Programmable, hardware-based

Routing and Scheduling

Speed × Flexibility

Laurent Vanbever

nsg.ee.ethz.ch

EuroP4

Tue Dec 1 2020

Routing

Computes and maintains the set of paths alongside which network traffic flows

Routing

Computes and maintains the set of paths alongside which network traffic flows

Scheduling

Controls how traffic flows alongside these paths

Routing

Computes and maintains the set of paths alongside which network traffic flows

Flexible, but slow

Scheduling

Controls how traffic flows alongside these paths

Fast, but fixed

Routing

Computes and maintains the set of paths alongside which network traffic flows

Flexible, but slow

Scheduling

Controls how traffic flows alongside these paths

We can divide routing into

four main tasks

We can divide routing into four main tasks

```
while(true); do

detect      network changes

notify      remote devices of the changes

compute     new paths

update      the local forwarding state
```

Except for notifying,

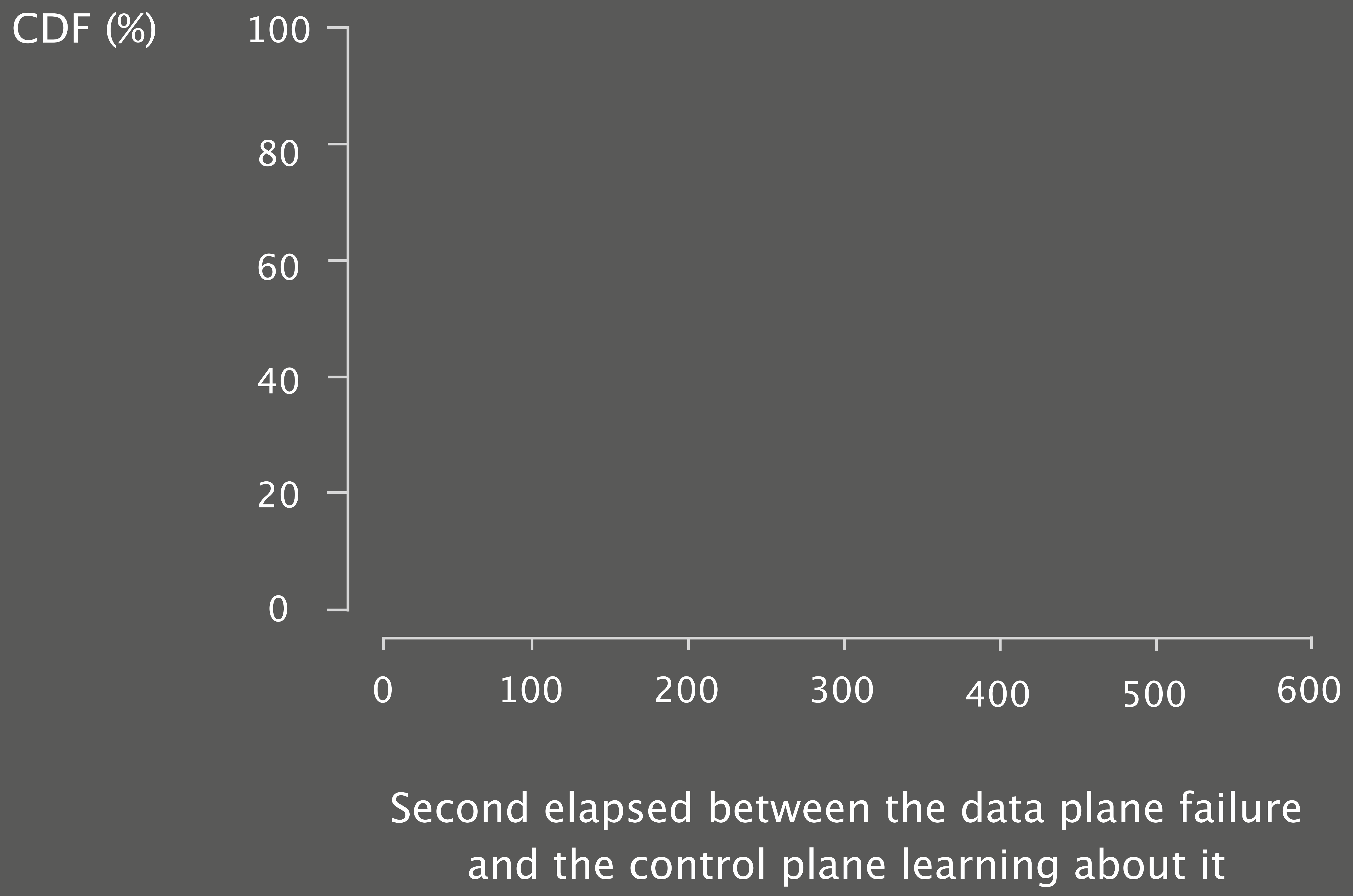
all tasks can take *minutes* to complete

In the worst-case,
it can take >1 min to…

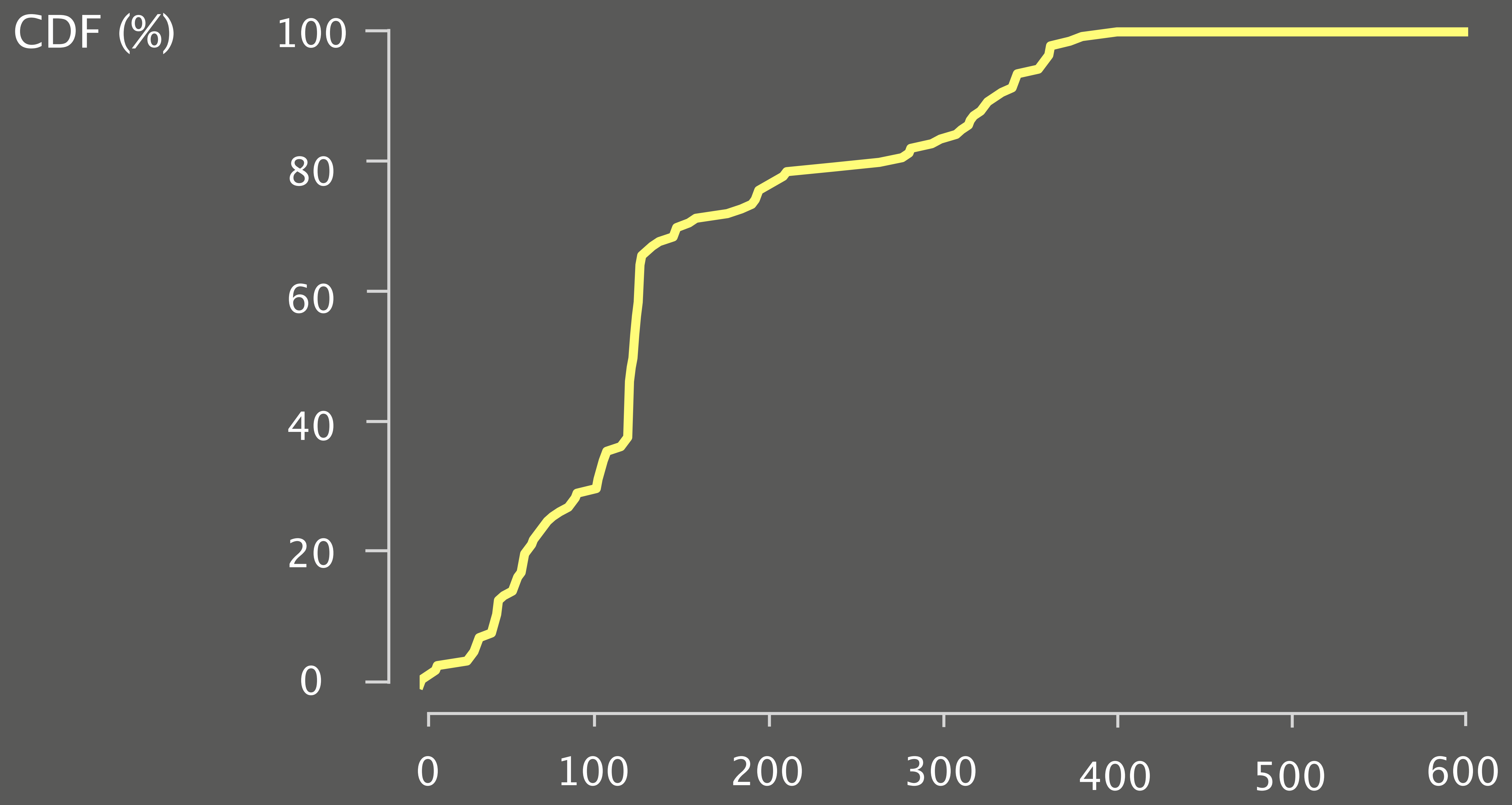
In the worst-case,
it can take >1 min to…

detect remote Internet failures

Internet control plane can take minutes to *learn* about remote failures



Internet control plane can take minutes to *learn* about remote failures



Second elapsed between the data plane failure and the control plane learning about it

In the worst-case,
it can take >1 min to...

detect remote Internet failures

compute the routing state for 100,000s destinations

In the worst-case,
it can take >1 min to…

detect remote Internet failures

compute the routing state for 100,000s destinations

update 100,000s of forwarding entries

Routing

Computes and maintains the set of paths alongside which network traffic flows

Flexible, but slow

Scheduling

Controls how traffic flows alongside these paths

Routing

Computes and maintains the set of paths alongside which network traffic flows

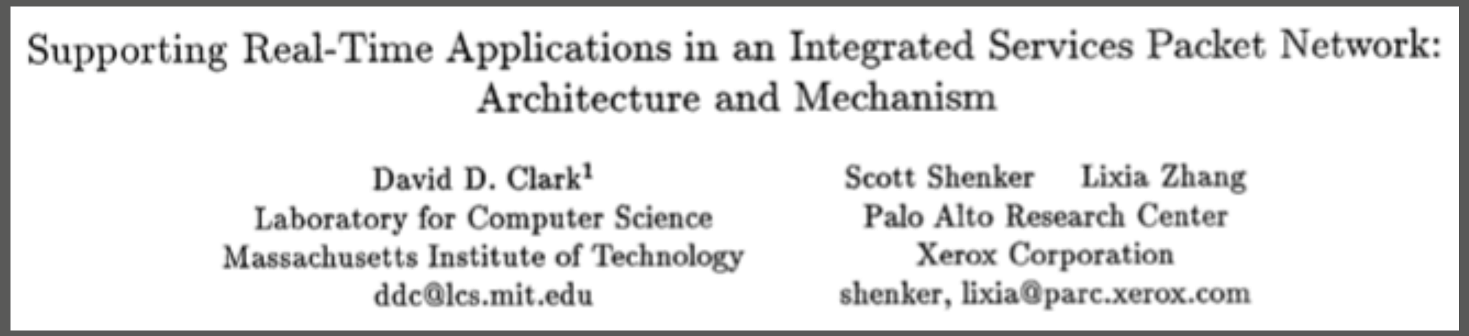
Scheduling

Controls how traffic flows alongside these paths

Fast, but fixed

There exists a wide variety of scheduling policies, optimizing for a wide variety of objectives

Minimize tail latency



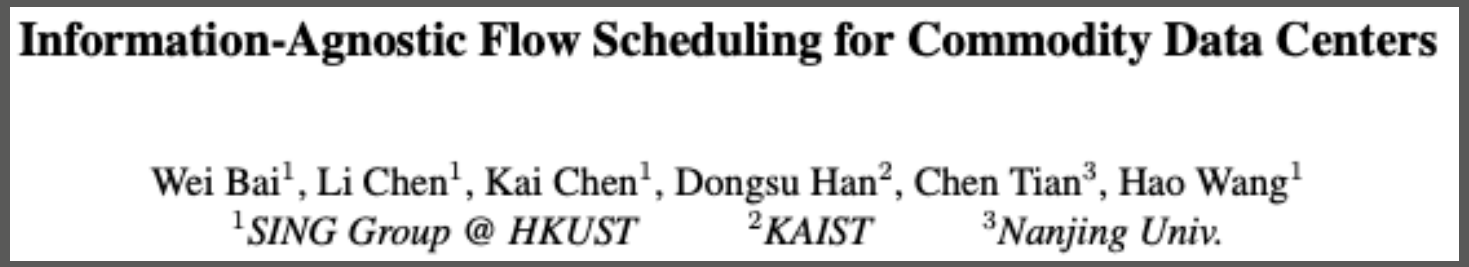
Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism

David D. Clark[1]
Laboratory for Computer Science
Massachusetts Institute of Technology
ddc@lcs.mit.edu

Scott Shenker Lixia Zhang
Palo Alto Research Center
Xerox Corporation
shenker, lixia@parc.xerox.com

SIGCOMM'92

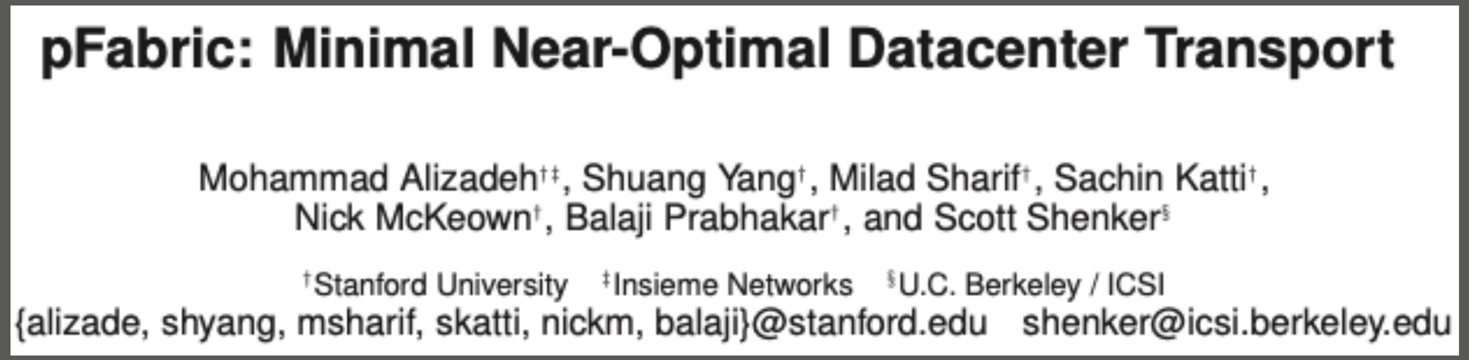
Minimize flow completion times



Information-Agnostic Flow Scheduling for Commodity Data Centers

Wei Bai[1], Li Chen[1], Kai Chen[1], Dongsu Han[2], Chen Tian[3], Hao Wang[1]
[1]*SING Group @ HKUST* [2]*KAIST* [3]*Nanjing Univ.*

NSDI'15



pFabric: Minimal Near-Optimal Datacenter Transport

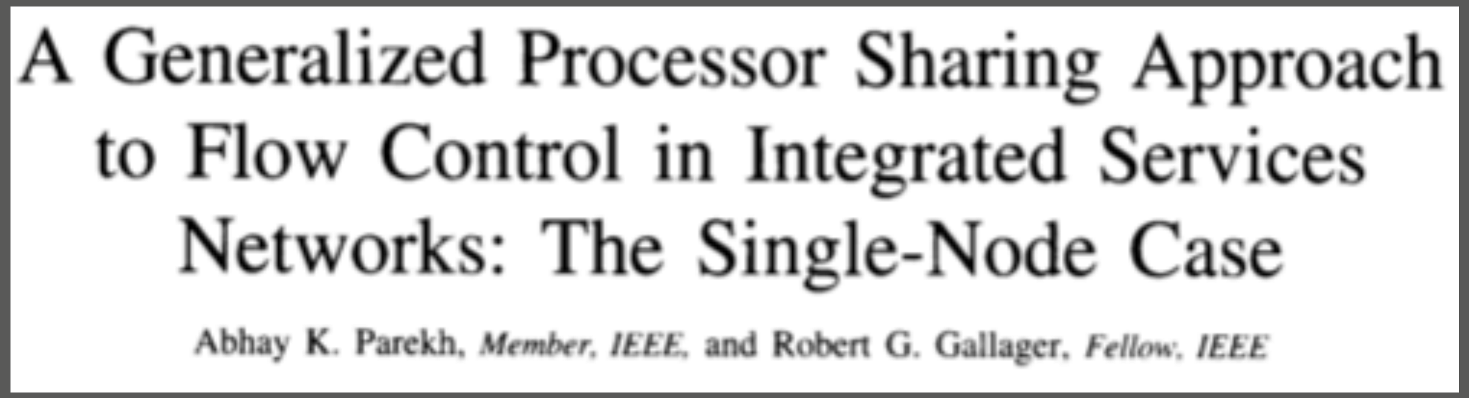
Mohammad Alizadeh[†‡], Shuang Yang[†], Milad Sharif[†], Sachin Katti[†],
Nick McKeown[†], Balaji Prabhakar[†], and Scott Shenker[§]

[†]Stanford University [‡]Insieme Networks [§]U.C. Berkeley / ICSI
{alizade, shyang, msharif, skatti, nickm, balaji}@stanford.edu shenker@icsi.berkeley.edu

SIGCOMM'13

Enforce max-min fairness

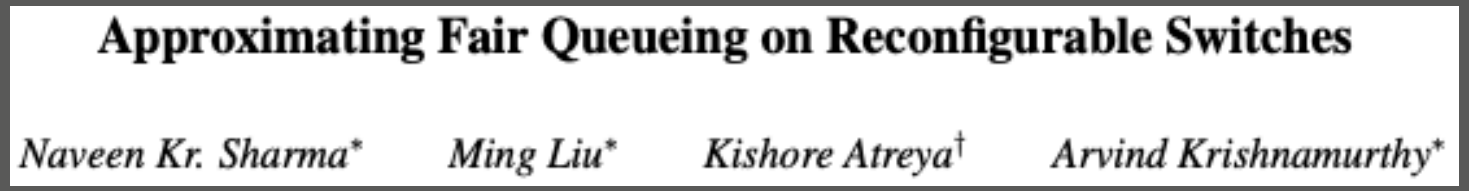
+ *many* more



A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case

Abhay K. Parekh, *Member, IEEE,* and Robert G. Gallager, *Fellow, IEEE*

ToN'93



Approximating Fair Queueing on Reconfigurable Switches

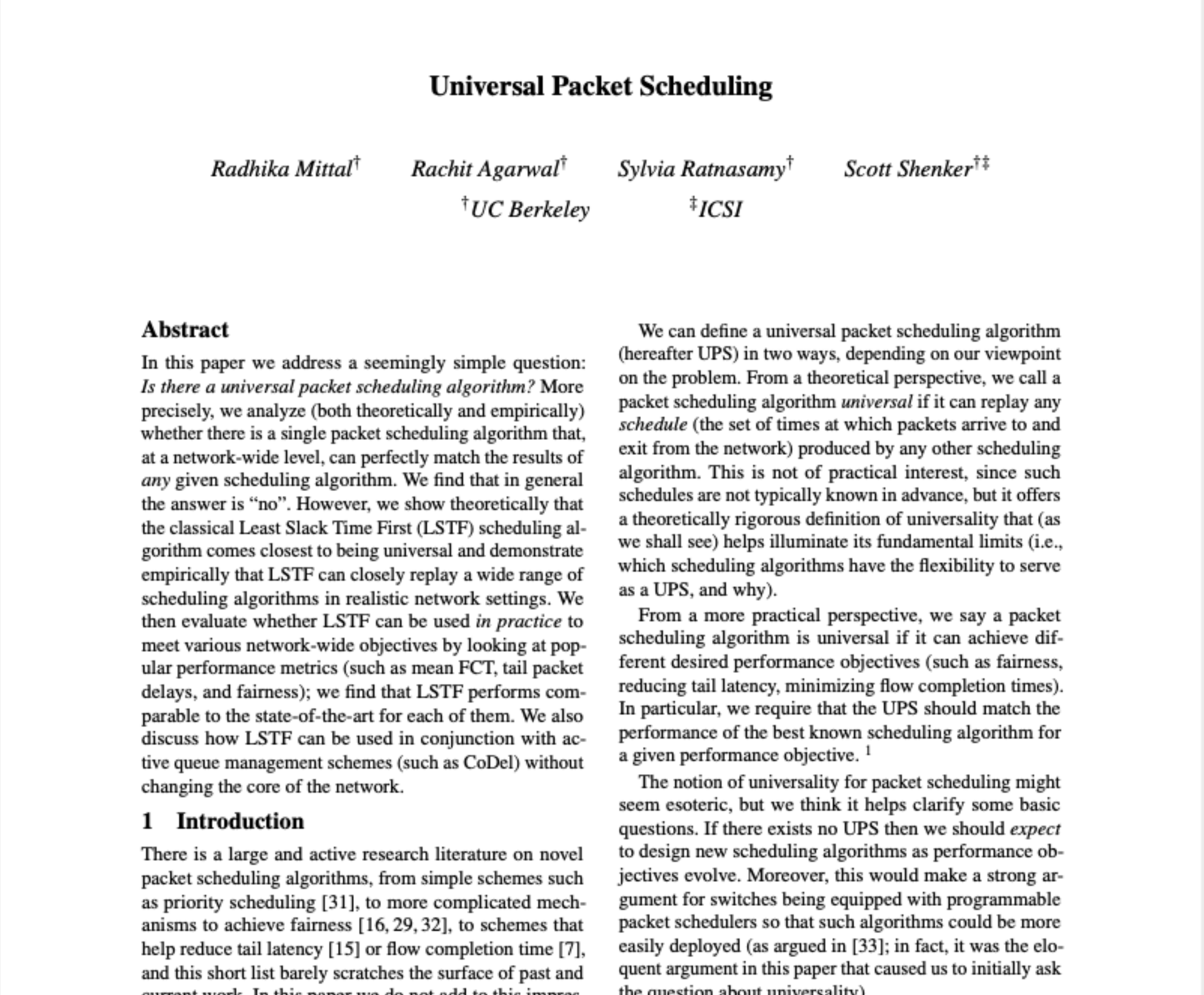
Naveen Kr. Sharma[*] *Ming Liu*[*] *Kishore Atreya*[†] *Arvind Krishnamurthy*[*]

NSDI'18

Quite unfortunately…

a universal scheduling algorithm does *not* exist

NSDI'16



Universal Packet Scheduling

Radhika Mittal[†] *Rachit Agarwal*[†] *Sylvia Ratnasamy*[†] *Scott Shenker*[†‡]

[†]*UC Berkeley* [‡]*ICSI*

Abstract

In this paper we address a seemingly simple question: *Is there a universal packet scheduling algorithm?* More precisely, we analyze (both theoretically and empirically) whether there is a single packet scheduling algorithm that, at a network-wide level, can perfectly match the results of *any* given scheduling algorithm. We find that in general the answer is "no". However, we show theoretically that the classical Least Slack Time First (LSTF) scheduling algorithm comes closest to being universal and demonstrate empirically that LSTF can closely replay a wide range of scheduling algorithms in realistic network settings. We then evaluate whether LSTF can be used *in practice* to meet various network-wide objectives by looking at popular performance metrics (such as mean FCT, tail packet delays, and fairness); we find that LSTF performs comparable to the state-of-the-art for each of them. We also discuss how LSTF can be used in conjunction with active queue management schemes (such as CoDel) without changing the core of the network.

1 Introduction

There is a large and active research literature on novel packet scheduling algorithms, from simple schemes such as priority scheduling [31], to more complicated mechanisms to achieve fairness [16, 29, 32], to schemes that help reduce tail latency [15] or flow completion time [7], and this short list barely scratches the surface of past and current work. In this paper we do not add to this impres-

We can define a universal packet scheduling algorithm (hereafter UPS) in two ways, depending on our viewpoint on the problem. From a theoretical perspective, we call a packet scheduling algorithm *universal* if it can replay any *schedule* (the set of times at which packets arrive to and exit from the network) produced by any other scheduling algorithm. This is not of practical interest, since such schedules are not typically known in advance, but it offers a theoretically rigorous definition of universality that (as we shall see) helps illuminate its fundamental limits (i.e., which scheduling algorithms have the flexibility to serve as a UPS, and why).

