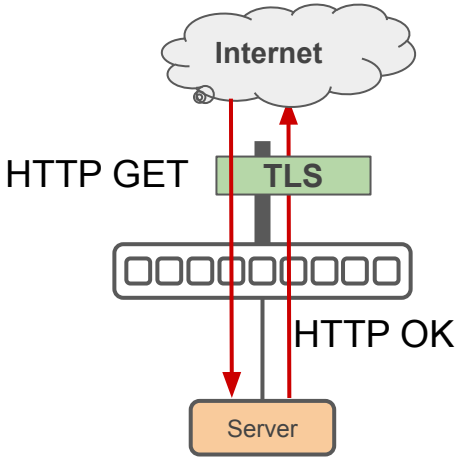
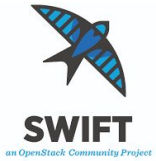
Prism: Proxies without the Pain

Yutaro Hayakawa (LINE Corporation)
Michio Honda (University of Edinburgh)
Douglas Santry (Apple Inc.)
Lars Eggert (NetApp)

Object Storage Systems

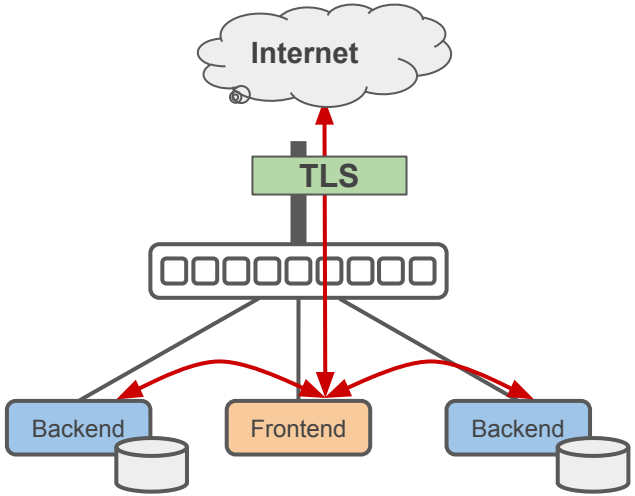




Protocol

S3 Protocol, HTTP/HTTPS etc...

Object Storage Systems



Protocol

S3 Protocol, HTTP/HTTPS etc...

Frontend servers

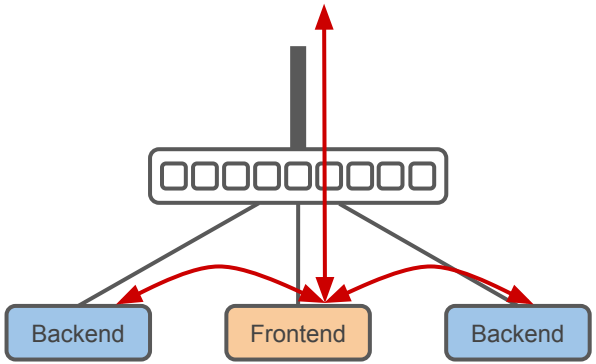
Proxy servers for load balacing, caching, filtering etc…

Backend servers

Storage servers

Object Storage Scale-out Architectures

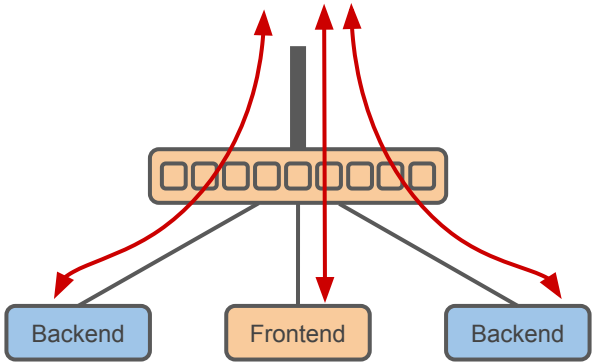
Frontend Proxy



OpenStack Swift, Minio, Mos [HPDC'16], Crystal [FAST'17], etc...

- ✔ Standard protocols (e.g. S3 / HTTP)
- ✔ TLS (Encryption / Authentication)
- ✖ Frontend becomes bottleneck

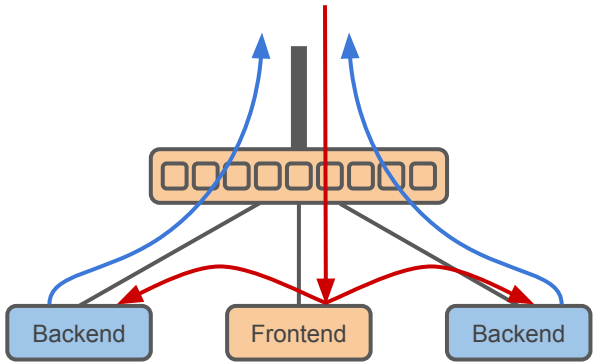
Content-based Routing



SwitchKV [NSDI'16], NetCache [SOSP'17], Pegasus [OSDI'20], etc...

- ✖ Requires special protocols in client-side
- ✖ Cannot use TLS
- ✔ No intermediate servers

Prism

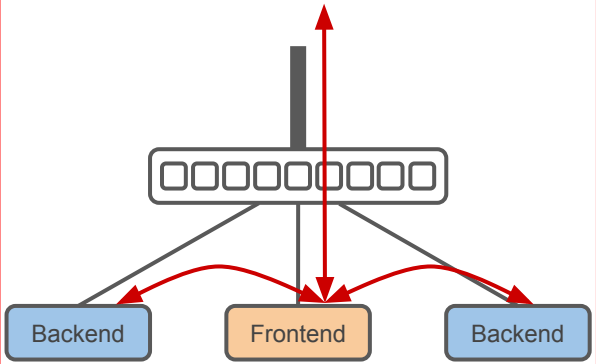


Our proposal

- ✔ Can use standard protocols
- ✔ TLS
- ✔ No intermediate servers (for return path)

Object Storage Scale-out Architectures

Frontend Proxy



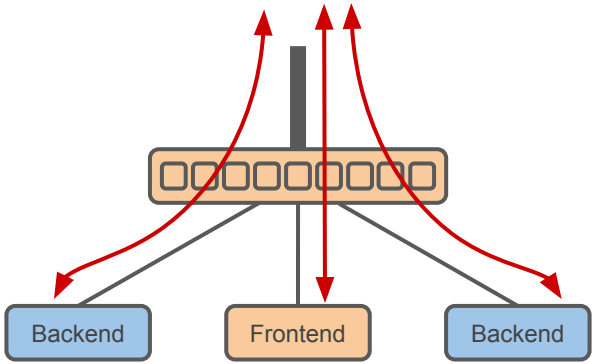
OpenStack Swift, Minio, Mos [HPDC'16], Crystal [FAST'17], etc...

✔ Standard protocols (e.g. S3 / HTTP)

✔ TLS (Encryption / Authentication)

✖ Frontend becomes bottleneck

Content-based Routing



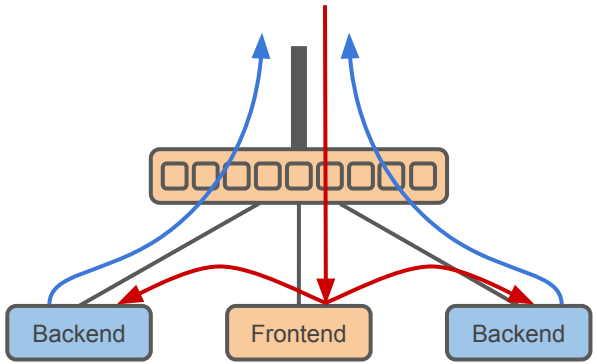
SwitchKV [NSDI'16], NetCache [SOSP'17], Pegasus [OSDI'20], etc...

