Overcoming End-to-End Challenges in Reconfigurable Datacenter Networks

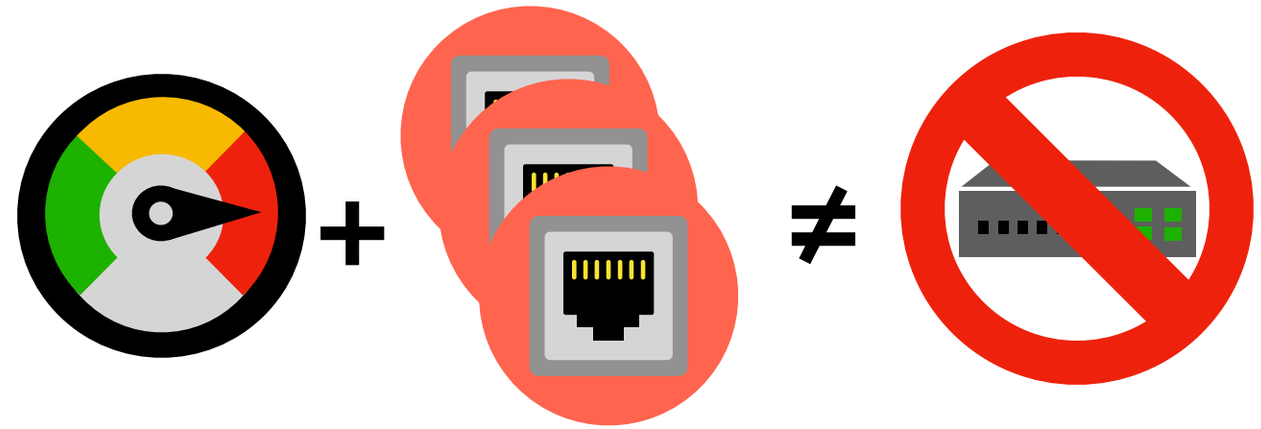
Matthew K. Mukerjee, Daehyeok Kim, Srinivasan Seshan

SIGCOMM ’18 (submisison)

Carnegie Mellon University

Hi my name is Matt Mukerjee and today I’ll presenting our work on Overcoming End-to-End Challenges in Reconfigurable Datacenter Networks.

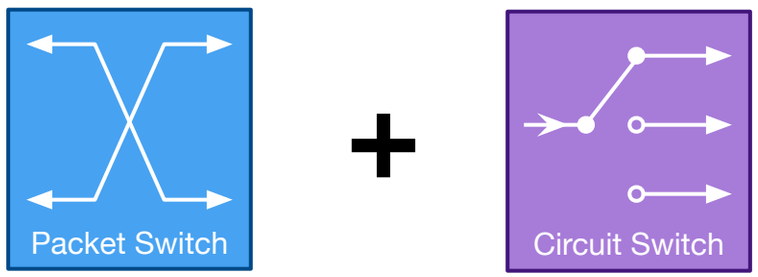
Difficult to scale datacenters with demand



P-FatTree: A multi-channel datacenter network topology. HotNets 2016.

Demand in datacenter is constantly growing. We want ** higher bandwidth, ** higher port count packet switches… but a ** 2016 HotNets paper is making our lives difficult. ** We’re starting to hit CMOS manufacturing limits, meaning it’s difficult to build larger, faster datacenter networks…

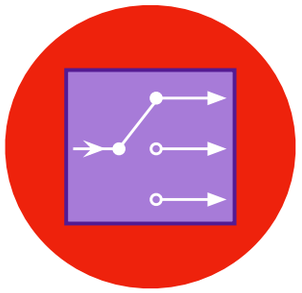
Use circuits to build bigger + faster networks!



Reconfigurable Datacenter Networks (RDCNs)

Well that’s not entirely true. We just can’t build a better packet switch… Thus, researchers have proposed augmenting traditional packet networks in datacenters with ** reconfigurable circuits that add bandwidth on demand. We call these networks ** “Reconfigurable Datacenter Networks”.

Is the problem solved?



Circuit Switch Design

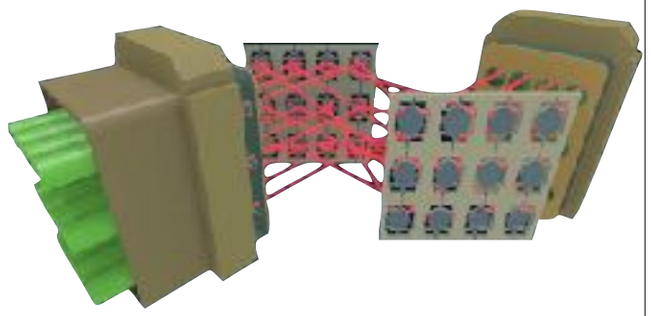
How do you physical build it?

So is the problem solved? Most prior work focuses on ** circuit switch design.

Is the problem solved?

60GHz Wireless

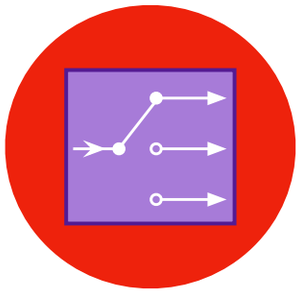
Free-space Optics



Optical Circuit Switching

Different projects have used different technologies, for example 60GHz wireless, free-space optics, or optical circuit switching. In this talk, we don’t worry about which technology is used, as the end-to-end challenges are generally the same.

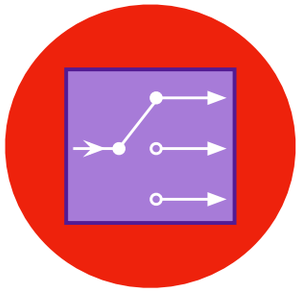
Is the problem solved?



Circuit Switch Design

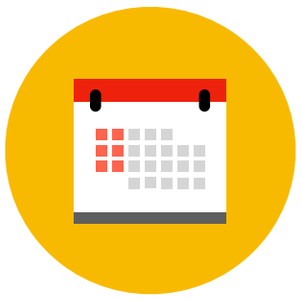
How do you physical build it?

Beyond circuit switch design, prior work have pointed out one other problem in this space…



Circuit Switch Design

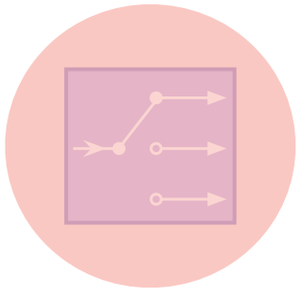
How do you physical build it?



Network Scheduling

How do you make use of it?

Network scheduling. We'll explain both of these pieces in more detail in a moment. We argue that there is a third problem that hasn't been looked at in detail…

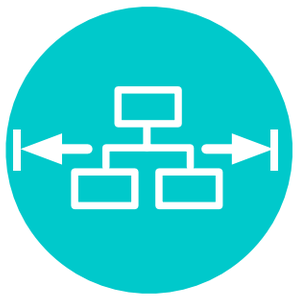


Circuit Switch Design

How do you physical build it?

Network Scheduling

How do you make use of it?

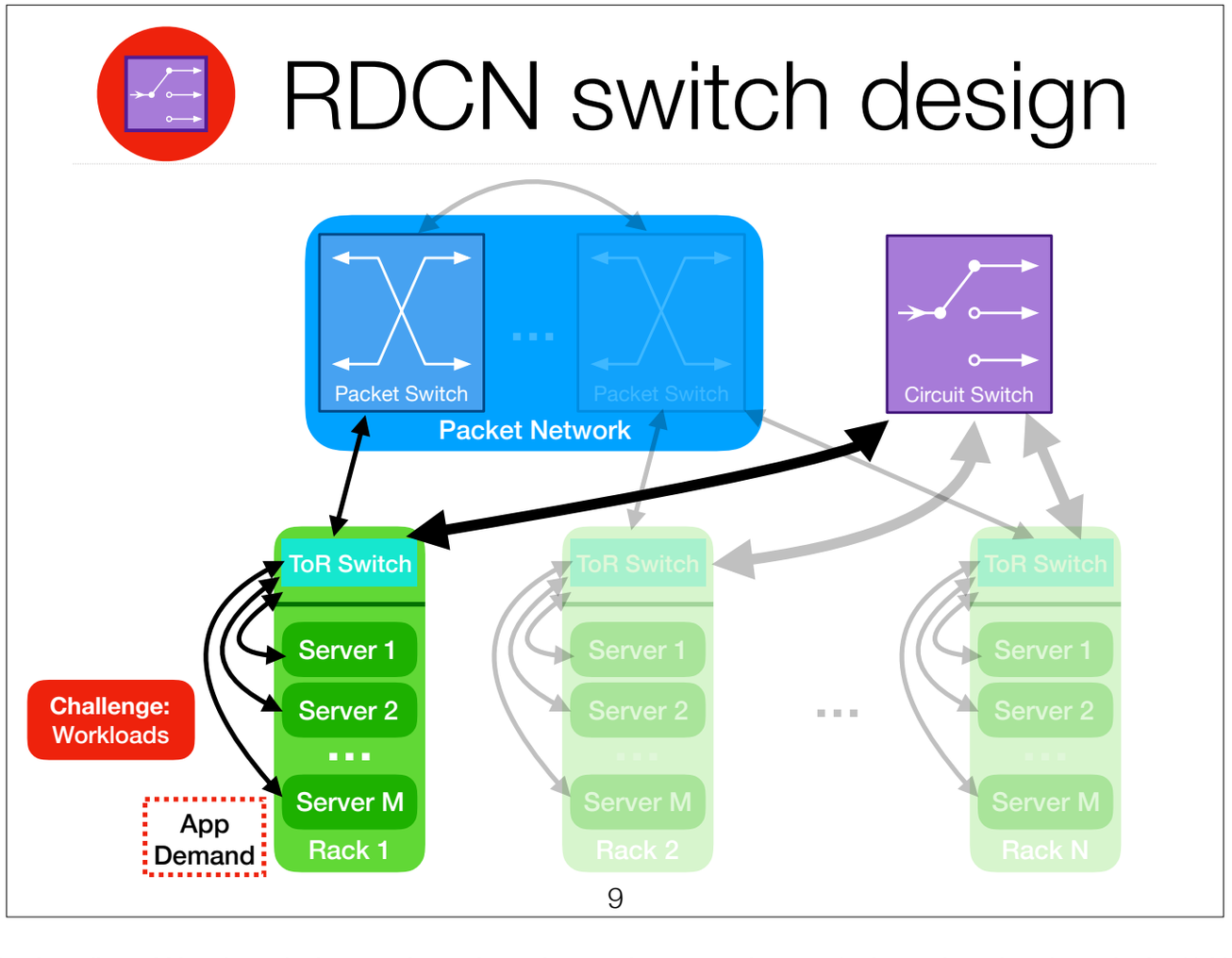


End-to-End Challenges

What existing things break?

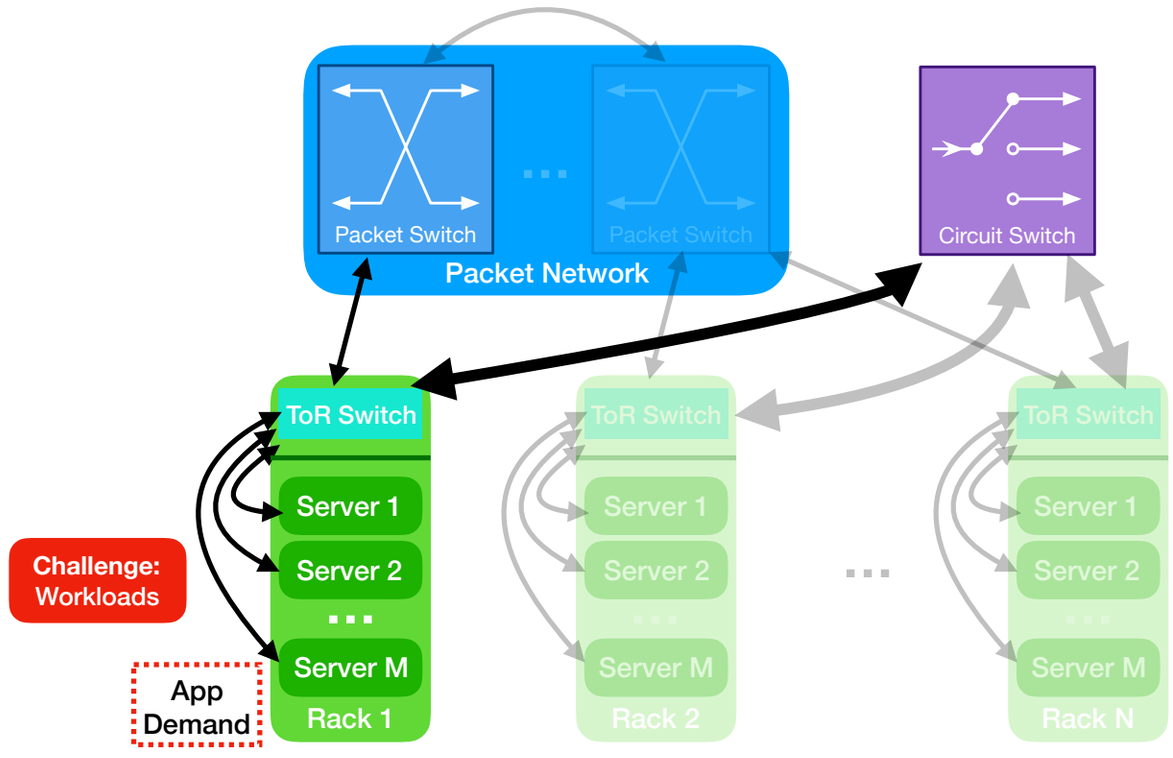
End-to-end challenges. For example, does TCP break on these networks?

Let’s walk through the first two problems in a bit more detail to explain why end-to-end challenges arise…



Let’s look at reconfigurable datacenter design first. We already know there is going to be a packet switch and a circuit switch. ** Well, the packet switch might really be a packet network. We have ** N racks ** of ** M servers. Each server connects to a ** Top-of-Rack (ToR) switch** . The ToR switches connect to the ** packet network and the ** circuit switch. The network needs to deliver ** application demand from each server. Figuring out what demand should be sent over the packet network and which over a circuit is challenging. Certain application ** workloads may be orders of magnitude more challenging to delivery than others, as the circuit switch has specific delivery constraints as we’ll show next. If we could simplify difficult workloads in the application themselves, we could greatly improve performance. We’ll talk about this in more detail later in the talk.

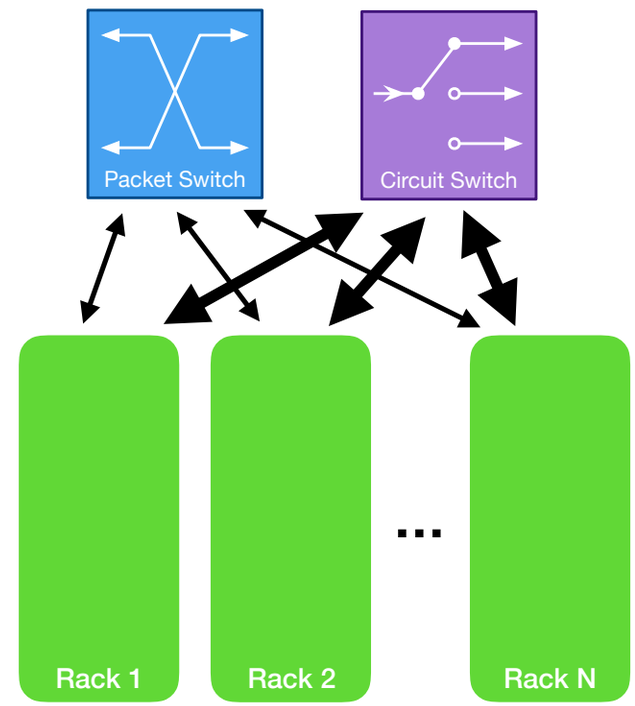
RDCN switch design



...

Challenge: Workloads

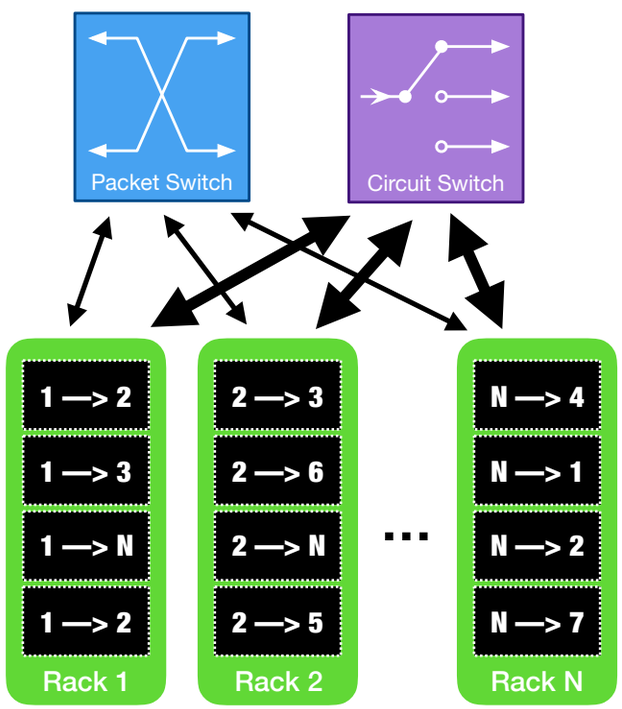
RDCN scheduling



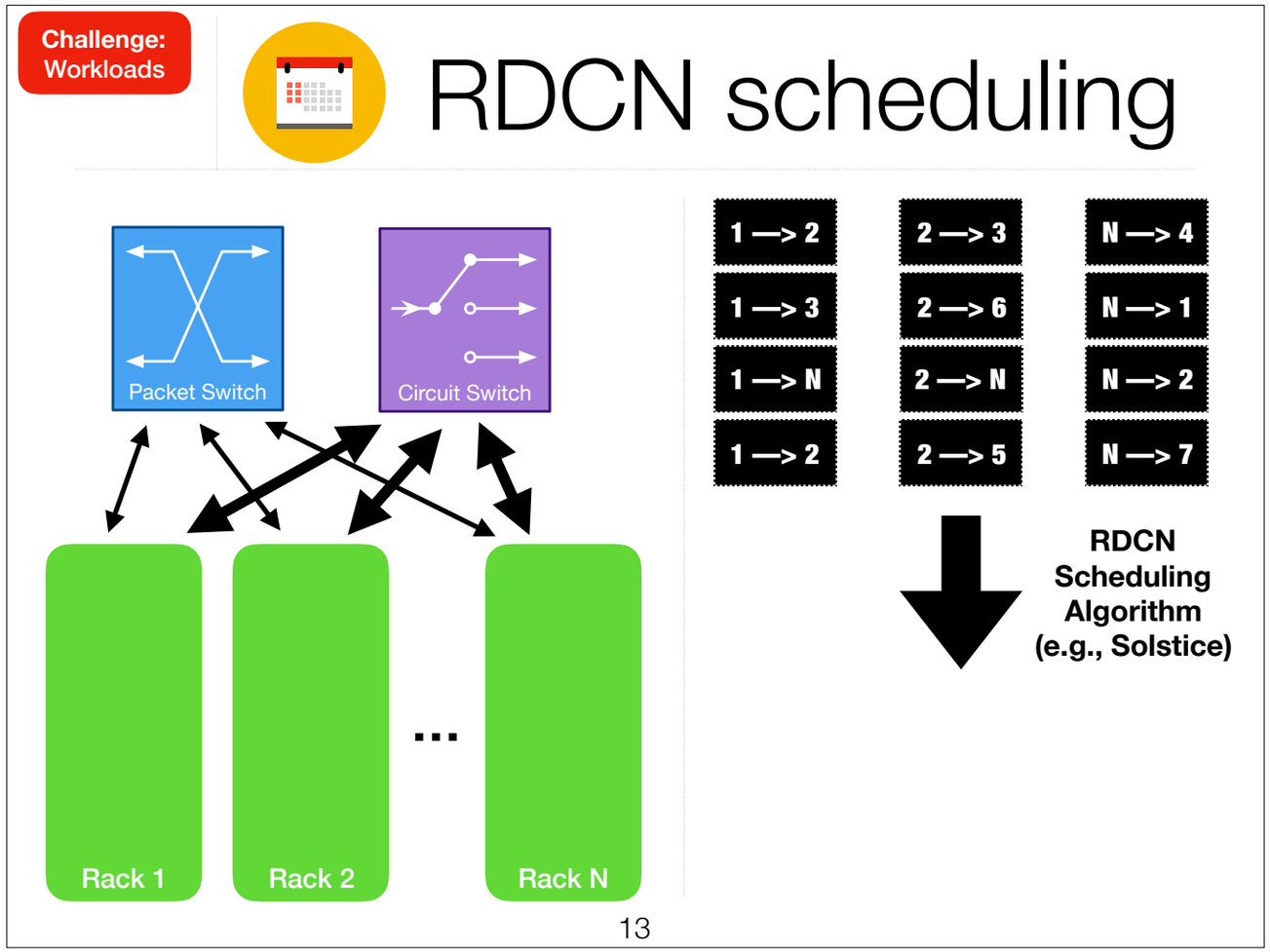
So let’s keep our “Workload” challenge at the top so we remember it, and now let’s talk about reconfigurable datacenter network scheduling. We’ve cleared out the clutter here to bring our network to its basic form, N racks that can communicate over a packet switch or a circuit switch. For scheduling, what we need to figure out how to best serve rack-to-rack demand. So let’s generate some demand…

Challenge: Workloads

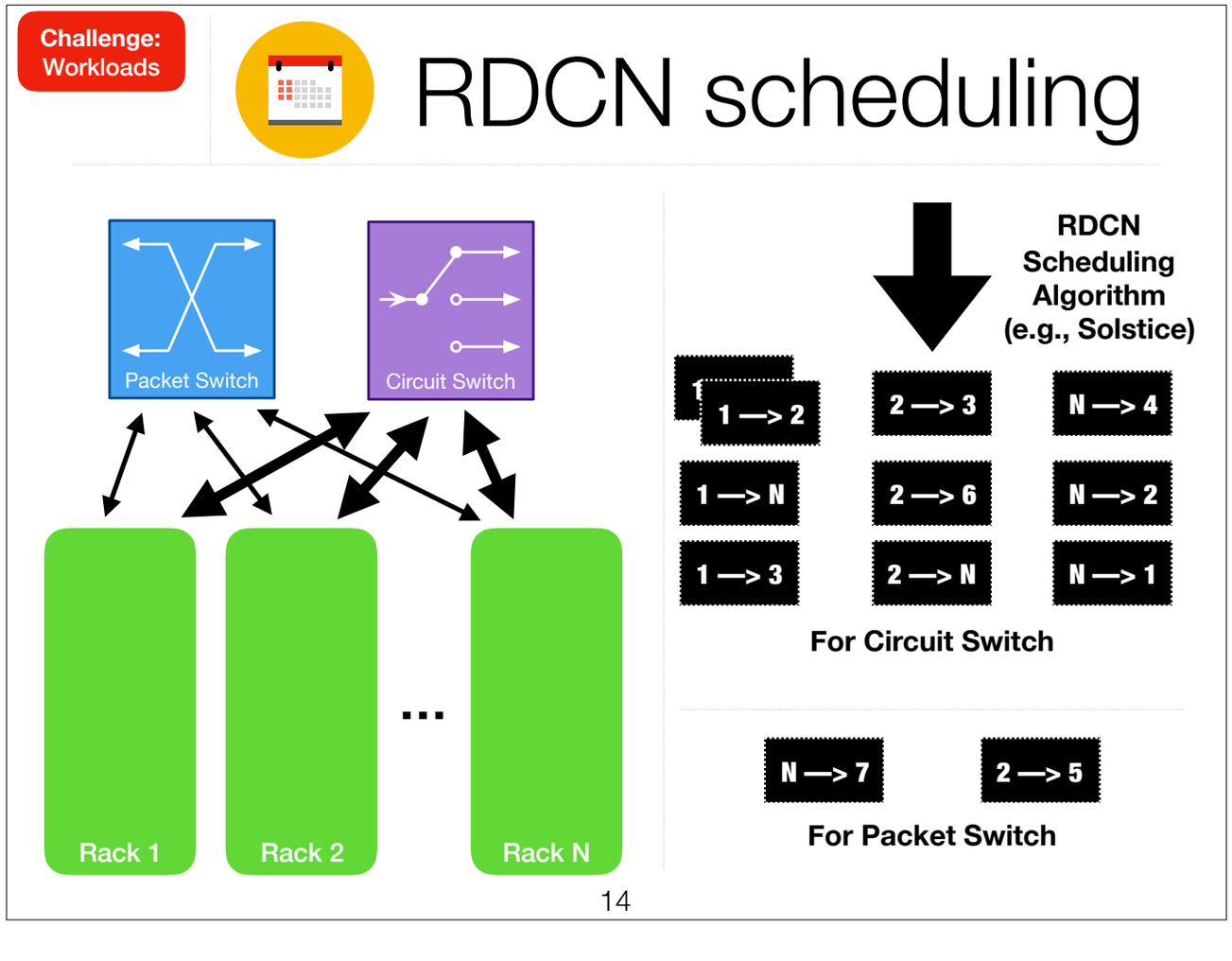
RDCN scheduling



So here we see some rack-to-rack demand…



We'll toss that into a reconfigurable datacenter scheduling algorithm, like our prior work Solstice…

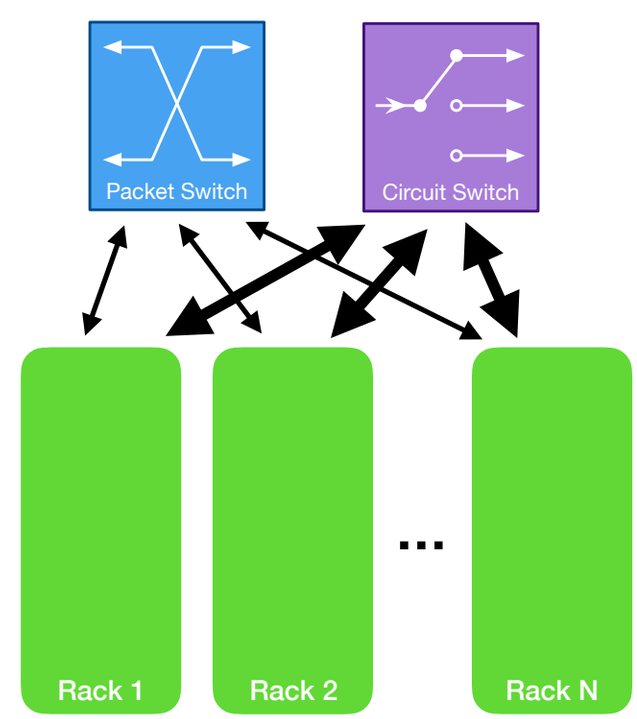


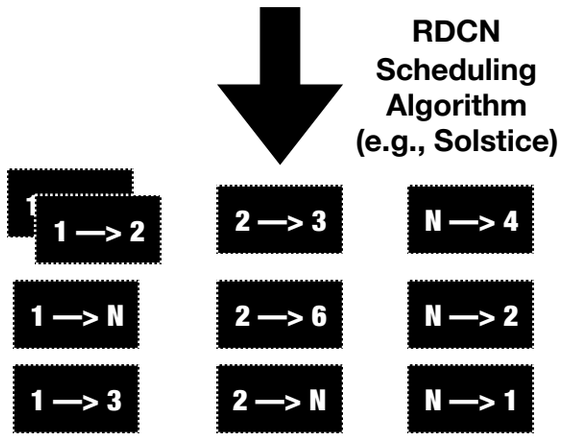
At a high level, these algorithms split demand, sending some to the circuit switch and some to the packet switch.