From a more practical perspective, we say a packet scheduling algorithm is universal if it can achieve different desired performance objectives (such as fairness, reducing tail latency, minimizing flow completion times). In particular, we require that the UPS should match the performance of the best known scheduling algorithm for a given performance objective. [1]

The notion of universality for packet scheduling might seem esoteric, but we think it helps clarify some basic questions. If there exists no UPS then we should *expect* to design new scheduling algorithms as performance objectives evolve. Moreover, this would make a strong argument for switches being equipped with programmable packet schedulers so that such algorithms could be more easily deployed (as argued in [33]; in fact, it was the eloquent argument in this paper that caused us to initially ask the question about universality).

The need for diversity × lack of generality

motivates the need for flexible scheduling

The need for diversity × lack of generality
motivates the need for flexible scheduling

You *can’t* have *everything* you want…

The need for diversity × lack of generality
motivates the need for flexible scheduling

You *can’t* have *everything* you want…

but you *can* have *anything* you want

Programmable, hardware-based

Routing and Scheduling

1 **Fast routing**

data-plane offloading

2 **Flexible scheduling**

dynamic packet adaptation

3 **Looking forward**

routing | forwarding

Programmable, hardware-based

Routing and Scheduling

1 Fast routing

data-plane offloading

Flexible scheduling

dynamic packet adaptation

Looking forward

routing | forwarding

What about we…

detect network changes

notify remote devices of the changes

compute new paths

update the local forwarding state

…in hardware?!

What about we…

detect network changes

notify remote devices of the changes

compute new paths

update the local forwarding state

…in hardware?!

While the control plane can take minutes to learn about a failure,

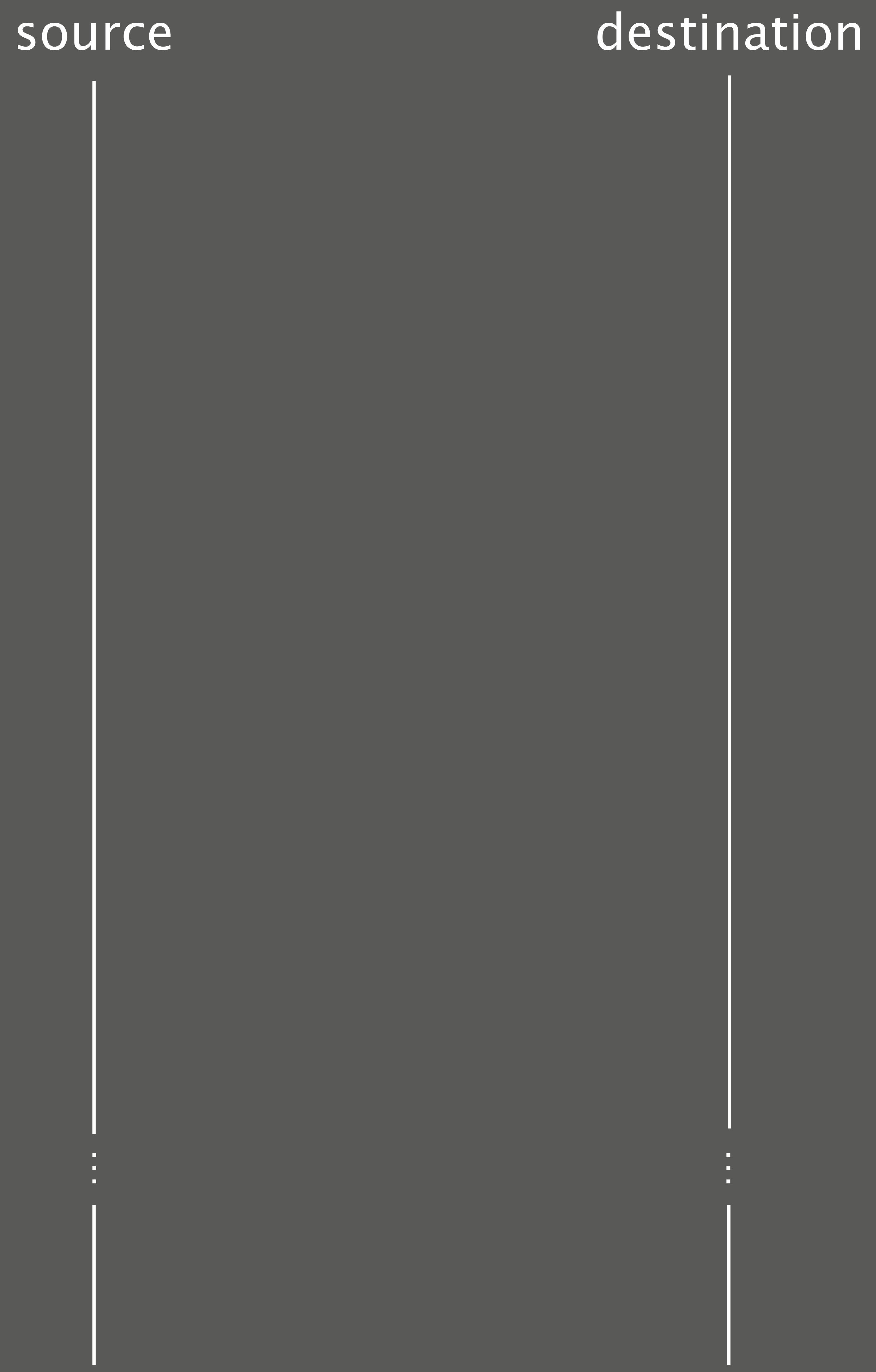
the traffic itself is *instantaneously* affected

While the control plane can take minutes to learn about a failure,

the traffic itself is *instantaneously* affected

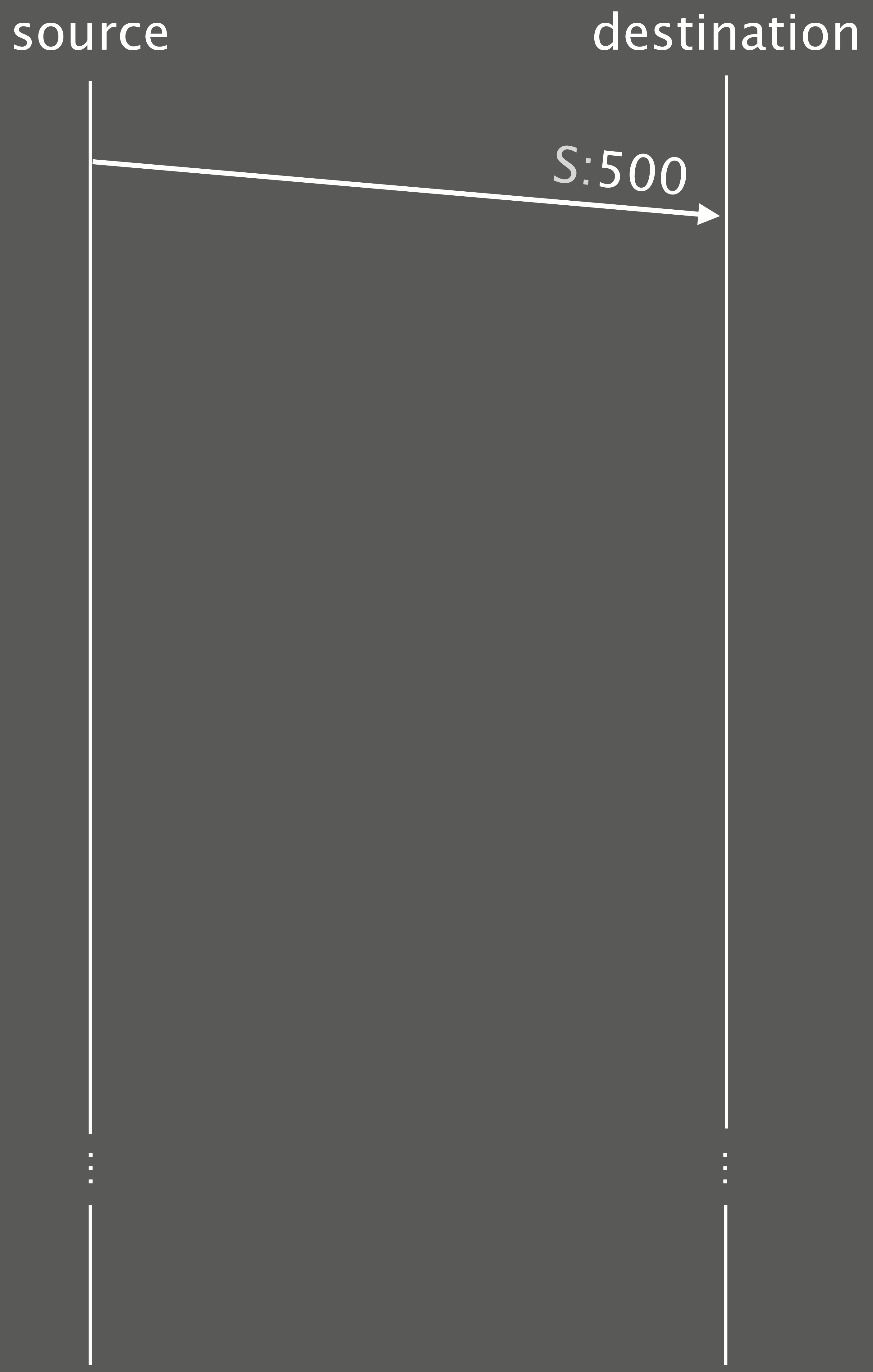
What about we track this signal instead?

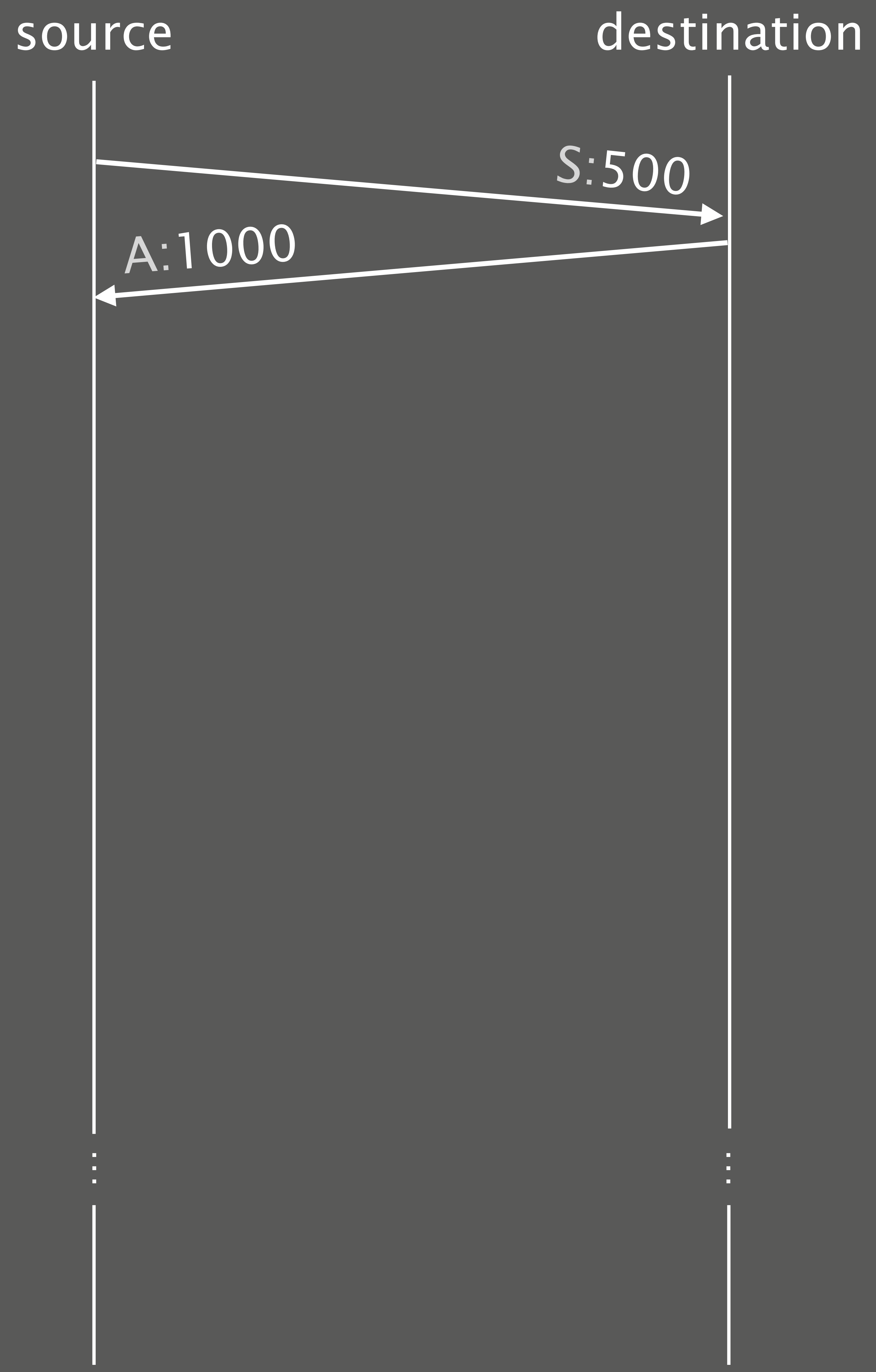
Internet traffic end-points retransmit packets upon experiencing packet drops

