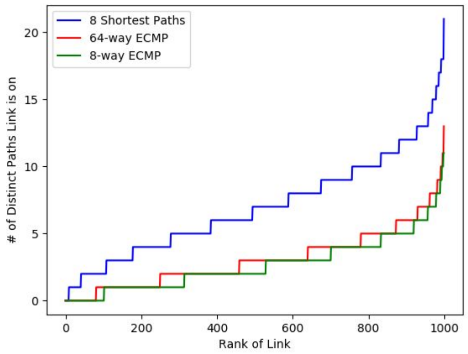
- Implemented a Jellyfish random topology using Networkx in Python
- Simulated “traffic” direction and computed ECMP and k-Shortest Paths



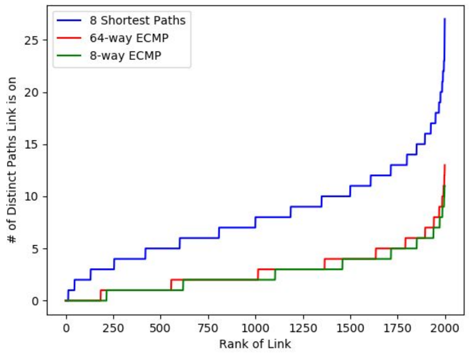


Experiments

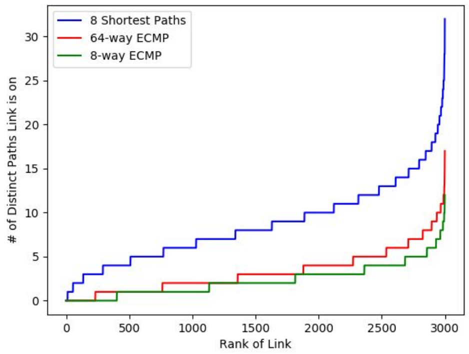
- Ran more experiments with a different number of switches and hosts



100 switches, 10 links



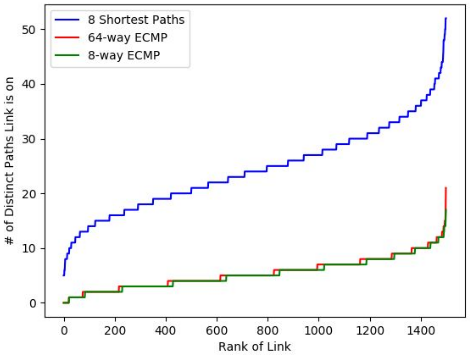
200 switches, 10 links



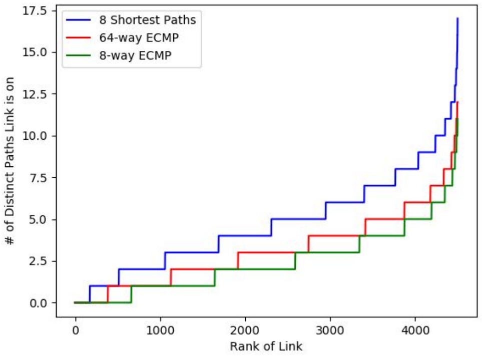
300 switches, 10 links

Experiments

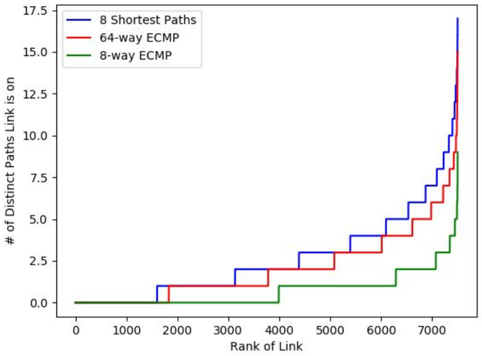
- Ran more experiments with a different number of switches and hosts



300 switches, 5 links

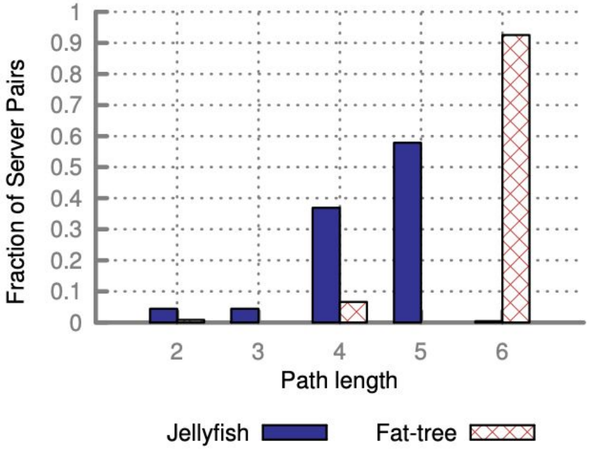


300 switches, 15 links



300 switches, 25 links

Reproducing Table 1



Congestion control	Fat-tree (686 svrs)	Jellyfish (780 svrs)	
	ECMP	ECMP	8-shortest paths
TCP 1 flow	48.0%	57.9%	48.3%
TCP 8 flows	92.2%	73.9%	92.3%
MPTCP 8 subflows	93.6%	76.4%	95.1%

Reproducing Table 1

- Built a Mininet jellyfish topology from Networkx graph

```
# Construct mininet
for n in graph.nodes:
    net.addSwitch("s_%s" % n)

    # Add single host on designated switches
    if int(n) in list(range(host_range)):
        net.addHost("h%s" % n)
        # directly add the link between hosts and their gateways
        net.addLink("s_%s" % n, "h%s" % n)

# Connect switches to each other as defined in networkx graph
for (n1, n2) in graph.edges:
    net.addLink('s_%s' % n1,'s_%s' % n2)
```

Reproducing Table 1

- Tested on Mininet with a pre-built Pox controller
    - forwarding.l2_learning

```
mininet> h0 ping h1
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=7.41 ms
```

```
--- 10.0.0.2 ping statistics ---
376 packets transmitted, 376 received, 0% packet loss, time 383574ms
rtt min/avg/max/mdev = 0.040/0.187/7.407/0.673 ms
mininet>
```

Reproducing Table 1

- Custom topology for Fat Tree
- Implement custom controllers with ECMP and 8-Shortest Paths routing
    - Pox + ripl vs. Ryu + OpenFlow
- Generate traffic flow using pre-constructed traffic matrix
- Measure throughput using iperf
- Parse and generate graphs based on data output

Conclusion

Challenges

- riplpox (Brandon Heller)
  - Version mismatch / permissions issues
- Pivoted to building everything from scratch
  - Mininet permission issues
  - Approach decisions
  - Custom controllers

Looking Forward

- Complete Fat Tree topology construction (for comparison to Jellyfish)
- Implement ECMP and K-Shortest Path routing in custom controllers
- Simulate network traffic in Mininet
    - Measure throughput with iperf
- Generate graphs for network throughput
- Generate graphs for average path length

Takeaways

- Difficult to test theories due to sheer size of applications needed and unavailability of testing frameworks / networks

- Systems programming is hard

Thank you!