✖ Requires special protocols in client-side

✖ Cannot use TLS

✔ No intermediate servers

Prism



Our proposal

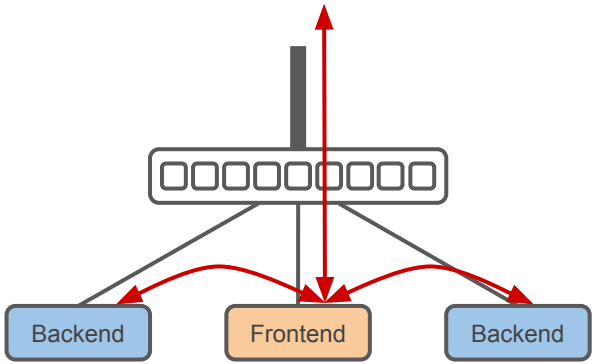
✔ Can use standard protocols

✔ TLS

✔ No intermediate servers (for return path)

Object Storage Scale-out Architectures

Frontend Proxy



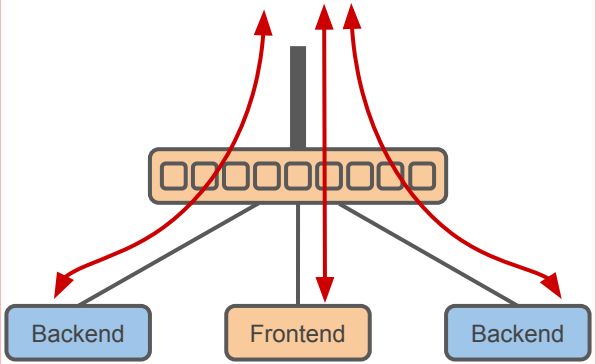
OpenStack Swift, Minio, Mos [HPDC'16], Crystal [FAST'17], etc...

✔ Standard protocols (e.g. S3 / HTTP)

✔ TLS (Encryption / Authentication)

✖ Frontend becomes bottleneck

Content-based Routing



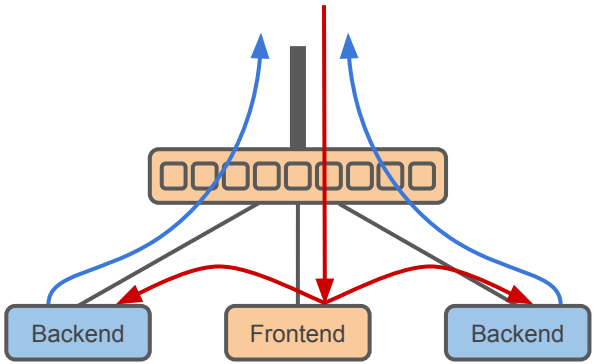
SwitchKV [NSDI'16], NetCache [SOSP'17], Pegasus [OSDI'20], etc...

✖ Requires special protocols in client-side

✖ Cannot use TLS

✔ No intermediate servers

Prism



Our proposal

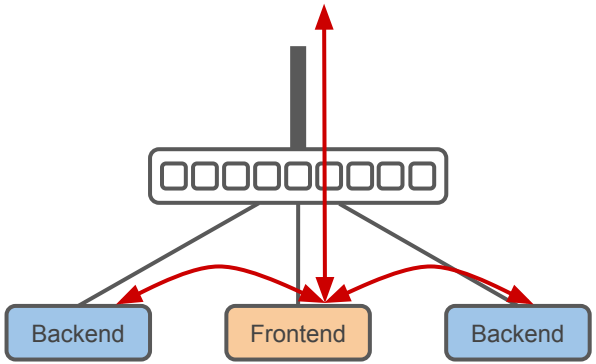
✔ Can use standard protocols

✔ TLS

✔ No intermediate servers (for return path)

Object Storage Scale-out Architectures

Frontend Proxy



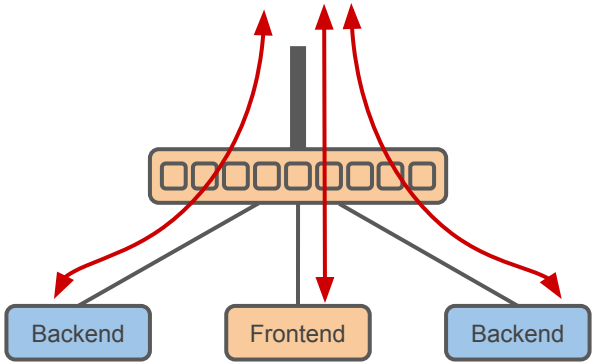
OpenStack Swift, Minio, Mos [HPDC'16], Crystal [FAST'17], etc...

✔ Standard protocols (e.g. S3 / HTTP)

✔ TLS (Encryption / Authentication)

✖ Frontend becomes bottleneck

Content-based Routing



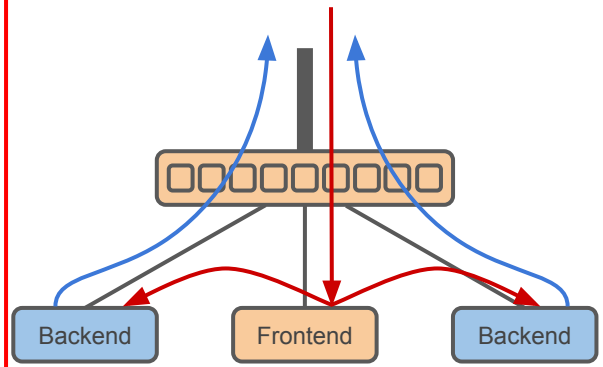
SwitchKV [NSDI'16], NetCache [SOSP'17], Pegasus [OSDI'20], etc...

✖ Requires special protocols in client-side

✖ Cannot use TLS

✔ No intermediate servers

Prism



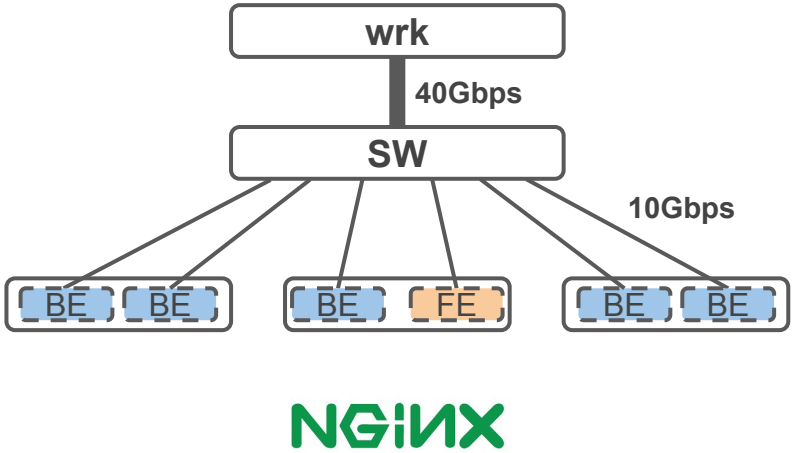
Our proposal

✔ Can use standard protocols

✔ TLS

✔ No intermediate servers (for return path)

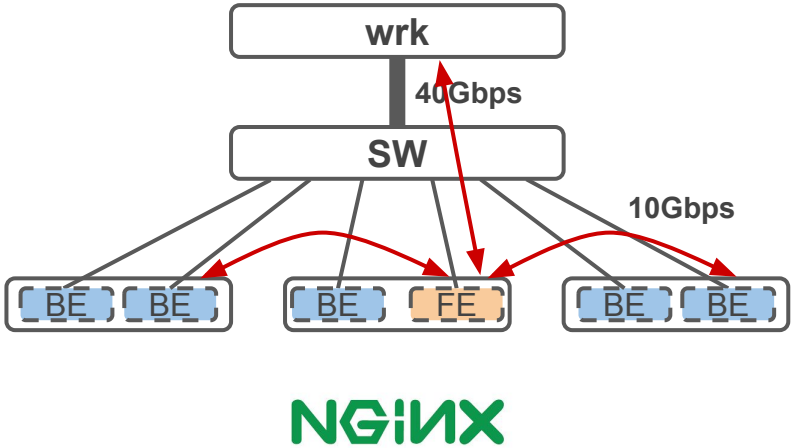
Performance Characterization of Frontend Proxy Architecture



Setup

- Physical Server (4 core, Dual Port 10G NIC)
- BE — Logical Backend Server (Nginx, 2 core, 1 NIC Port)
- FE — Logical Frontend Server (Nginx, 2 core, 1 NIC Port)
- SW — Software L2 switch (Netmap based)
- wrk — Benchmark client (wrk)