The interesting part is what goes on with the circuit. Due to technology constraints, demand sent to the circuit switch must be explicitly scheduled…

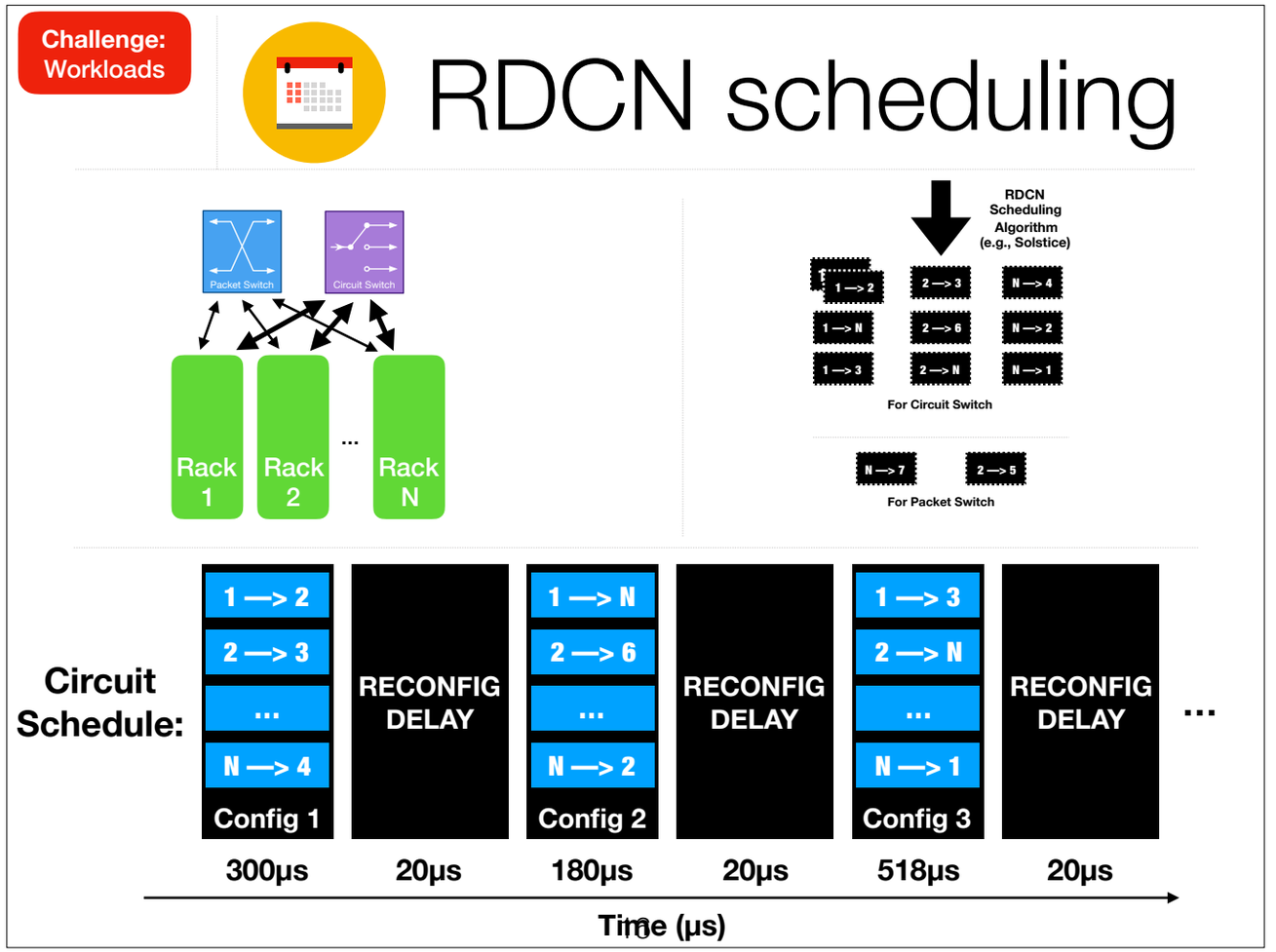
Challenge: Workloads

RDCN scheduling

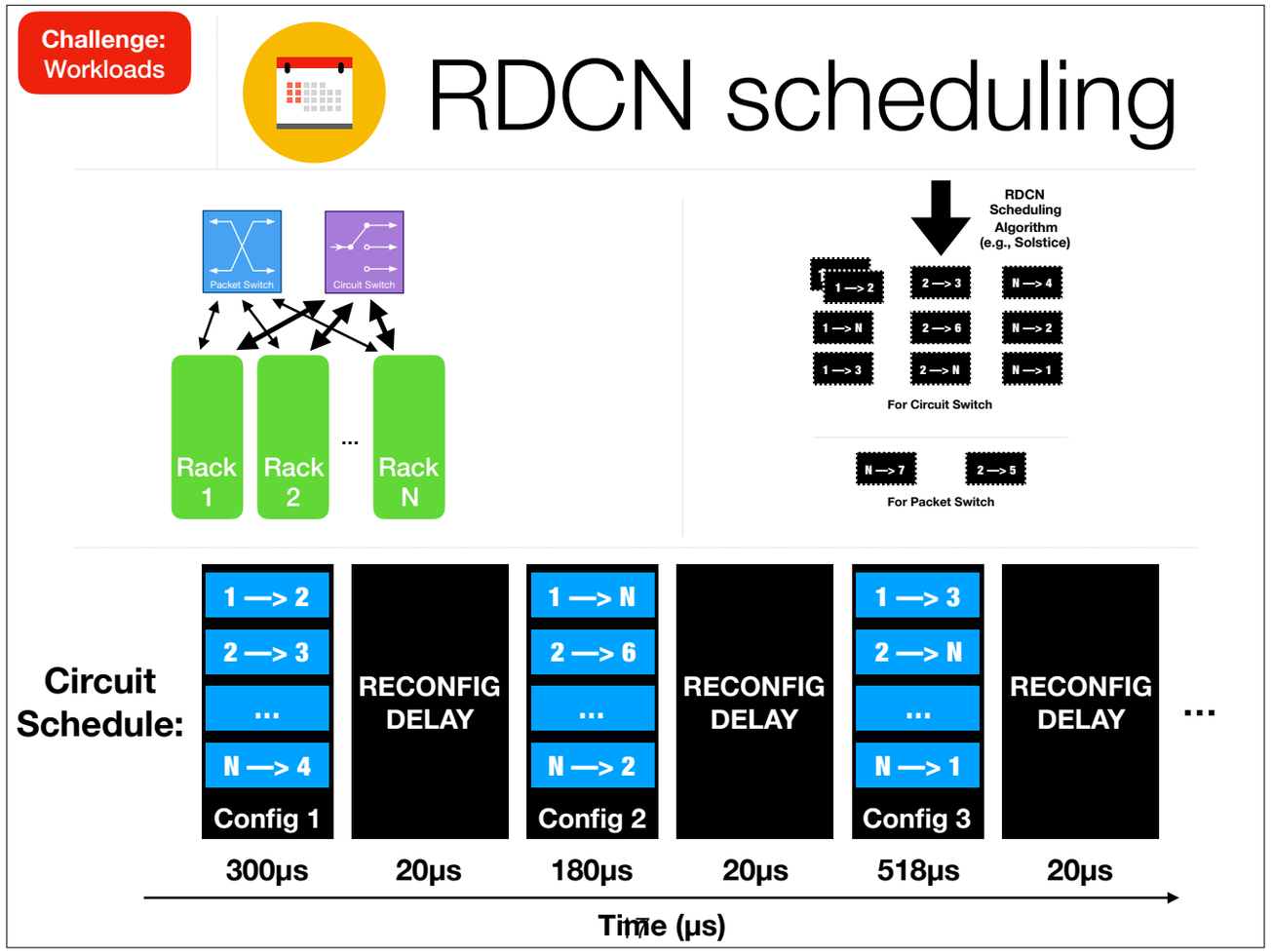




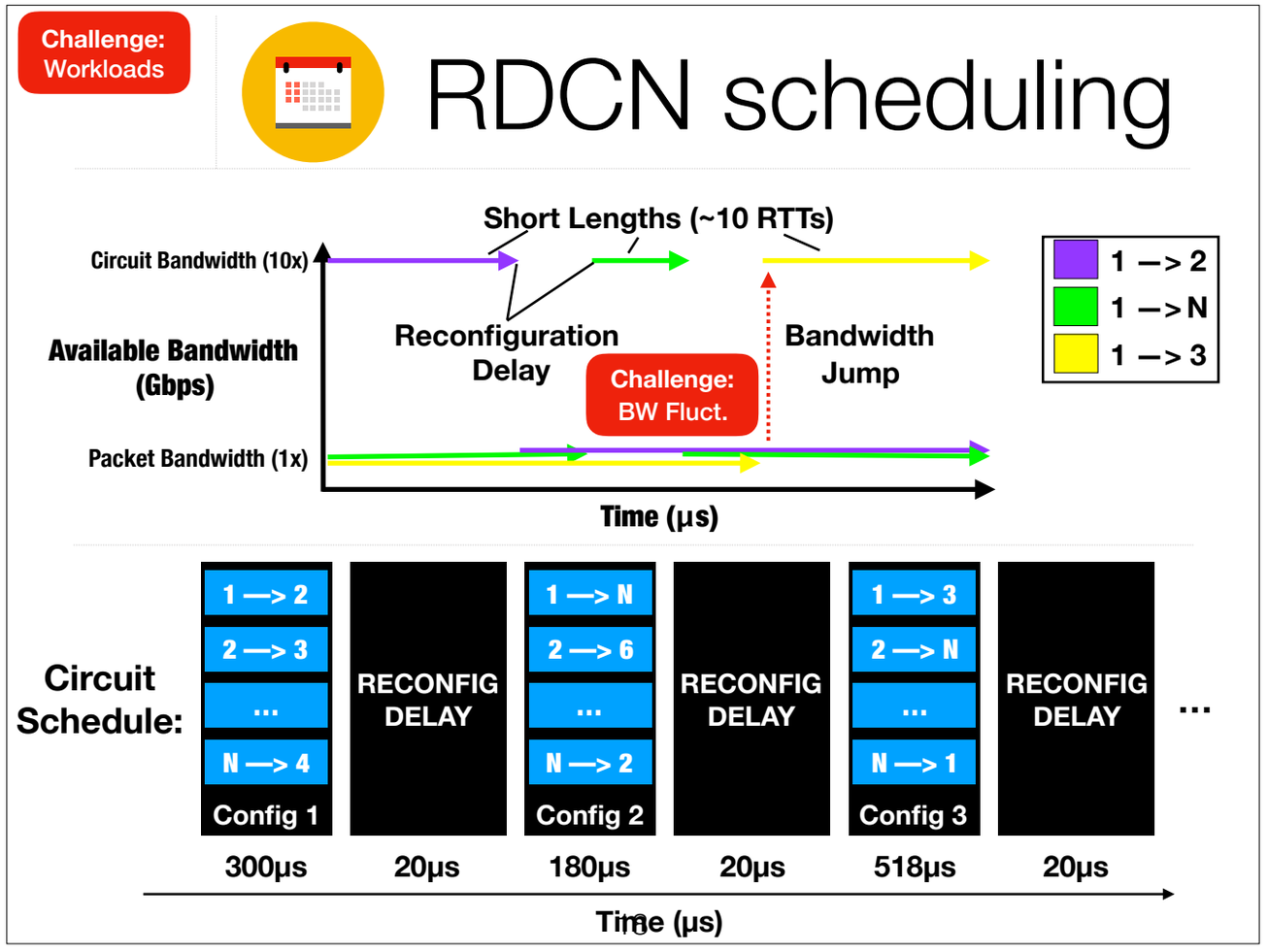
...



So, for this example, we might have a schedule like this. ** Over the next so amount of microseconds, first configure ** these circuits, for ** 300 microseconds. Then ** reconfigure the circuits (** taking some non-trivial reconfiguration delay). Then ** do this next config, etc. This example highlights the two constraints for the circuit switch: reconfiguration requires taking down all circuits for a non-trivial reconfiguration delay, and each rack can only send to (or receive from) one other rack within a configuration.



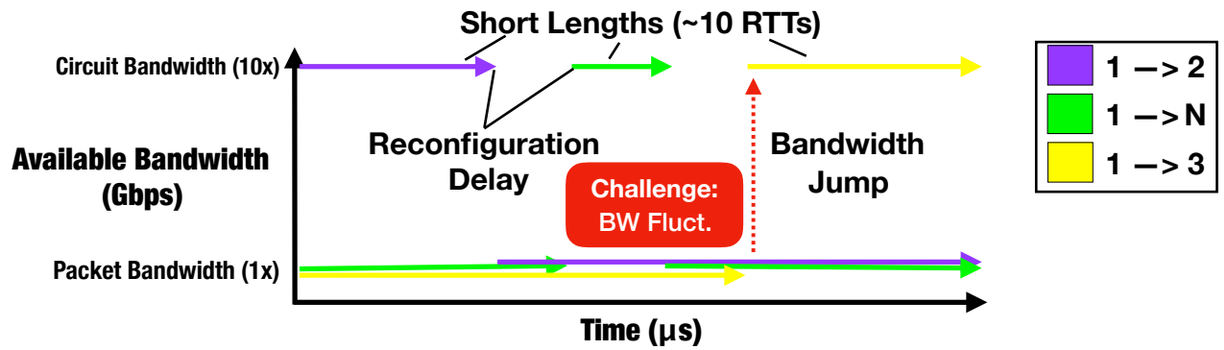
Finally, now that we understand scheduling, how do reconfigurable datacenter networks impact the life of TCP flows?

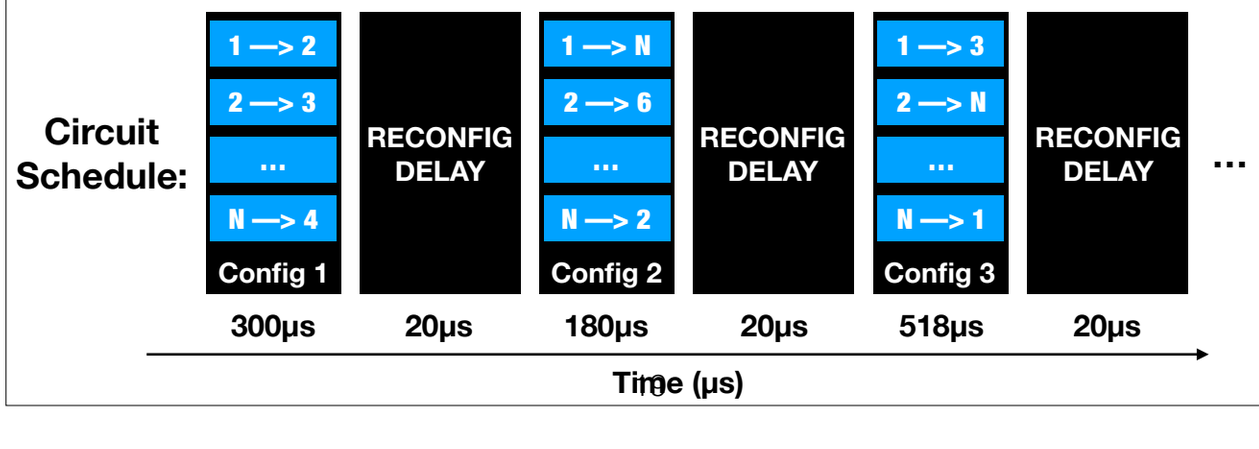


Let’s draw an example. ** On the ** x-axis we’re showing time in microseconds, and on the ** y-axis we’re showing available bandwidth in Gbps. Remember, the ** packet switch bandwidth is low and the ** circuit switch bandwidth is high. Let’s examine three flows ** purple, ** green, and ** yellow, all from the same source rack. The available bandwidth they see may be something like ** this. We see that purple starts on the circuit, then green gets a turn, then yellow. The amount of time spent on the circuit switch is ** short (about 10 RTTs). We also clearly see that ** the circuits go down during reconfiguration. Finally, and most importantly, we see that flows are expected to make use of ** a large bandwidth jump. TCP congestion control likely would have problems with this. We’ll walk through a detailed TCP example later in the talk. This makes up our final end-to-end challenge, ** rapid bandwidth fluctuations.

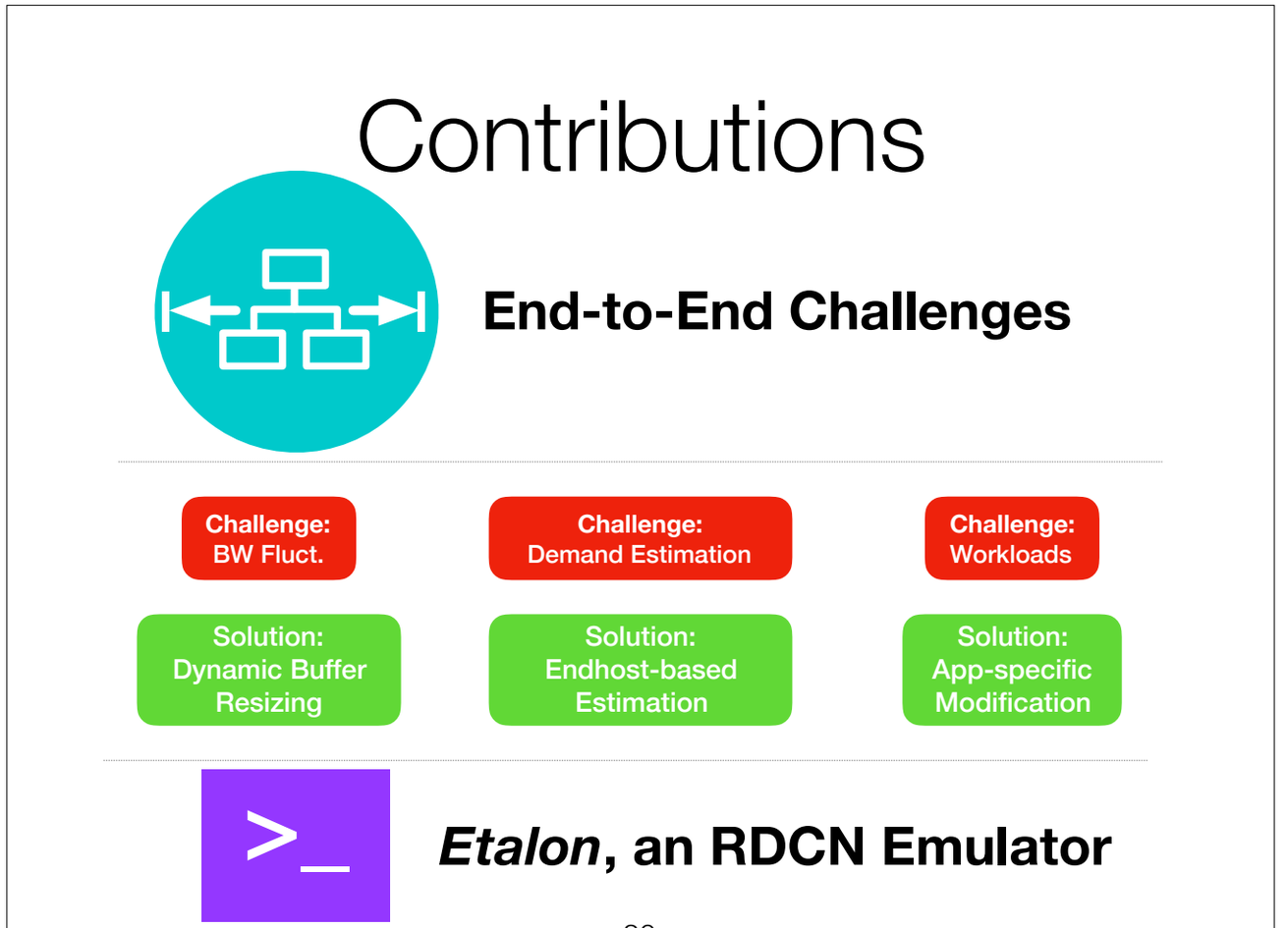
Challenge: Workloads

RDCN scheduling



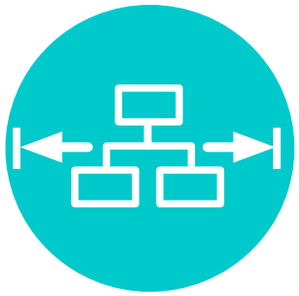


...



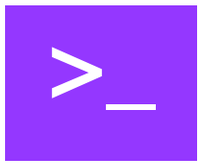
This leads us to our contributions. First-and-foremost we argue that a new problem, “end-to-end challenges” in reconfigurable datacenter that hasn’t been explored in detail. Second, we analyze three critical end-to-end challenges and find solutions based on cross-layer awareness. Finally, we design and implement an open-source evaluation platform for public testbeds, Etalon, for evaluating reconfigurable datacenter networks end-to-end with real applications and endhost stacks. This forms the…

Overview



End-to-End Challenges

Challenge: BW Fluct.	Challenge: Demand Estimation	Challenge: Workloads
Solution: Dynamic Buffer Resizing	Solution: Endhost-based Estimation	Solution: App-specific Modification



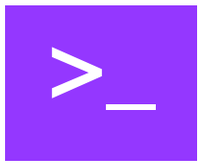
***Etalon*, an RDCN Emulator**

…overview for the rest of the talk. We've already talked about where the end-to-end challenges come from…

Overview

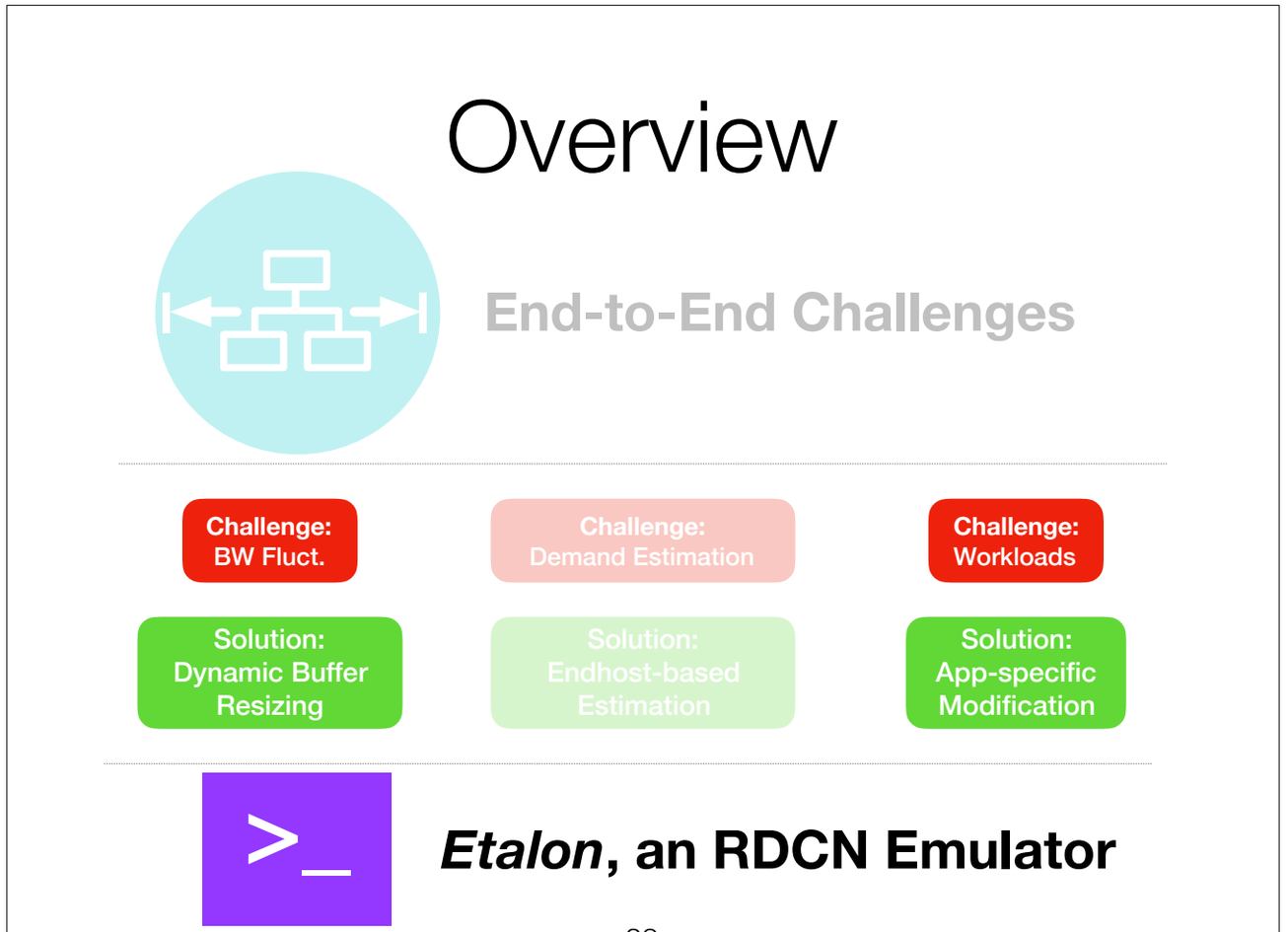
End-to-End Challenges

Challenge: BW Fluct.	Challenge: Demand Estimation	Challenge: Workloads
Solution: Dynamic Buffer Resizing	Solution: Endhost-based Estimation	Solution: App-specific Modification



***Etalon*, an RDCN Emulator**

Also, we’re going to skip demand estimation and it’s solution for this talk.