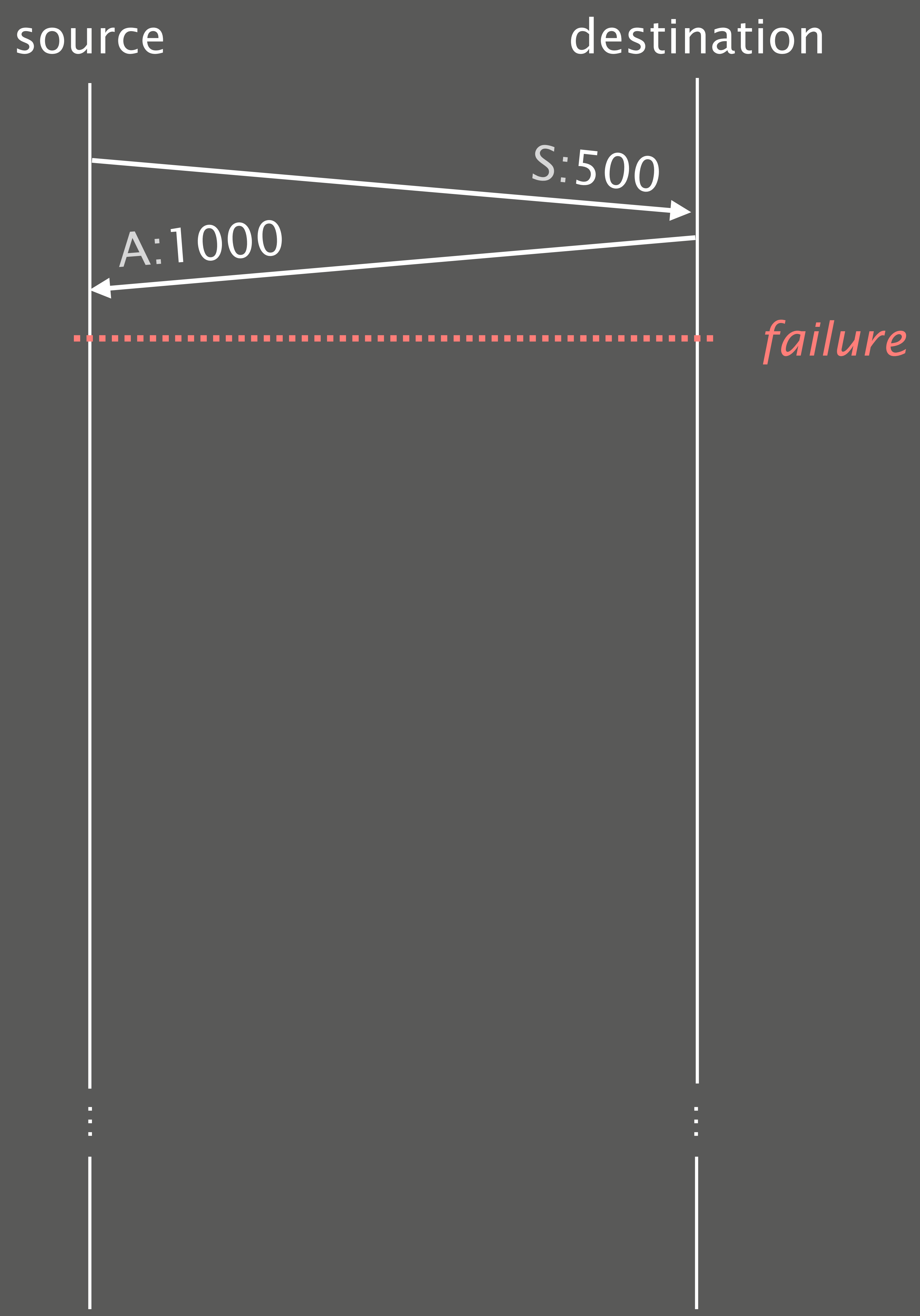


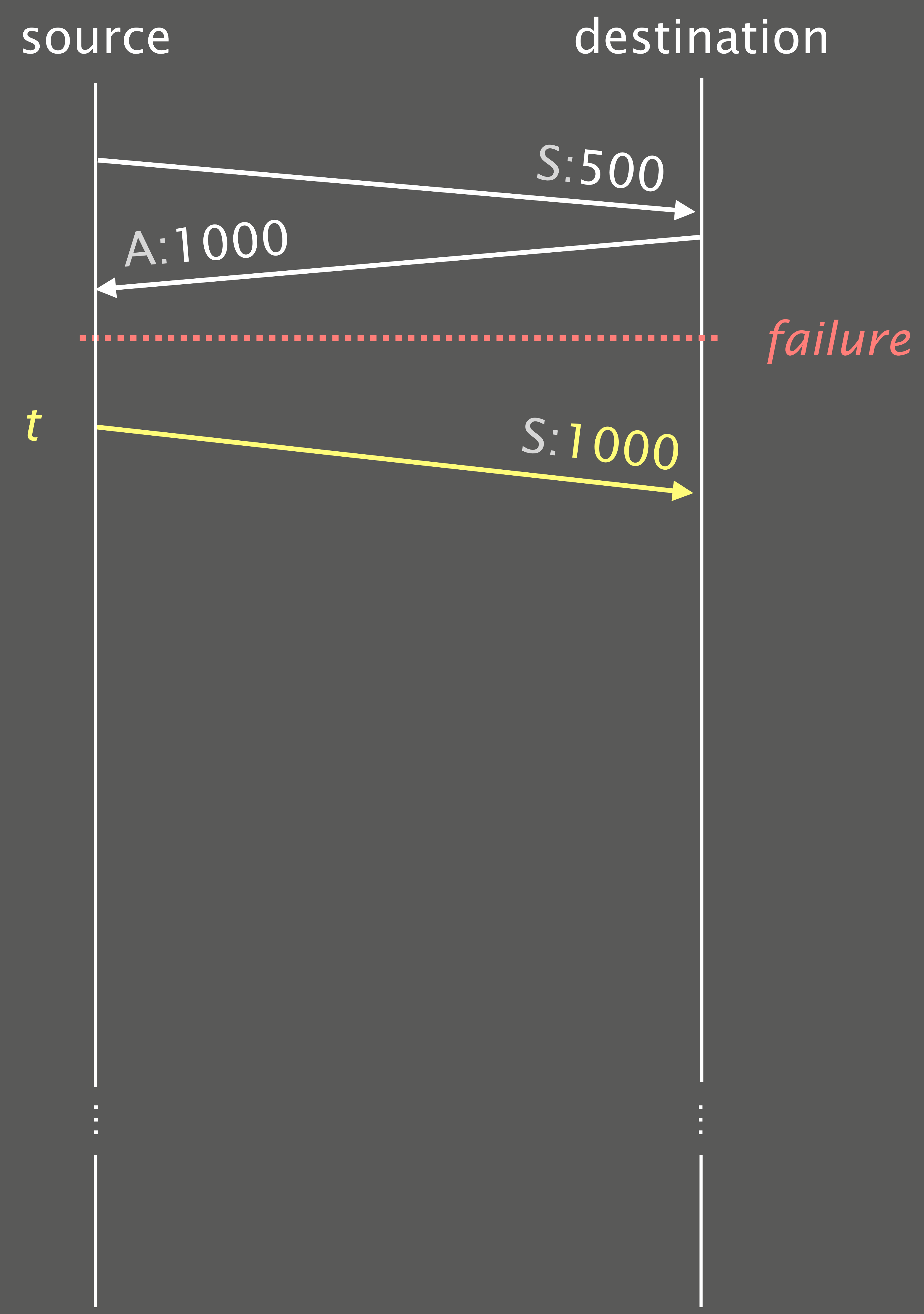
source

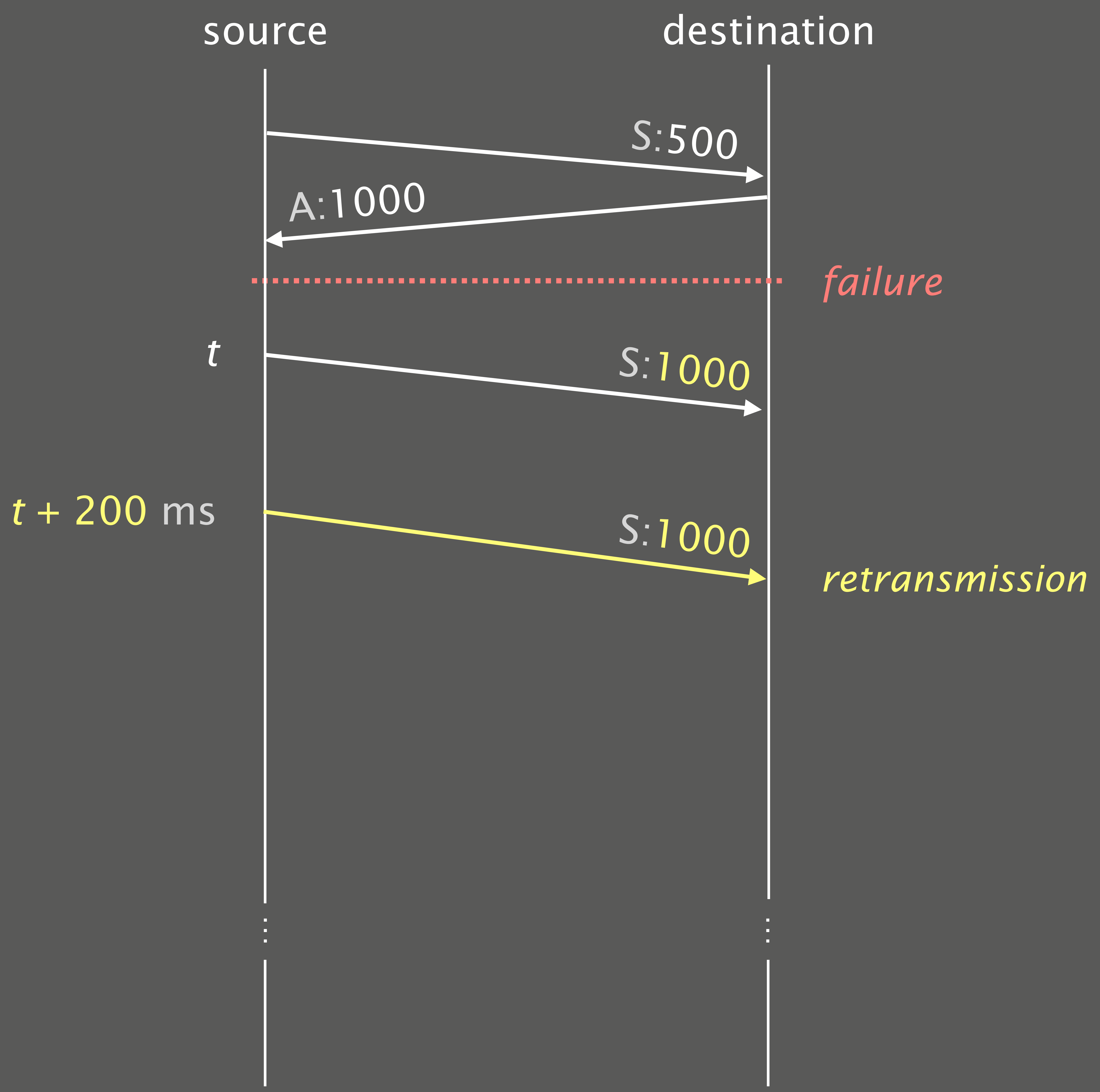
destination

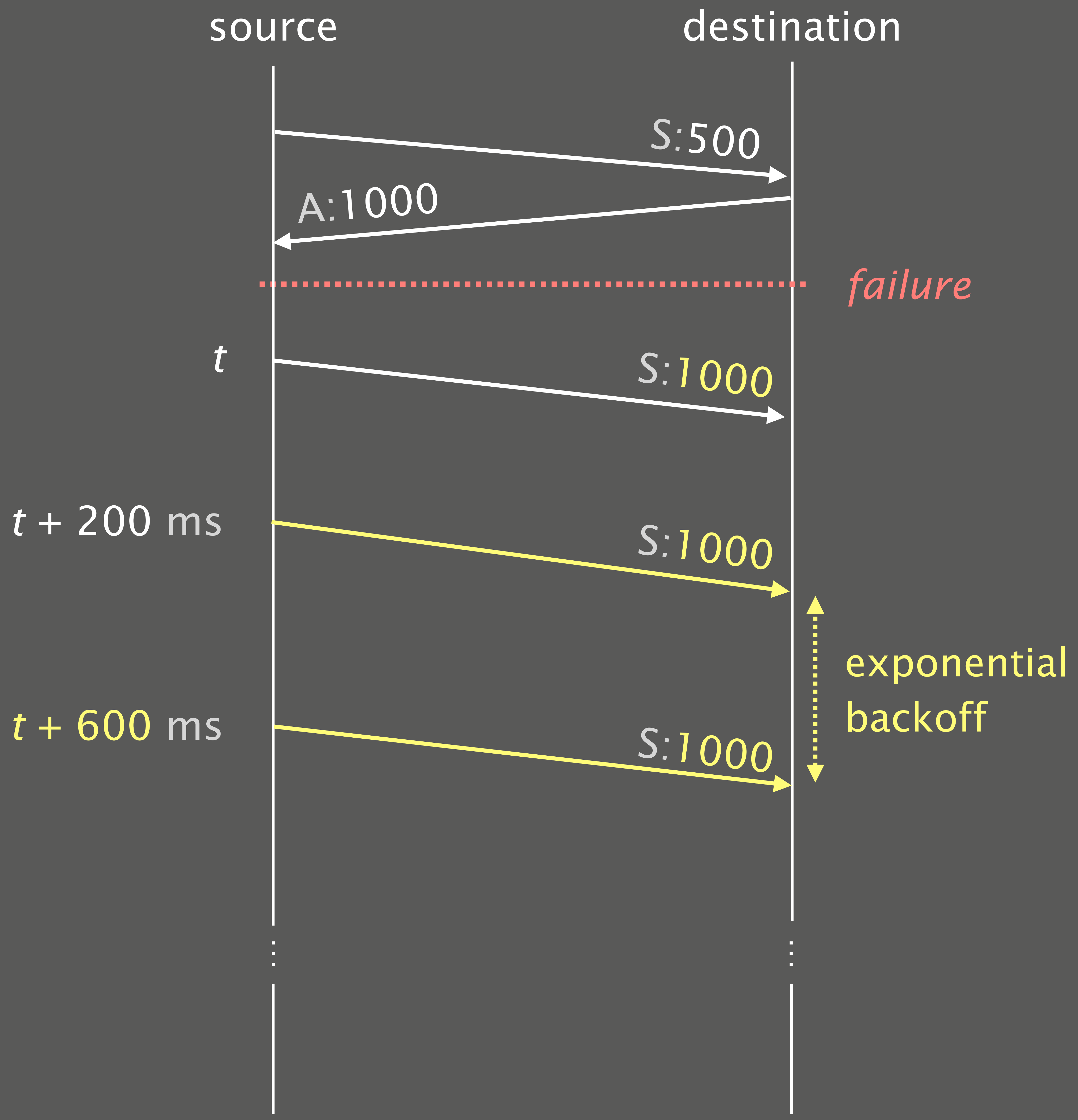

source
destination
S:500

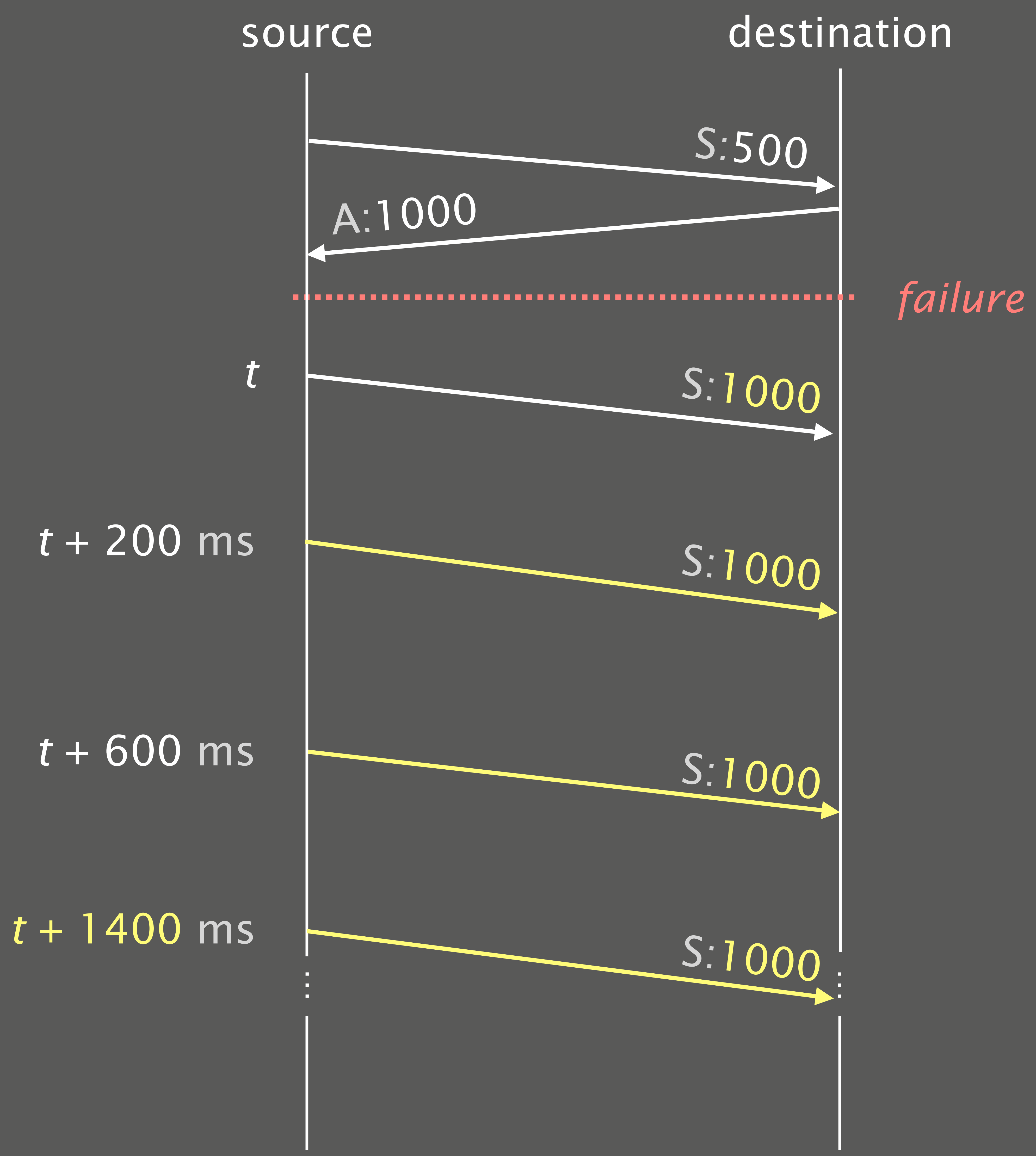

source
destination
S:500
A:1000


source
destination
S:500
A:1000
failure

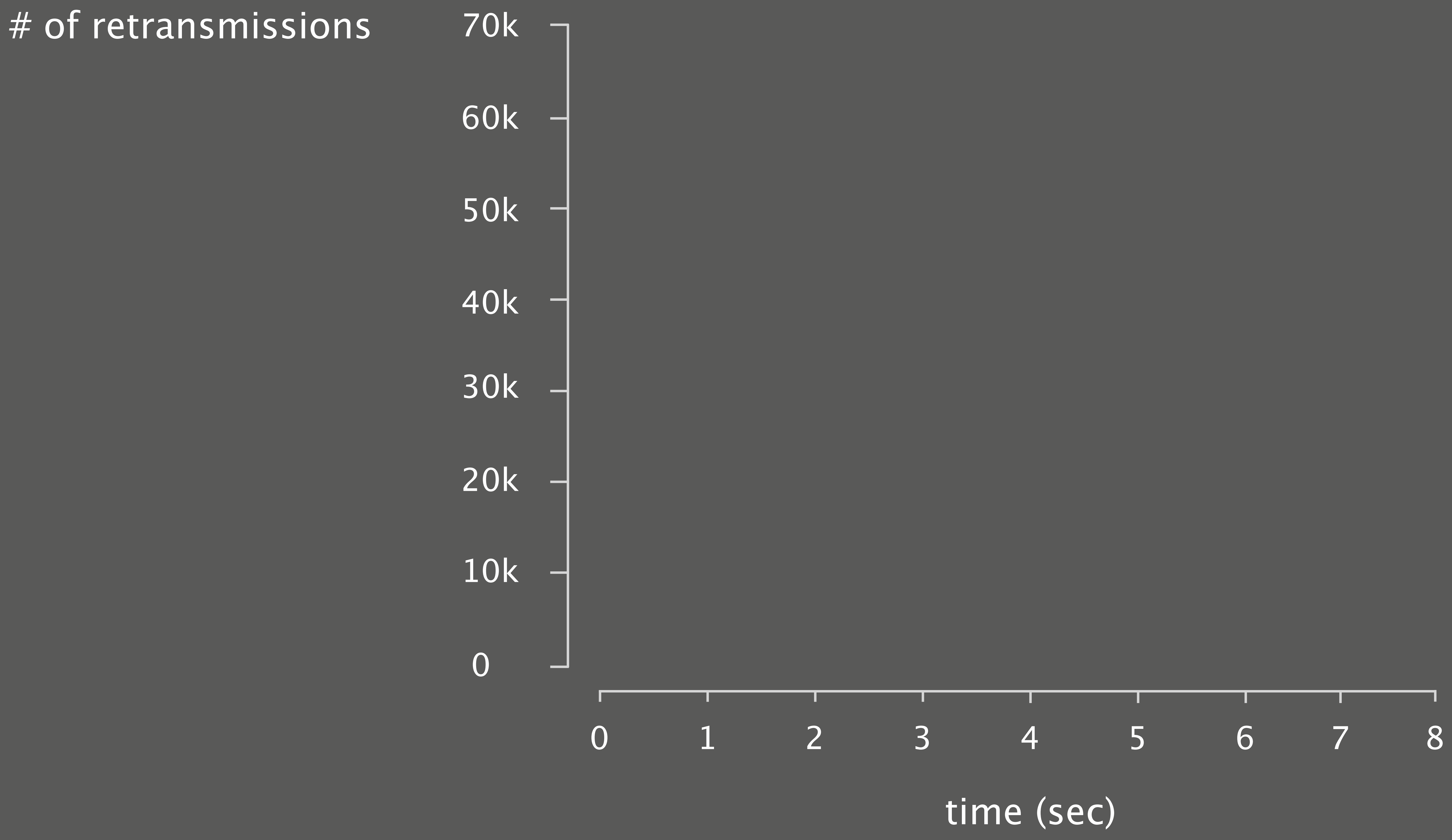

source
destination
S:500
A:1000
failure
t
S:1000

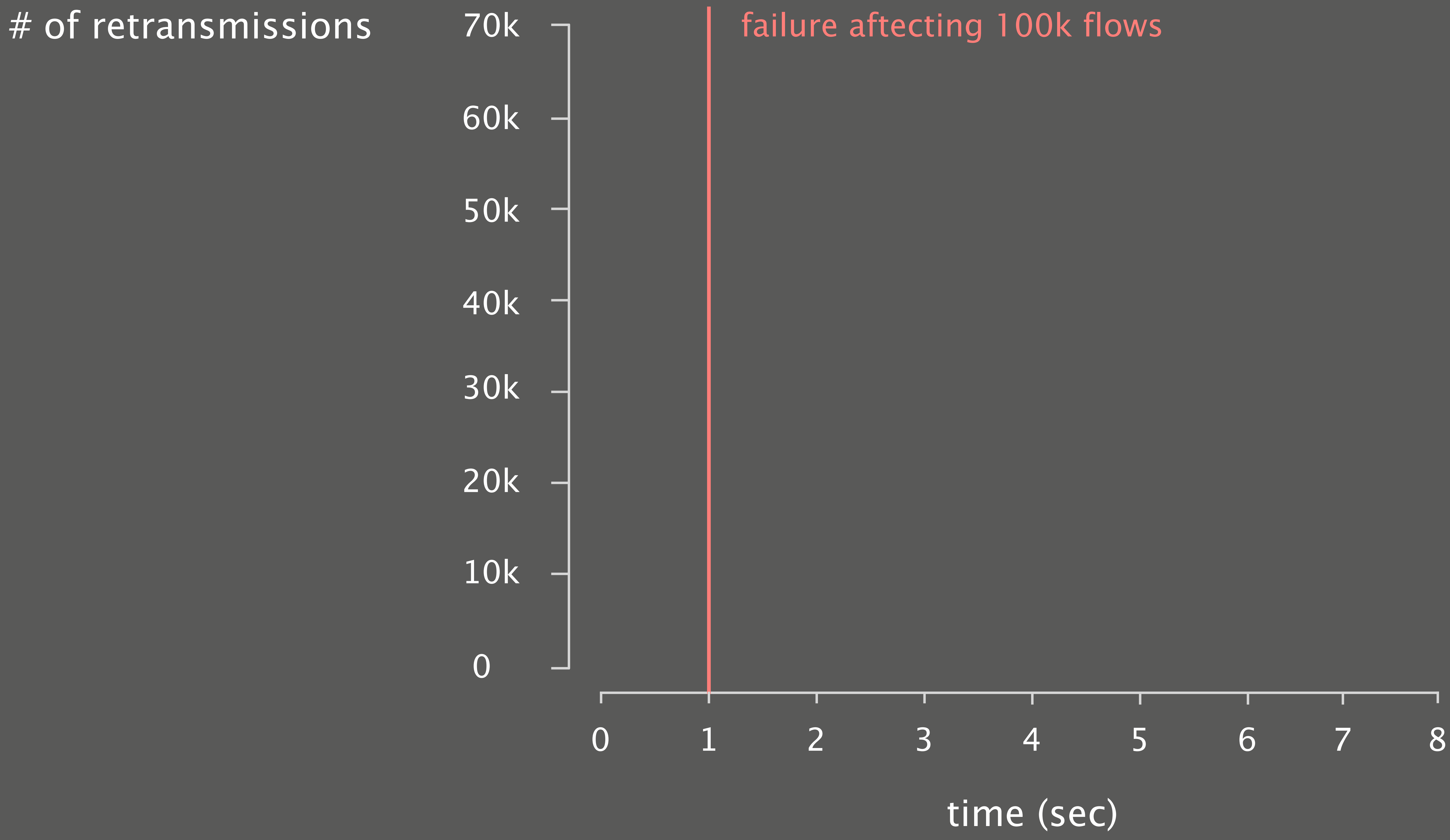

source
destination
S:500
A:1000
failure
t
S:1000
t + 200 ms
S:1000
retransmission

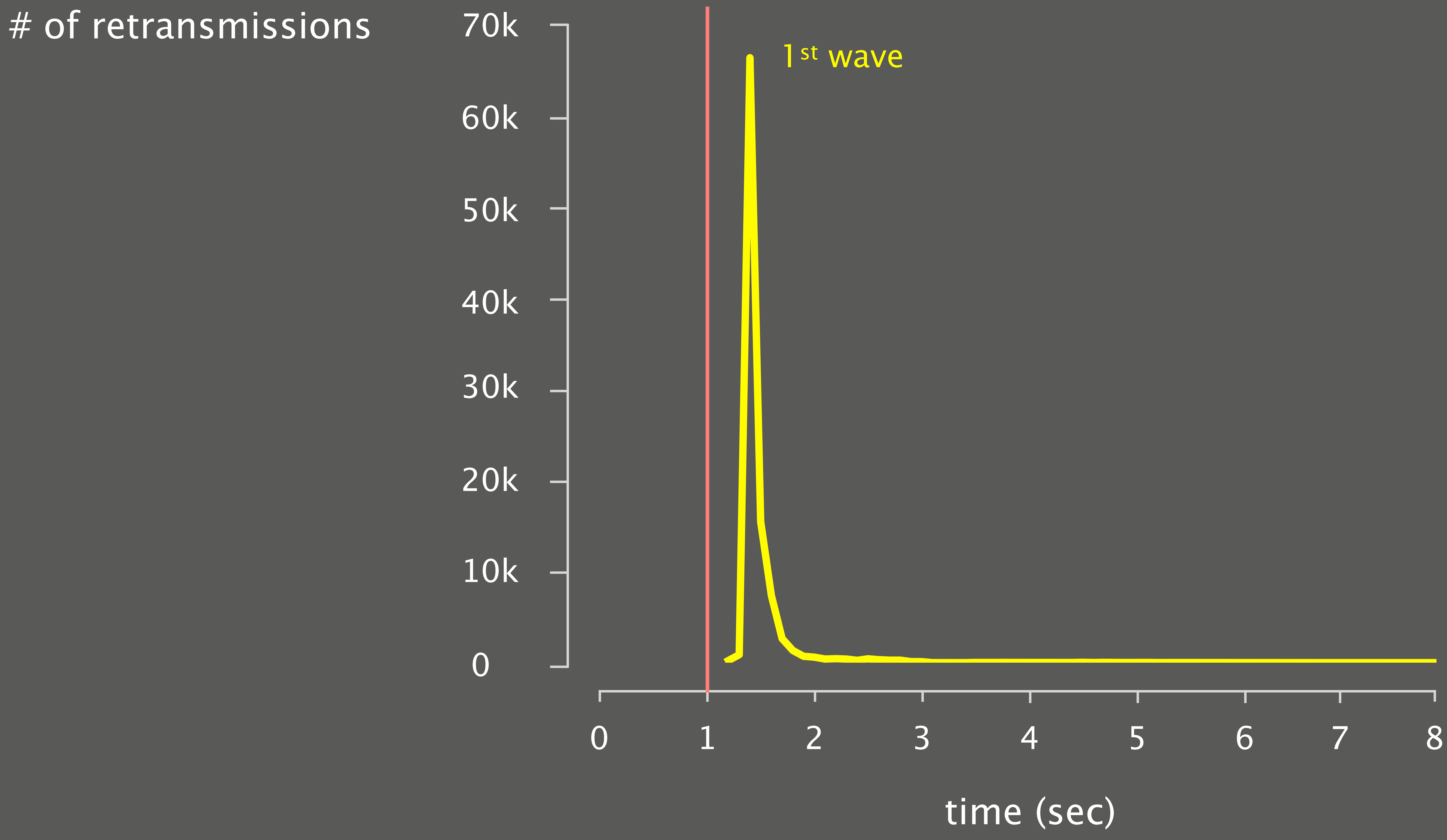

source
destination
S:500
A:1000
failure
t
S:1000
t + 200 ms
S:1000
exponential
backoff
t + 600 ms
S:1000


source
destination
S:500
A:1000
failure
t
S:1000
t + 200 ms
S:1000
t + 600 ms
S:1000
t + 1400 ms
S:1000

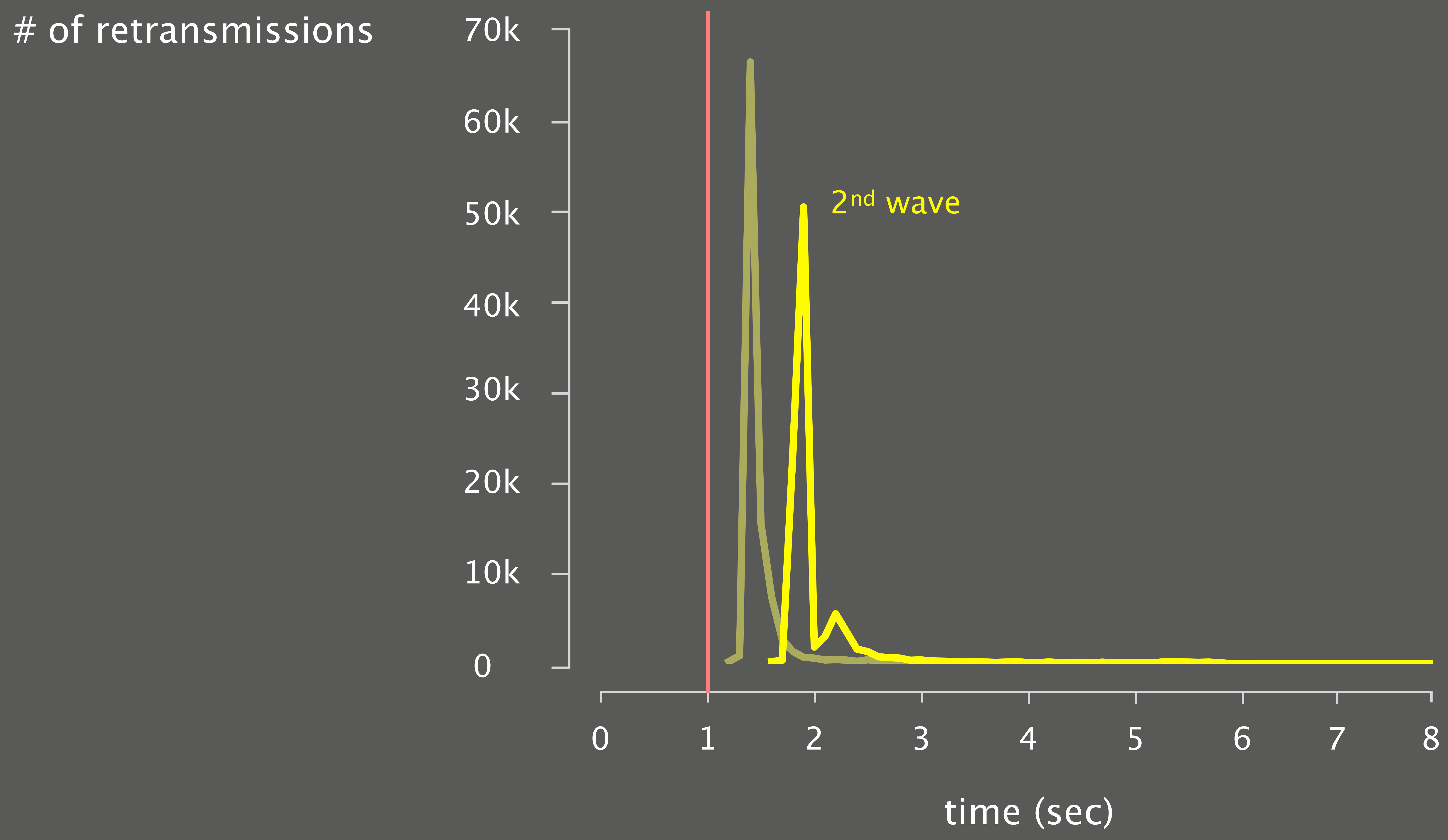
Many end points retransmitting simultaneously leads to waves of retransmission