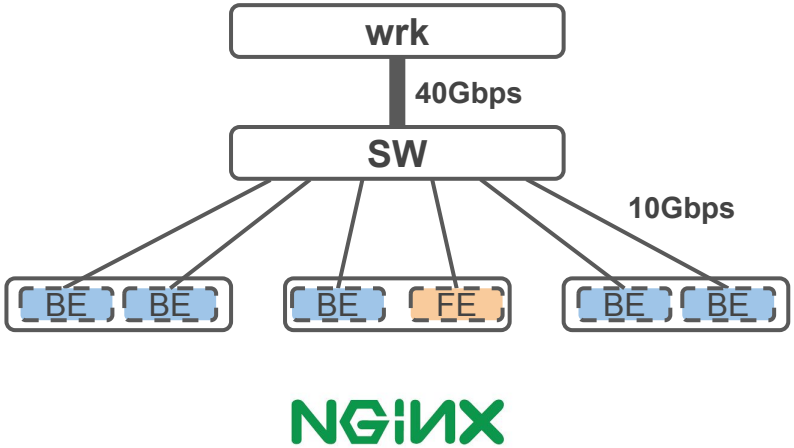
Performance Characterization of Frontend Proxy Architecture

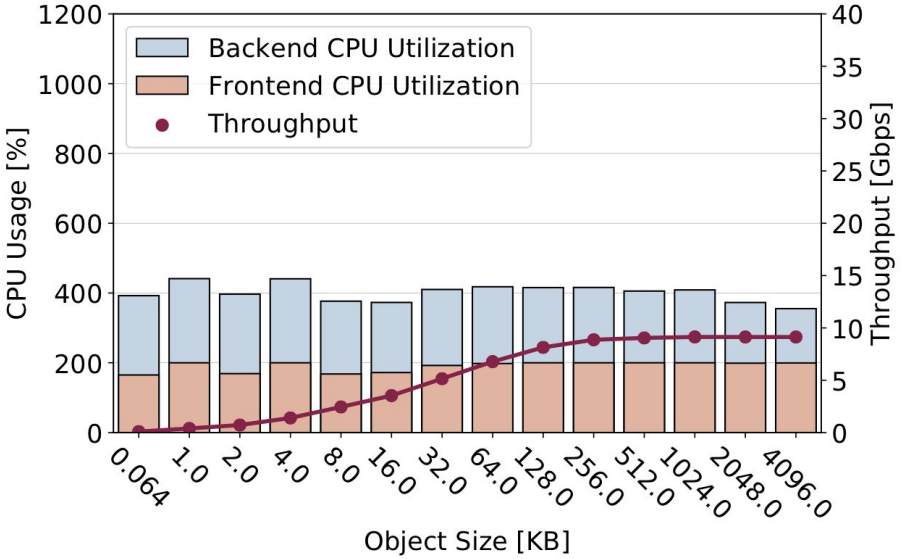


Benchmark

1. S3 protocol with HTTP(S) keep-alive
2. 100 parallel TCP connections
3. Round-robin load balancing on frontend
4. Fetches in-memory objects

Performance Characterization of Frontend Proxy Architecture





Performance Characterization of Frontend Proxy Architecture

Left Y-Axis: Cluster CPU Utilization

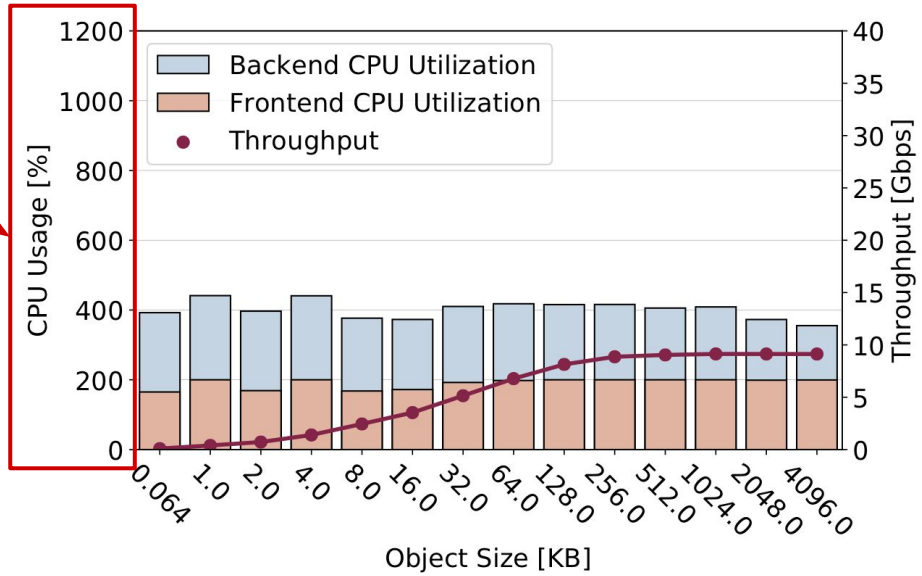
1200% = 2cores (200%) * 5 backends
+ 2cores (200%) * 1 frontend

Right Y-Axis: Throughput observed at client

Max 40Gbps (attached bandwidth of the client)

X-Axis: Object size with KB

0.064 => 64bytes



Performance Characterization of Frontend Proxy Architecture

Left Y-Axis: Cluster CPU Utilization

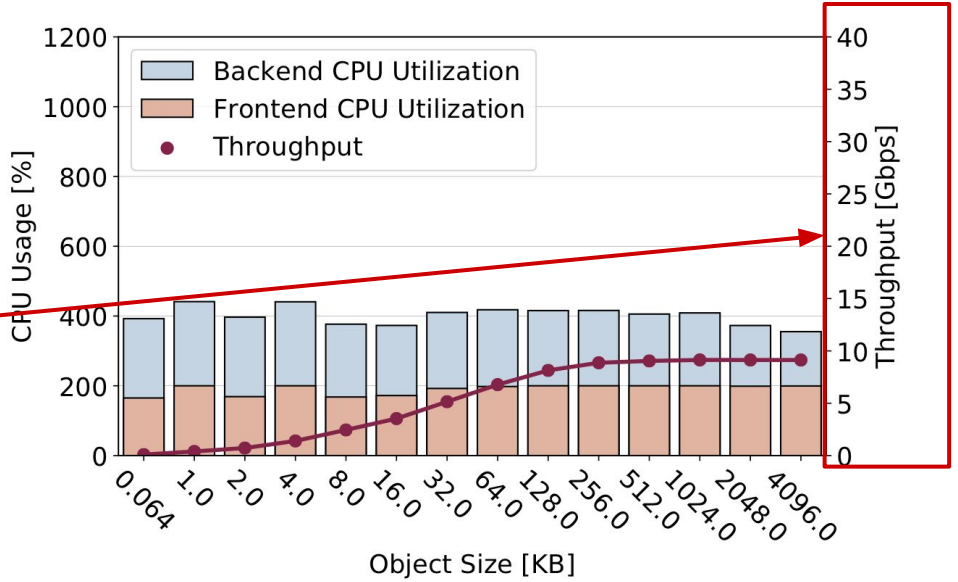
1200% = 2cores (200%) * 5 backends
+ 2cores (200%) * 1 frontend

Right Y-Axis: Throughput observed at client

Max 40Gbps (attached bandwidth of the client)

X-Axis: Object size with KB

0.064 => 64bytes



Performance Characterization of Frontend Proxy Architecture

Left Y-Axis: Cluster CPU Utilization

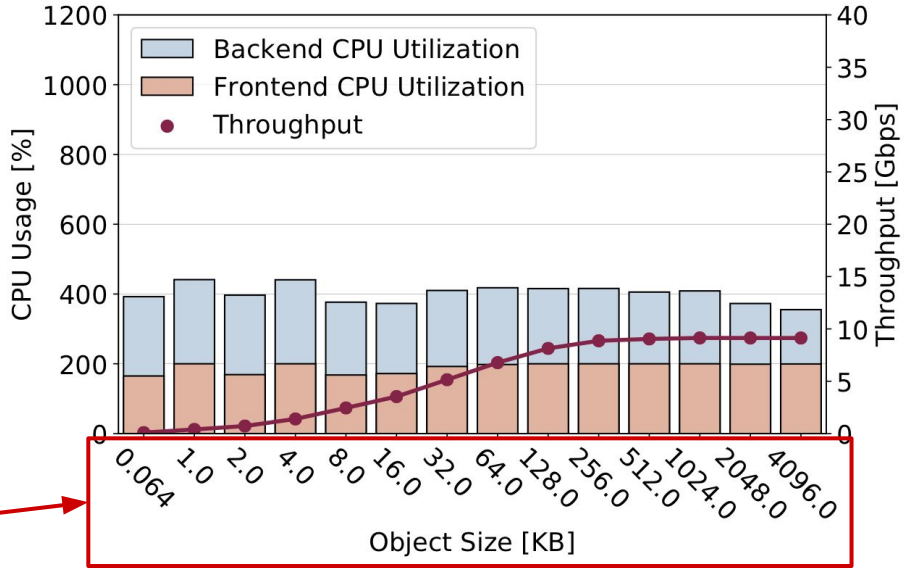
1200% = 2cores (200%) * 5 backends
+ 2cores (200%) * 1 frontend