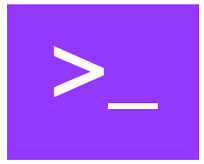


If you’re interested in that, see the paper. Okay let’s briefly touch on our emulator, Etalon…

Overview

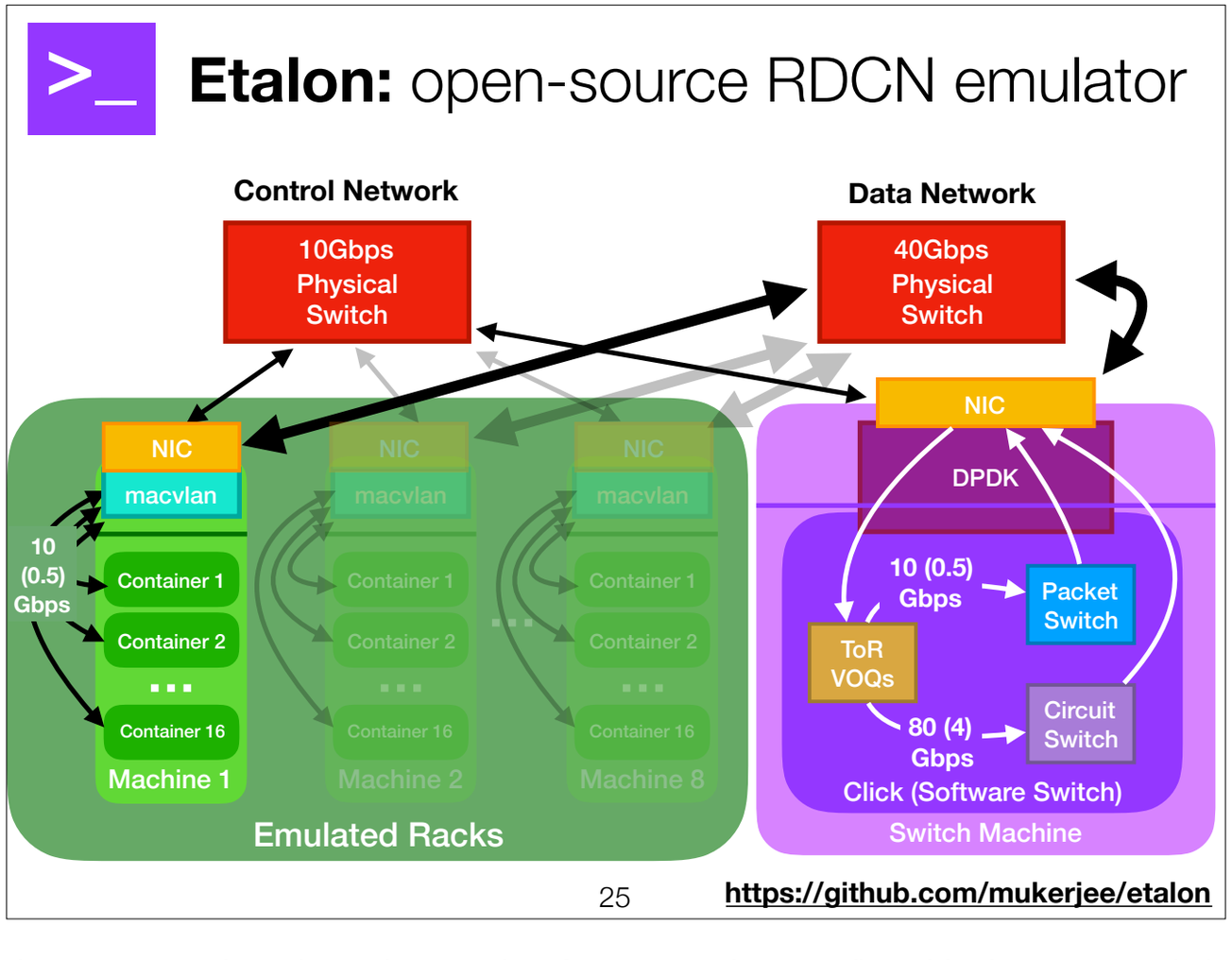
End-to-End Challenges

Challenge: BW Fluct.	**Challenge:** Demand Estimation	**Challenge:** Workloads
Solution: Dynamic Buffer Resizing	**Solution:** Endhost-based Estimation	**Solution:** App-specific Modification



***Etalon*, an RDCN Emulator**

...

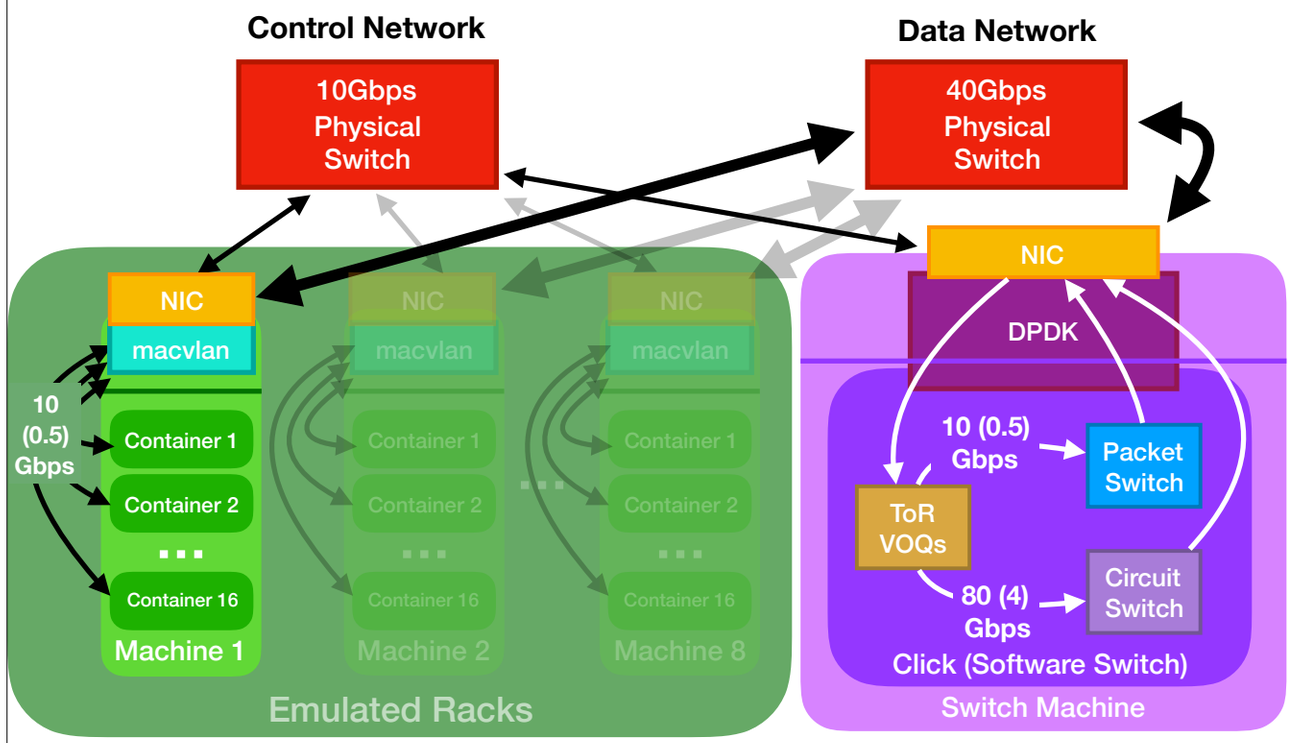


Etalon is an open-source reconfigurable datacenter emulator. It emulates racks of servers and a reconfigurable datacenter switch across multiple physical machines. First, the emulated racks **. ** Each rack is a different physical machine. We emulate “hosts” in the rack using Docker containers **. In our experiments, we have 8 racks with 16 emulated hosts each. The emulated hosts talk to each other using a virtualized switch, Docker’s built-in macvlan **. But something like openvswitch could easily be swapped in. Each container talks to macvlan with a 10Gbps link. Unfortunately, we can’t actually support that many links simultaneously, so we use time dilation (with a time dilation factor of 20x) to instead dilate half gig links to 10Gbps **. Each physical host has a NIC **.

We emulate the reconfigurable datacenter switch on a separate physical machine **, in software using click **. There are three key pieces we emulate in click: the ** ToR VOQs, the ** packet switch, and the ** circuit switch. We emulate the ToR VOQs here, rather than in the emulated racks for convenience. Click gets packets in and out with ** DPDK, to and from the ** NIC. Specifically, packets ** flow from the NIC through DPDK to the proper ToR VOQ. Then either go through ** 10 Gbps links to the packet switch, or through ** 80 Gbps links to the circuit switch. Regardless, packets ** leave the software switch through DPDK and exit through the NIC.

To connect both emulated racks and the emulated switch, we have an underlay physical network. Using Cloudlab’s apt cluster in Utah, our physical network is two switches ** one 10Gbps and one 40Gbps. We use the 40Gbps switch for our ** data network. And the 10Gbps switch for our ** control network, but this split is not strictly necessary. All machine NICs connect to both **.

Etalon: open-source RDCN emulator



https://github.com/mukerjee/etalon

...

Overview

End-to-End Challenges

Challenge: BW Fluct.

Challenge: Demand Estimation

Challenge: Workloads

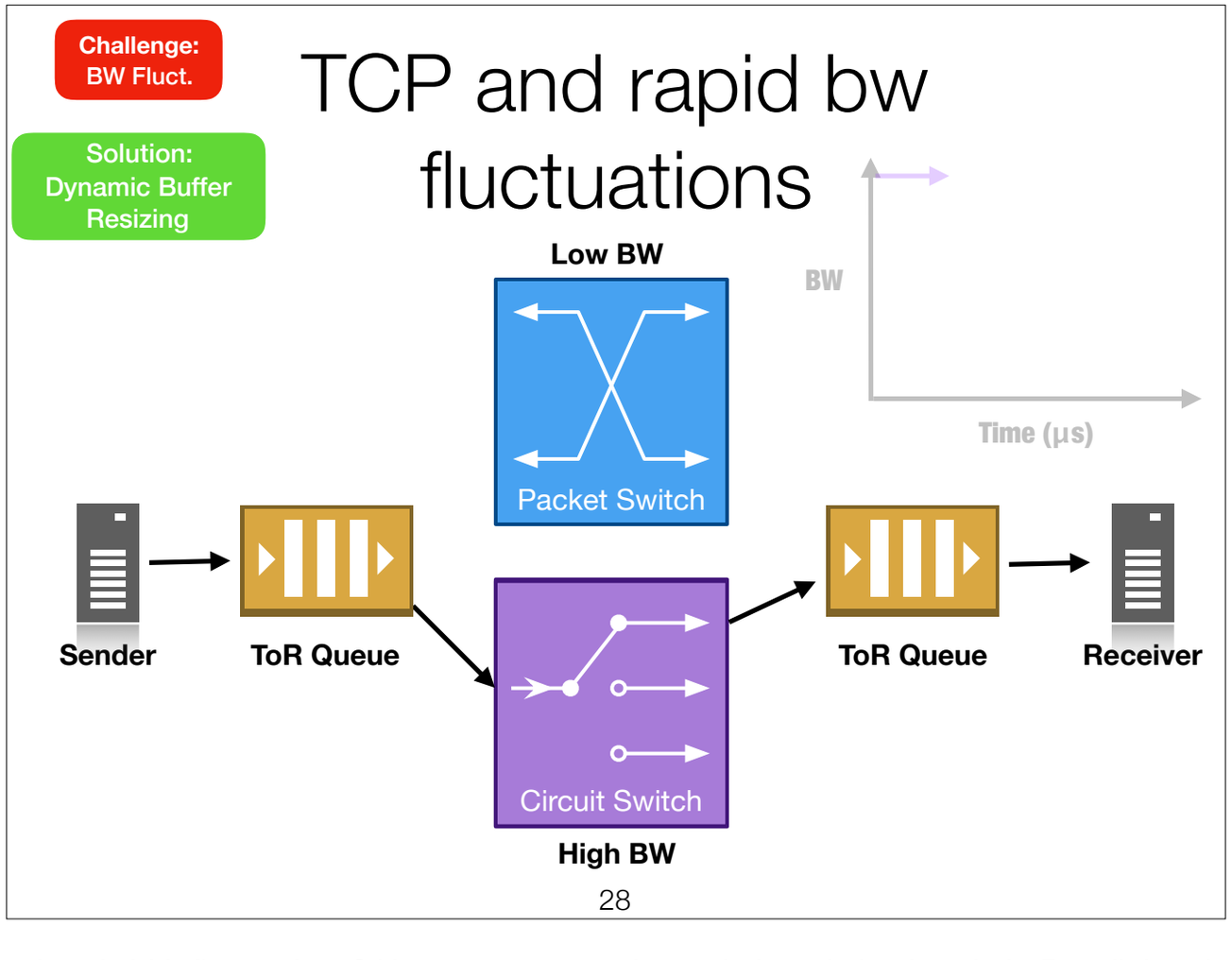
Solution: Dynamic Buffer Resizing

Solution: Endhost-based Estimation

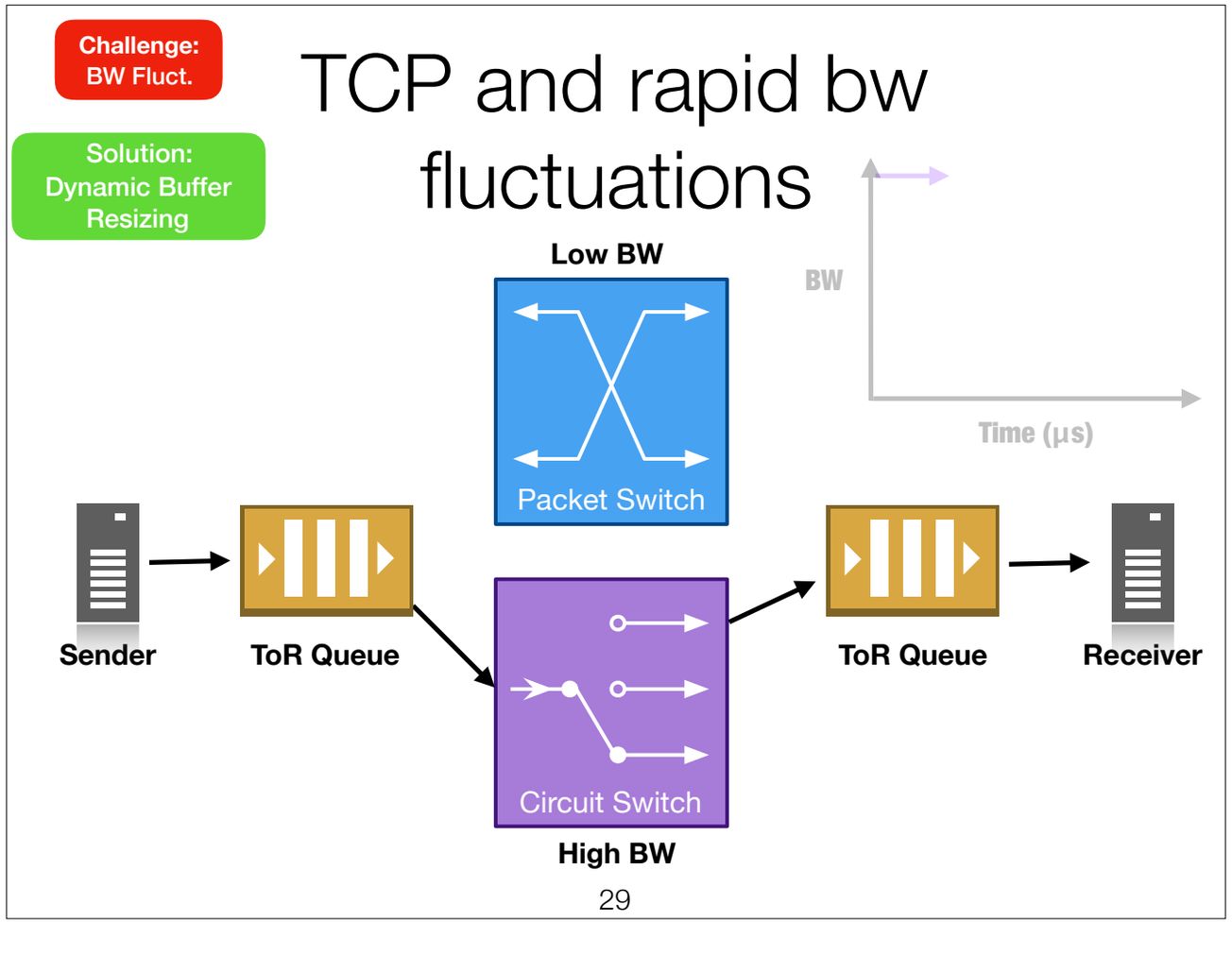
Solution: App-specific Modification

Etalon, an RDCN Emulator

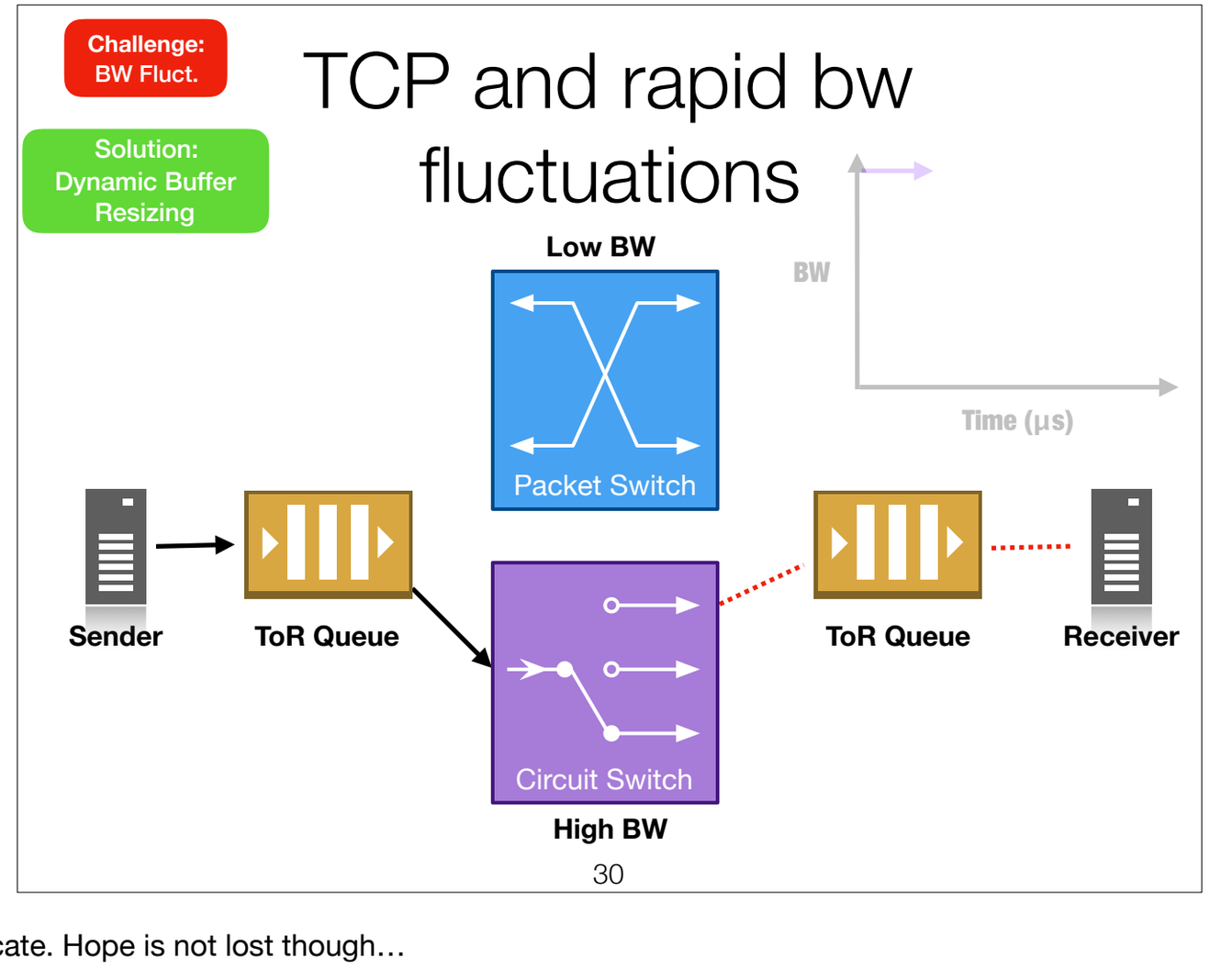
Okay, now that we understand our emulation platform, let’s talk about how we solve two of our challenges. Let’s first look at how we solve rapid bandwidth fluctuation.



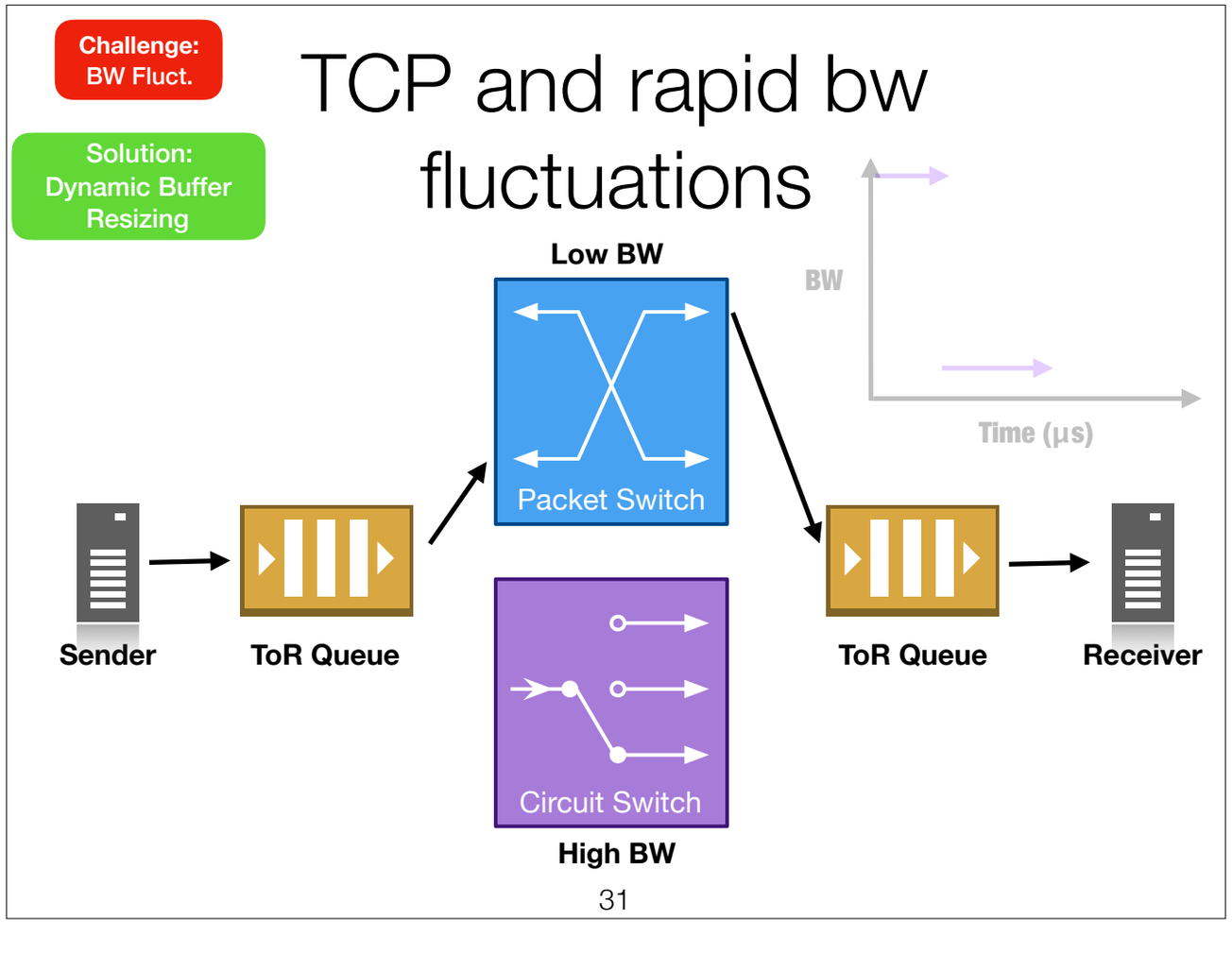
So let’s be really clear. Why are we getting bandwidth fluctuations? Here we see a packet switch and circuit switch. Recall that the circuit switch is ** high bandwidth, and the packet switch is ** low bandwidth. Given some ** sender and some ** receiver that wish to communicate, this picture isn’t really complete. There’s also a ** ToR queue on the sender’s side and on the ** receivers side. If the circuit is current up, packets may flow like this **. We can represent this on a time series plot like this **.