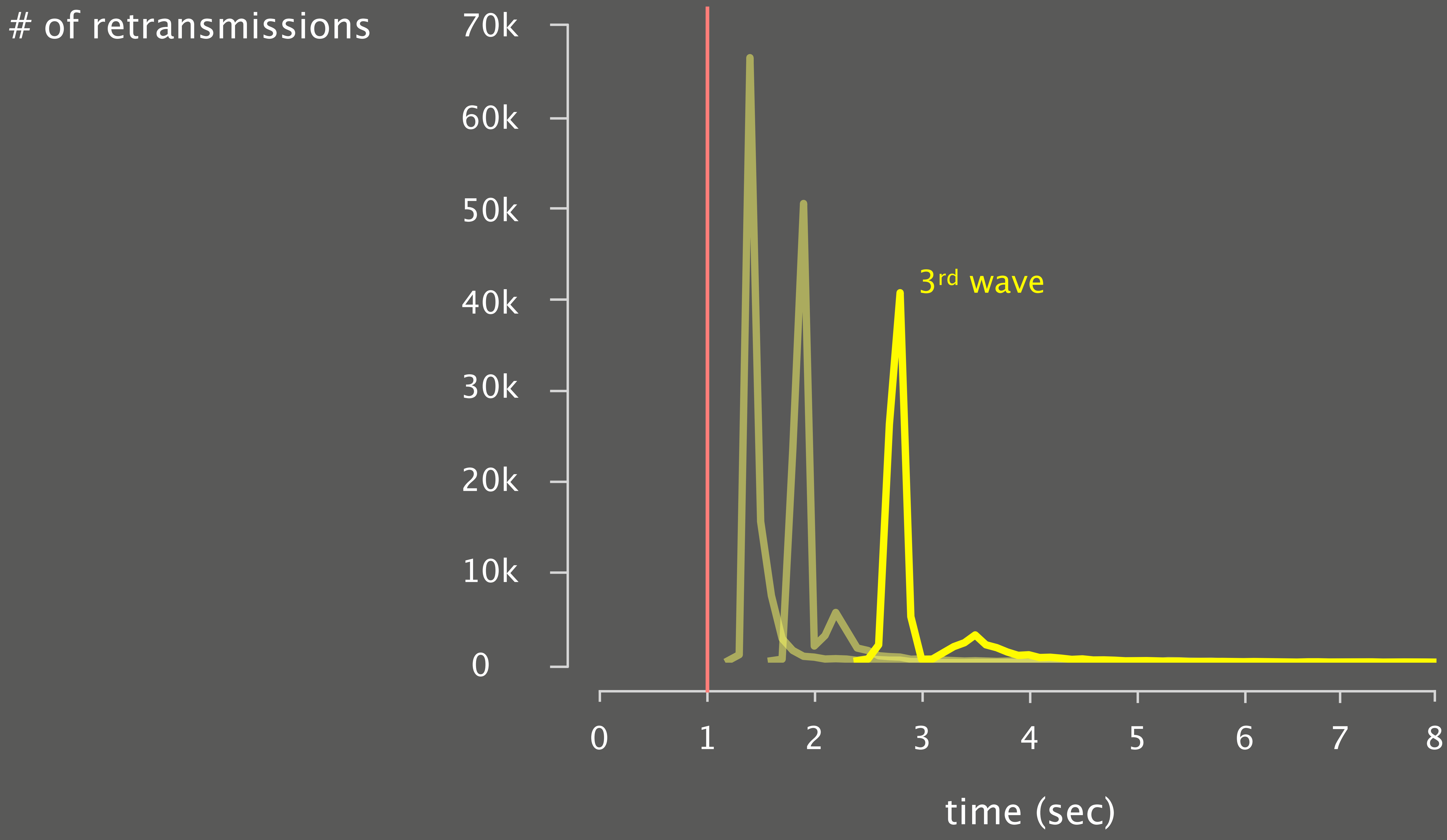

# of retransmissions
70k
60k
50k
40k
30k
20k
10k
0
0
1
2
3
4
5
6
7
8
time (sec)


# of retransmissions
70k
60k
50k
40k
30k
20k
10k
0
failure affecting 100k flows
0
1
2
3
4
5
6
7
8
time (sec)

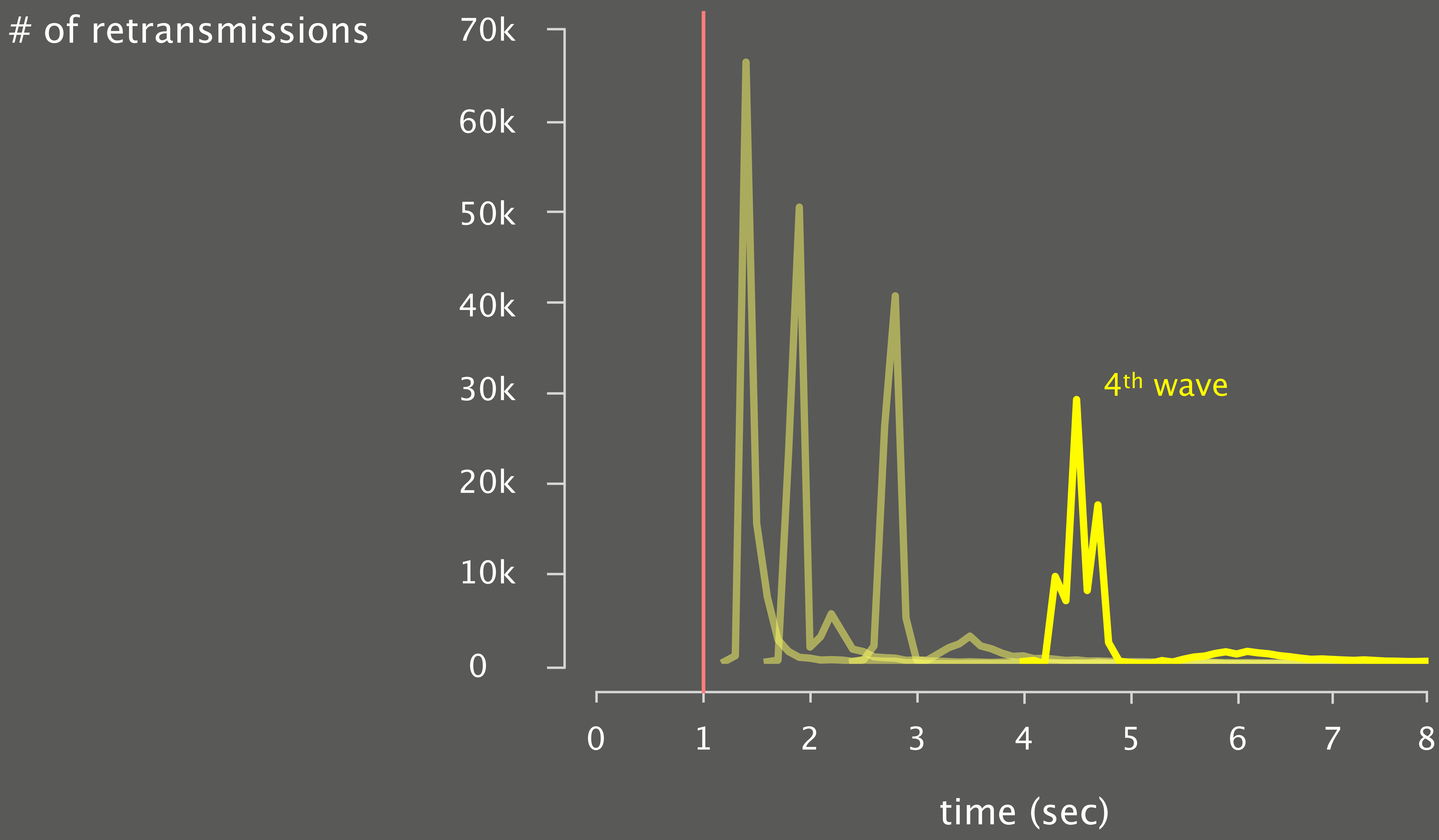


# of retransmissions
70k
60k
50k
40k
30k
20k
10k
0
1st wave
0
1
2
3
4
5
6
7
8
time (sec)


# of retransmissions
70k
60k
50k
40k
30k
20k
10k
0
2nd wave
0
1
2
3
4
5
6
7
8
time (sec)



# of retransmissions
70k
60k
50k
40k
30k
20k
10k
0
3rd wave
0
1
2
3
4
5
6
7
8
time (sec)



Tracking this signal in the data plane is challenging

Challenges

Signal is noisy

packets loss is routinely observed

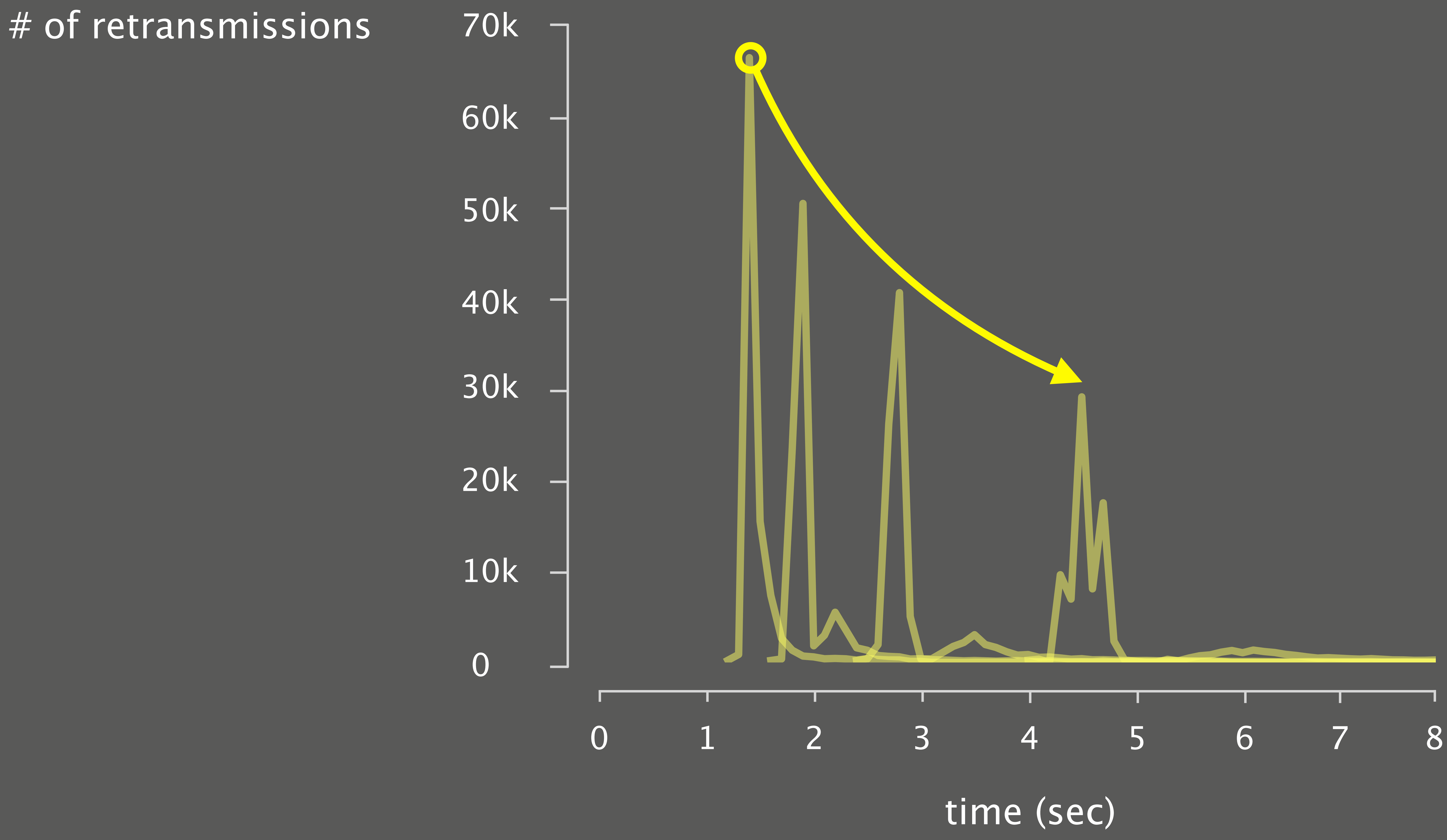
Signal fades away

due to the exponential backoff

Signal is composed of many small ones

requires per-connection tracking

The signal quickly fades away as subsequent waves spread over longer periods of time



To solve these challenges, we consider a subset of the signal for which we maximize the signal-to-noise ratio

To solve these challenges, we consider a subset of the signal for which we maximize the signal-to-noise ratio

Solutions

Signal is noisy

Focus on retransmissions due to bursty losses

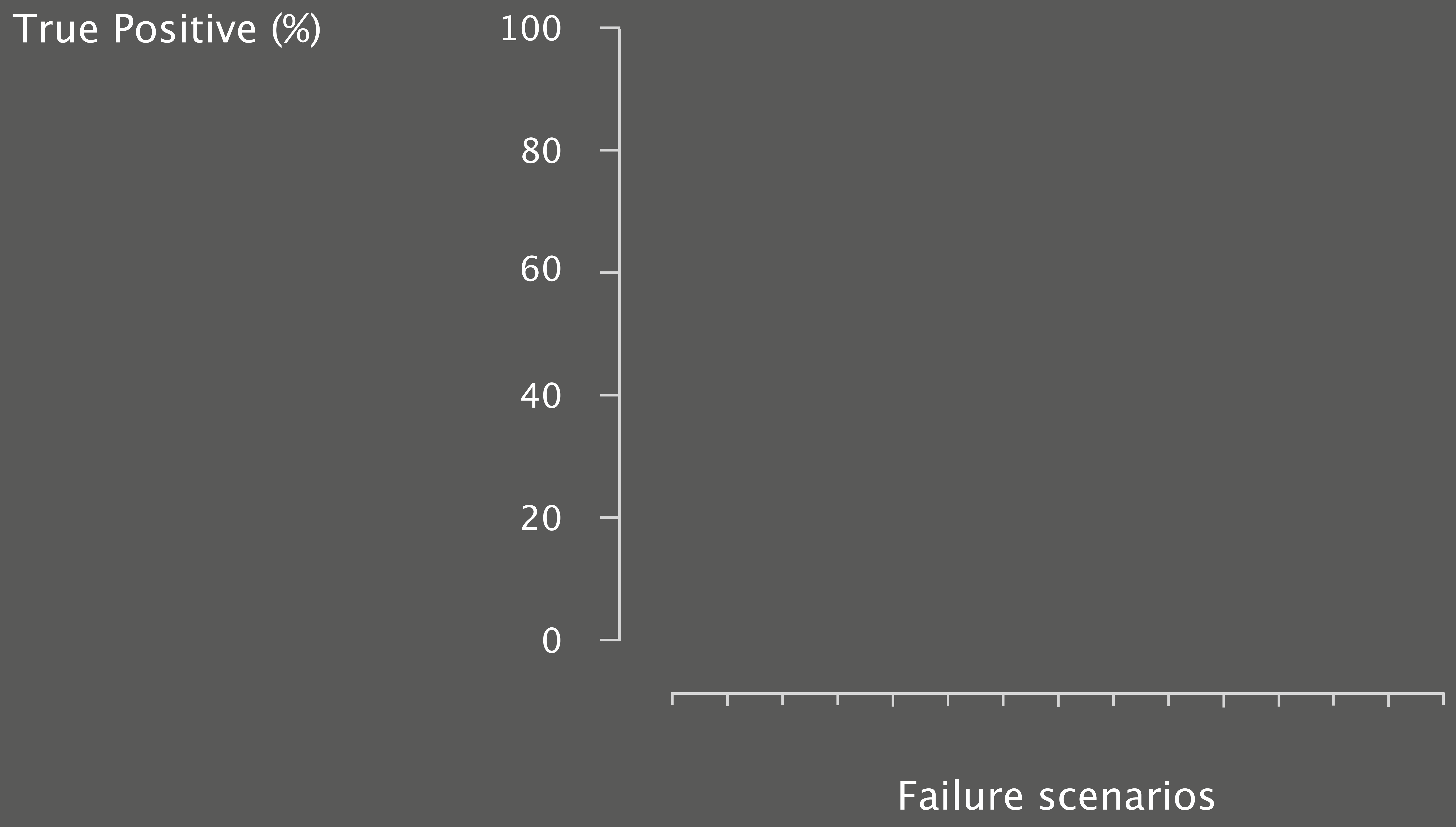
Signal fades away

Focus on active flows

Signal is composed of many small ones

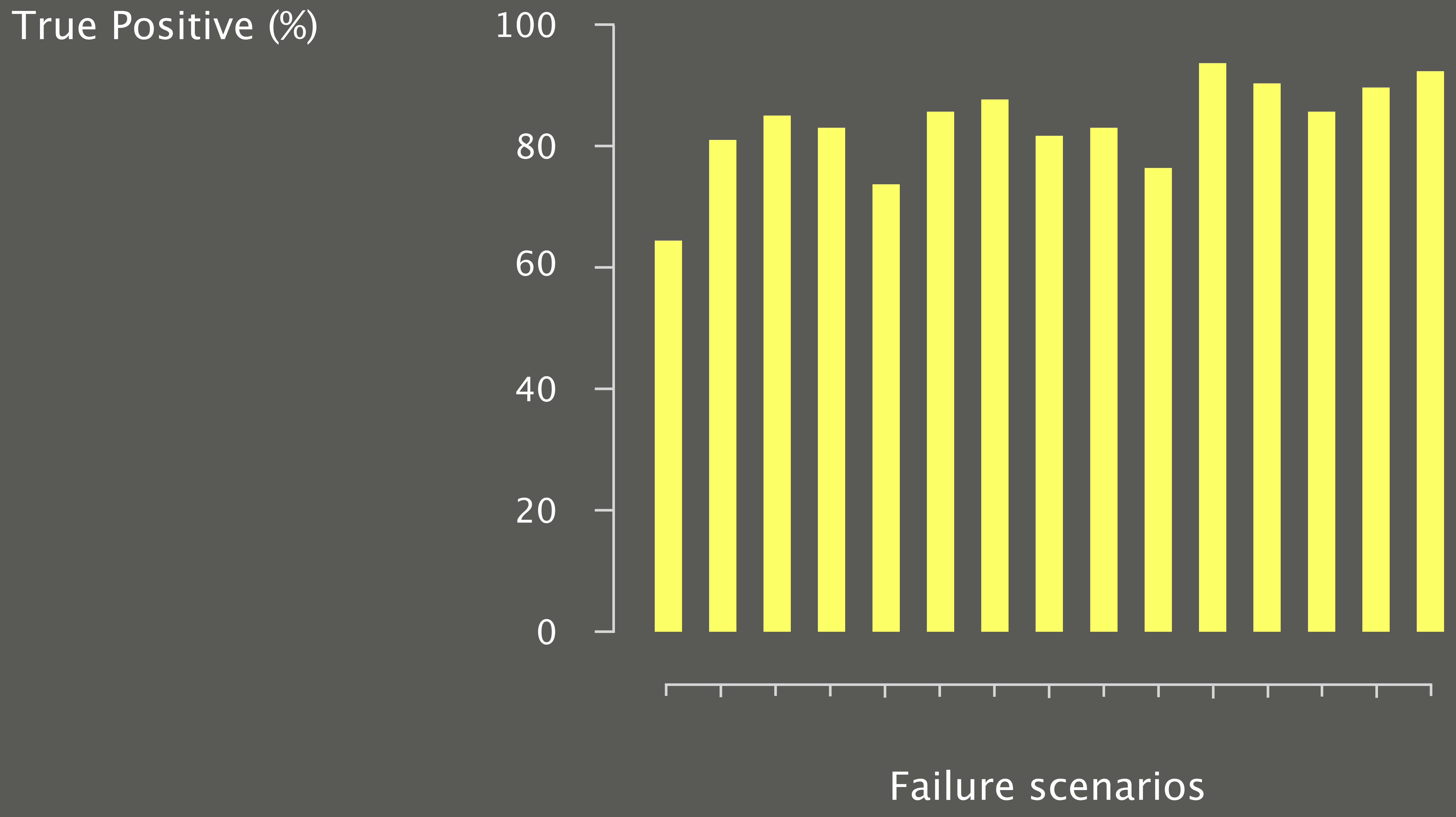
Rely on scalable data structures and sampling