Right Y-Axis: Throughput observed at client

Max 40Gbps (attached bandwidth of the client)

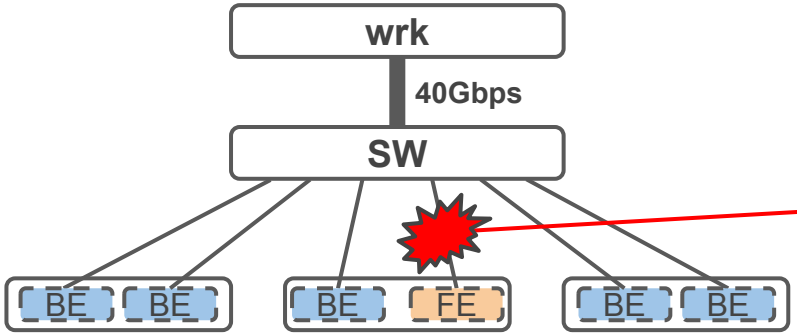
X-Axis: Object size with KB

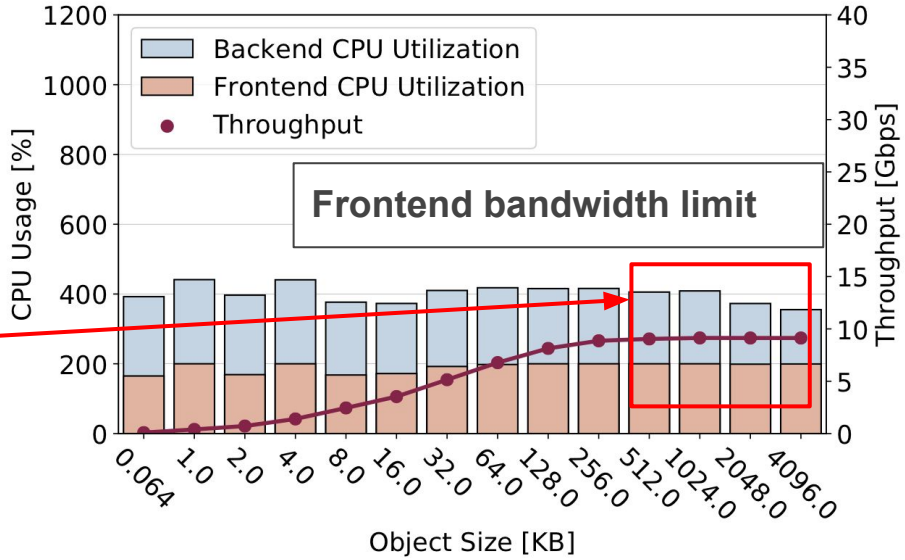
0.064 => 64bytes



Underutilized Uplink Bandwidth

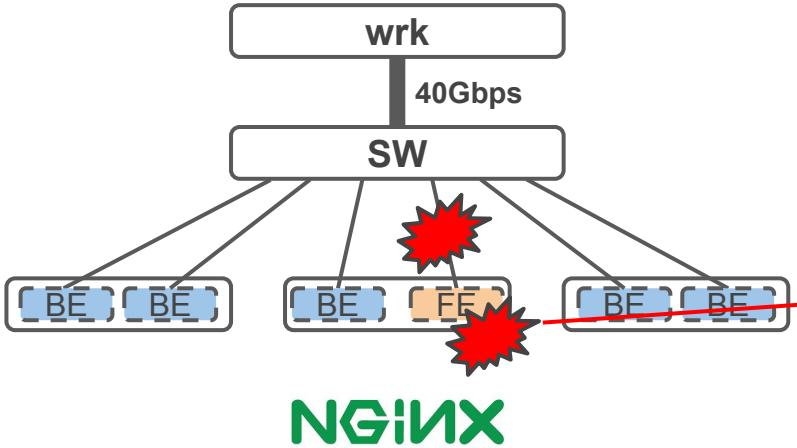
30Gbps out of 40Gbps uplink bandwidth is unsed

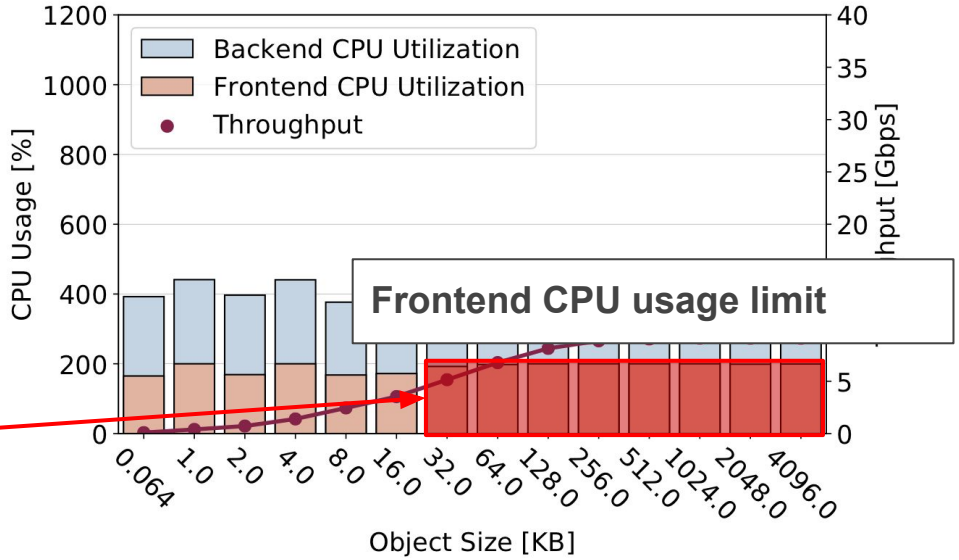




Overutilized Frontend CPU Resources

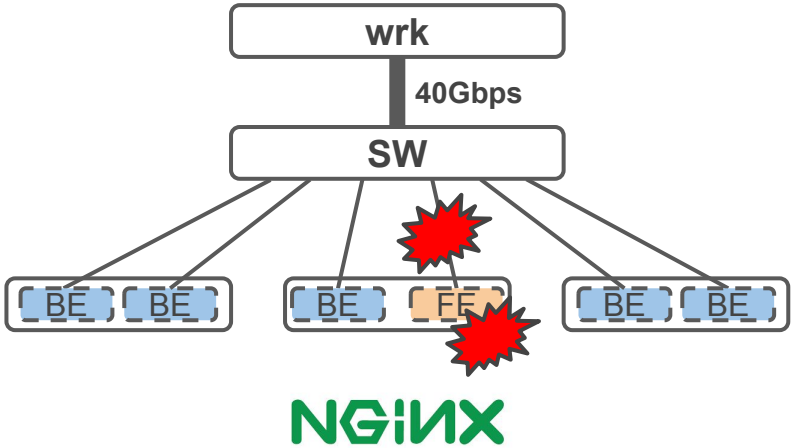
Frontend CPU is fully utilized in most of the cases