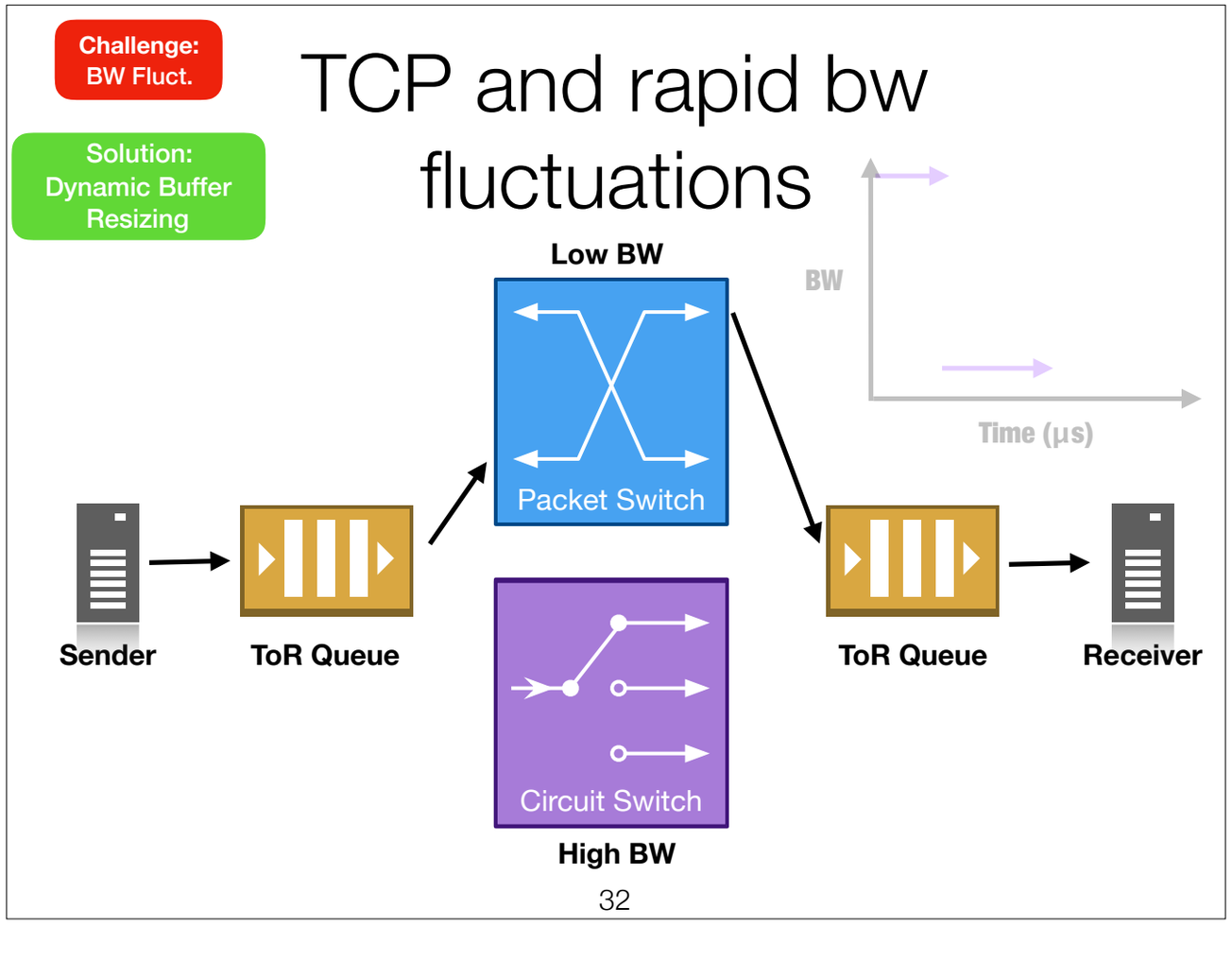
Now that the circuit is down…



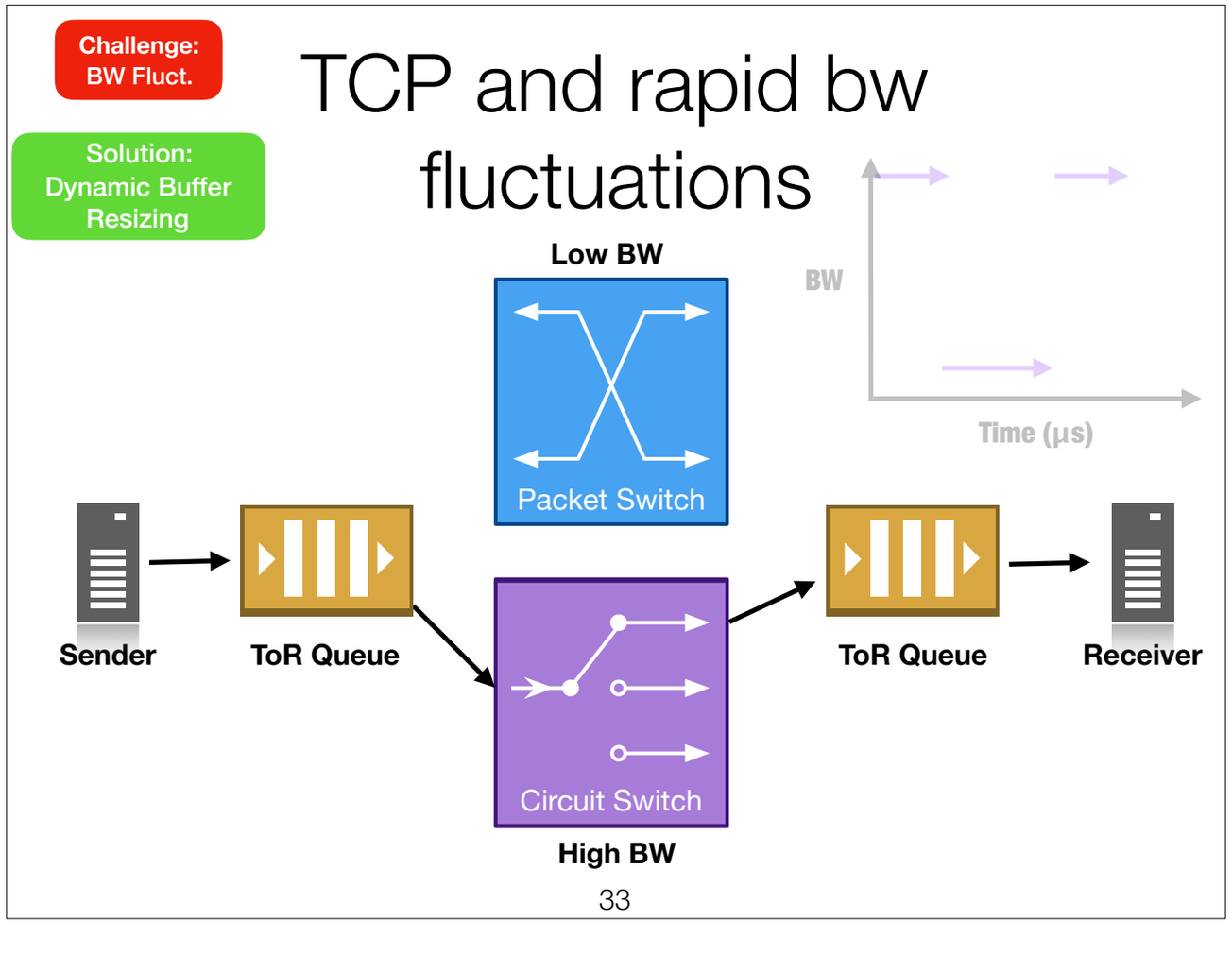
The two hosts can no longer communicate. Hope is not lost though…



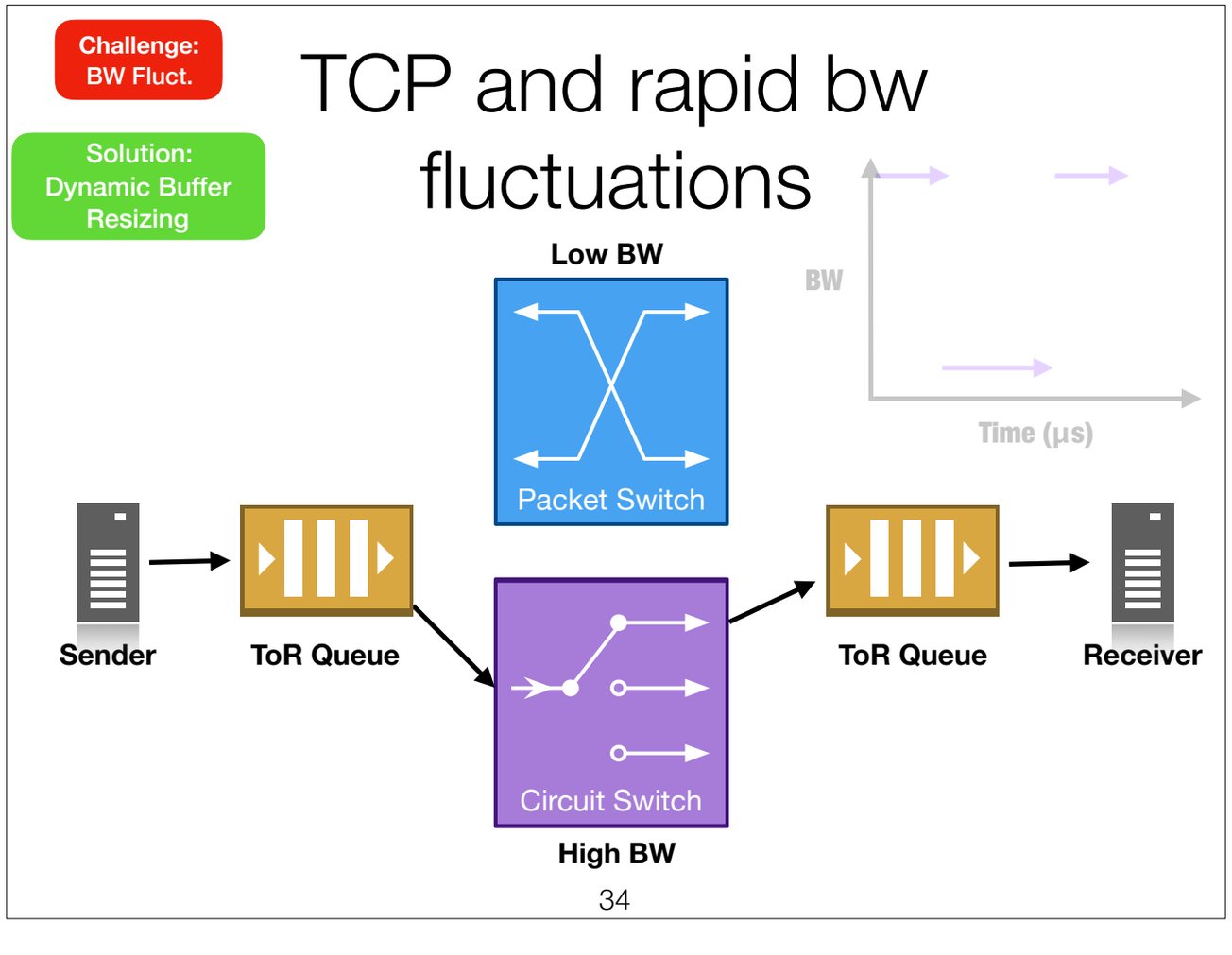
The flow moves to the much slower packet switch and communication can continue **. When the circuit comes back up…



The flow can then move…



back to the high bandwidth circuit switch **.

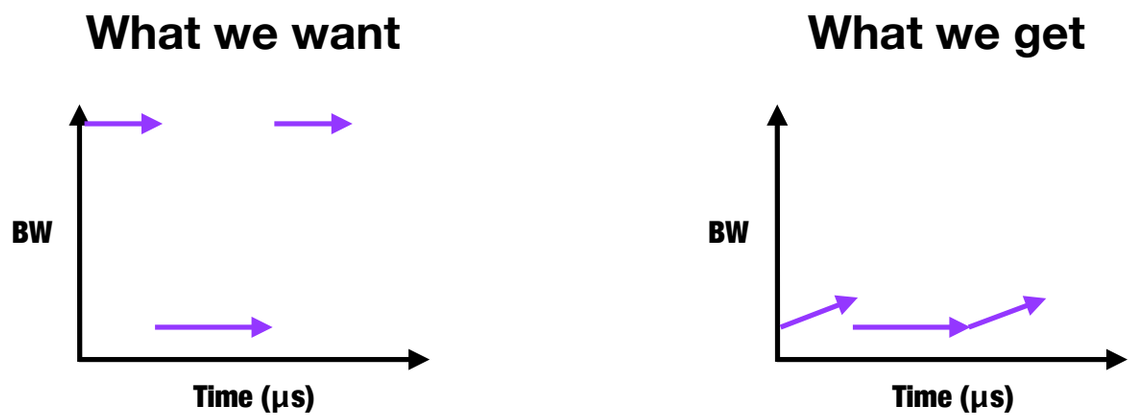

Challenge:
BW Fluct.
Solution:
Dynamic Buffer
Resizing
TCP and rapid bw fluctuations
Low BW
BW
Time (µs)
Packet Switch
Sender
ToR Queue
Circuit Switch
ToR Queue
Receiver
High BW

...

Challenge: BW Fluct.

Solution: Dynamic Buffer Resizing

TCP and rapid bw fluctuations

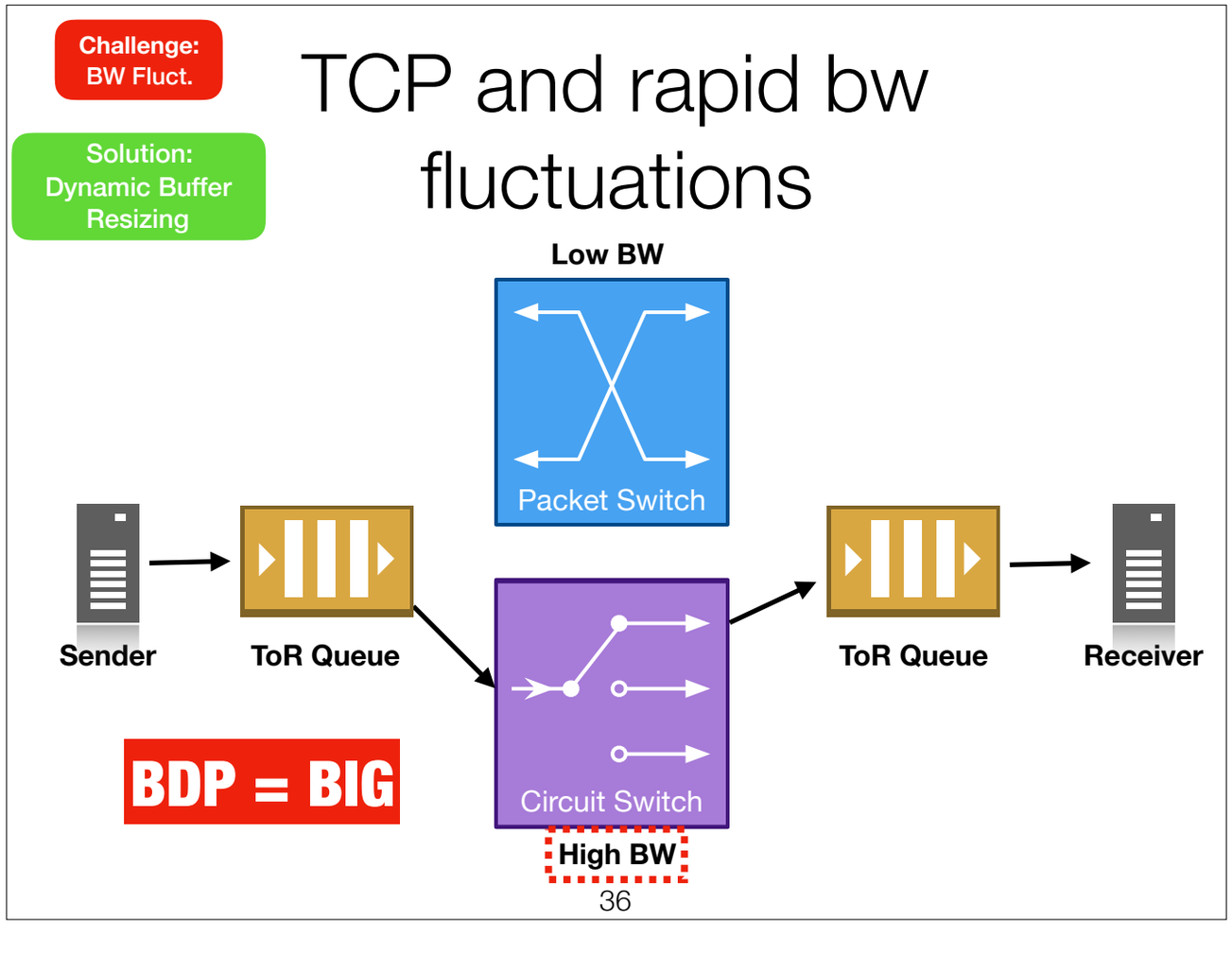


In theory this all works. However, in practice TCP's additive congestion control makes overcoming these bandwidth fluctuations challenging. We would like to see the bandwidth results shown on the left, but in practice we see those on the right as the circuit is simply not up for enough RTTs for TCP to grow large enough to use the circuit bandwidth.

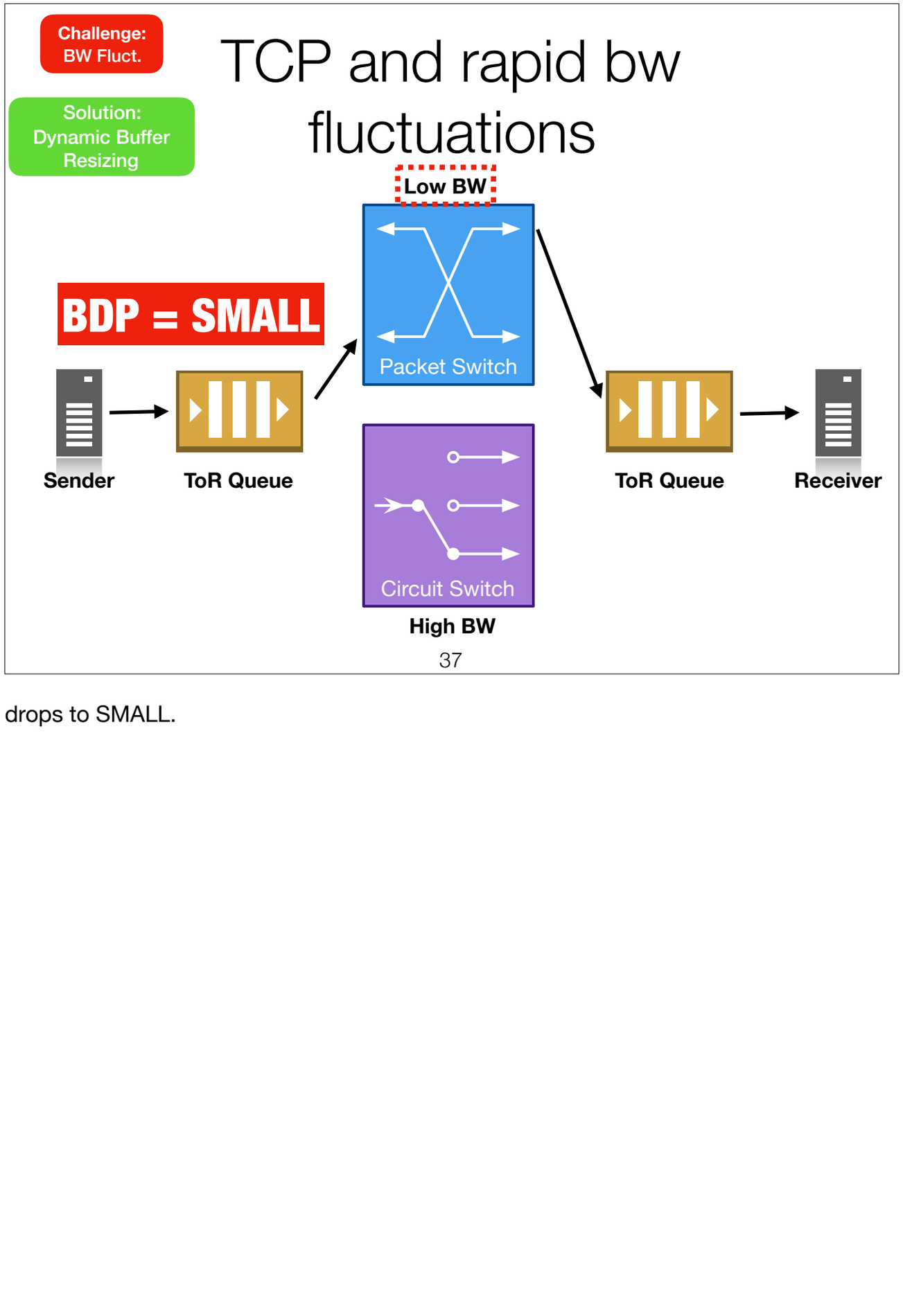
We argue two key insights help us overcome this challenge:

1. While the bandwidth fluctuations are frequent, we know when they're going to happen _in advance_.
2. Since we know in advance when available bandwidth is going to change, we can take advantage of TCPs interaction with bottleneck buffer.