Traffic-driven failure inference is accurate

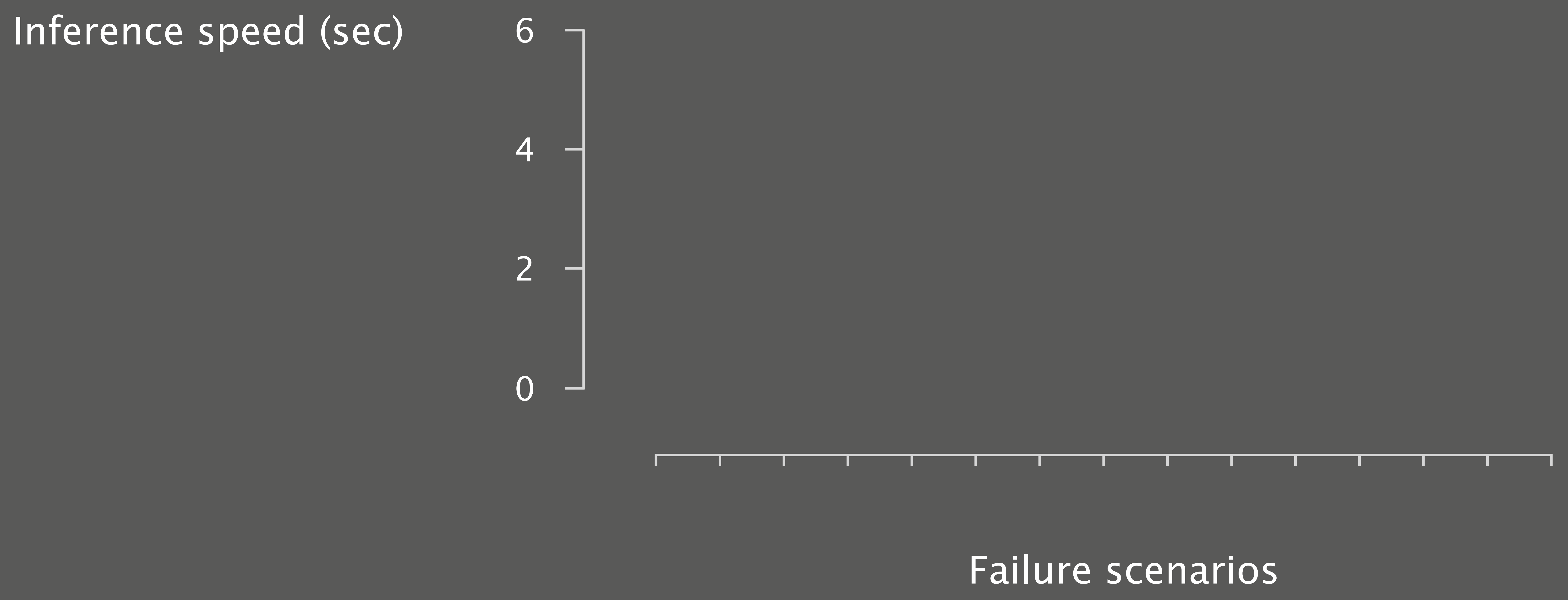


Traffic-driven failure inference is accurate

Accuracy is >80% in most cases

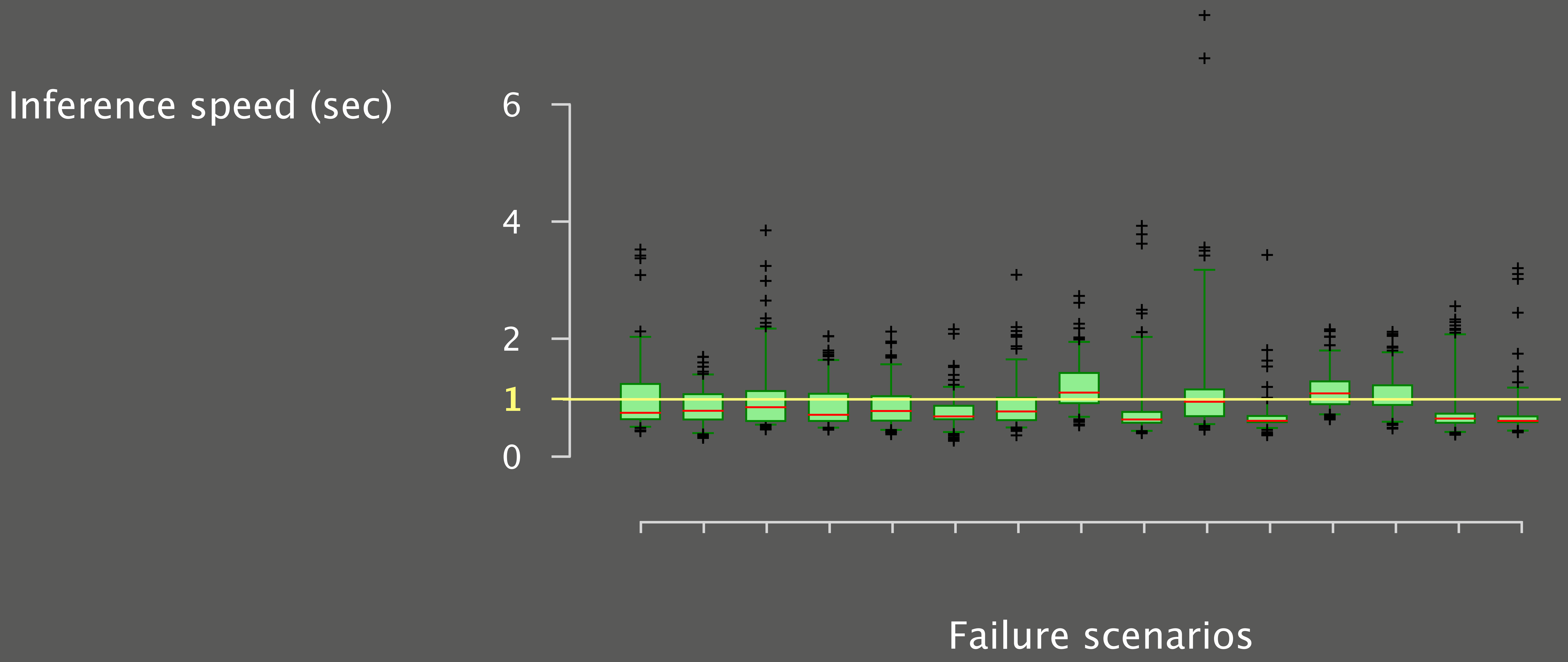


Traffic-driven failure inference is fast



Traffic-driven failure inference is fast

Inference is made <1s in most cases



What about we…

detect network changes

notify remote devices of the changes

compute new paths

update the local forwarding state

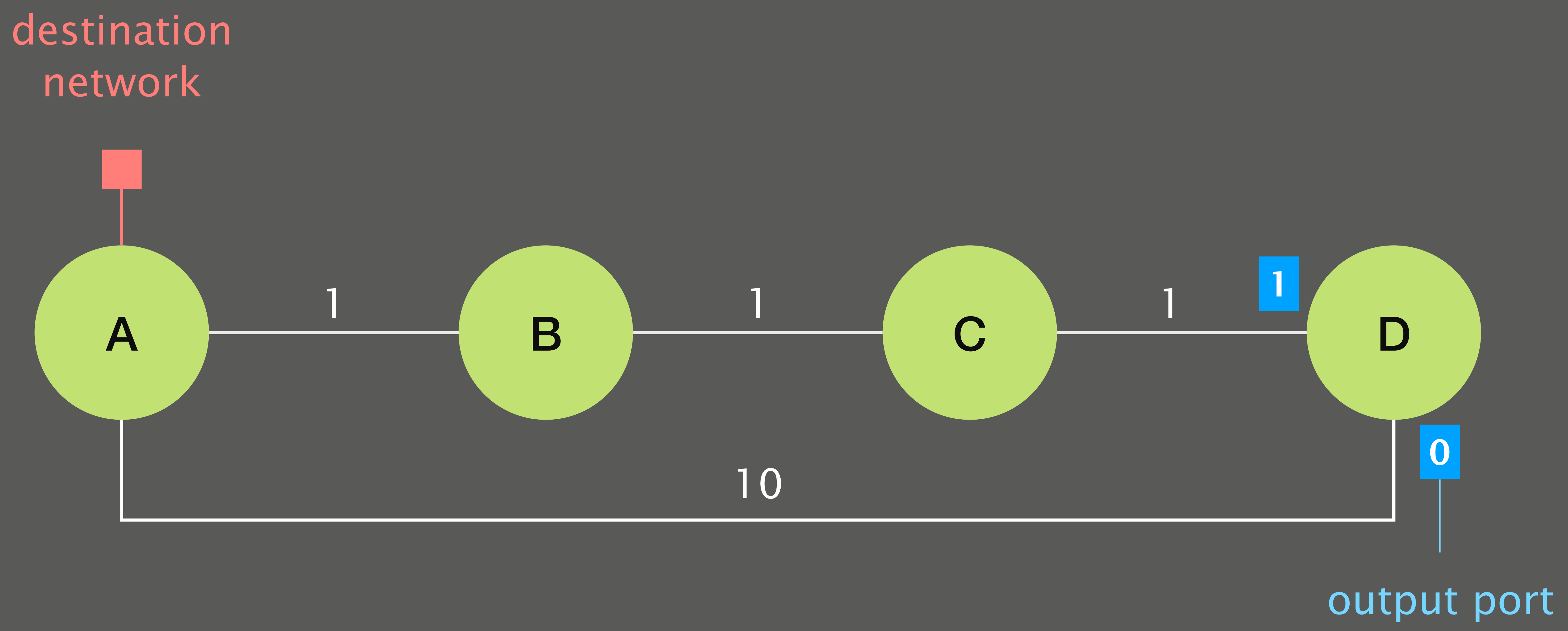
…in hardware?!

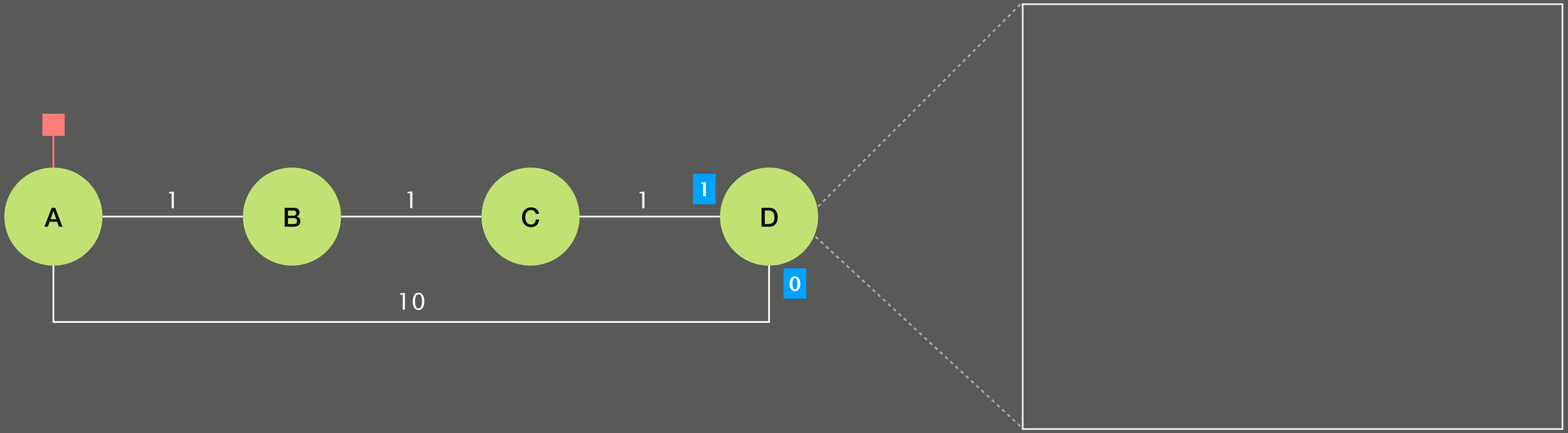
Switches can compute *and* update their forwarding state in hardware, without a controller

Switches can compute *and* update their forwarding state

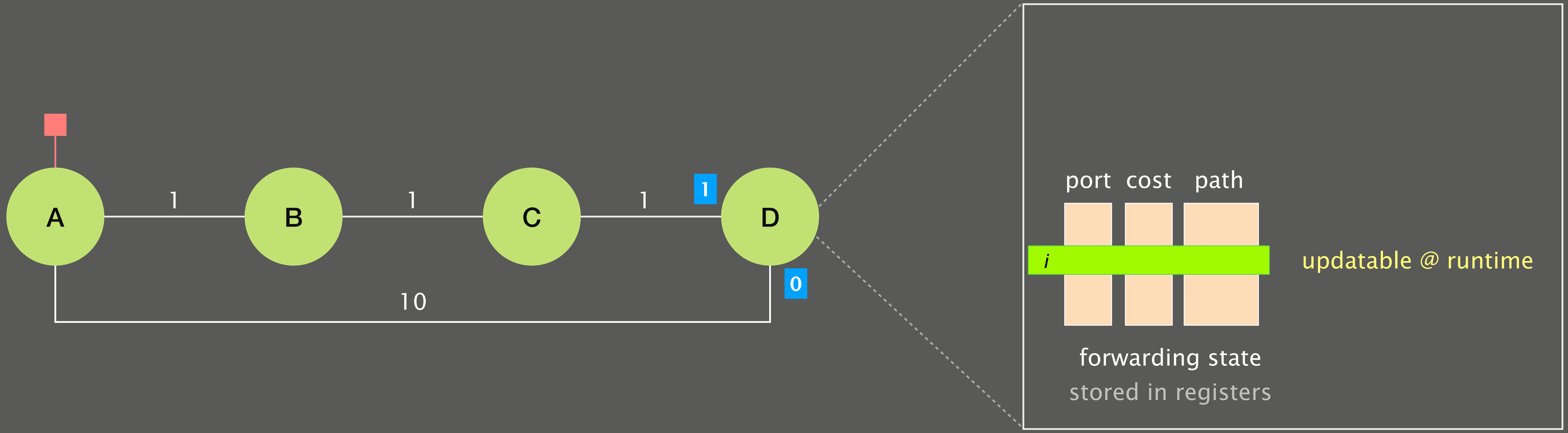
in hardware, without a controller

Let me show you how (in P4)

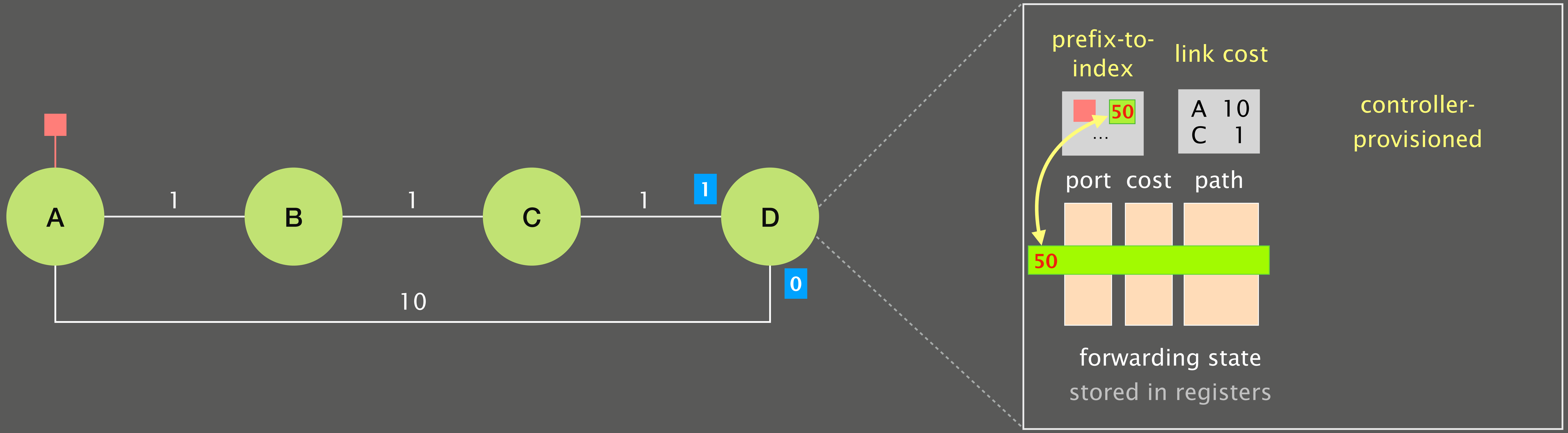

destination network
A
B
C
D
1
1
1
10
1
0
output port


A
B
C
D
1
1
1
10
1
0

Use registers to store the best paths and their attributes

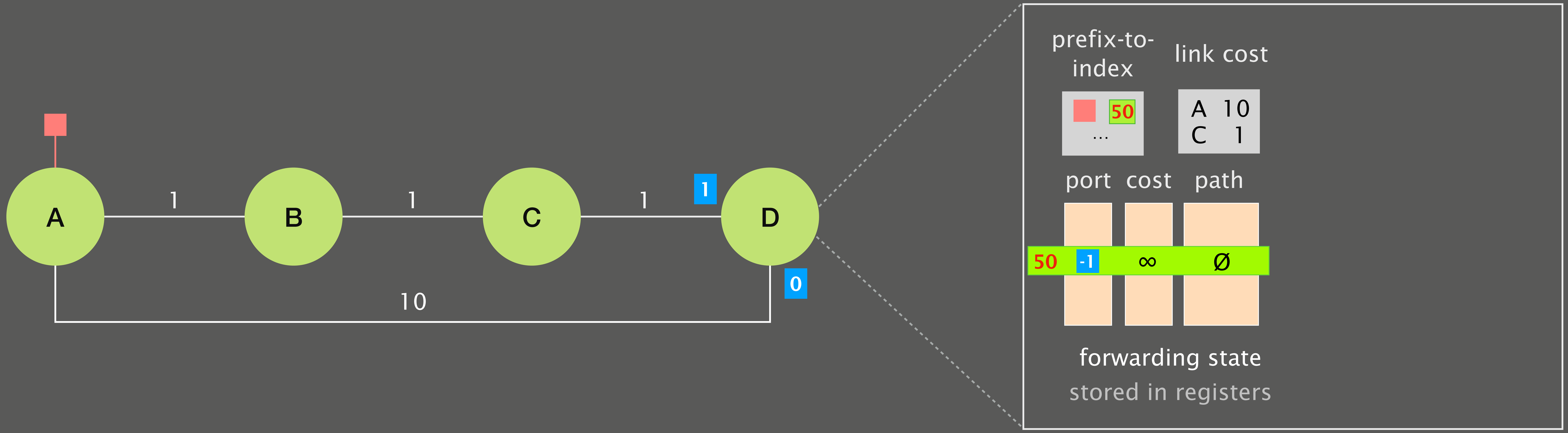


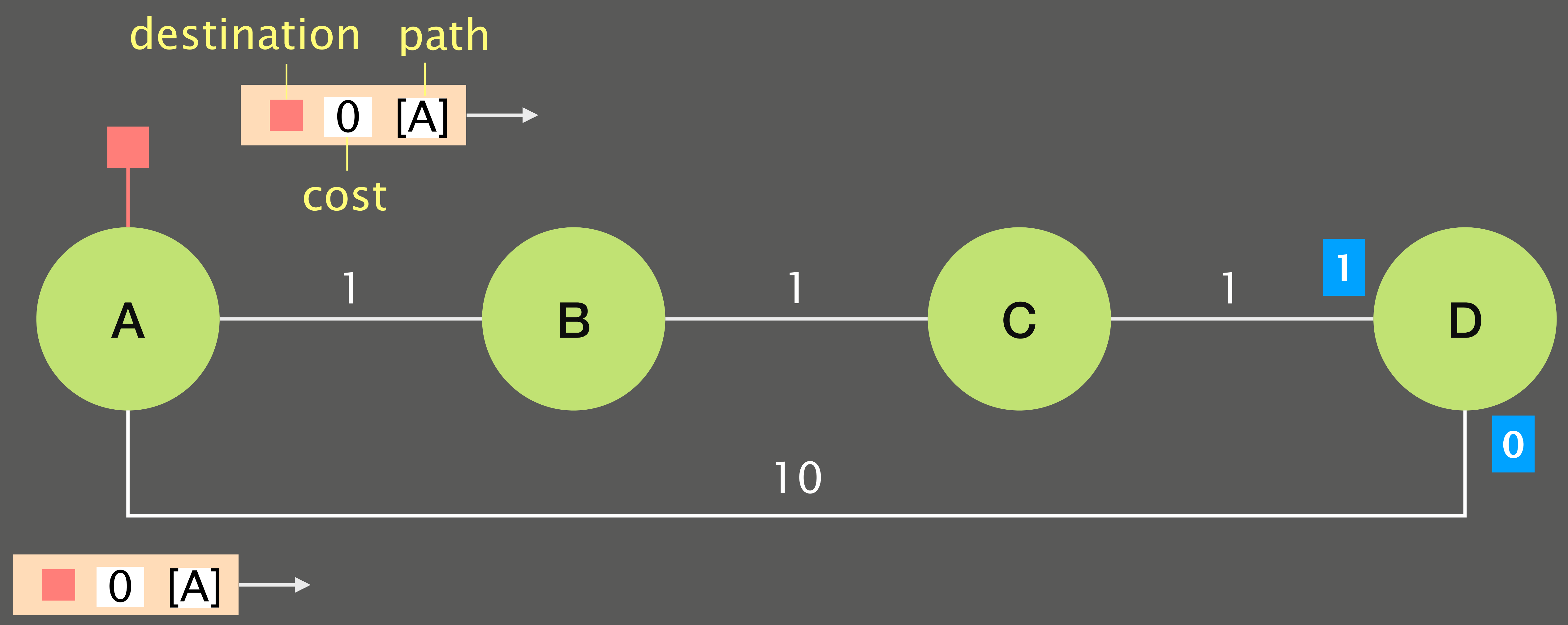
Use tables to store link costs and to map destinations to their register entries

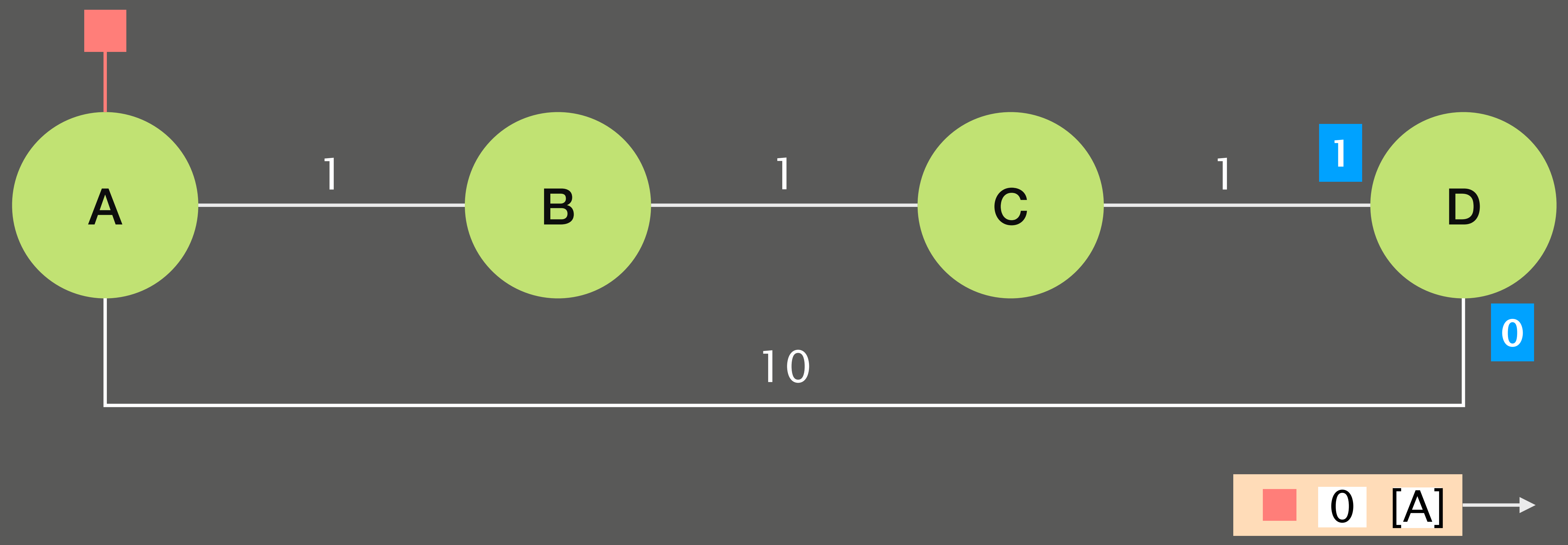


Let the switches flood their forwarding state,

one destination at time

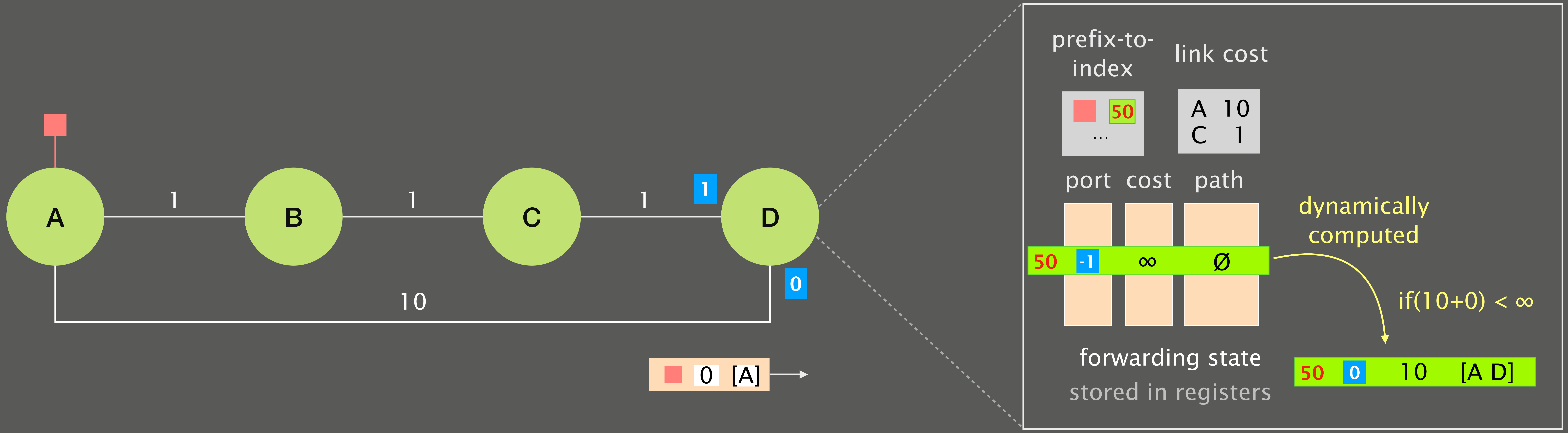



destination
path
🟥 0 [A]
cost
A
B
C
D
1
1
1
1
0
10
🟥 0 [A]

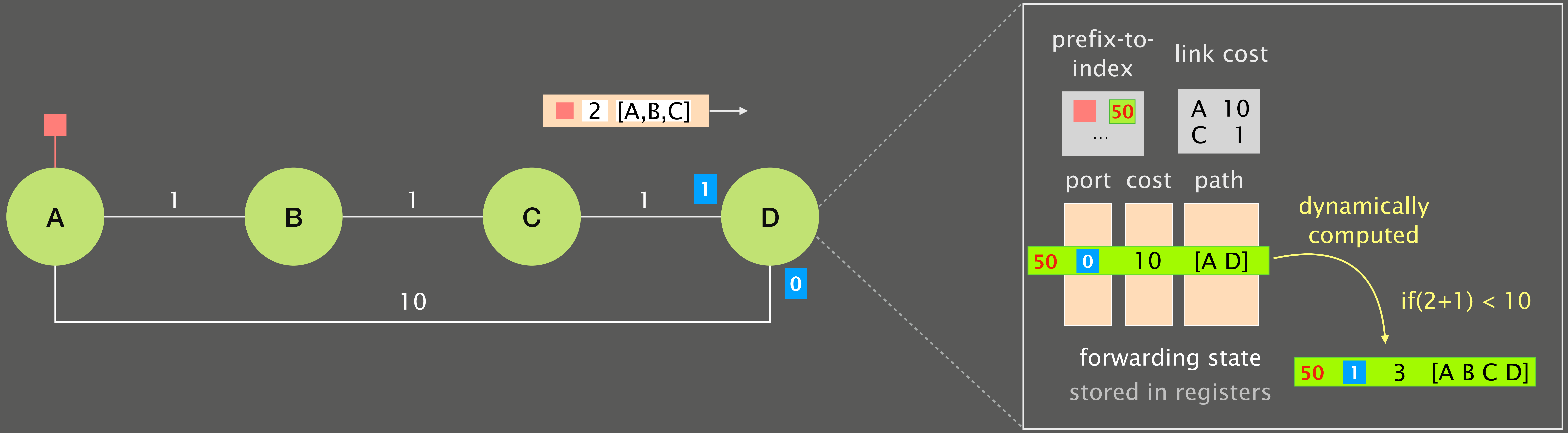

A
1
B
1
C
1
1
D
0
10
🟥 0 [A]

Upon receiving a routing message,

have the switches recompute their forwarding state



Upon receiving a routing message, have the switches recompute their forwarding state



We have a working P4$_{16}$ prototype

Tofino implementation coming up

Implementation	>2000 lines of code run on software model (bmv2)
Capabilities	Intra-domain destinations path-vector routing (a la RIP) Inter-domain destinations policy-based routing (a la BGP)

What about we…

detect network changes

notify remote devices of the changes

compute new paths

update the local forwarding state

…in hardware?!