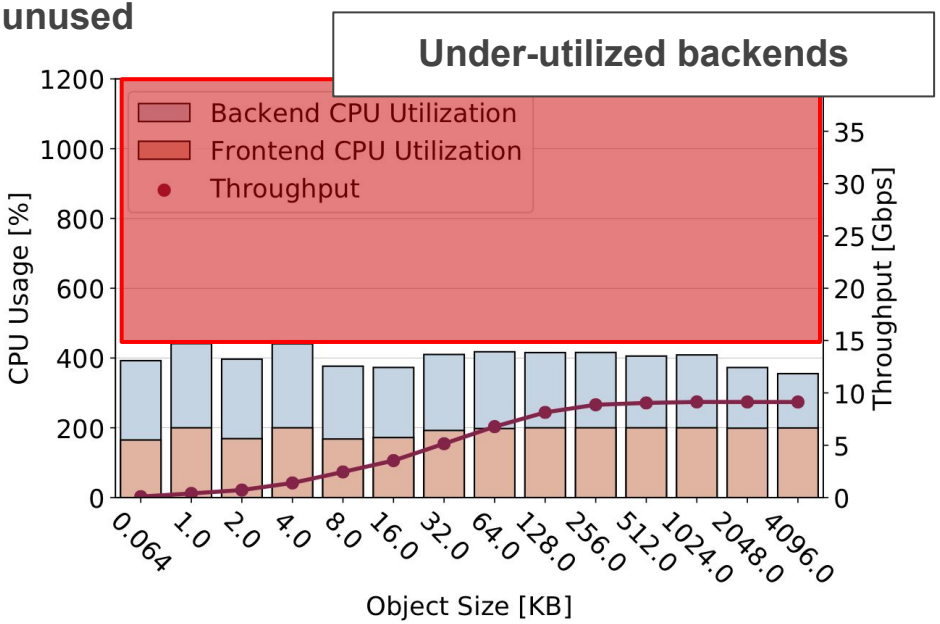




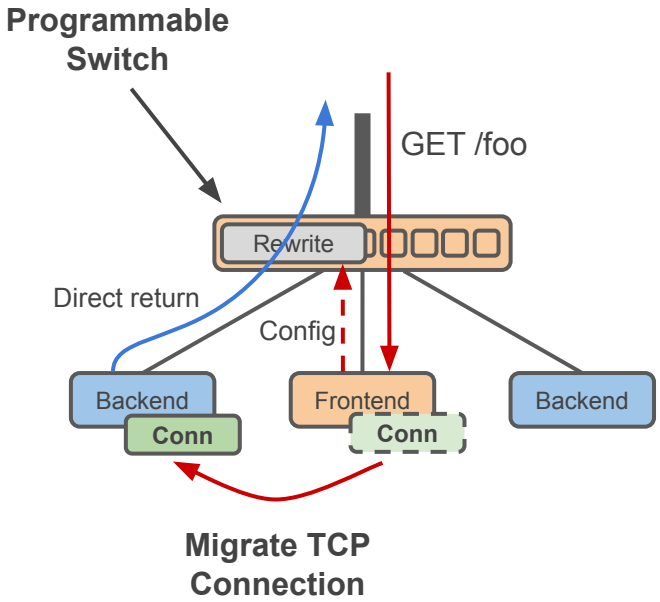
Underutilized Backend CPU Resources

800% out of 1000% backend CPU resources are unused





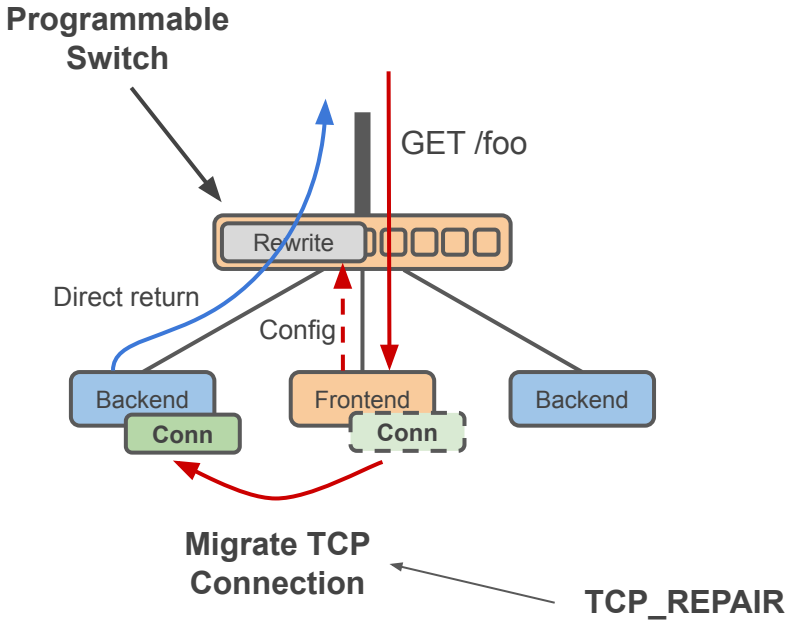
Motivation: TCP Hand-off [ASPLOS'98]



Benefit

1. Don't need special protocol on client
2. Effectively utilizes the uplink bandwidth
3. Effectively utilizes the backend CPU resource

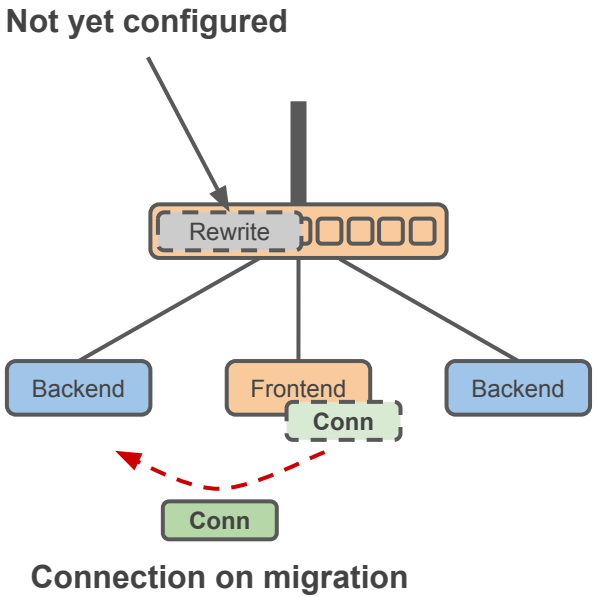
Motivation: TCP Hand-off [ASPLOS'98]