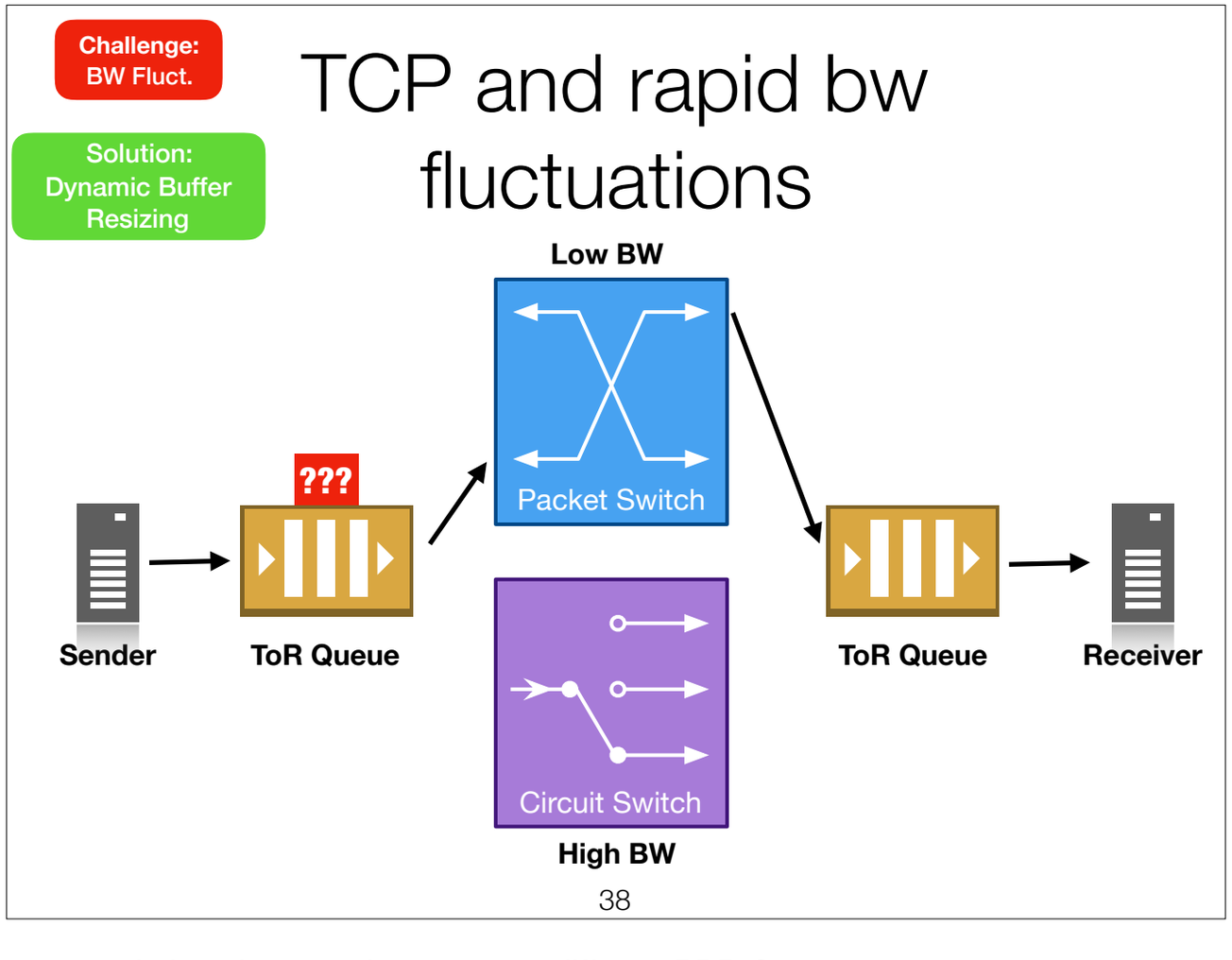
Recall that the rule-of-thumb for optimal TCP bottleneck buffer sizing is the bandwidth-delay product (BDP). Here's where things are a little weird in reconfigurable networks…



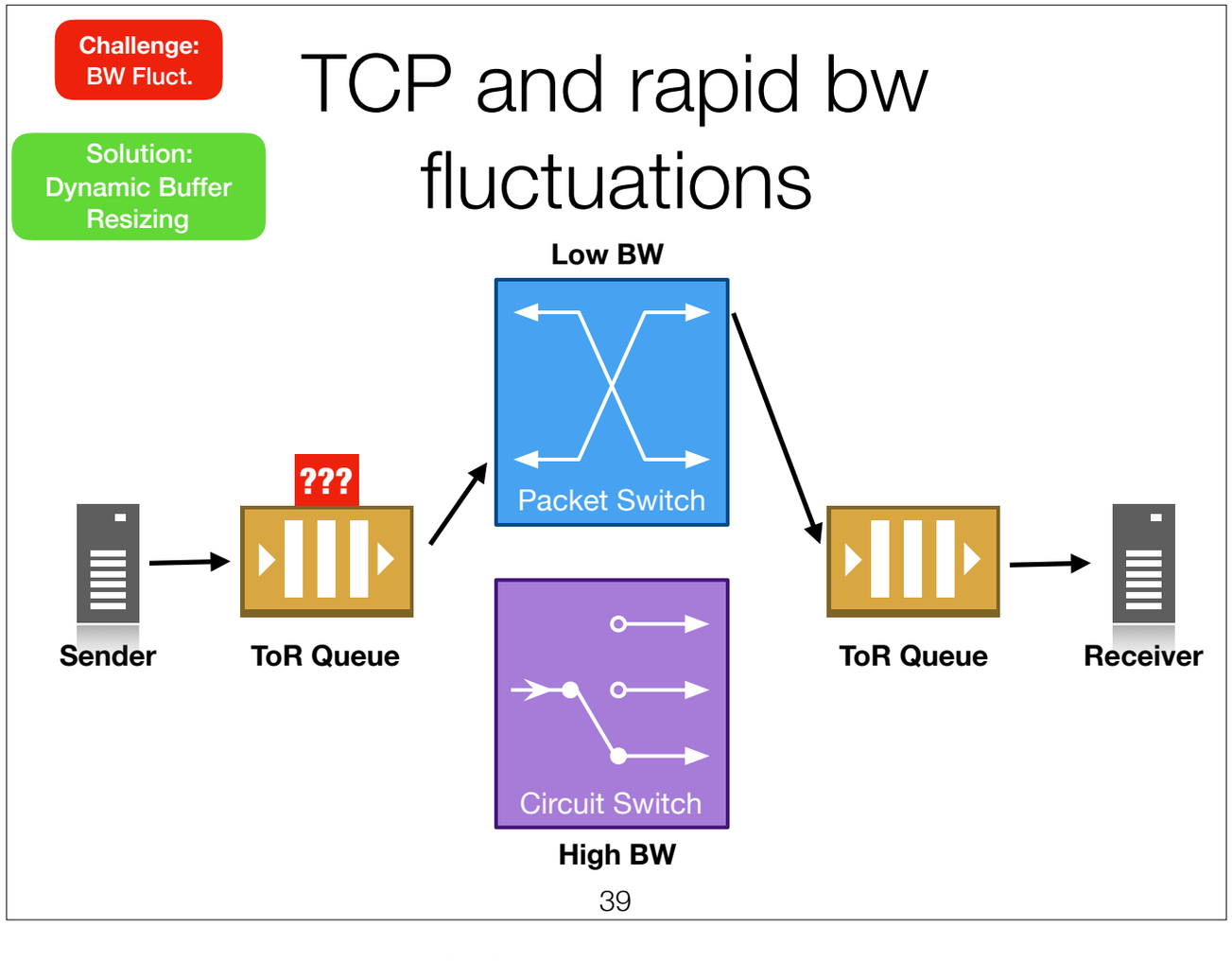
While the circuit is up, our BPD is BIG. If the circuit is down…



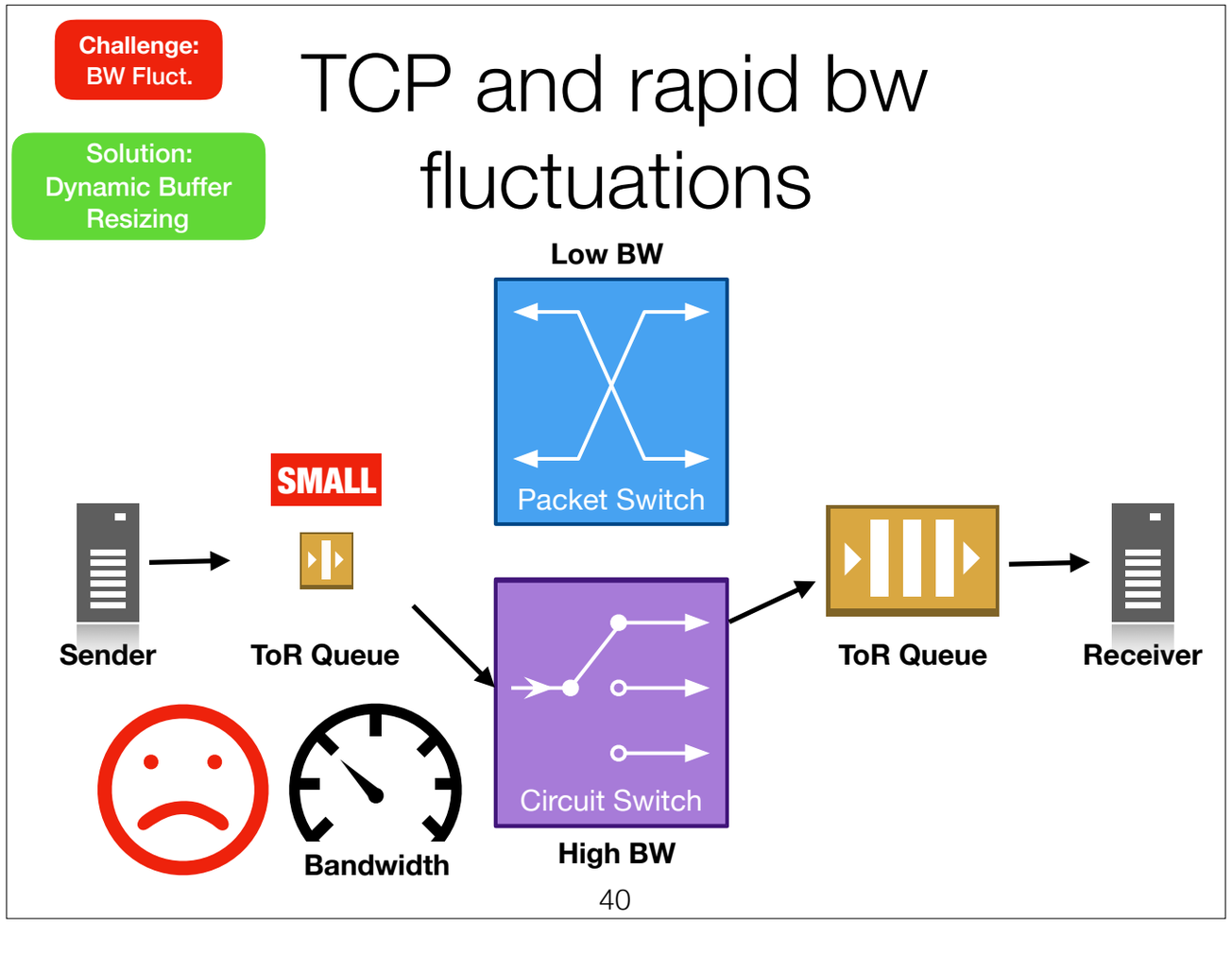
Our available bandwidth (and thus BDP) drops to SMALL.



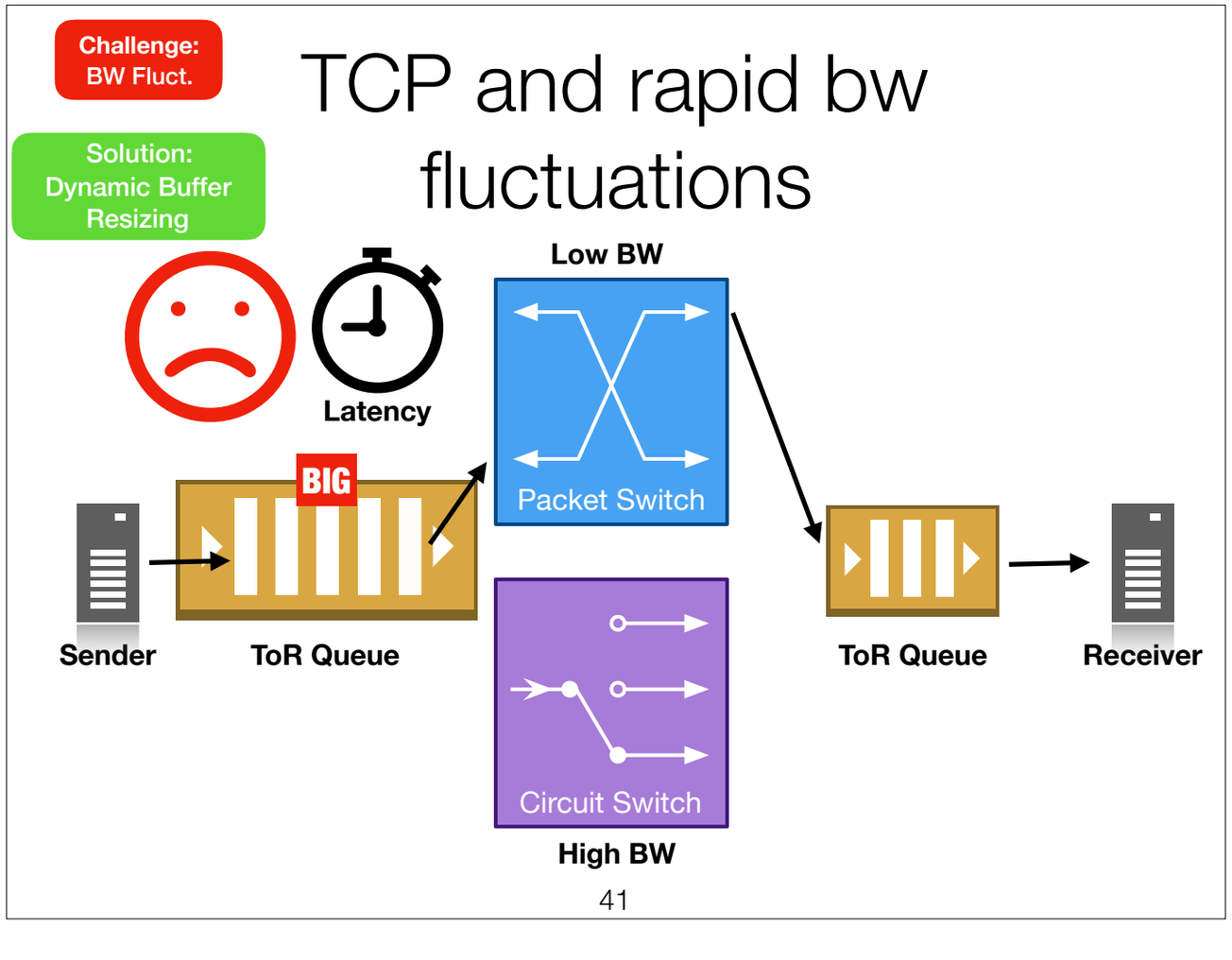
The key question is, what size is right for a network that alternates between two different BDPs?



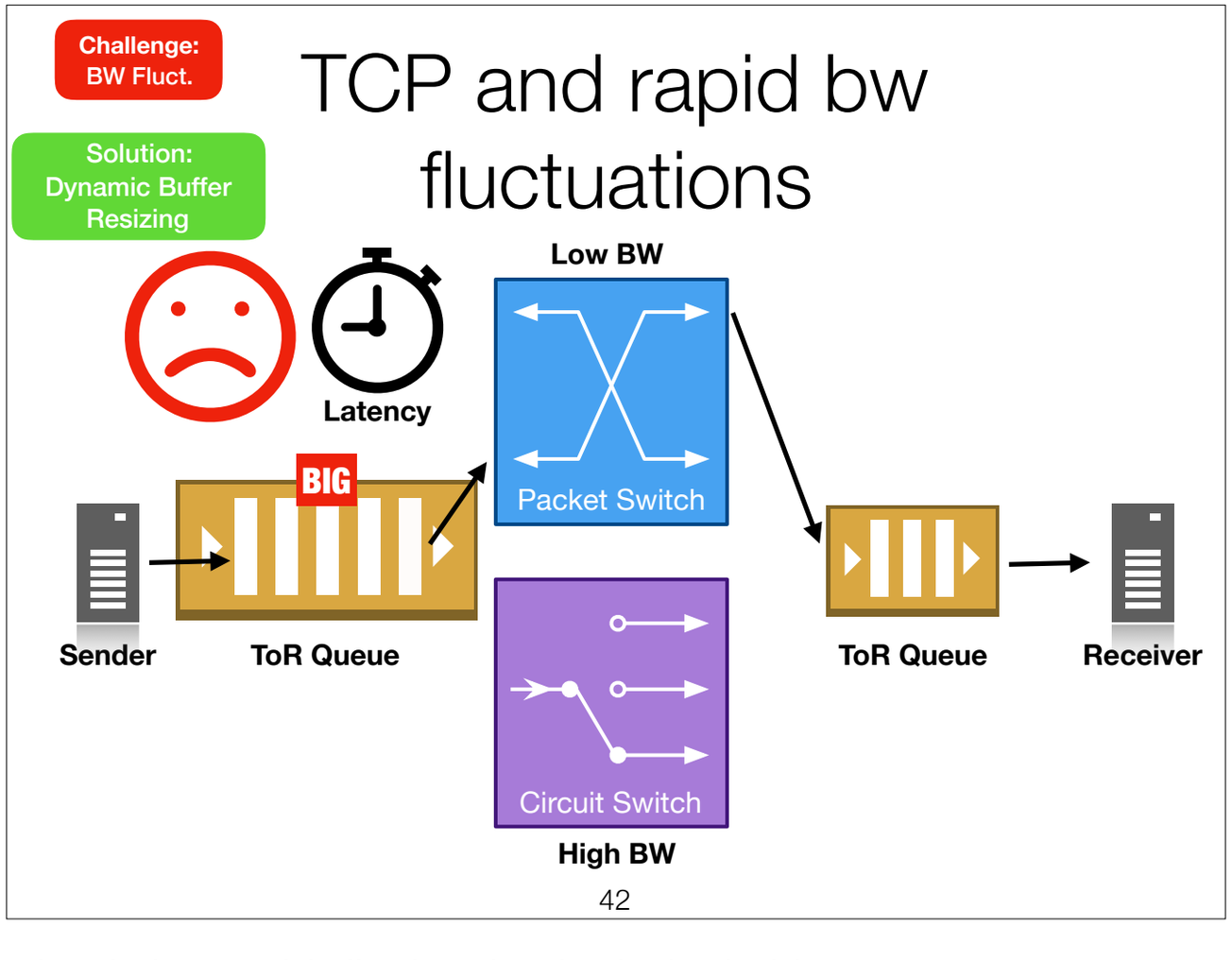
What size is the proper size for a network with two different (alternating) BDPs?



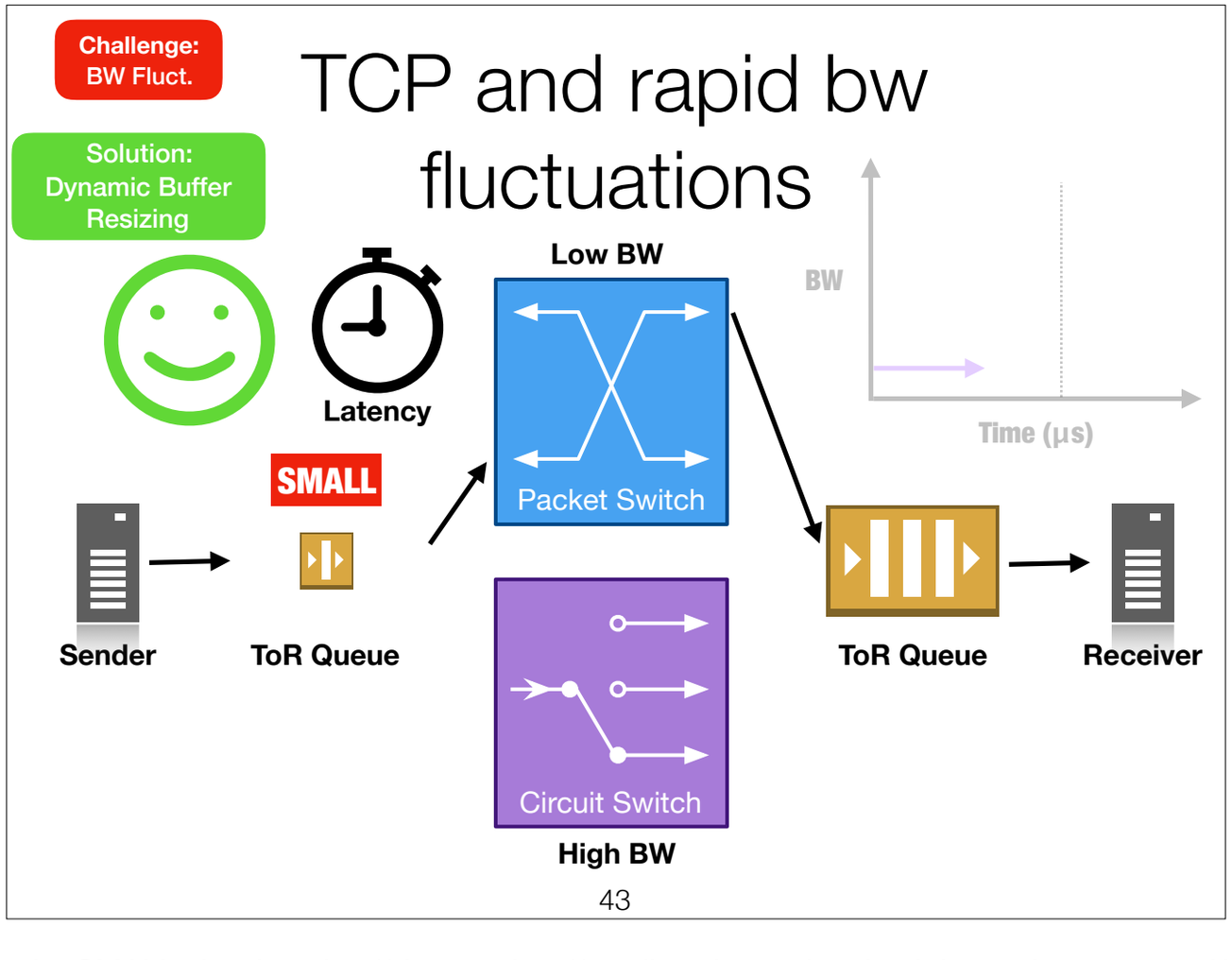
Picking the SMALL size will provide low latency, but not enough ** bandwidth when the circuit switch is up **…



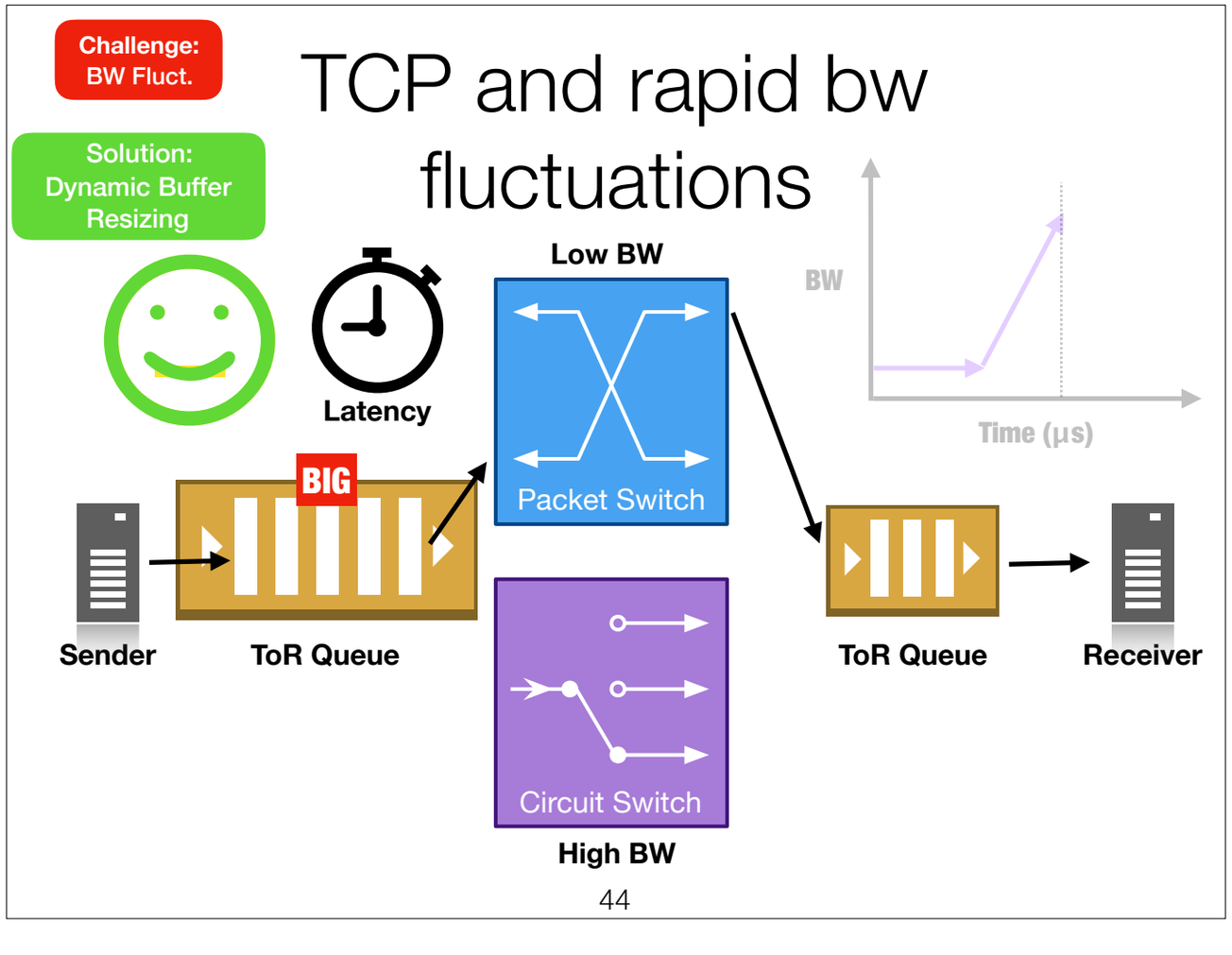
but picking the BIG size will ** inflate latency when using the packet switch **.