Offloading routing tasks does not come for free

It is not *always* a good idea—think first!

Some tasks cannot be offloaded

e.g. crypto operations

Some tasks should probably not be offloaded

e.g. implementing the entire BGP/TCP protocol

Offloading routing tasks consumes hardware ressources

and those cannot be reused for other applications…

Programmable, hardware-based

Routing and Scheduling

Fast routing

data-plane offloading

2 Flexible scheduling

dynamic packet adaptation

Looking forward

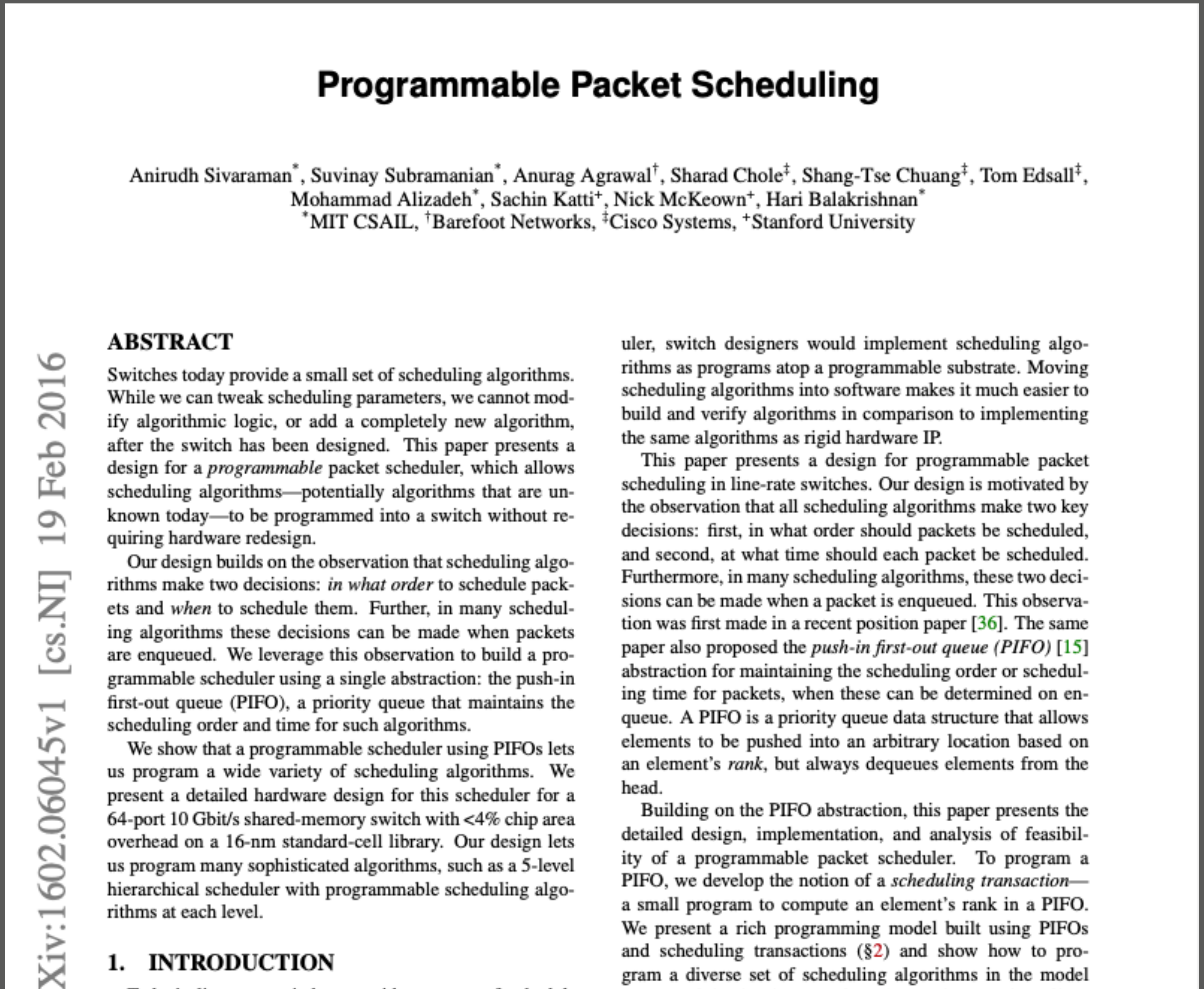
routing | forwarding

The need for diversity × lack of generality

motivates the need for flexible scheduling

Push-In First-Out Queue (PIFO) is a data structure that enables programmable packet scheduling

SIGCOMM'16



Programmable Packet Scheduling

Anirudh Sivaraman*, Suvinay Subramanian*, Anurag Agrawal†, Sharad Chole‡, Shang-Tse Chuang‡, Tom Edsall‡,
Mohammad Alizadeh*, Sachin Katti+, Nick McKeown+, Hari Balakrishnan*
*MIT CSAIL, †Barefoot Networks, ‡Cisco Systems, +Stanford University

ABSTRACT

Switches today provide a small set of scheduling algorithms. While we can tweak scheduling parameters, we cannot modify algorithmic logic, or add a completely new algorithm, after the switch has been designed. This paper presents a design for a *programmable* packet scheduler, which allows scheduling algorithms—potentially algorithms that are unknown today—to be programmed into a switch without requiring hardware redesign.

Our design builds on the observation that scheduling algorithms make two decisions: *in what order* to schedule packets and *when* to schedule them. Further, in many scheduling algorithms these decisions can be made when packets are enqueued. We leverage this observation to build a programmable scheduler using a single abstraction: the push-in first-out queue (PIFO), a priority queue that maintains the scheduling order and time for such algorithms.

We show that a programmable scheduler using PIFOs lets us program a wide variety of scheduling algorithms. We present a detailed hardware design for this scheduler for a 64-port 10 Gbit/s shared-memory switch with <4% chip area overhead on a 16-nm standard-cell library. Our design lets us program many sophisticated algorithms, such as a 5-level hierarchical scheduler with programmable scheduling algorithms at each level.

1. INTRODUCTION

uler, switch designers would implement scheduling algorithms as programs atop a programmable substrate. Moving scheduling algorithms into software makes it much easier to build and verify algorithms in comparison to implementing the same algorithms as rigid hardware IP.

This paper presents a design for programmable packet scheduling in line-rate switches. Our design is motivated by the observation that all scheduling algorithms make two key decisions: first, in what order should packets be scheduled, and second, at what time should each packet be scheduled. Furthermore, in many scheduling algorithms, these two decisions can be made when a packet is enqueued. This observation was first made in a recent position paper [36]. The same paper also proposed the *push-in first-out queue (PIFO)* [15] abstraction for maintaining the scheduling order or scheduling time for packets, when these can be determined on enqueue. A PIFO is a priority queue data structure that allows elements to be pushed into an arbitrary location based on an element's *rank*, but always dequeues elements from the head.

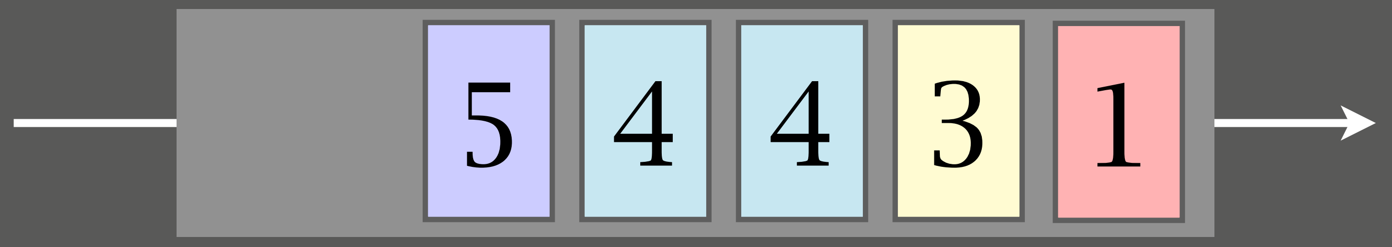
Building on the PIFO abstraction, this paper presents the detailed design, implementation, and analysis of feasibility of a programmable packet scheduler. To program a PIFO, we develop the notion of a *scheduling transaction*—a small program to compute an element's rank in a PIFO. We present a rich programming model built using PIFOs and scheduling transactions (§2) and show how to program a diverse set of scheduling algorithms in the model

Push-In First-Out Queue (PIFO) is a data structure that enables programmable packet scheduling

A PIFO queue…

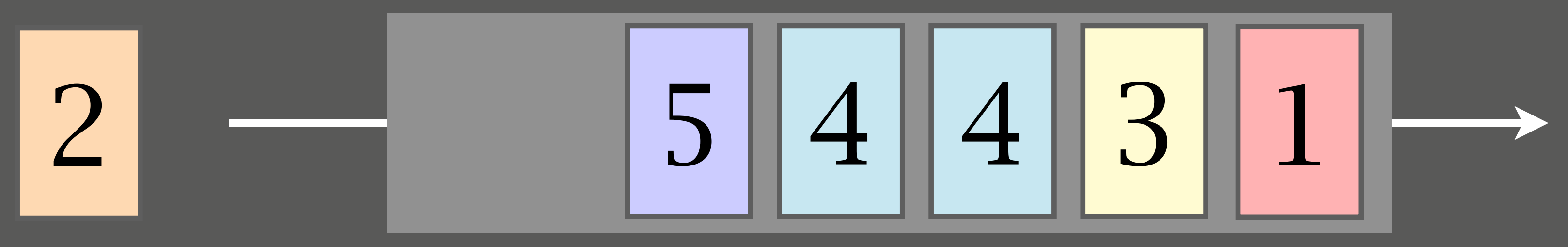
- perfectly reorders packets according to their ranks
- drains packets from the head

PIFO queue



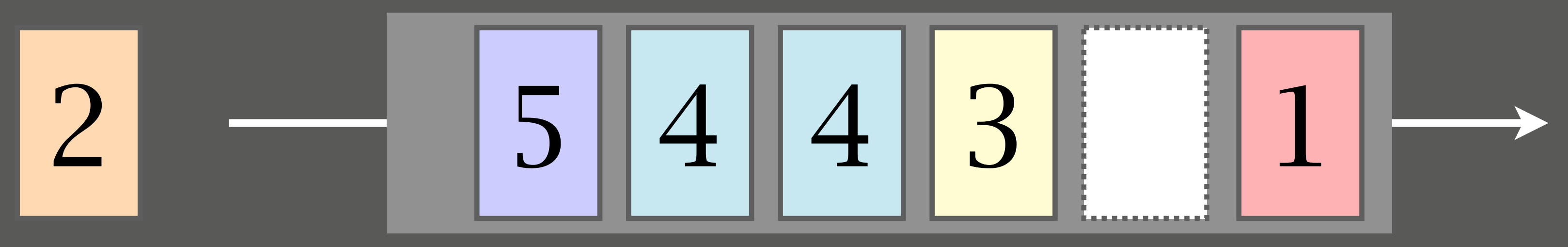
Incoming packets

PIFO queue



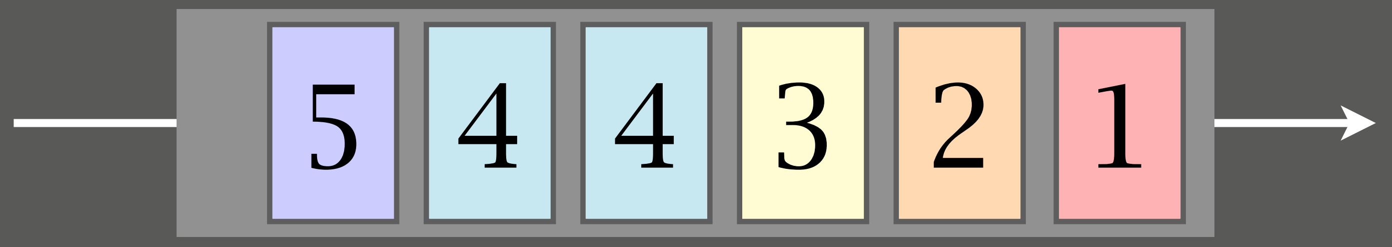
Incoming packets

PIFO queue



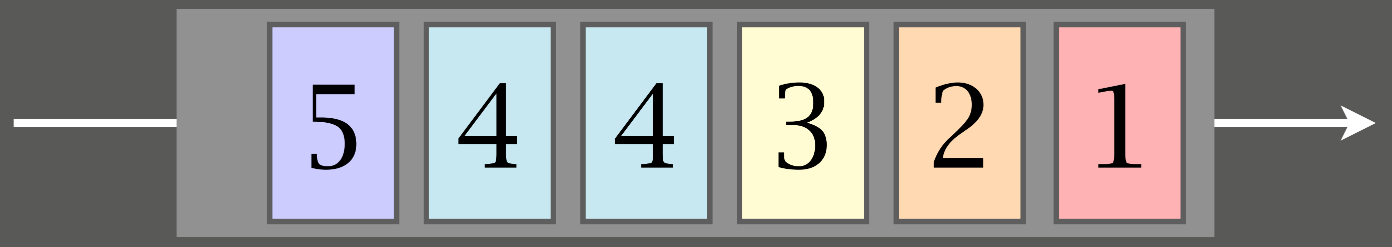
Incoming
packets

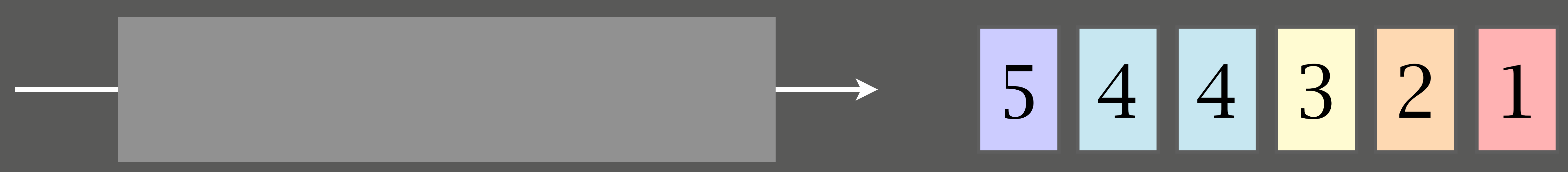
PIFO queue



PIFO queue

Outgoing packets





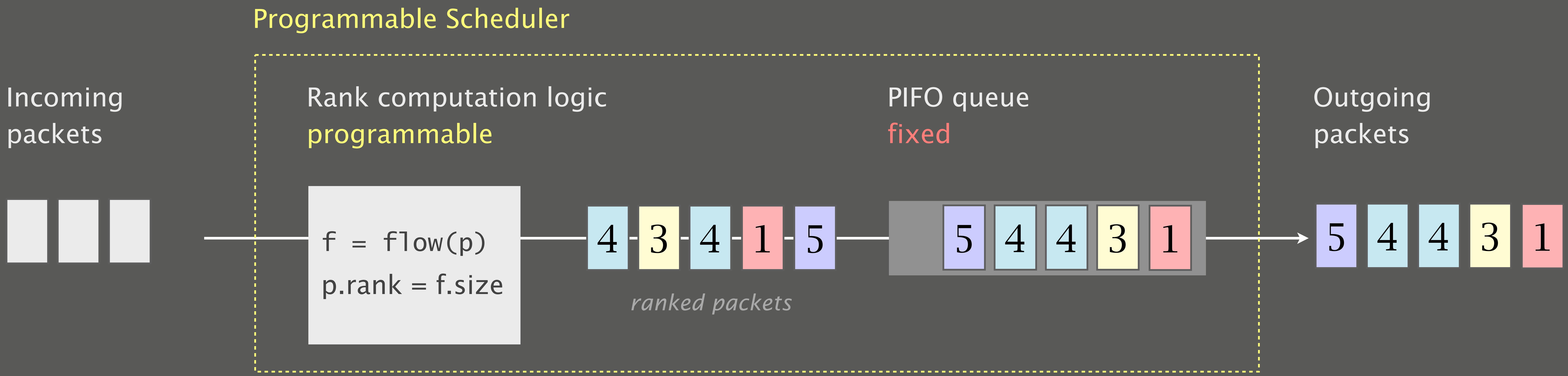
PIFO queue

Outgoing packets

Push-In First-Out Queue (PIFO) is a data structure that enables programmable packet scheduling

How exactly?

Implementing a new algorithm simply requires to adapt the rank computation logic



```
f = flow(p)
p.rank = f.size
```

Implementing PIFO queues in hardware is challenging

	Existing proposal…
Scalability	supports ~1k flows and ~10 Gbps
Flexibility	assumes monotonically increasing ranks
	Moreover…
Deployability	implementing ASICs takes years

Can we approximate PIFO queues…

- at line rate;
- at scale;
- on existing devices?

Can we approximate PIFO queues…

- at line rate;
- at scale;
- on existing devices?

Yep!

Can we approximate PIFO queues…

- at line rate;
- at scale;
- on existing devices?

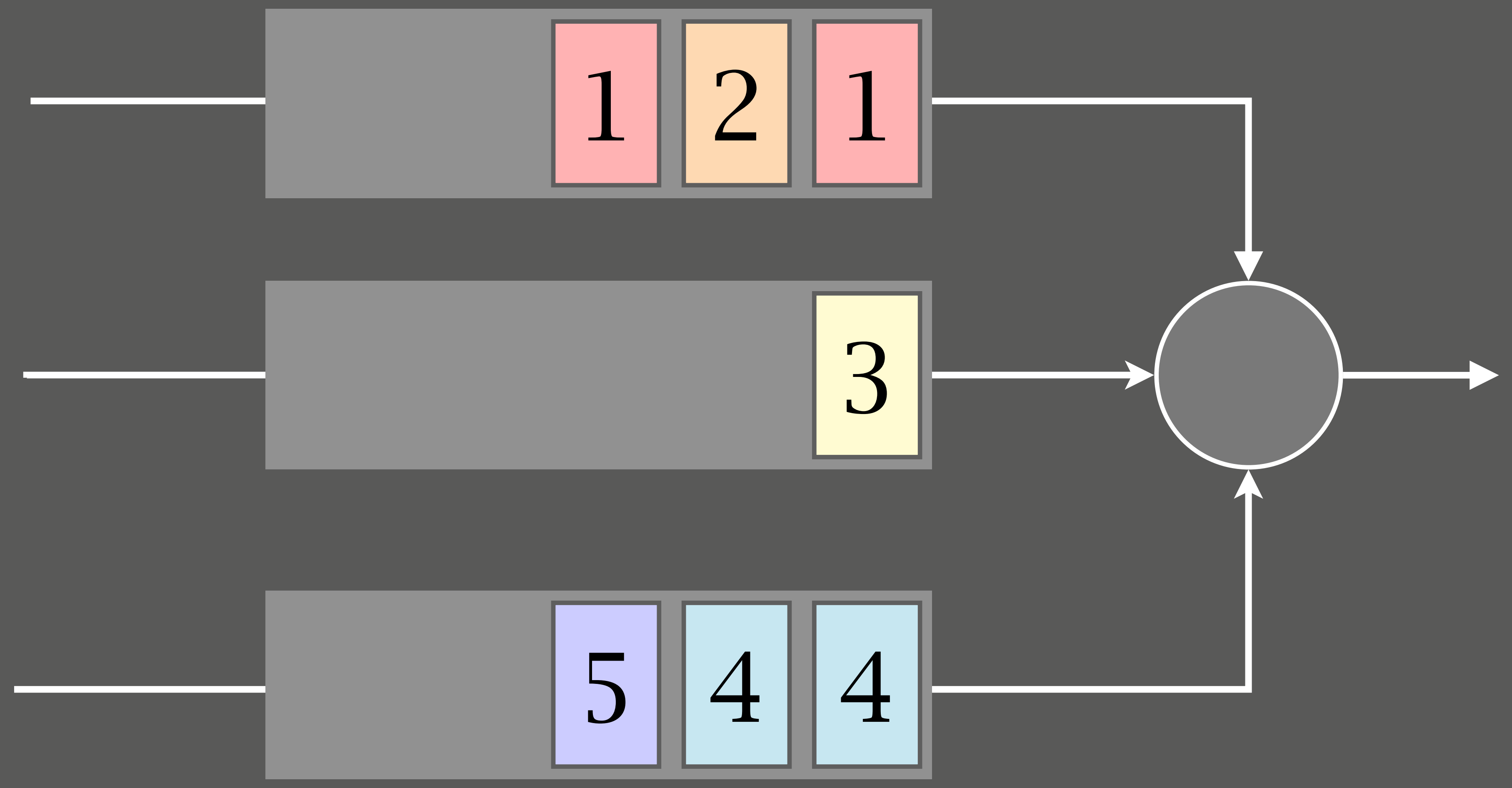
Yep!