Now it is more feasible than as of 1998

1. Linux TCP_REPAIR
2. Programmable DPlane eco-system (P4, eBPF...)

TCP Hand-off is Non-trivial

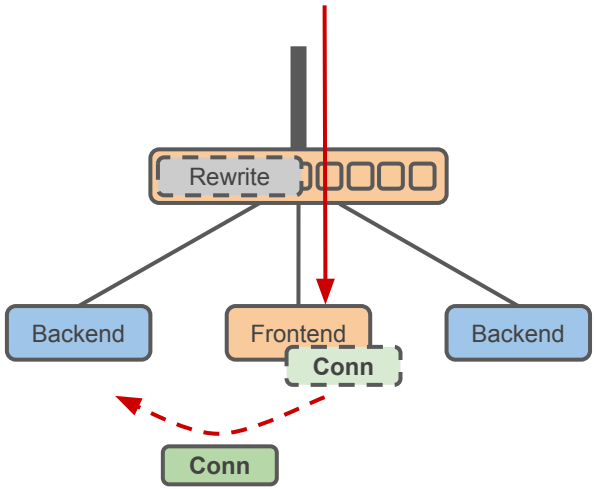


Example: "Leaked" packet problem

Connection state shouldn't progress during migration

Unexpected packet from client may changes it

TCP Hand-off is Non-trivial

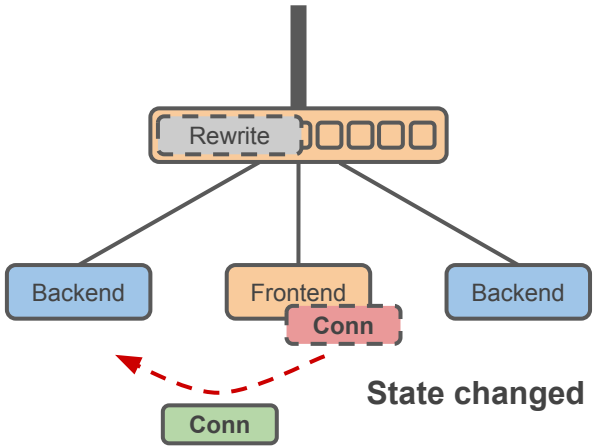


Example: "Leaked" packet problem

Connection state should't changed during migration

Unexpected packet from client may changes it

TCP Hand-off is Non-trivial

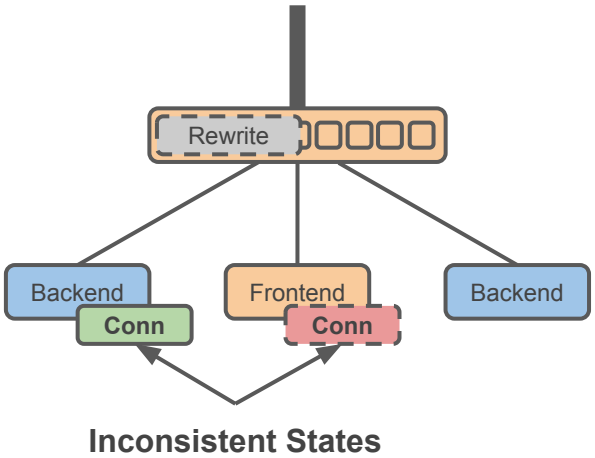


Example: "Leaked" packet problem

Connection state should't changed during migration

Unexpected packet from client may changes it

TCP Hand-off is Non-trivial

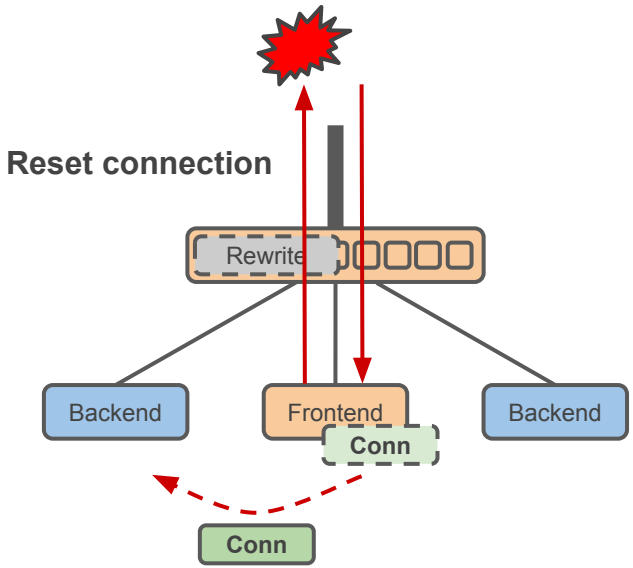


Example: "Leaked" packet problem

Connection state should't changed during migration

Unexpected packet from client may changes it

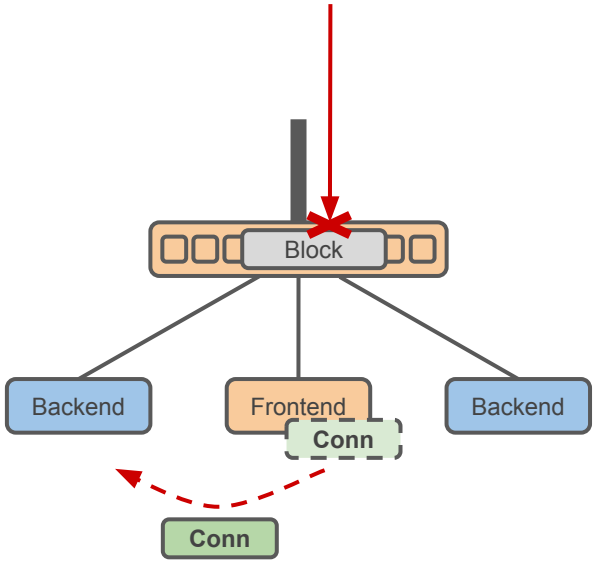
TCP Hand-off is Non-trivial



TCP_REPAIR sends RST when it get packet in "REPAIR" state

=> Breaks the client

TCP Hand-off is Non-trivial



Need "lock" in network to prevent the "leak"

Original TCP Hand-off paper doesn't cover this issue

Our Contributions

1. Robust hand-off protocol which overcomes correctness issues

Two Phase Hand-off Protocol,

2. Prism switch

Efficient switch interaction / inline switch coniguration protocol

3. Prism stack

Event loop based software stack to integrate TCP Hand-off to the applications,

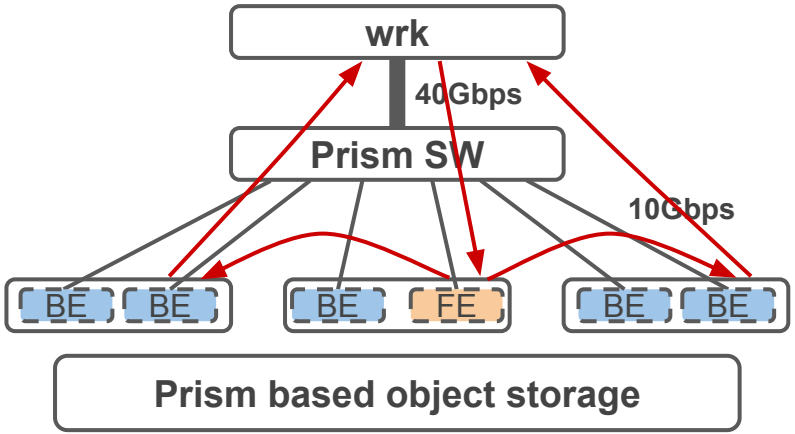
FIN_WAIT1/CLOSE_WAIT handling

4. Evaluation with realistic use cases

Replicated / Partitioned object storage

And more in our paper

Prism Performance

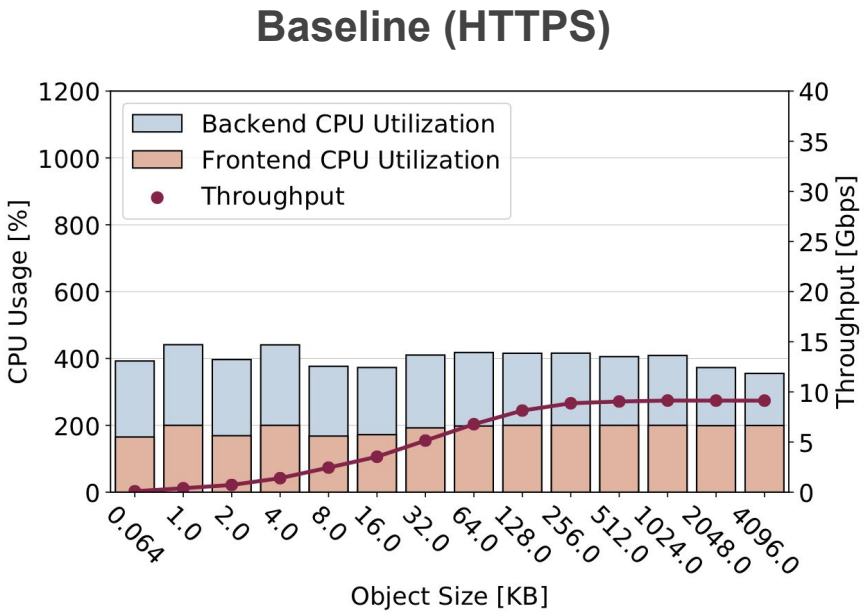


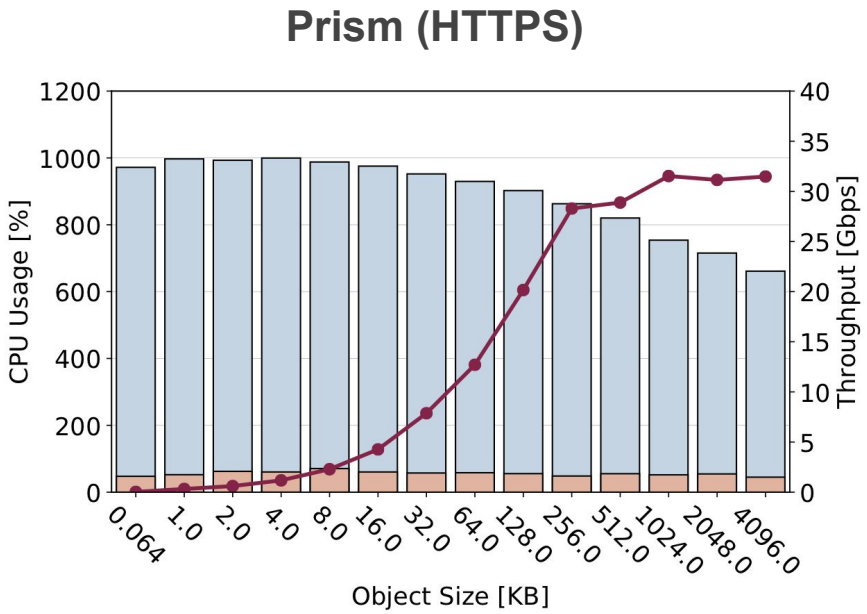
Benchmark

1. S3 protocol with HTTP(S) keep-alive
2. 100 parallel TCP connections
3. Round-robin load balancing on frontend
4. Fetches in-memory objects