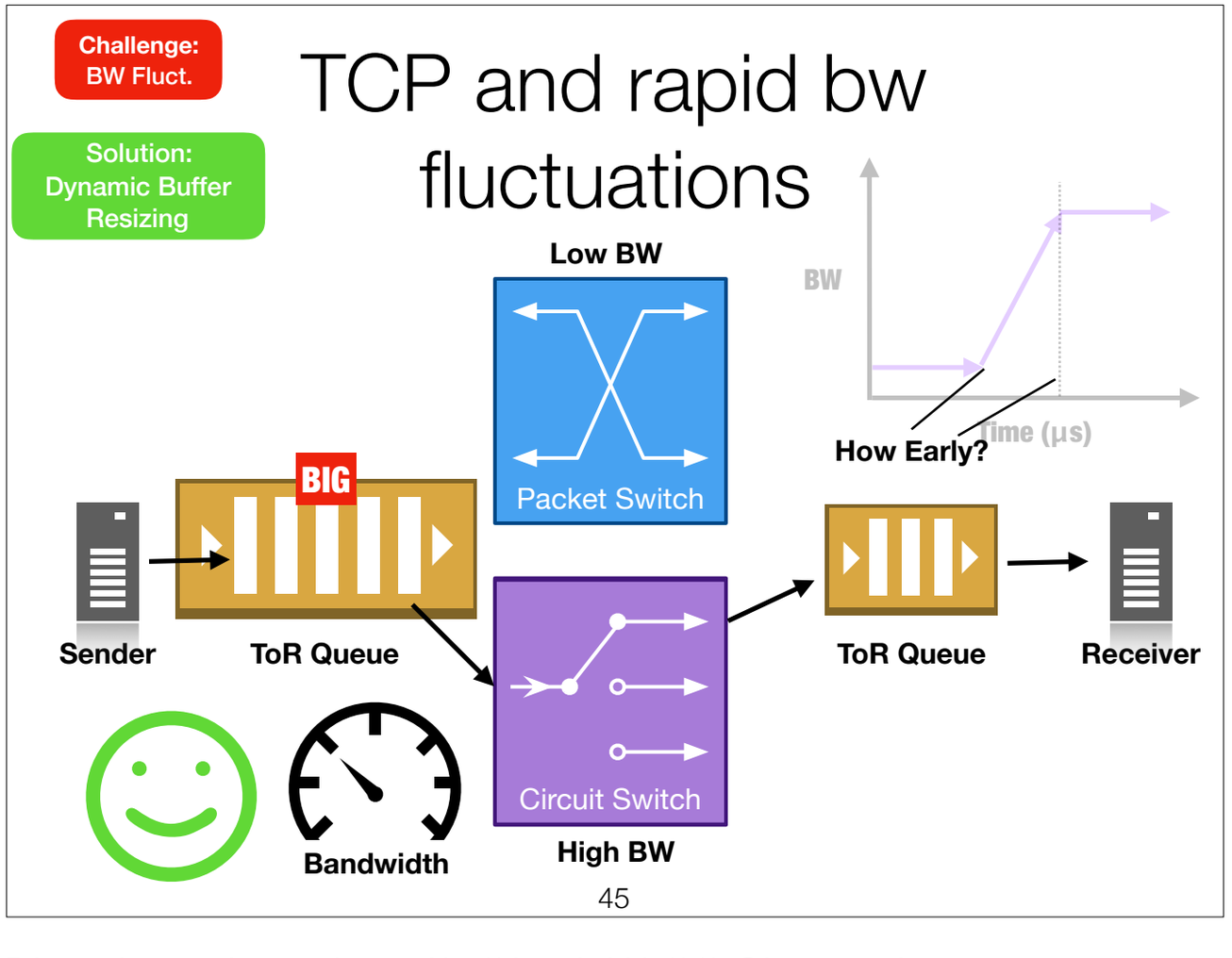
We argue that we need to dynamically _resize_ the in-network buffers based on the circuit schedule.



So, while using the packet switch we use the SMALL size, keeping ** latency low ** until we know the circuit is coming up soon. We can show that with a time series graphs **. The dotted line represents when the circuit is going to come up. While still using the packet switch…



we transition to the BIG buffer size, allowing TCP to ramp up, giving us some latency increase ** (but increases our TCP bandwidth **)…

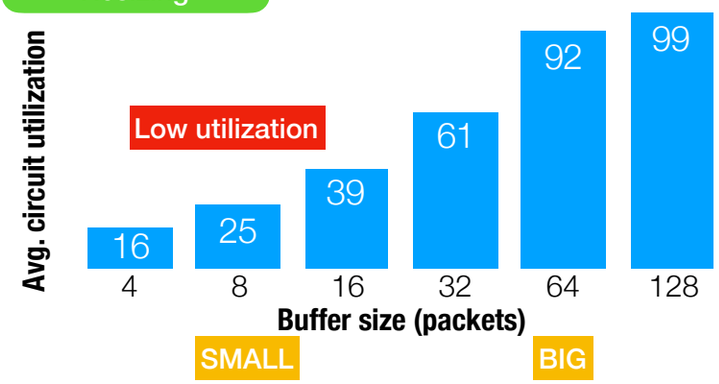


Until the circuit finally comes up, and TCP is ready to make good use of its ** bandwidth ** **. Of course, the parameter we may need to tweak is ** “how early”, we resize the buffer before the circuit starts.

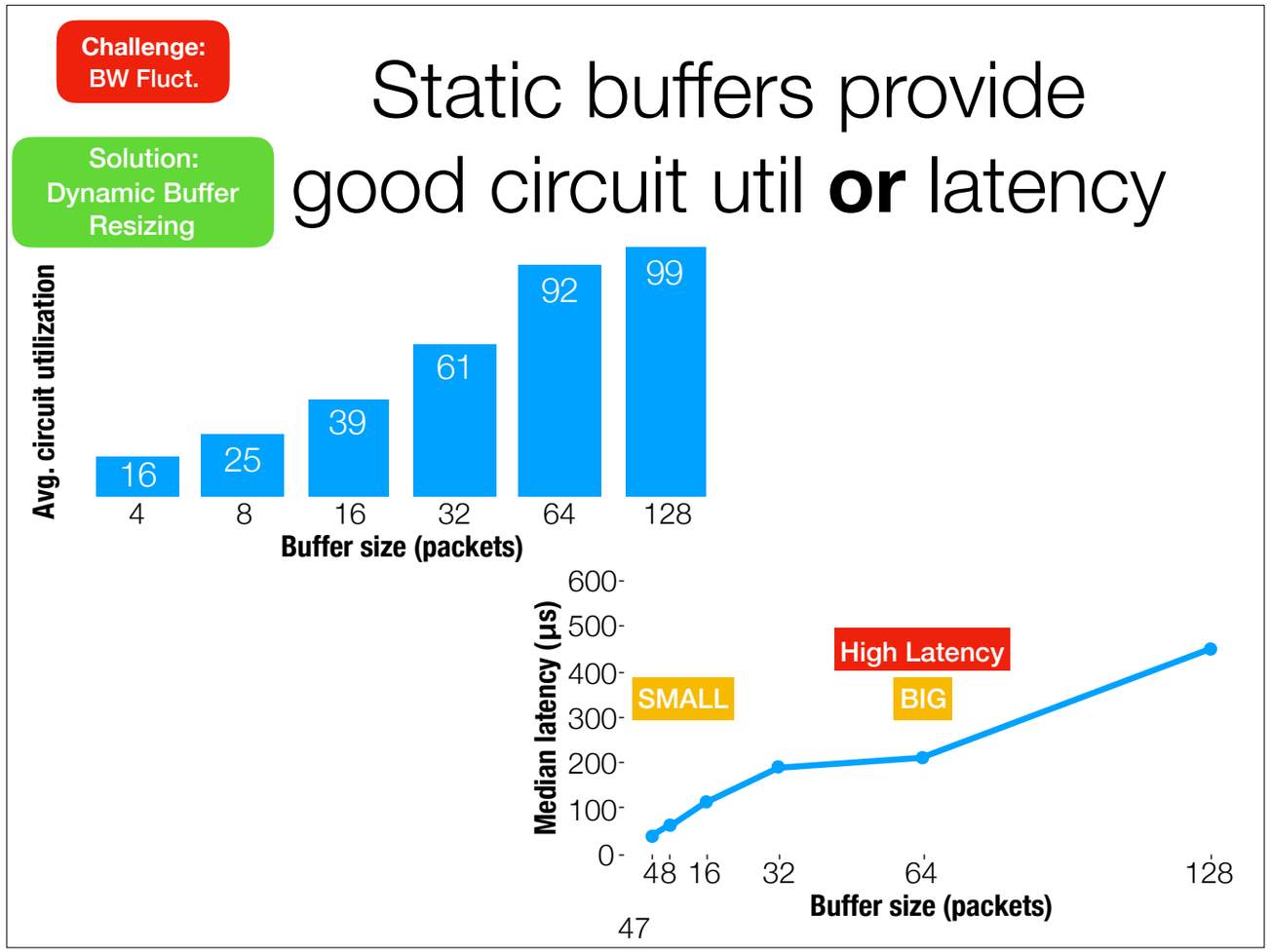
Challenge: BW Fluct.

Solution: Dynamic Buffer Resizing

Static buffers provide good circuit util **or** latency



Let’s look at the key results. First we’re showing how 16 TCP flows between two racks in our emulator behave with static buffers. ** The x-axis shows buffer size. We look at static buffer sizes ** 4 to 128. The ** SMALL buffer size from before would be 8 packets, and the ** BIG buffer size would be around 64 packets. ** The y-axis shows average circuit utilization. Here are the results **. As explained prior, ** the SMALL buffer size not enough to provide good circuit utilization…

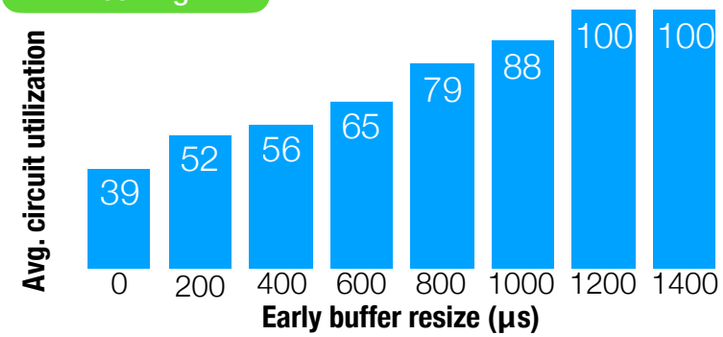


Now let’s look at latency for the same experiment. Again we’re going to look at ** static buffer size **, with ** small still being 8 and ** big still being 64. Now the y-axis will be median latency in microseconds ** **. The key point is what we explained prior. For SMALL buffer size, latency is low, but for the BIG buffer size, ** latency is quite inflated.

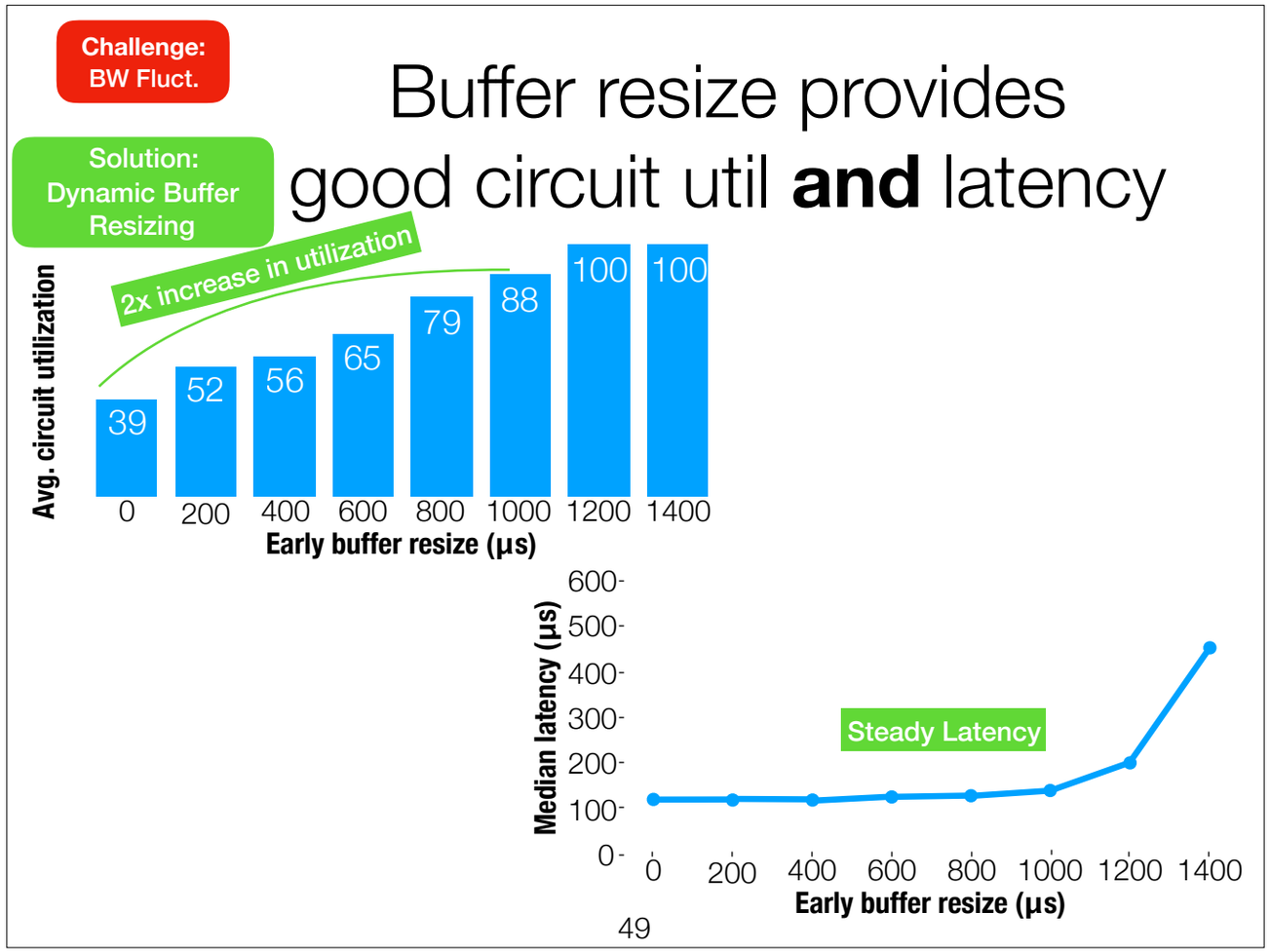
Challenge: BW Fluct.

Solution: Dynamic Buffer Resizing

Buffer resize provides good circuit util **and** latency



Here are there results for our buffer resizing mechanism. The ** x-axis shows how early you resize the buffer before the circuit comes up, in microseconds. ** 0 is no resizing, and 1400 is so early that the buffer stays resized to the BIG size constantly. The ** y-axis again shows circuit utilization. ** We see that we can achieve fairly good circuit utilization around 1000+ microseconds. This of course is uninteresting if we inflate the latency…



So let’s look at the median latency for this same experiment. Again the ** x-axis shows how far in advance we resize our buffer **. The ** y-axis again shows median latency in microseconds **. ** The key takeaway is that up to about 1000 microseconds, ** the median latency stays relatively steady, allowing us to more than ** double our circuit utilization without noticeably increasing median latency.

Overview

End-to-End Challenges

Challenge: BW Fluct.

Challenge: Demand Estimation

Challenge: Workloads

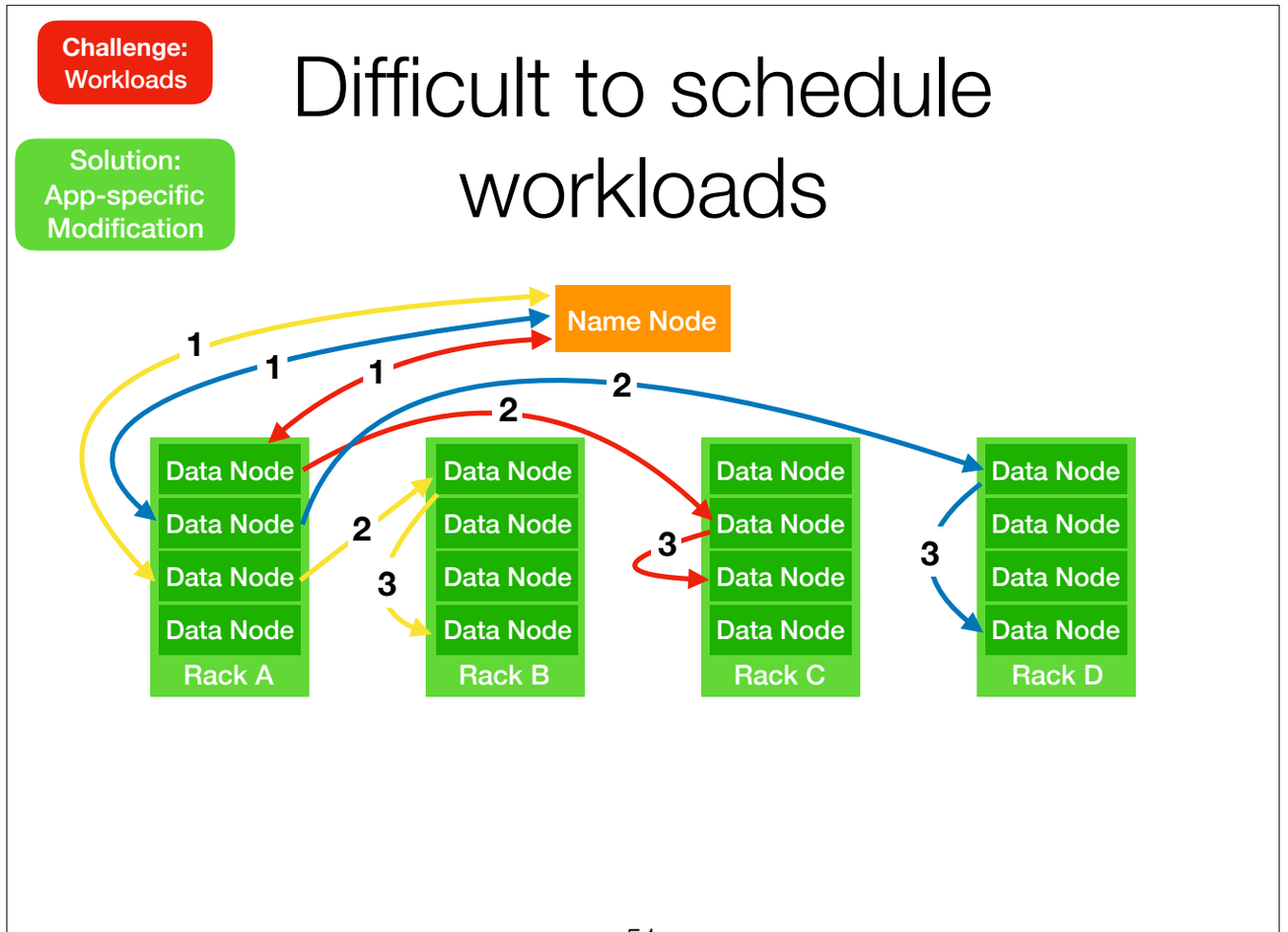
Solution: Dynamic Buffer Resizing

Solution: Endhost-based Estimation

Solution: App-specific Modification

Etalon, an RDCN Emulator

Finally, let’s look at how we combat challenging-to-schedule workloads with simple application-specific modifications.

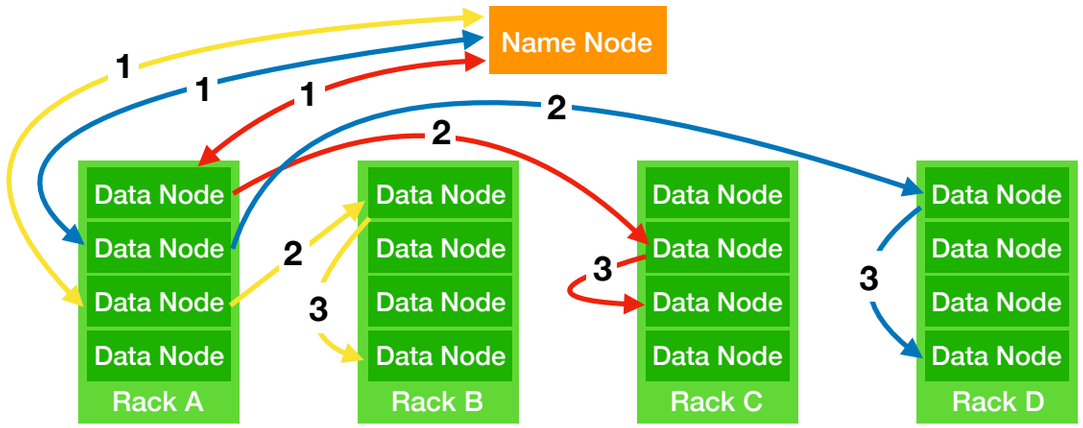


HDFS is a good example. ** HDFS has a bunch of data nodes and ** a logically centralized name node. The goal of HDFS is to write multiple replicas of files to a distributed file system, for fault tolerance. When a data node wants to write a file, it asks the name node where it should place additional replicas **. By default the name node tells the data node to place on copy locally, then the second copy in a _random_ rack, and a third copy, in the same rack as the second copy. This means that different data nodes in the same rack may write to different random racks ** **, or even different files written to by the same data node may go to different random racks, or even chunks within an individual file may be in different racks. While this is generally desired behavior, it makes a strong assumption about what is easy for the network to do…

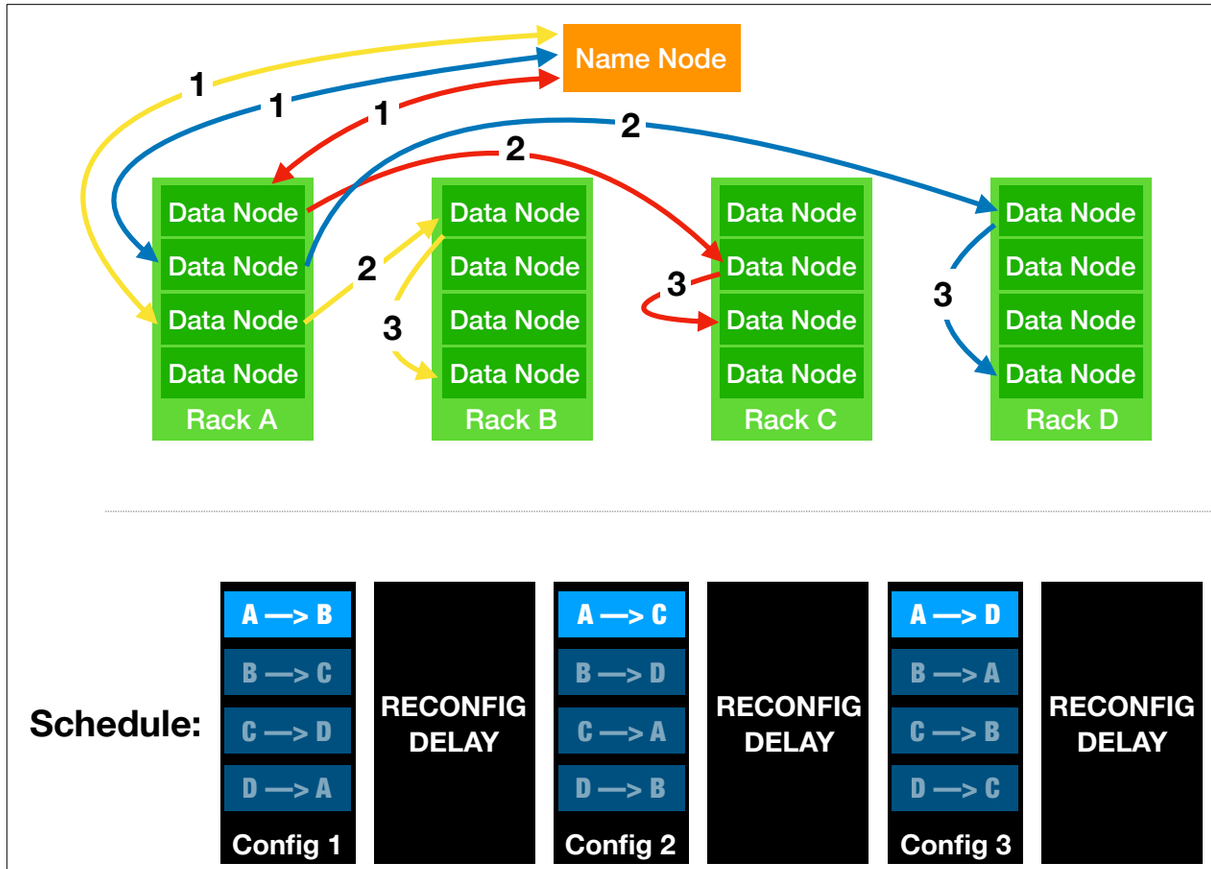
Challenge: Workloads

Solution: App-specific Modification

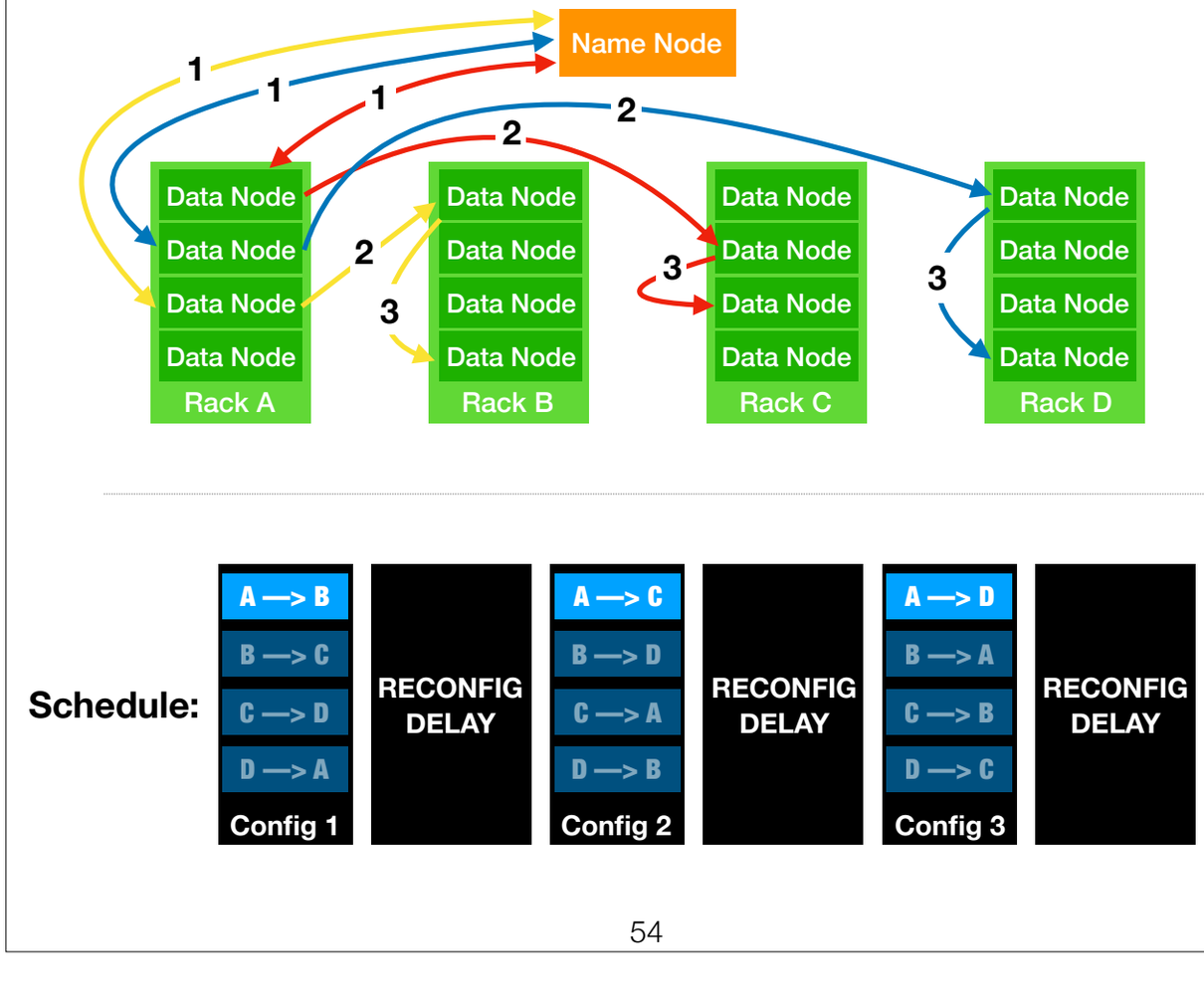
Difficult to schedule workloads



...