Introducing **SP-PIFO**

SP-PIFO approximates PIFO queues using
strict-priority queues and a dynamic mapping strategy

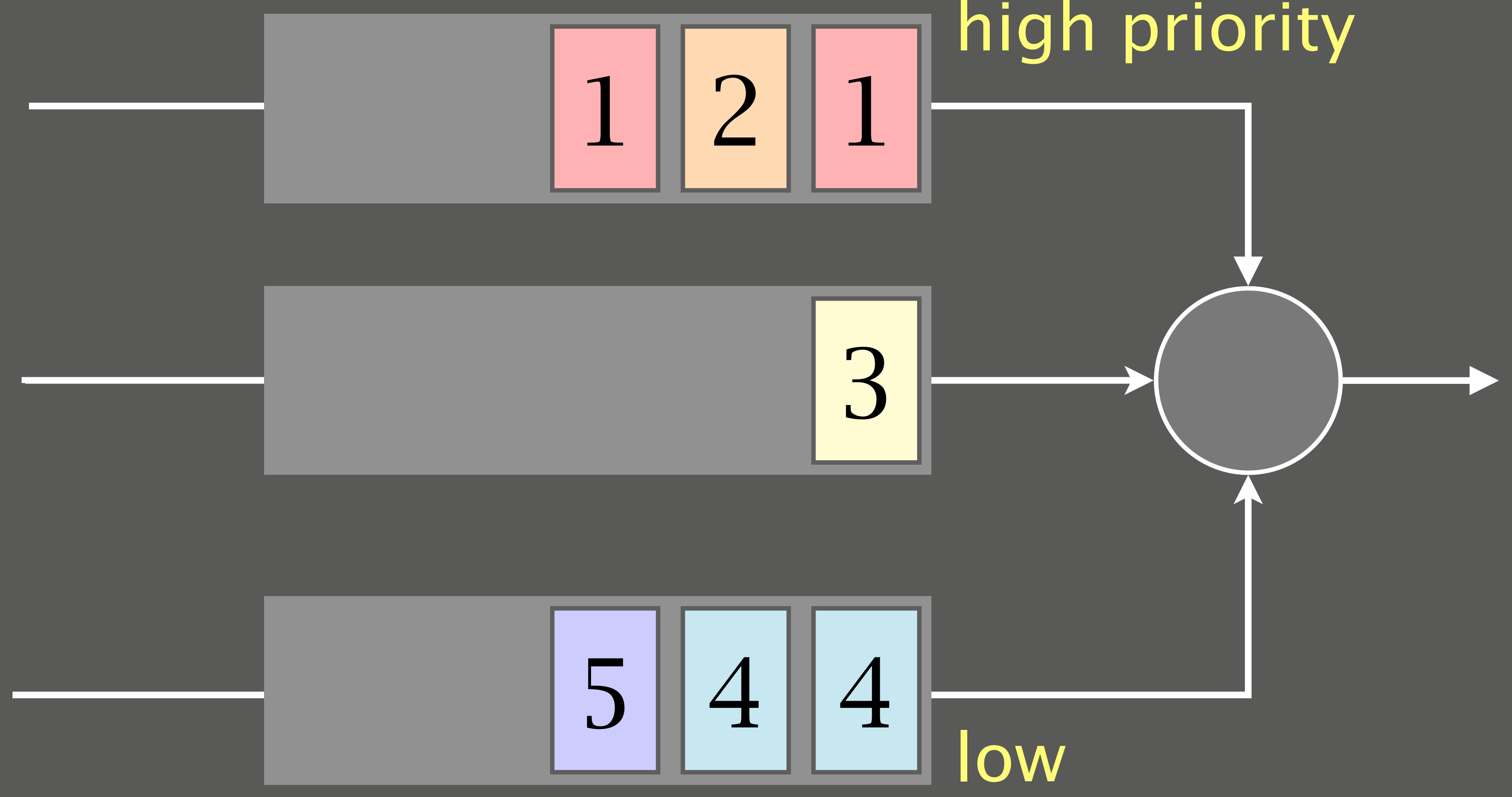
SP-PIFO approximates PIFO queues using

strict-priority queues and a dynamic mapping strategy



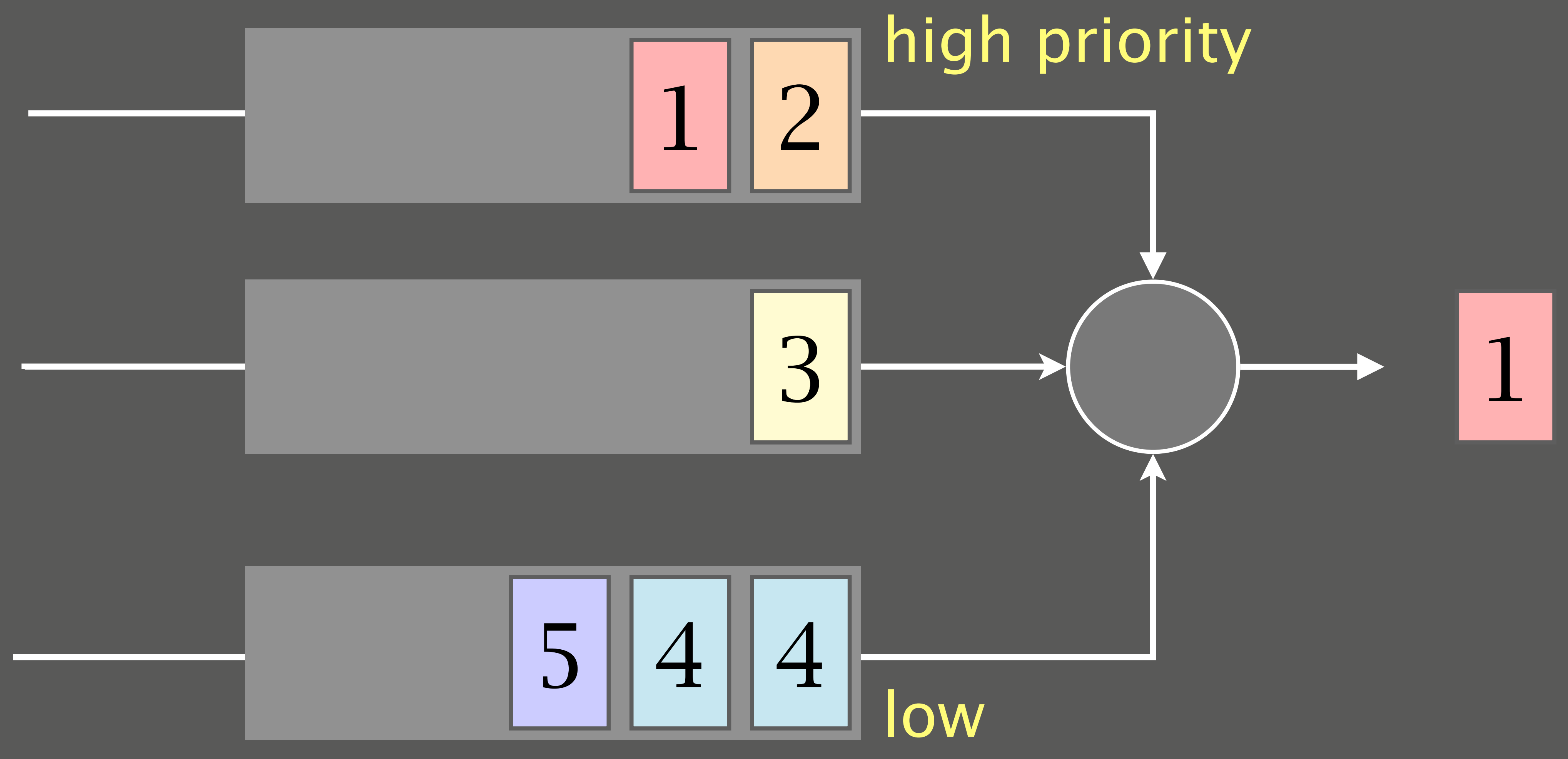
SP-PIFO approximates PIFO queues using

strict-priority queues and a dynamic mapping strategy



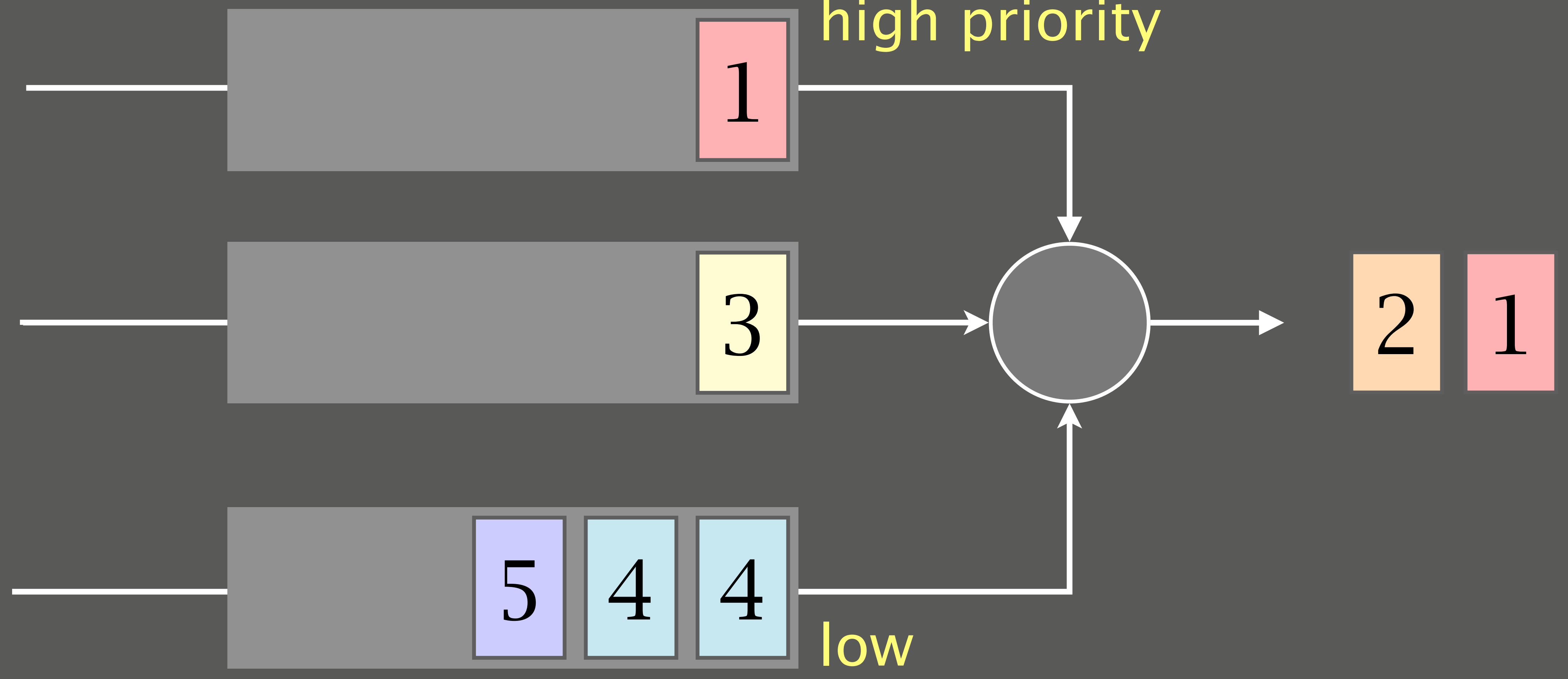
SP-PIFO approximates PIFO queues using

strict-priority queues and a dynamic mapping strategy



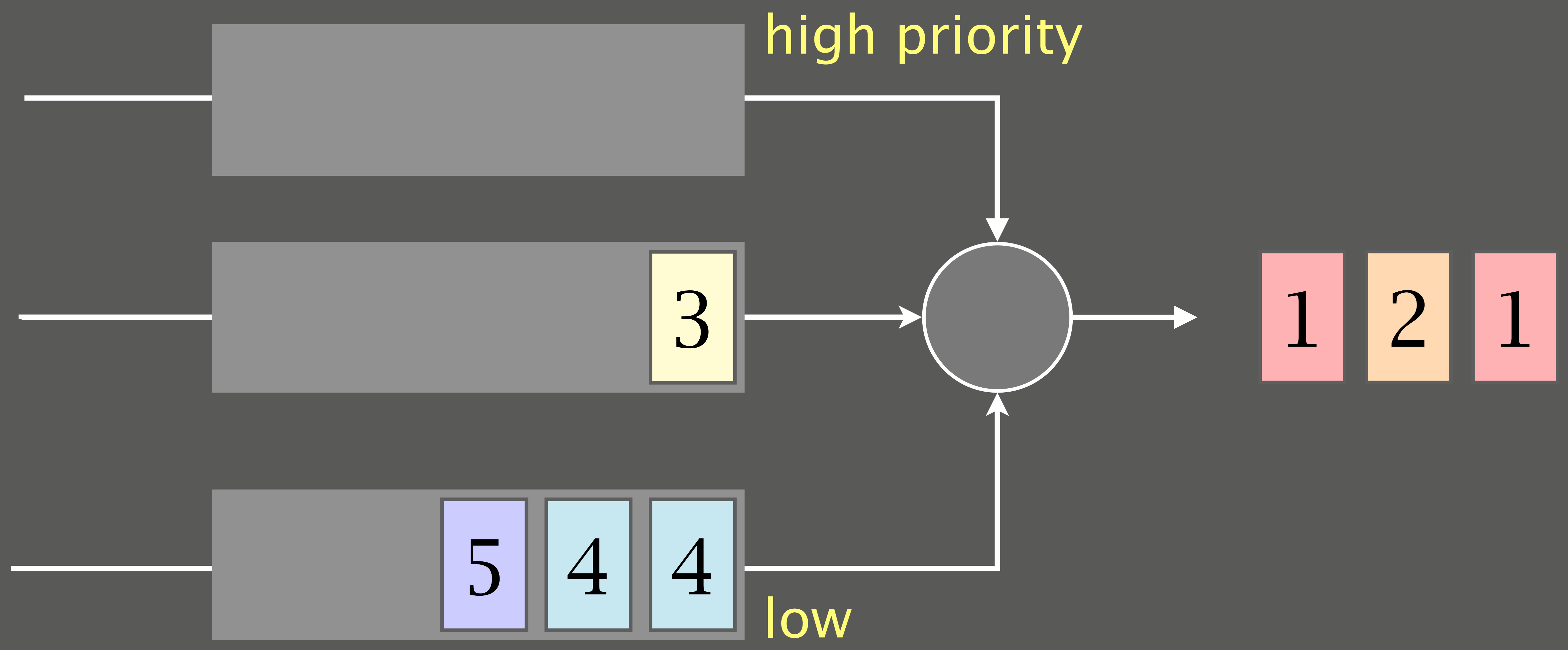
SP-PIFO approximates PIFO queues using

strict-priority queues and a dynamic mapping strategy

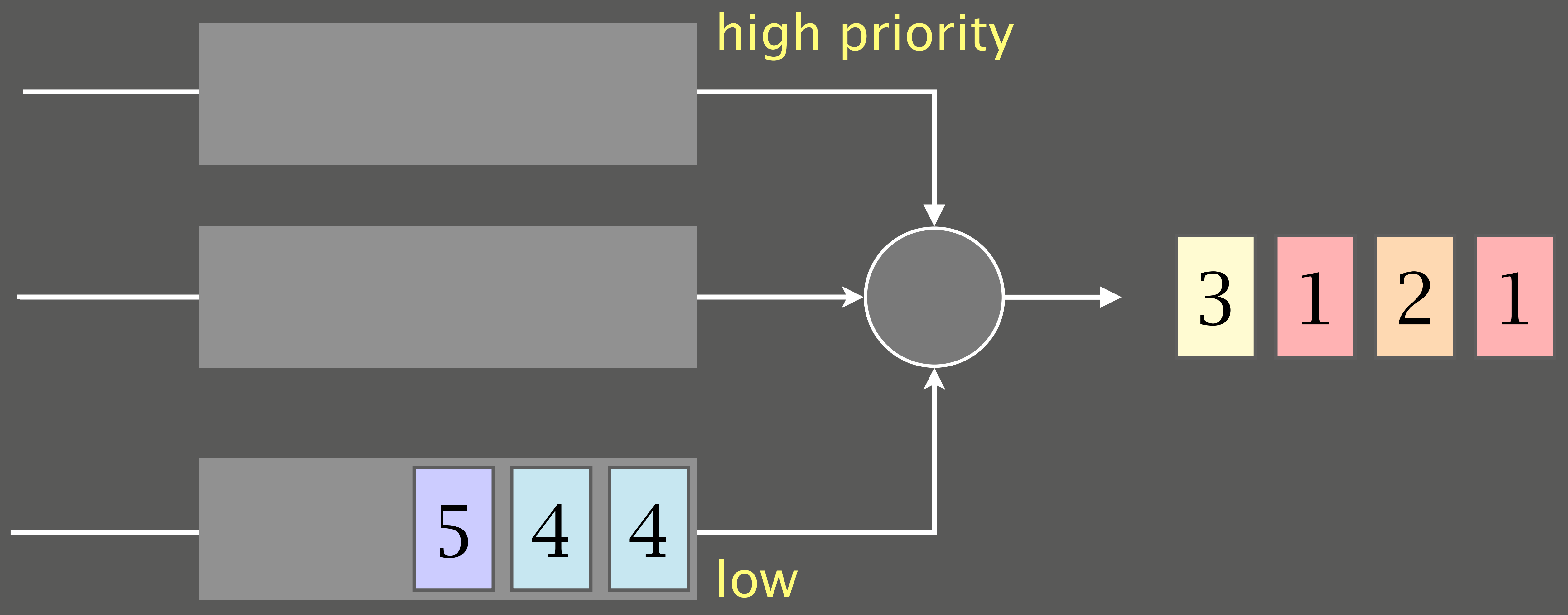


SP-PIFO approximates PIFO queues using

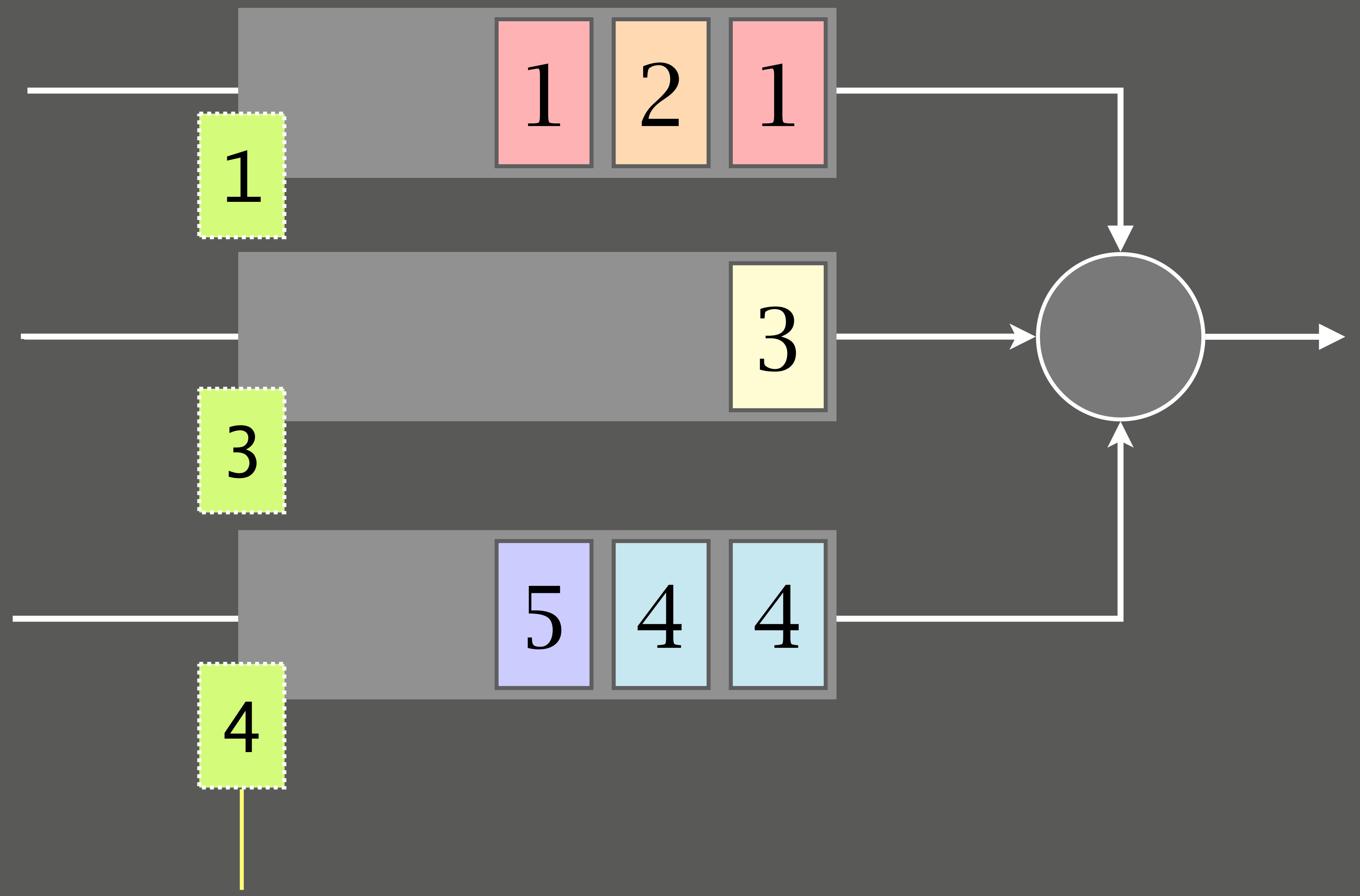
strict-priority queues and a dynamic mapping strategy



SP-PIFO approximates PIFO queues using strict-priority queues and a dynamic mapping strategy



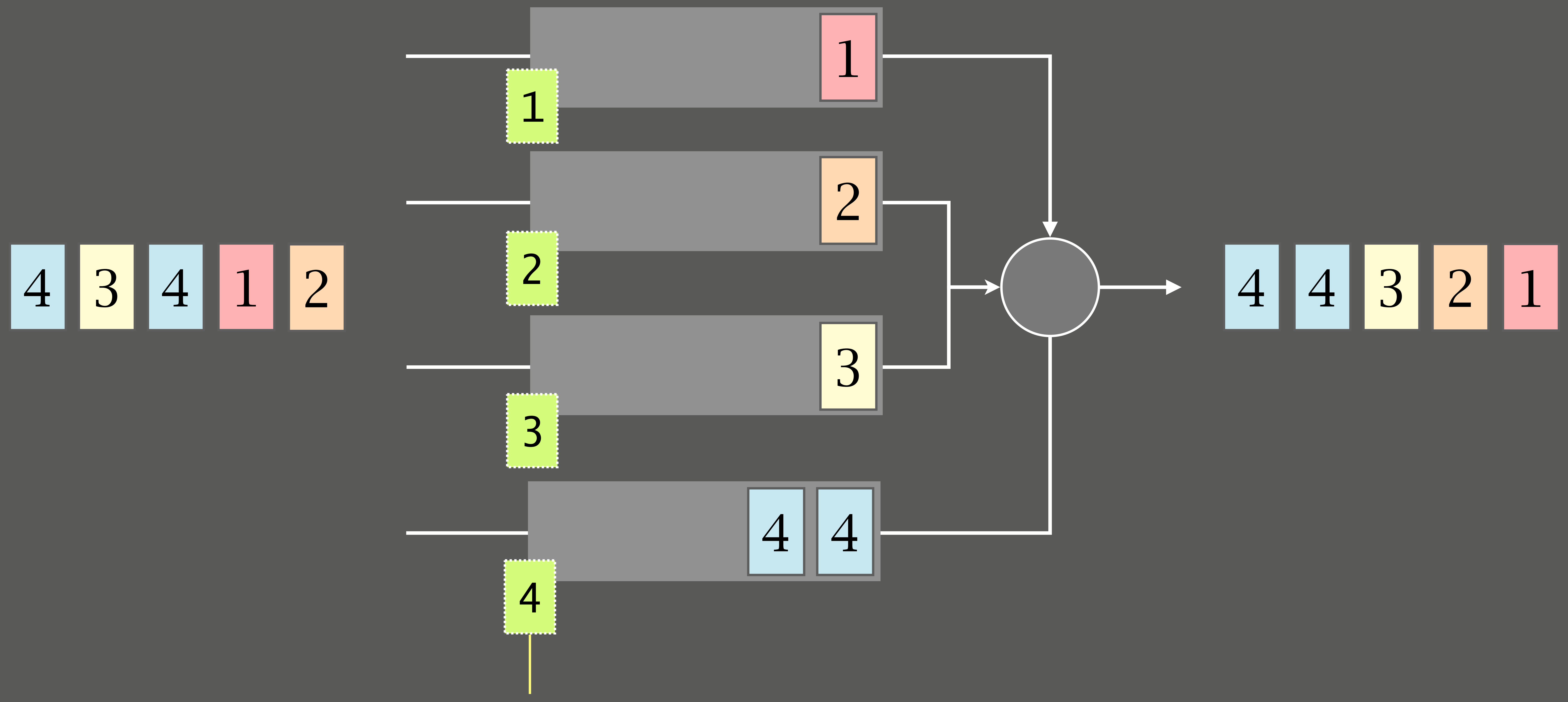
SP-PIFO approximates PIFO queues using strict-priority queues and a dynamic mapping strategy



queue mapping policy:

enqueues if rank ≥ queue_{idx}
when scanning bottom-up

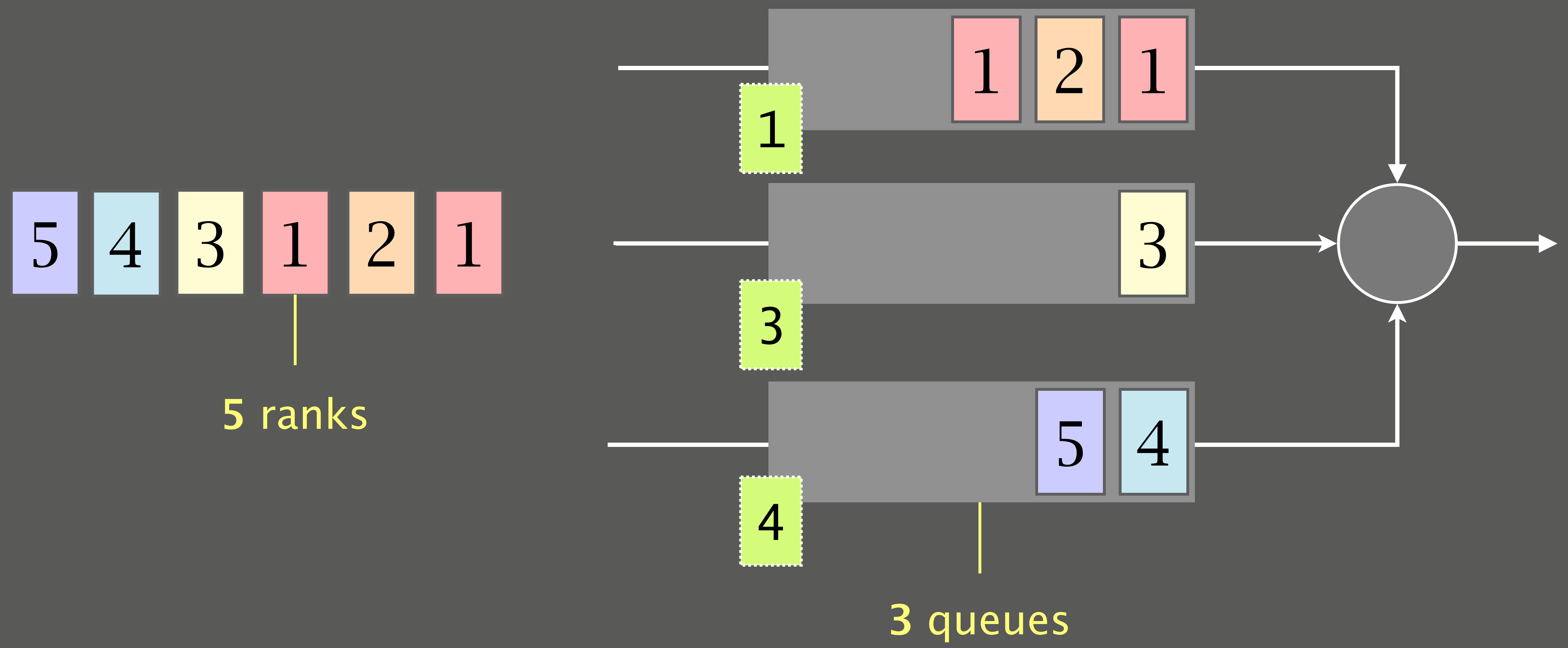
If there are as many queues as ranks,
SP-PIFO is equivalent to PIFO