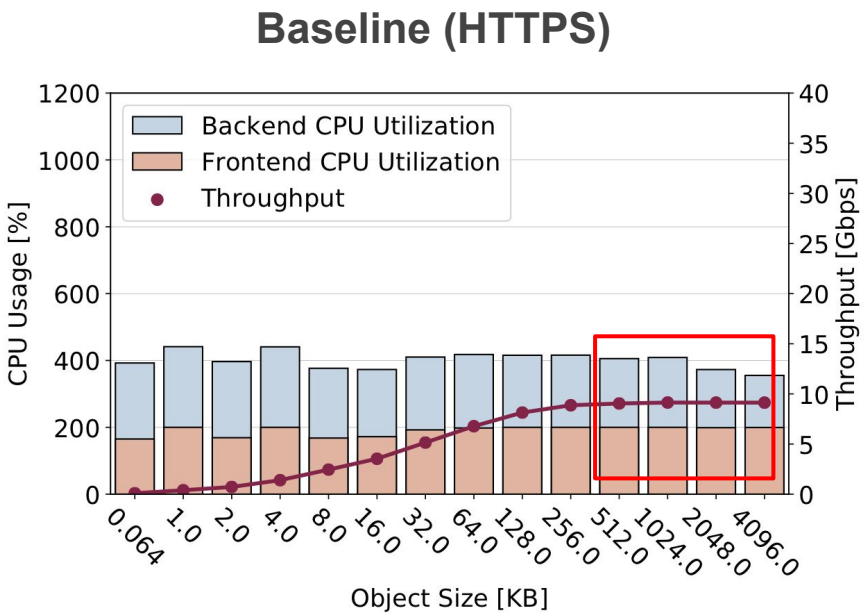
Same as baseline, but using Prism

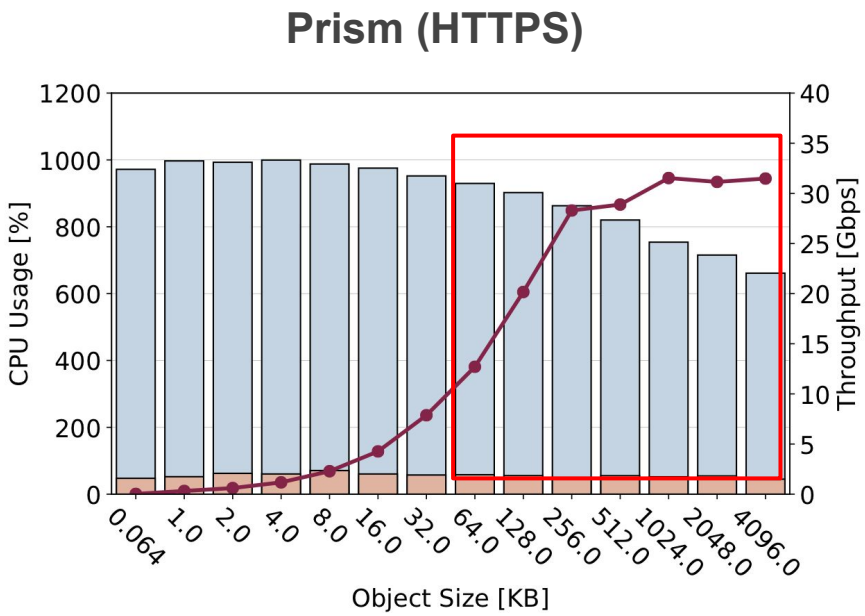
Prism Performance



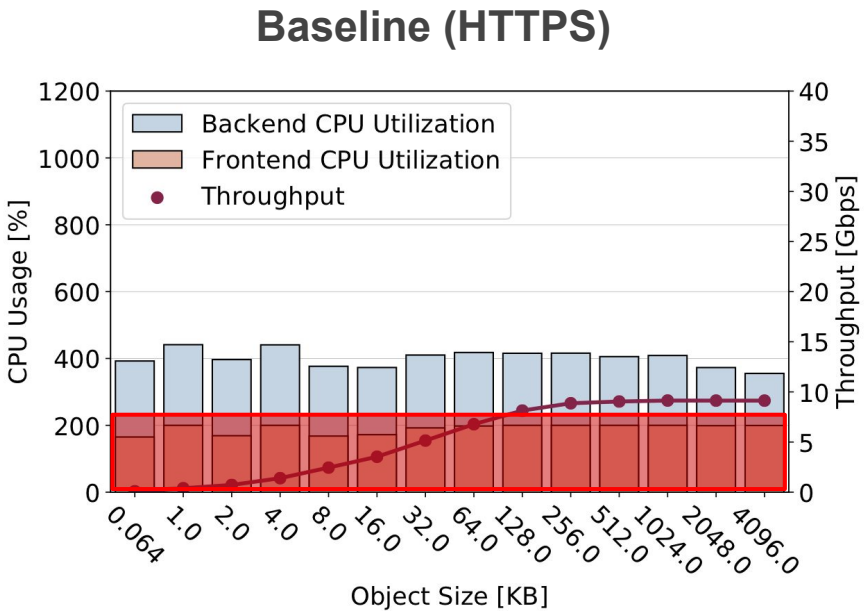


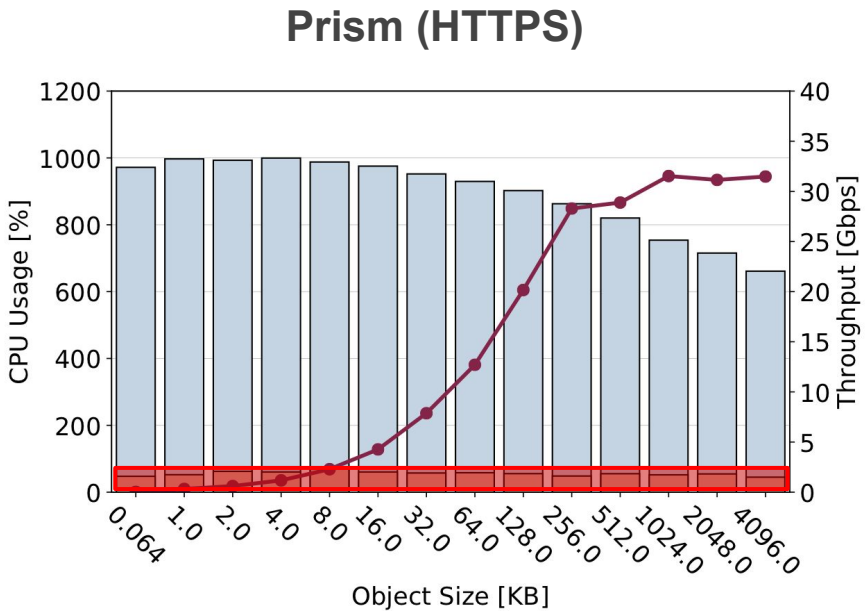
Prism Effectively Utilizes Uplink Bandwidth