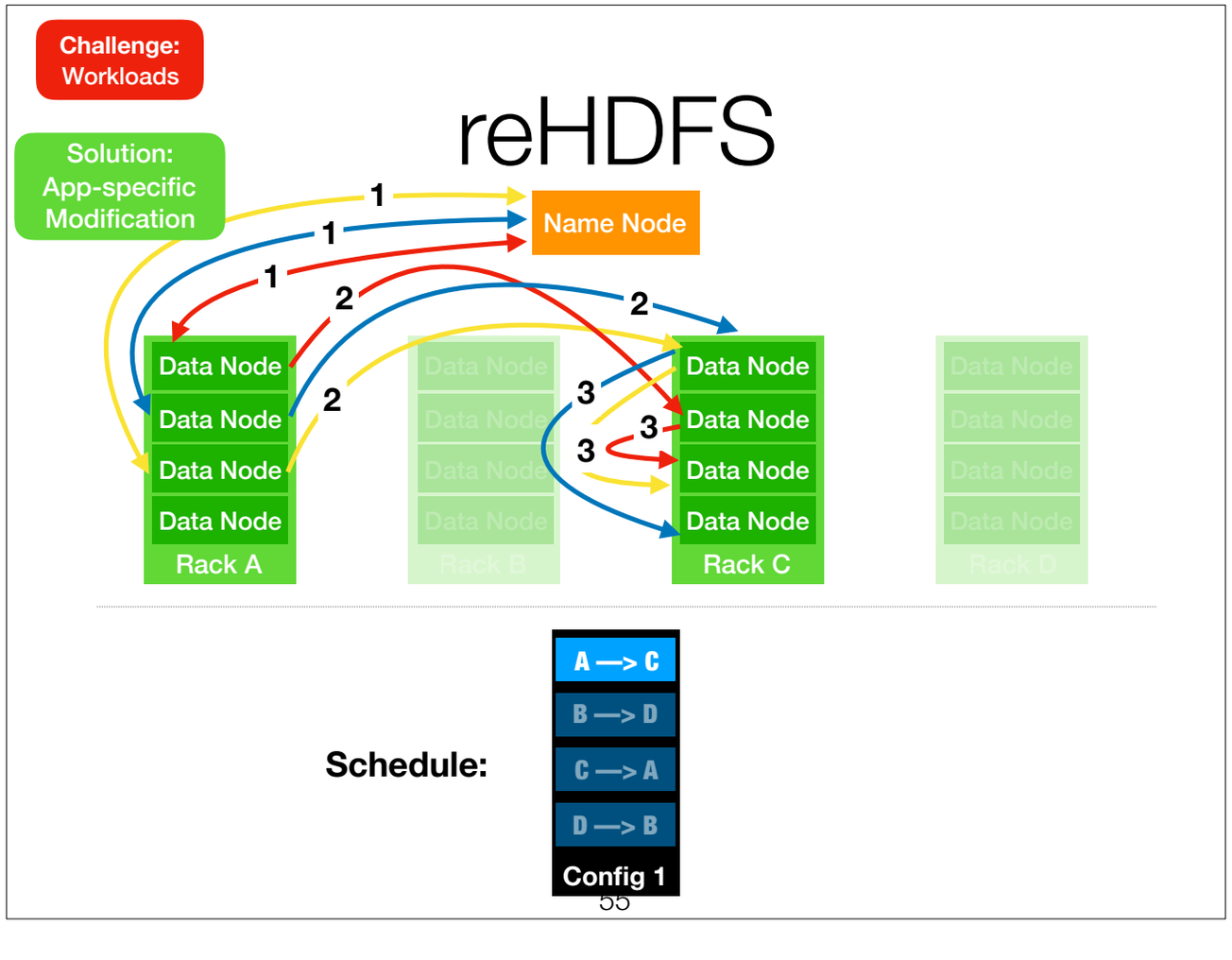
If we examine the workload created by this algorithm, it requires rack A to talk to each other rack. This produces a schedule like this **, requiring a lot of costly reconfiguration delays. The insight is that HDFS assumes that this “all rack to all rack” workload is just as easy to deliver as any other traffic pattern (which is maybe a fair assumption for networks today). We argue that HDFS writes, however, don’t fundamentally require an all-to-all workload, and through a simple tweak we can greatly improve it’s performance in reconfigurable datacenters.


Name Node
1
1
1
2
2
2
3
3
3
Data Node
Data Node
Data Node
Data Node
Rack A
Rack B
Rack C
Rack D

Schedule:

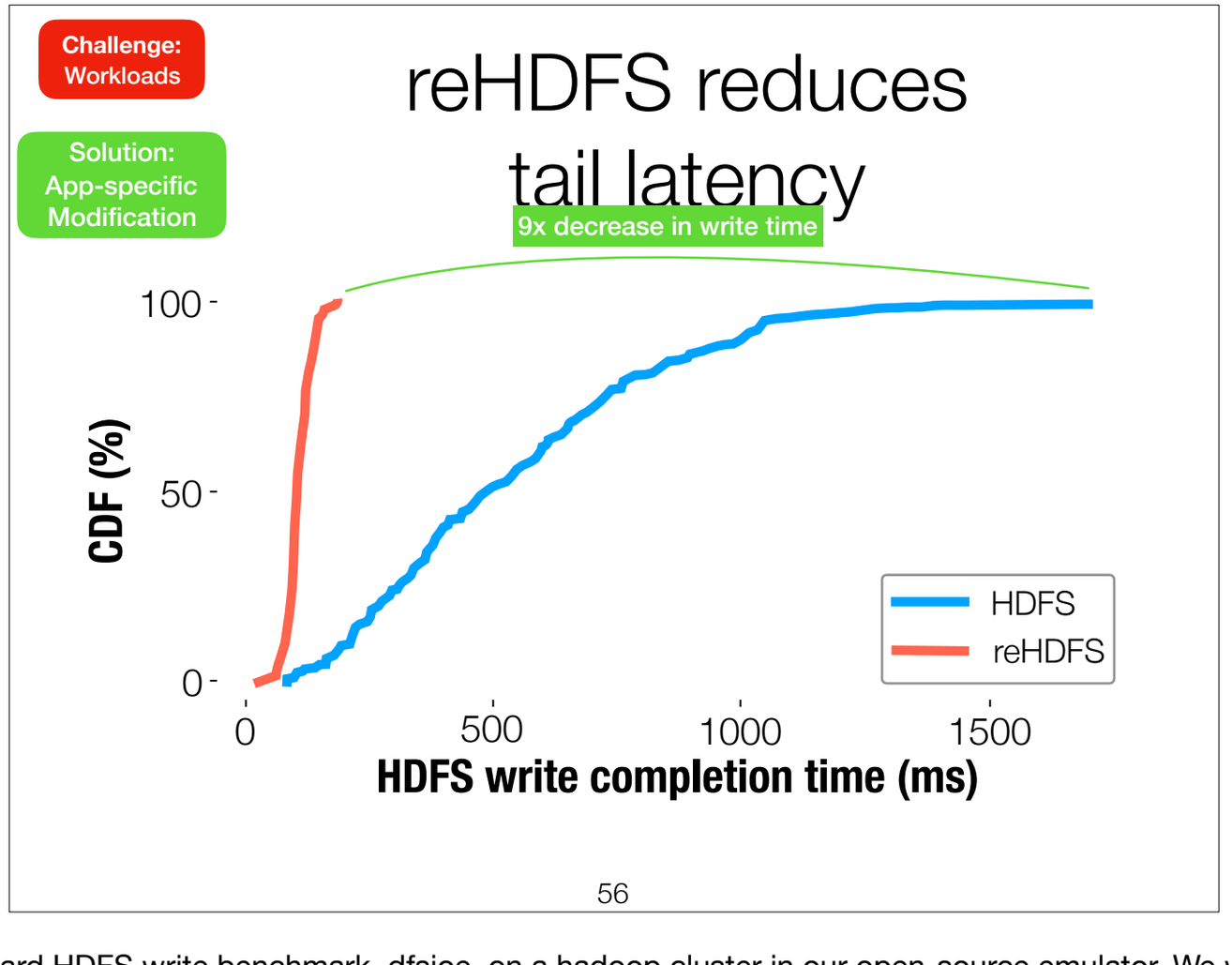
A —> B
B —> C
C —> D
D —> A
Config 1
RECONFIG DELAY
A —> C
B —> D
C —> A
D —> B
Config 2
RECONFIG DELAY
A —> D
B —> A
C —> B
D —> C
Config 3
RECONFIG DELAY

...



We make a modified version of HDFS' write placement algorithm, requiring only 5 additional lines of code. We call this version reHDFS for "reconfigurable datacenter network HDFS". The change is simple; when a data node asks the name node where it should place replicas, instead of selecting a random additional rack, it selects one based on the rack the requesting data node is in. For example, if the data node is in rack A it always writes to a random node in rack C ** ** **, if the initial data node is in rack B it writes to rack D, etc. This significantly changes the workload; ** rack A only needs to talk to rack C, requiring only 1 circuit configuration.

It might sound like this change may tie two racks together during failures, but you could just change the function every second (e.g., now rack A writes to rack D, etc.) and that would give you almost all the benefit without the worry.

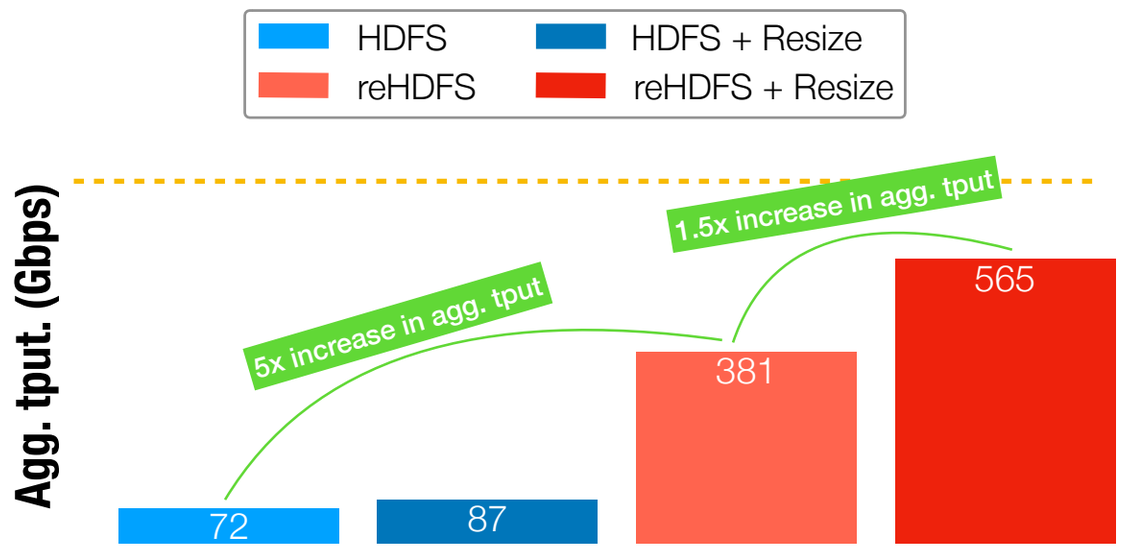


Here are the key results. We run a standard HDFS write benchmark, dfsioe, on a hadoop cluster in our open-source emulator. We write 64 files to HDFS with size 400MB each. On the ** x-axis we show HDFS write completion time in milliseconds **. On the ** y-axis we show the CDF **. We compare ** vanilla HDFS to our ** reHDFS. ** We find that in the tail ** we can reduce HDFS write completion time by 9x.

Challenge: Workloads

Solution: App-specific Modification

reHDFS + resize increases agg. tput.



Next we show cumulative results combining reHDFS with buffer resizing. We're going to look at this in terms of ** aggregate throughput in Gbps. ** This line represents the calculated maximum cluster throughput. We compare ** HDFS, ** reHDFS, ** HDFS + buffer resizing, and ** reHDFS + buffer resizing. ** We see a ** 5x increase in aggregate throughput moving from HDFS to reHDFS, and an additional ** 1.5x increase when we add buffer resizing.
The takeaway is that higher-layer optimizations, while painful, can provide more benefit than lower-layer ones.

Other results in the paper

- Bandwidth fluctuations + TCP
  - Comparisons with our custom TCP CC for RDCNs
  - 50th and 99th percentile latency comparisons
  - link delay sensitivity analysis
- Endhost-based demand estimation
  - Multiple workloads
  - Buffer resizing + ADU cumulative experiments
  - Comparisons with packet switches of different sizes

Overcoming End-to-End Challenges in Reconfigurable Datacenter Networks

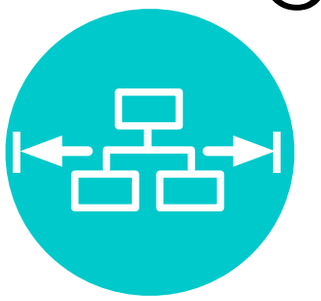
Paper # 58, 13 pages

Abstract

1 Introduction

We have more results in the paper. For bandwidth fluctuations + TCP, we have additional comparisons to our own custom TCP congestion control variant for reconfigurable datacenters, some additional latency comparisons, and links delay sensitivity analysis. We also have a whole additional challenge regarding demand estimation which we solve by involving endhosts directly.

Conclusion

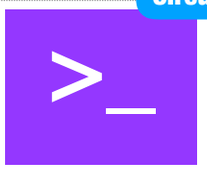


End-to-End Challenges

Challenge: BW Fluct. → Solution: Dynamic Buffer Resizing → **2x Circuit Util**

Challenge: Demand Estimation → Solution: Endhost-based Estimation → **8x FCT**

Challenge: Workloads → Solution: App-specific Modification → **9x Write Time**



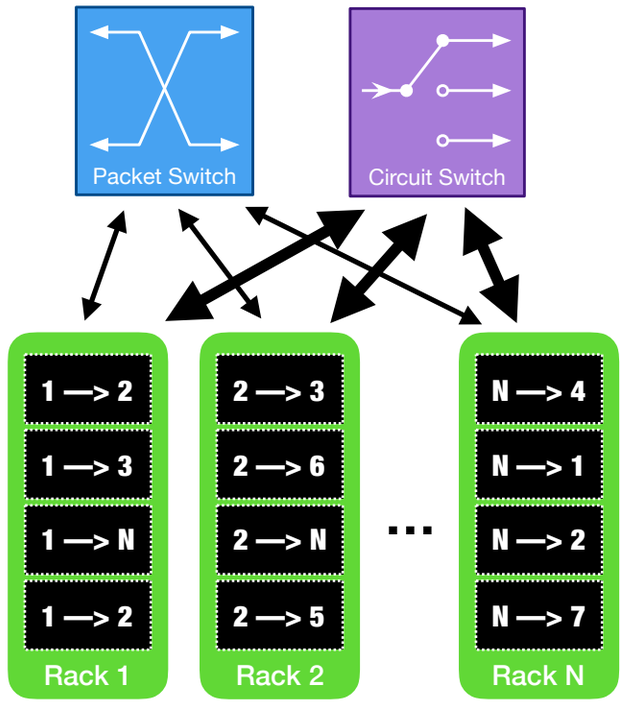
***Etalon*, an RDCN Emulator**

https://github.com/mukerjee/etalon

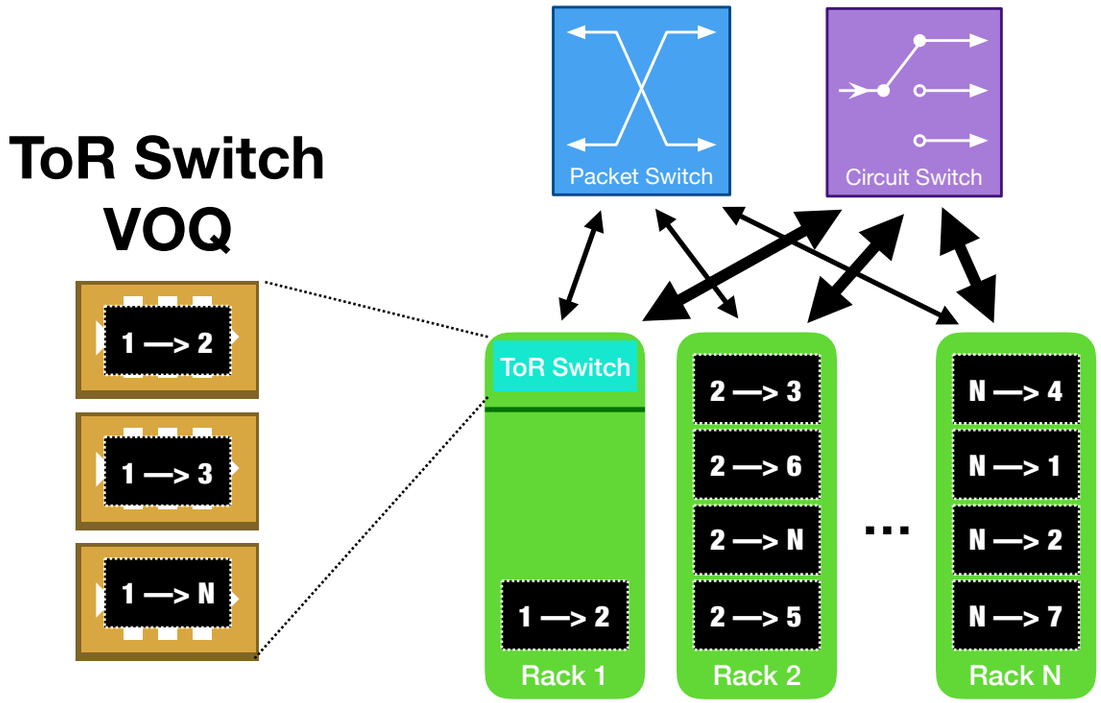
In conclusion, in this work we point to a new set of problems in reconfigurable datacenter networks which is end-to-end challenges. We point to three critical end-to-end challenges and their solutions. We find that rapid bandwidth fluctuation can be solved by dynamic buffer resizing, providing us with ** 2 times the circuit utilization, we find that demand estimation can be solved using endhost-based estimation, giving us ** 8 times better flow completion time, and we find that challenging workloads can be fixed with app-specific modification, providing us with ** 9 times faster HDFS write times. We did all these experiments in our open-source reconfigurable datacenter emulator, Etalon. If you are at all interested in reproducing these experiments or doing further research in this space, please take a look at Etalon. With that, I'm happy to take questions, thank you.

Backup Slides

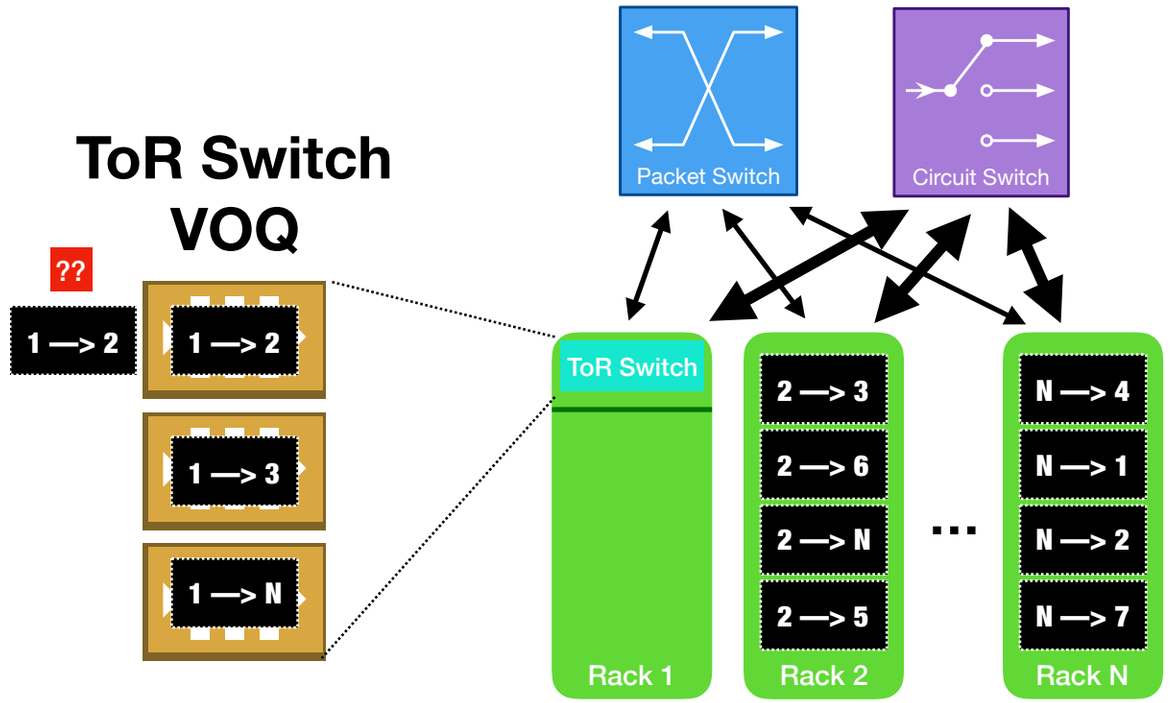
Use PFC instead of dynamic buffer resize?



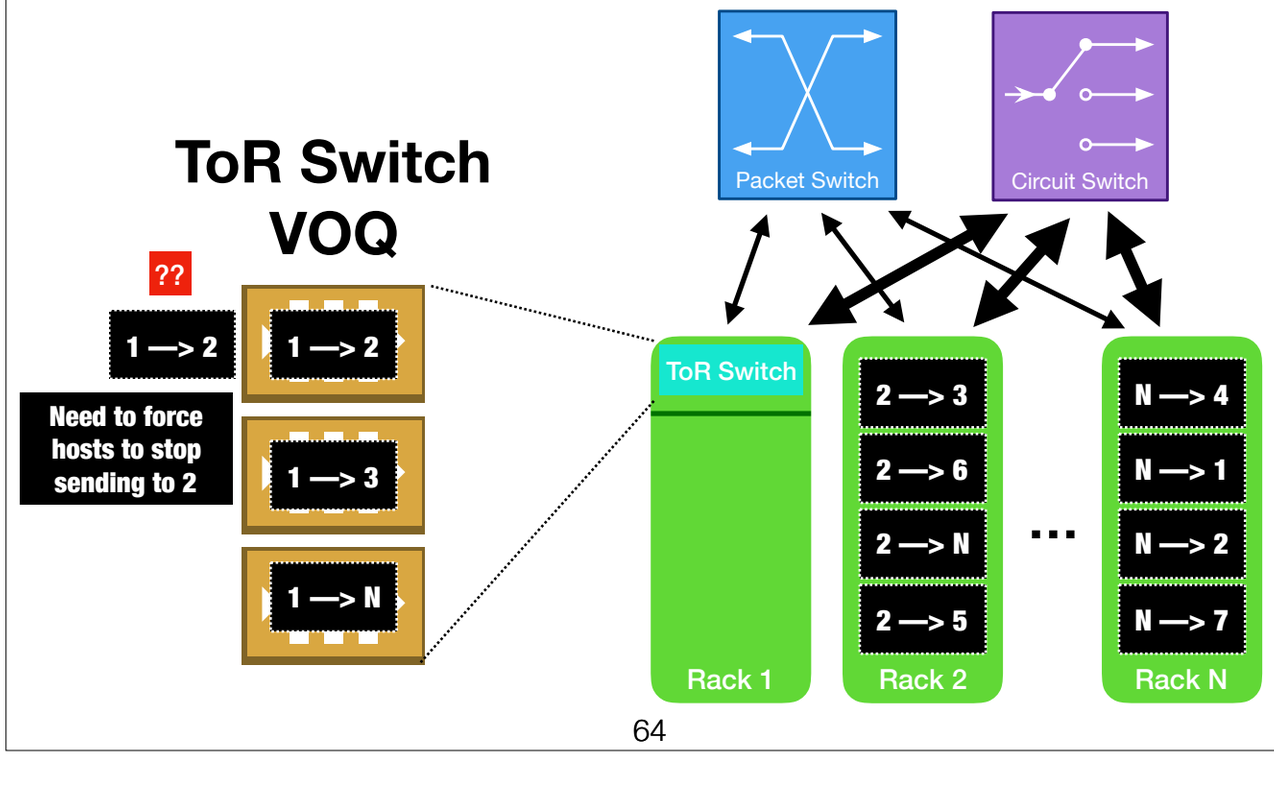
Use PFC instead of dynamic buffer resize?



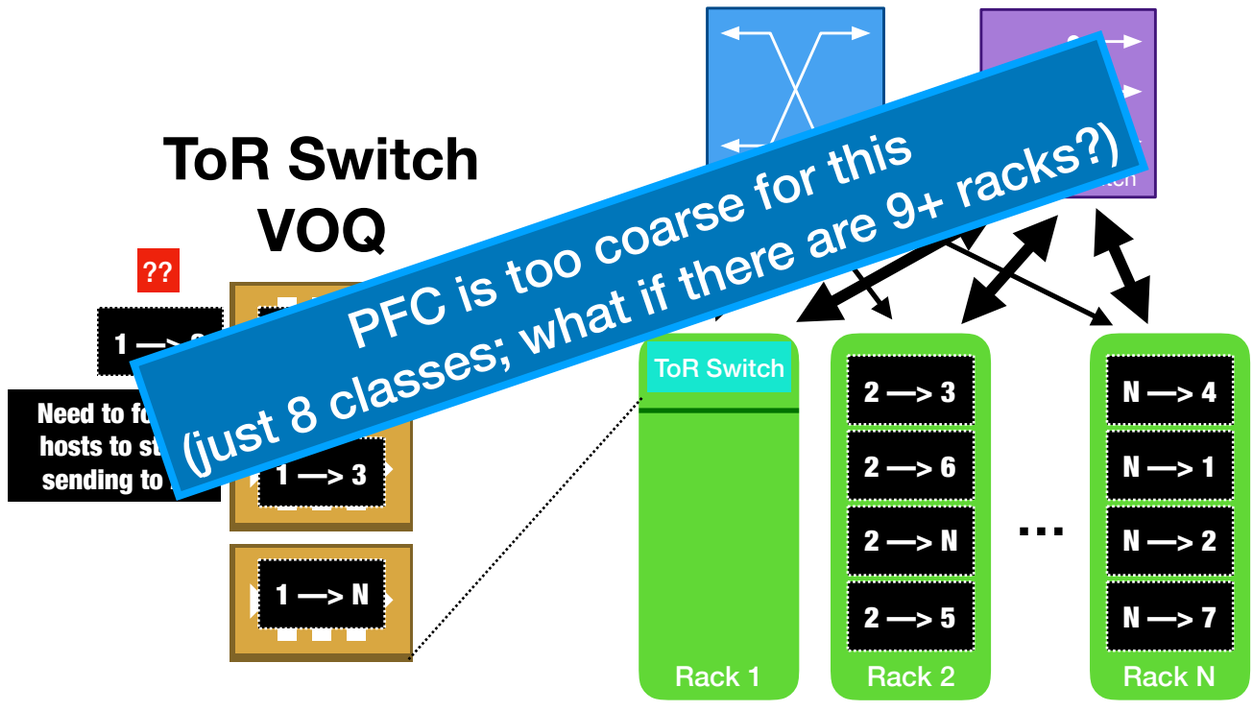
Use PFC instead of dynamic buffer resize?



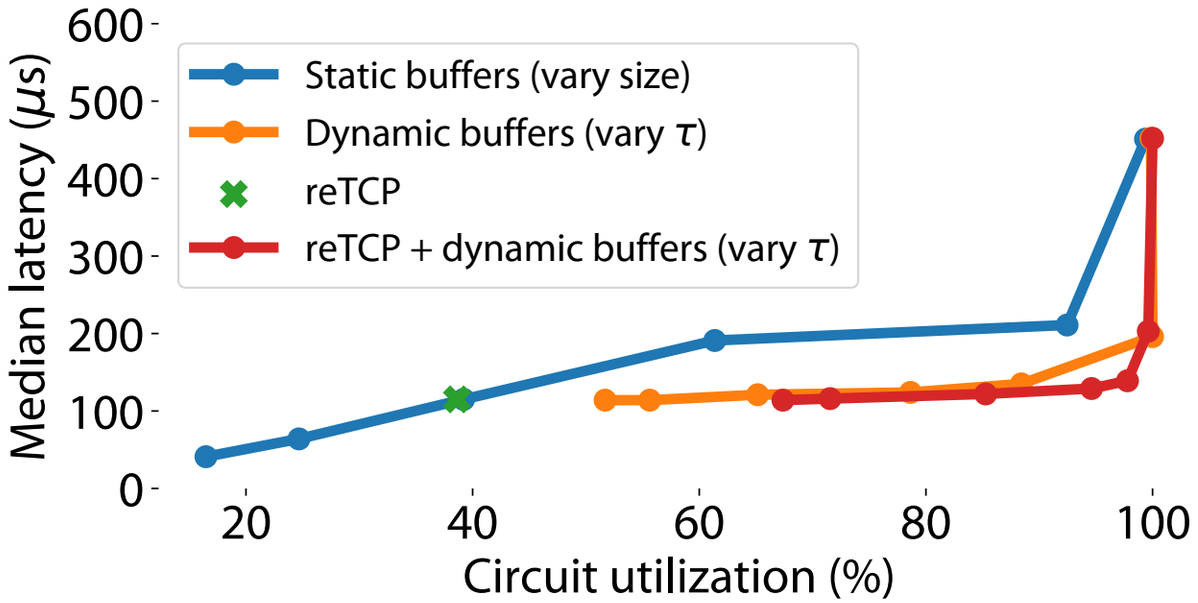
Use PFC instead of dynamic buffer resize?



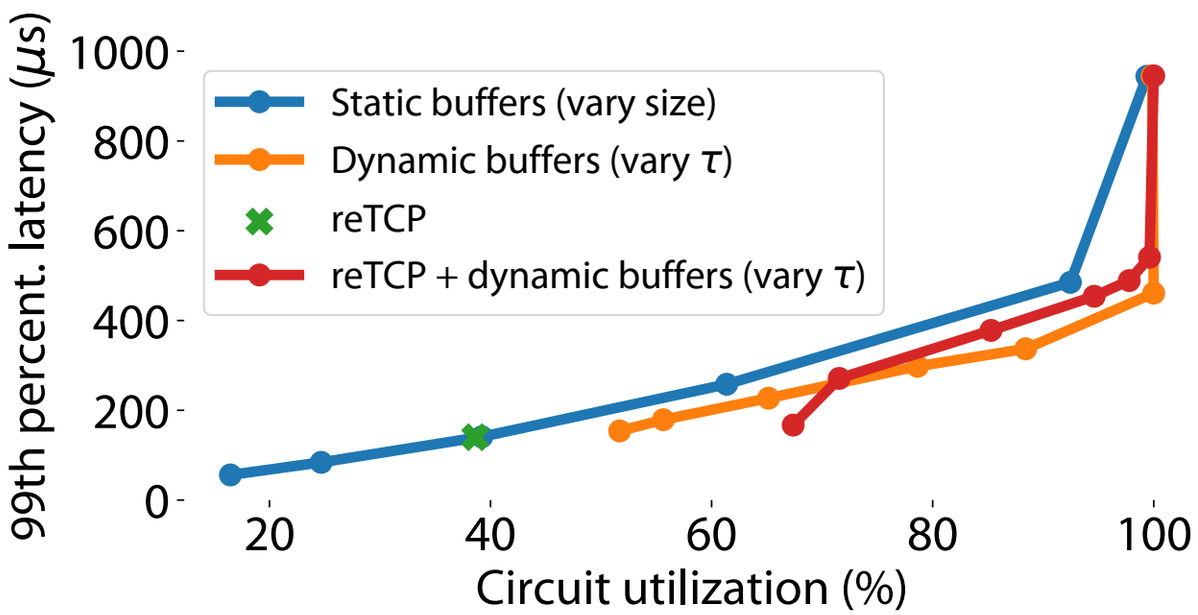
Use PFC instead of dynamic buffer resize?



Buffering and median latency



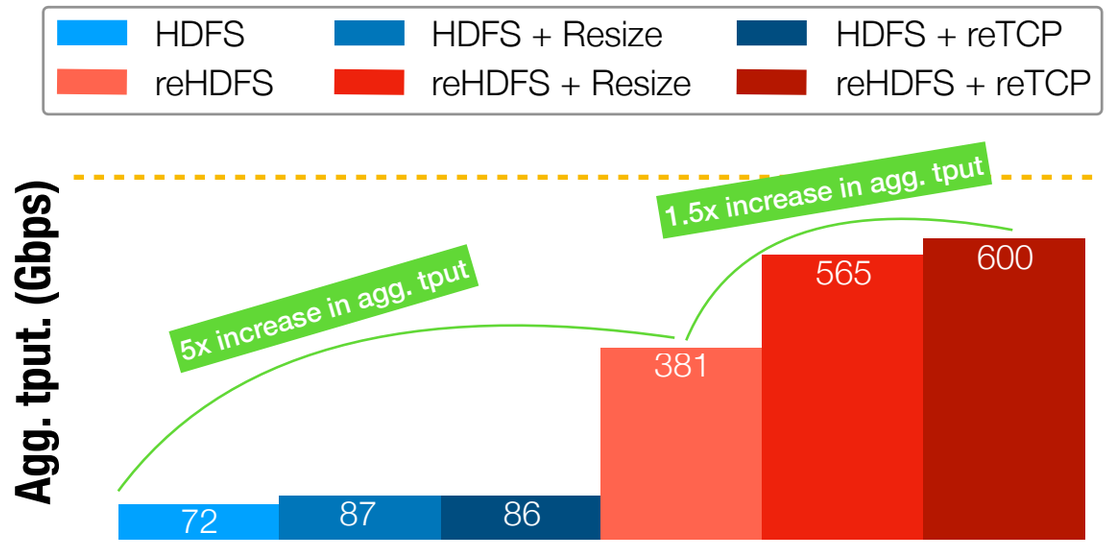
Buffering and tail latency



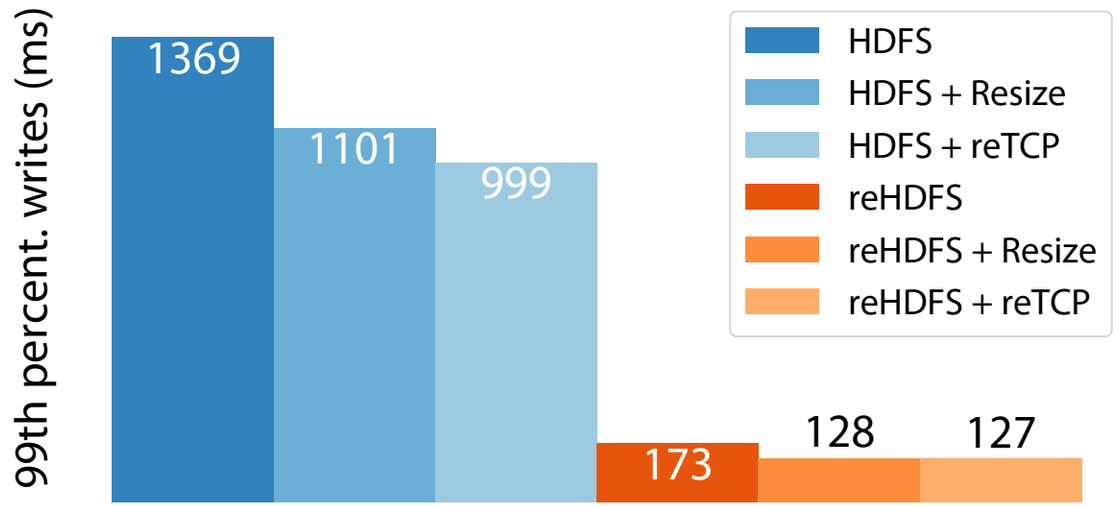
Challenge: Workloads

Solution: App-specific Modification

reHDFS cumulative results



reHDFS tail latency



The problem of poor demand estimation

.	B	s	s
s	.	B	s
s	s	.	B
B	s	s	.

Actual Demand

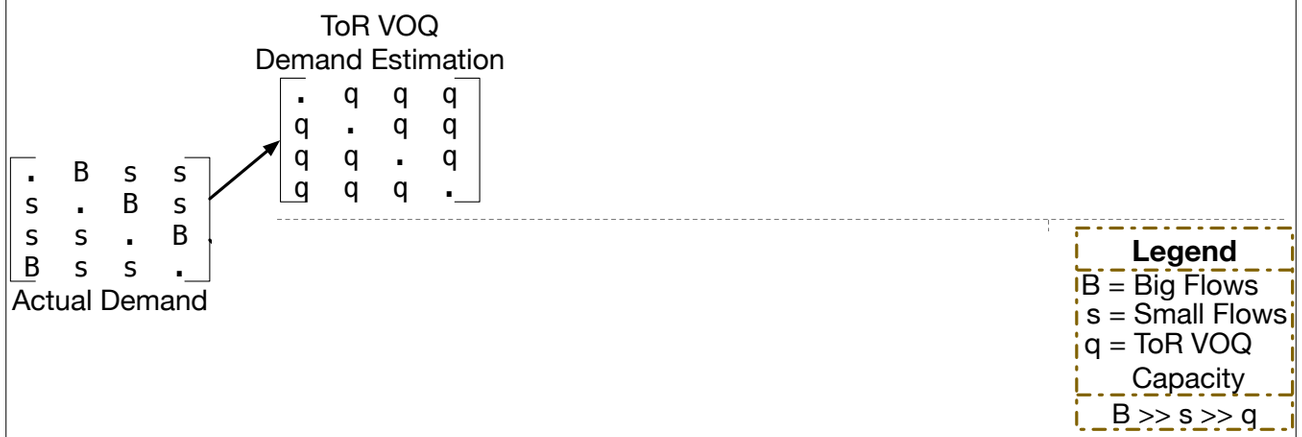
Legend

B = Big Flows
s = Small Flows
q = ToR VOQ Capacity

B >> s >> q

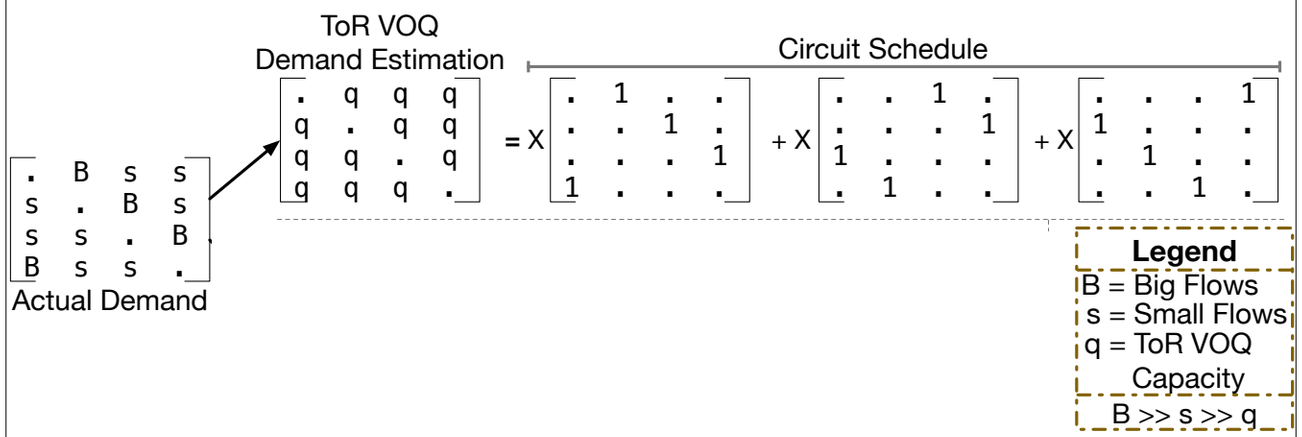
Why is poor demand estimation even an issue? Let’s say you have a demand matrix like this one, some small demand “s” and some big demand “B”.

The problem of poor demand estimation



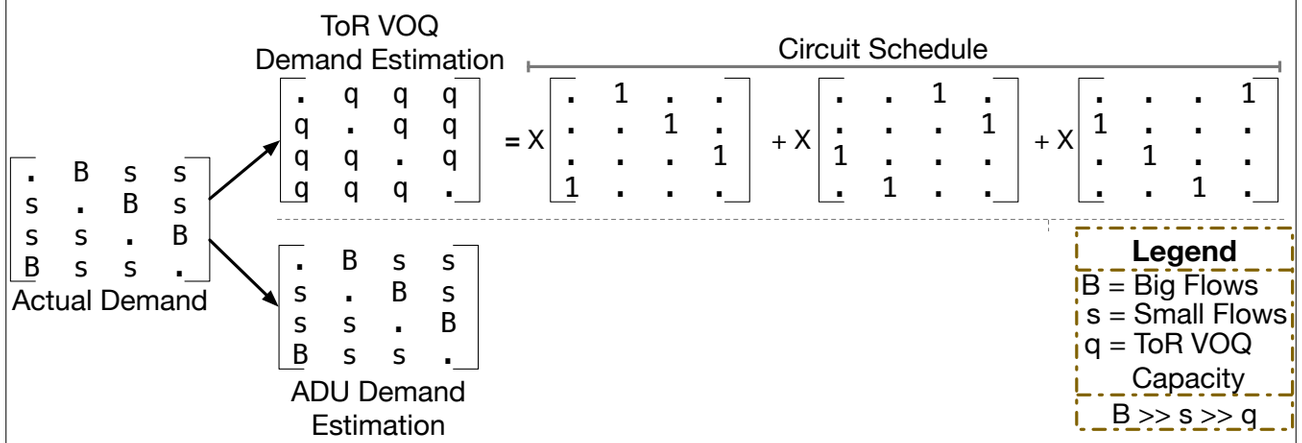
Let’s say that you have small network buffers to avoid inflating your latency, like we saw previously. Specifically let’s say that the buffer is size, “q”, is smaller than the small “s” demand in our matrix. This is reasonable as most RDCN schedulers will schedule for fairly long blocks of time (milliseconds) to amortize their runtime. Given “B” is much bigger than “s” is much bigger than “q”, if we estimate our demand matrix solely from what we seen our ToR queues, it will look like all “q”s…

The problem of poor demand estimation



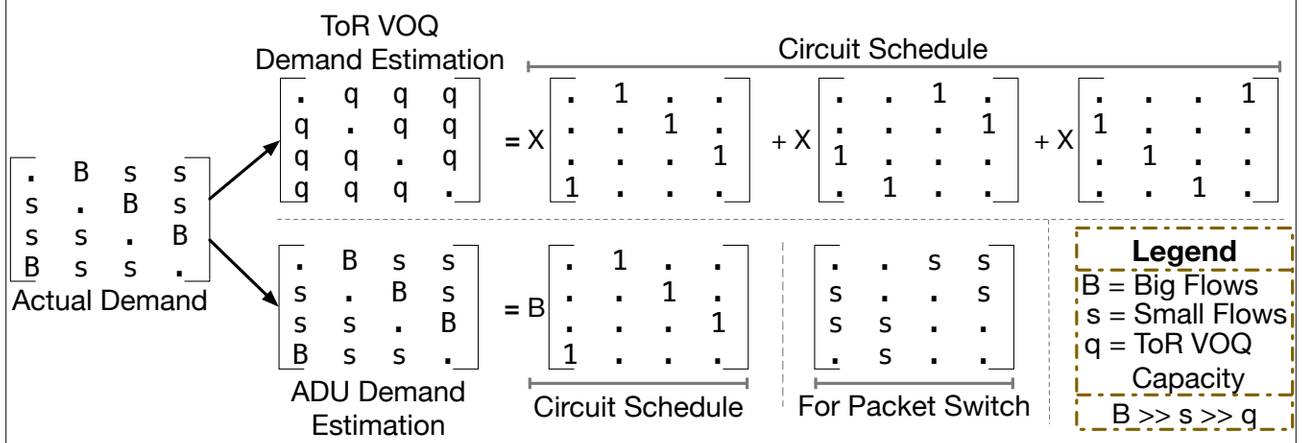
Giving us a schedule with many configurations, requiring downtime to reconfigure the circuits, as well as time wasted with idle circuits as it can’t differentiate between big and small demand. The problem is that demand is sitting on the endhost and we have no way to know that it’s there until it is too late…

The problem of poor demand estimation



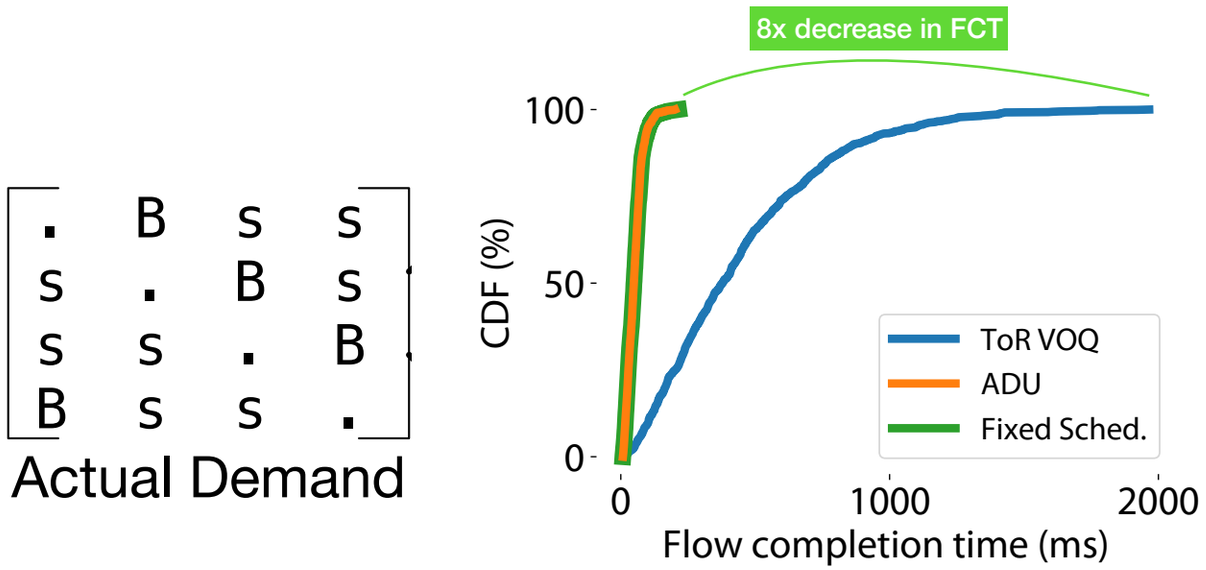
We propose “application data unit (or ADU)-based Demand Estimation” to solve this problem. We run an interposition library on endhosts that catch “write” and “send” calls to tell the scheduler exactly how much demand is sitting on the endhosts…

The problem of poor demand estimation



This results in a much more efficient schedule with just one circuit configuration as the scheduler can now differentiate between big and small demand, sending the small demand for the packet switch.

ADU demand estimation results



We briefly touch on the results for ADU-based demand estimation. We build a demand matrix like the one seen previously. A single “ring” of big demand is created with small demand everywhere else (unnecessary on the diagonal). Our matrix, however is 8 x 8. Using our open-source emulator, we find the results to be as expected.

In the graph on the right, we show a CDF of flow-completion times for the big flows. As big flows must send more data to complete, they are more susceptible to being slowed down by inefficient circuit schedules than small flows. We compare the flow-completion time using ToR queue-based estimation, versus our ADU-based scheme, versus a hand-made “fixed” schedule built using a perfect knowledge of the demand matrix. We find that ** ADU-based estimation works significantly better than ToR queue-based estimation (8x in the tail), and does virtually the same as the hand-made “fixed” schedule for this workload…