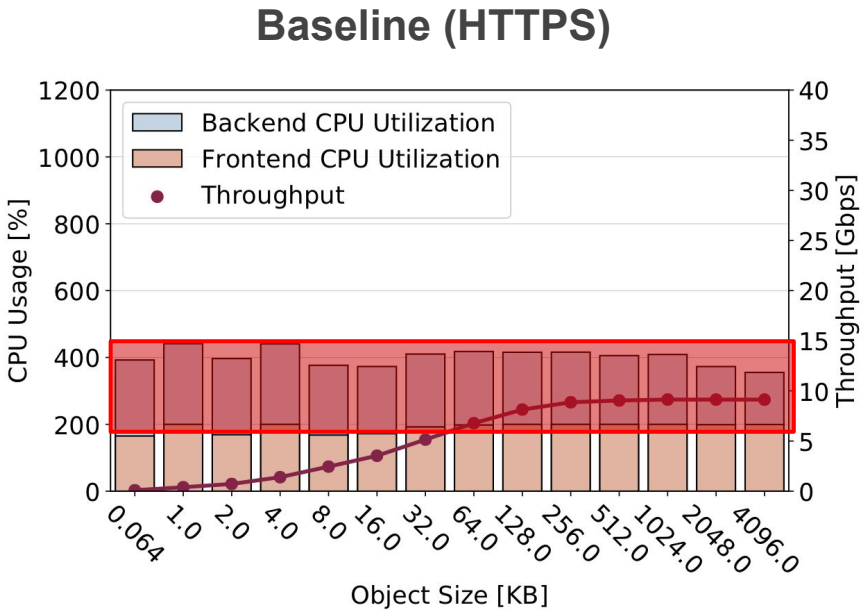


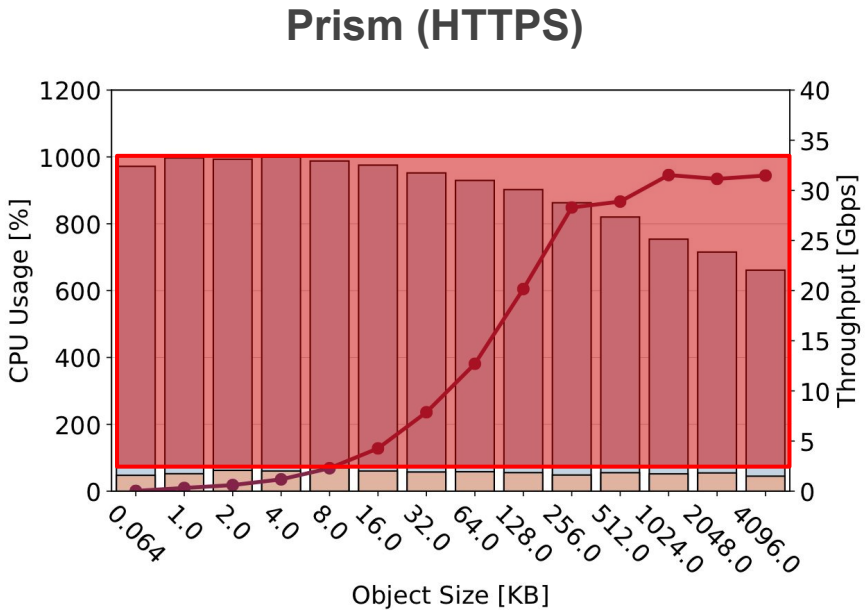
Prism Effectively Reduces Frontend CPU Utilization





Prism Effectively Utilizes Backend CPU Resources





Hand-off Latency

Operation	Latency (μs)
Block all traffic (switch config)	22
Serialize protocol states	14
Send states	51
Deserialize protocol states	123
Configure rewrite (switch config)	22
Total	**232**

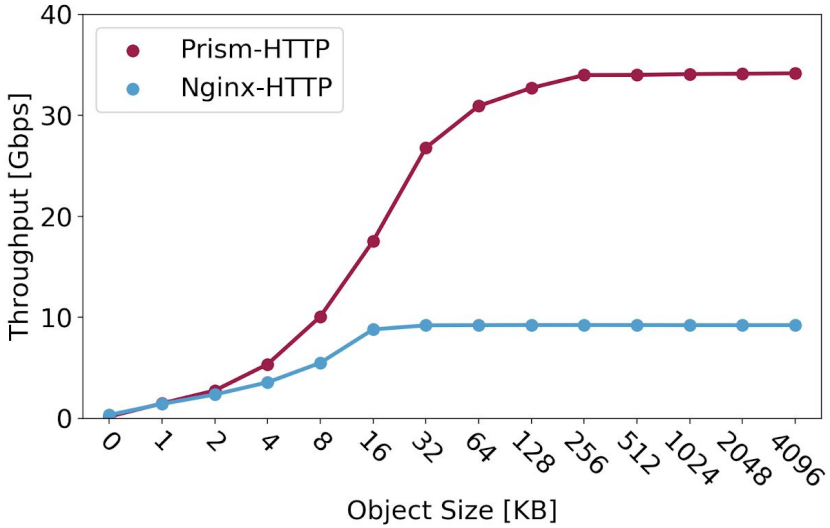
Extra latency for hand-off

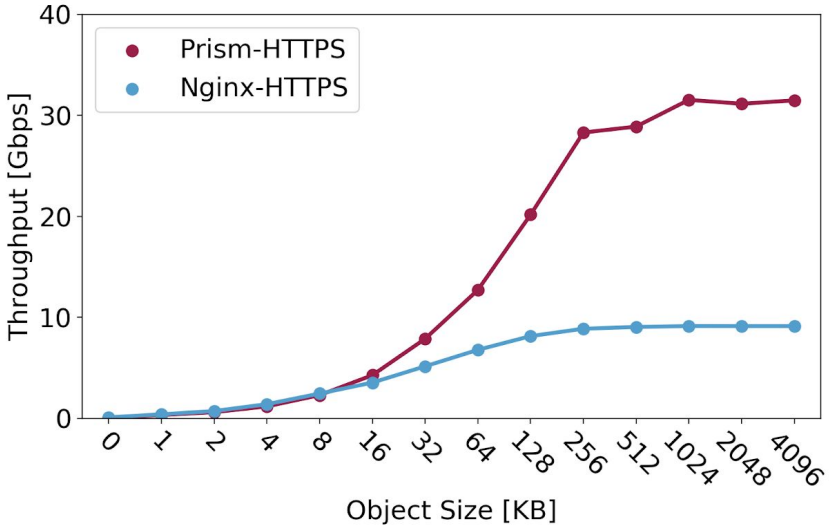
Time takes for hand-off is **fixed** for read

How it costs depends on the **object size**

Cost of Hand-off Latency

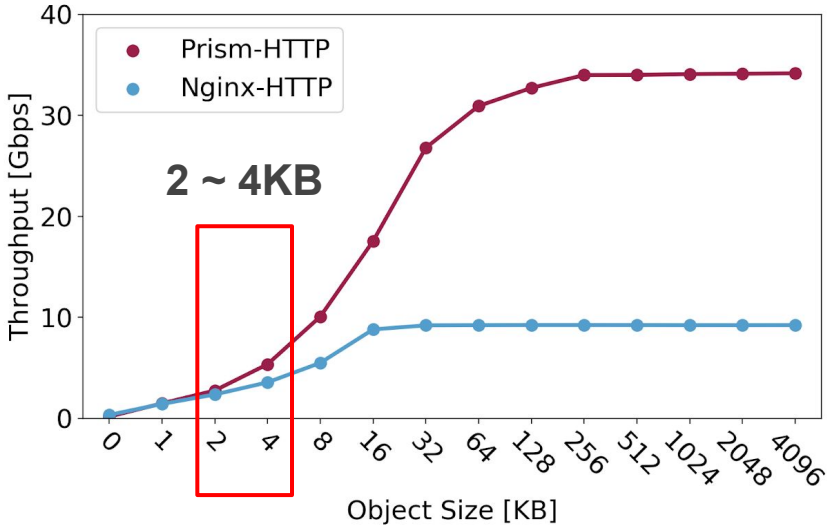
How large does the object size need to be to amortize the connection hand-off cost?

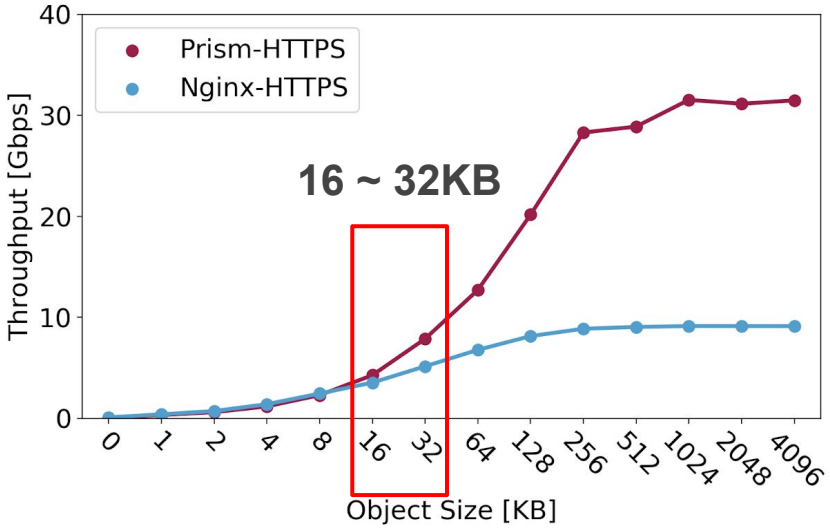




Cost of Hand-off Latency

How large does the object size need to be to amortize the connection hand-off cost?





Recap

Object storage performance is capped by the frontend intermedeates all traffic

Limitation of attached bandwidth, high CPU usage due to relaying large traffic

TCP hand-off based solution is fairly feasible in these days, but non-trivial

Need to take many "correctness" issues into account (e.g. "leaked" packet problem)

Prism is a framework to integrate TCP hand-off approach into the applications

Robust connection hand-off mechanism, switch implementation, software stack, etc…

Source is available: https://github.com/YutaroHayakawa/Prism-HTTP