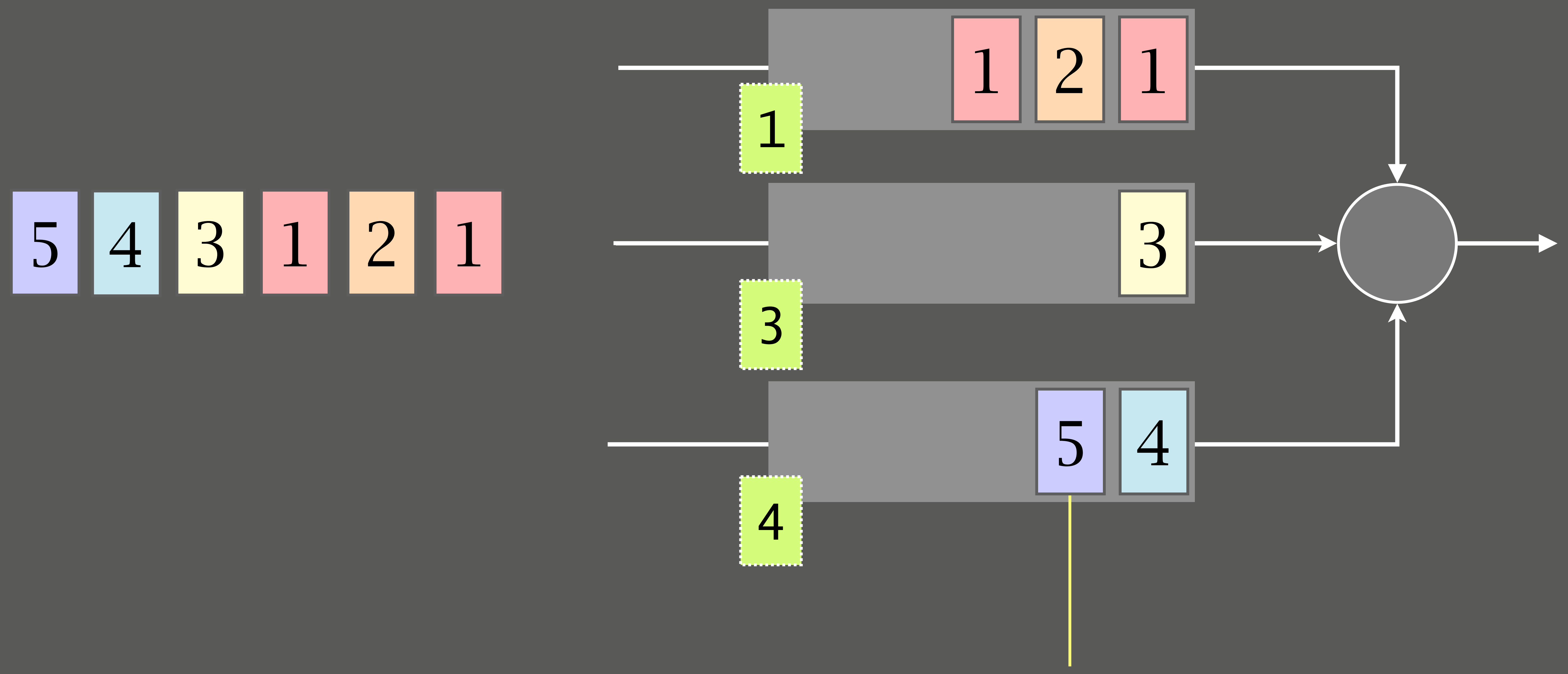


exactly one rank per queue

In practice though,

number of ranks >> number of queues

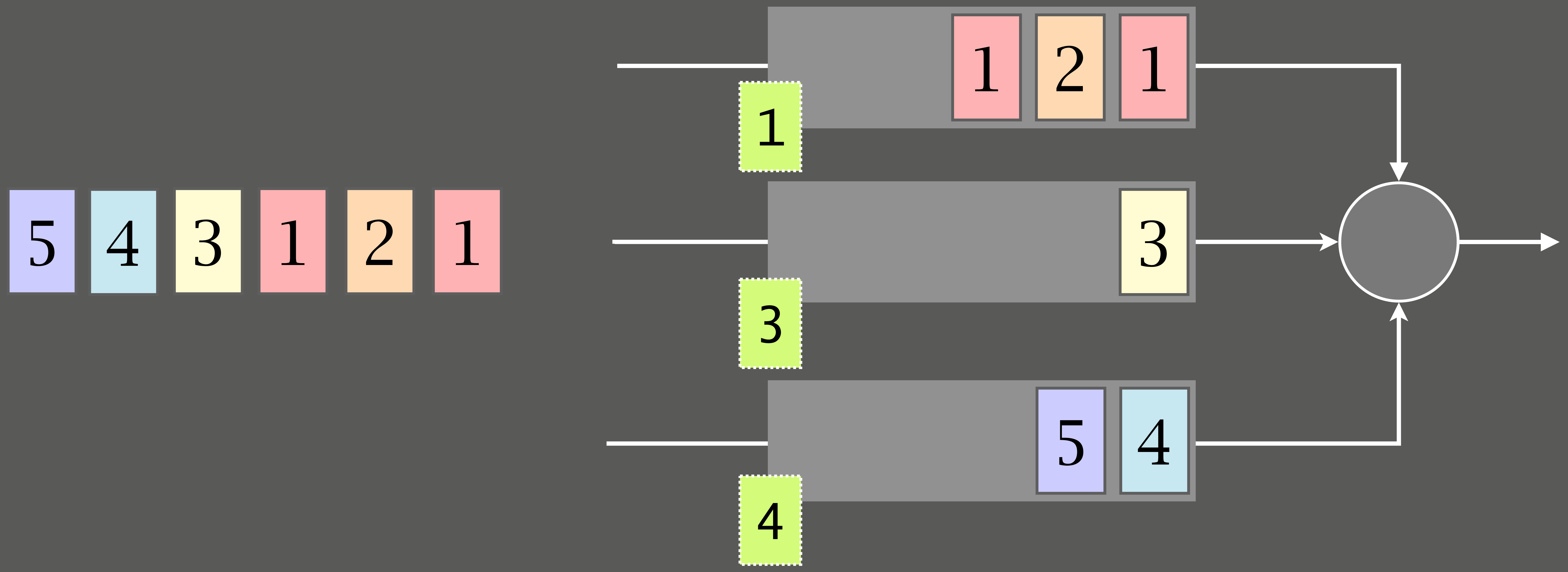


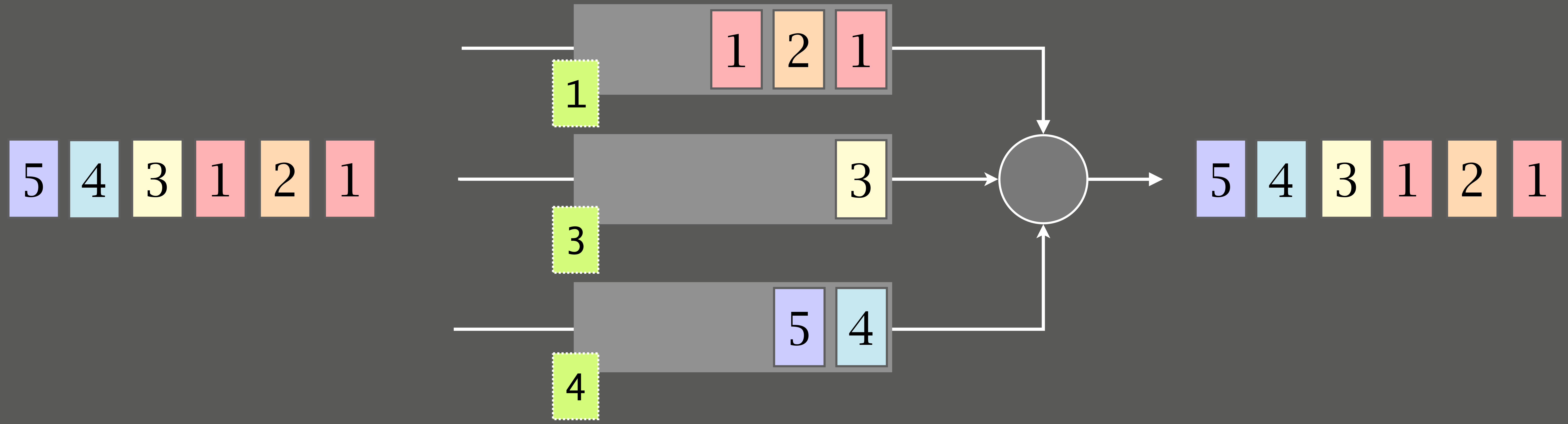


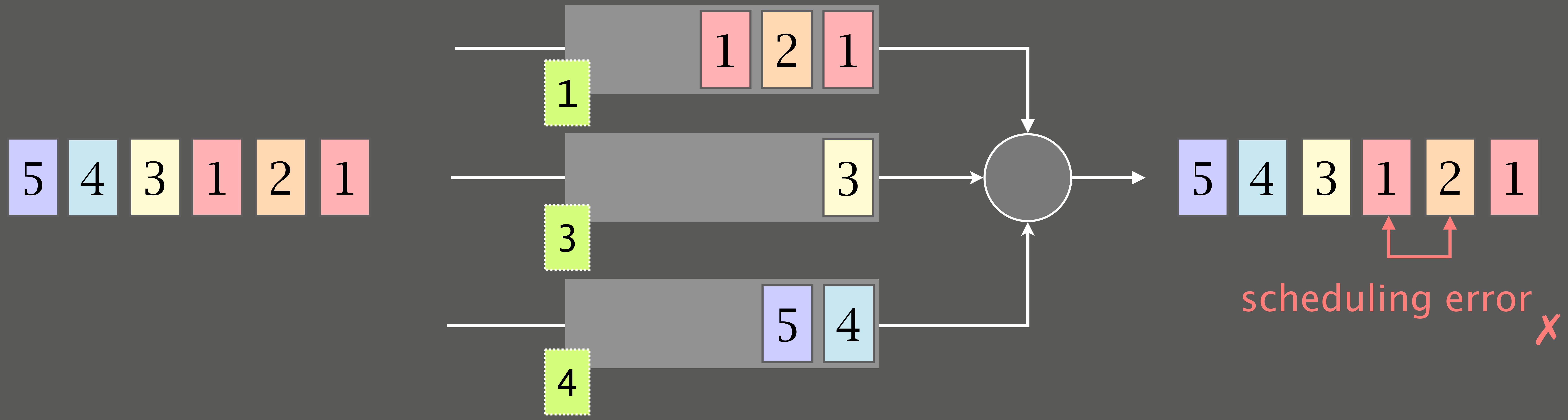
Different ranks share the same queues

ranks {1,2} and ranks {4,5}

Depending on the packets order, having distinct ranks in one queue can lead to scheduling errors


5
4
3
1
2
1
1
2
1
1
3
3
5
4
4


5
4
3
1
2
1
1
2
1
1
3
3
5
4
4
5
4
3
1
2
1

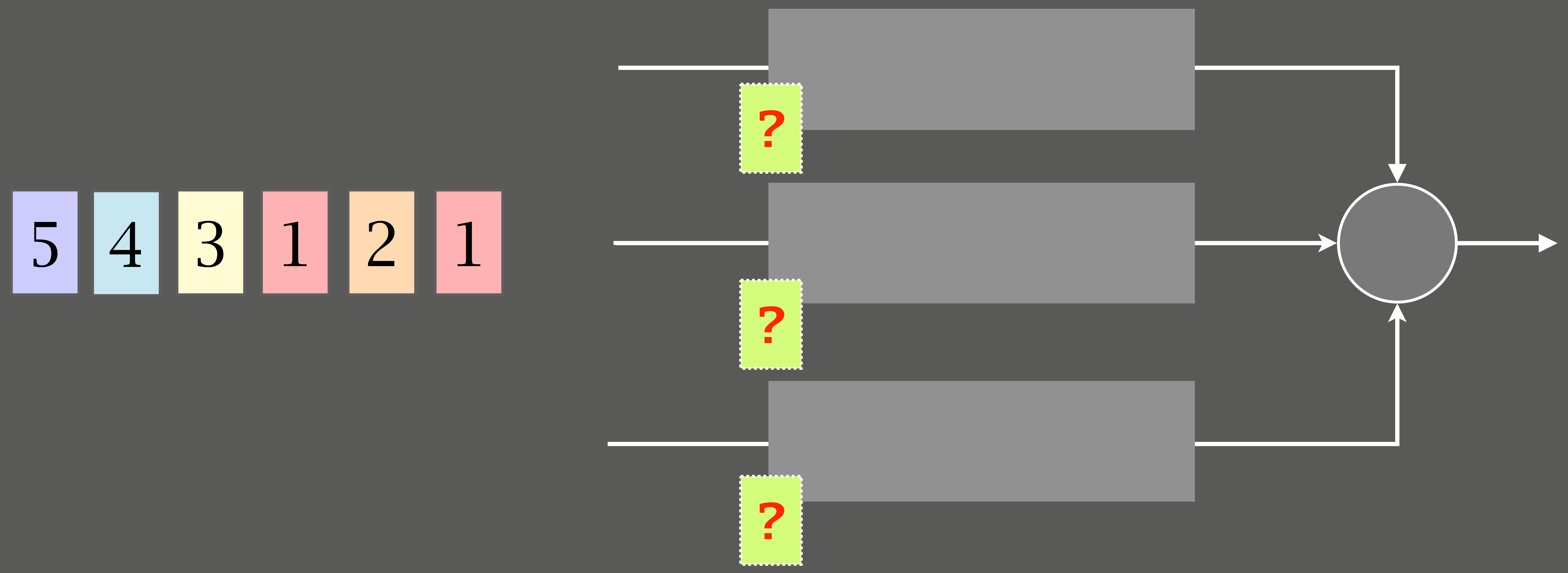


scheduling error ✗

We can minimize the number of scheduling errors
by dynamically adapting the mapping policy

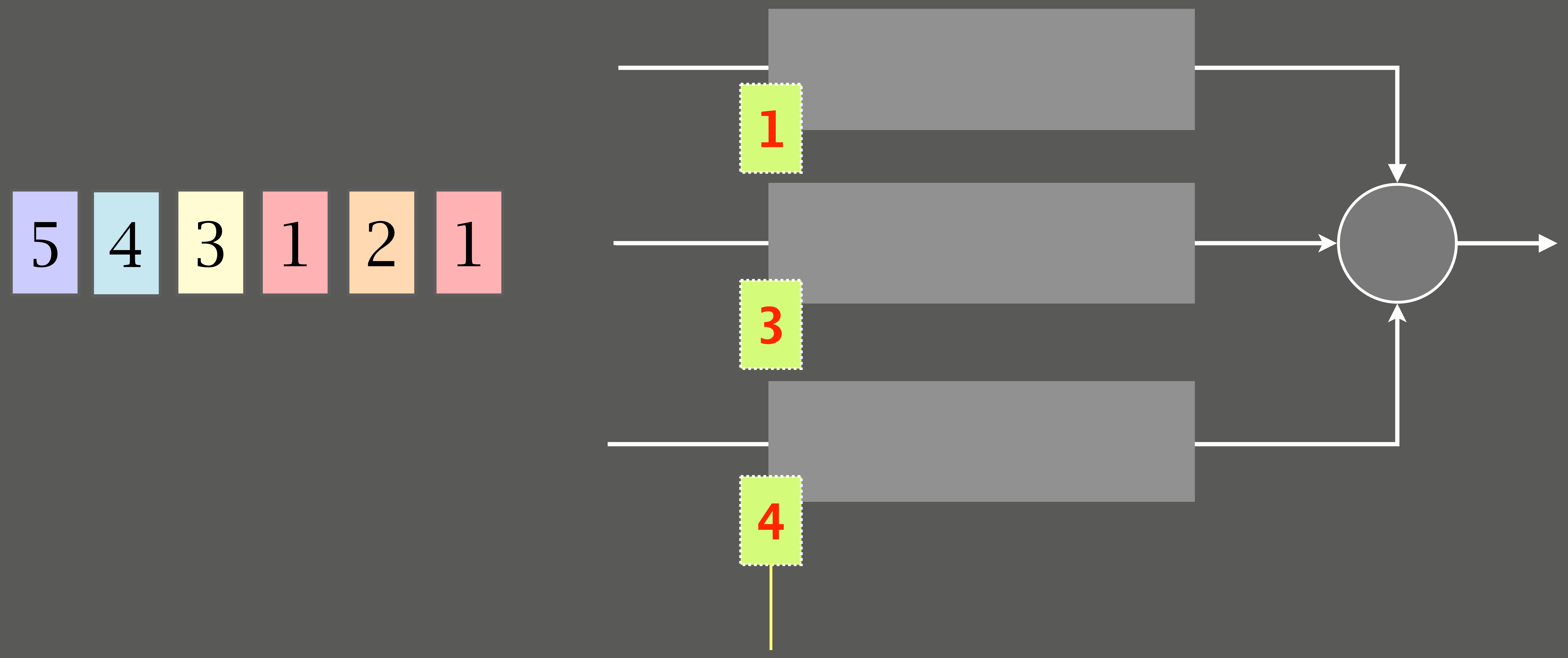
We can minimize the number of scheduling errors

by dynamically adapting the mapping policy



We can minimize the number of scheduling errors

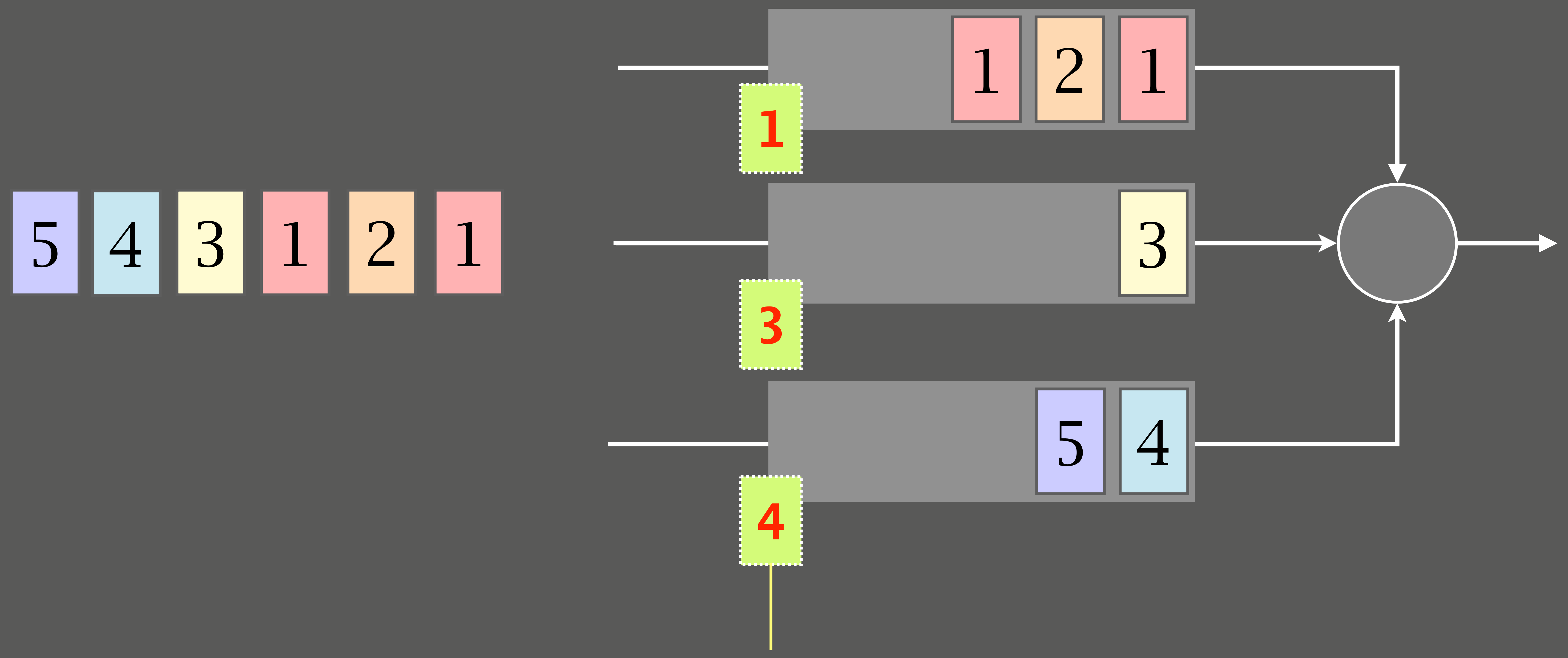
by dynamically adapting the mapping policy



mapping policy q* = [1,3,4]

We can minimize the number of scheduling errors

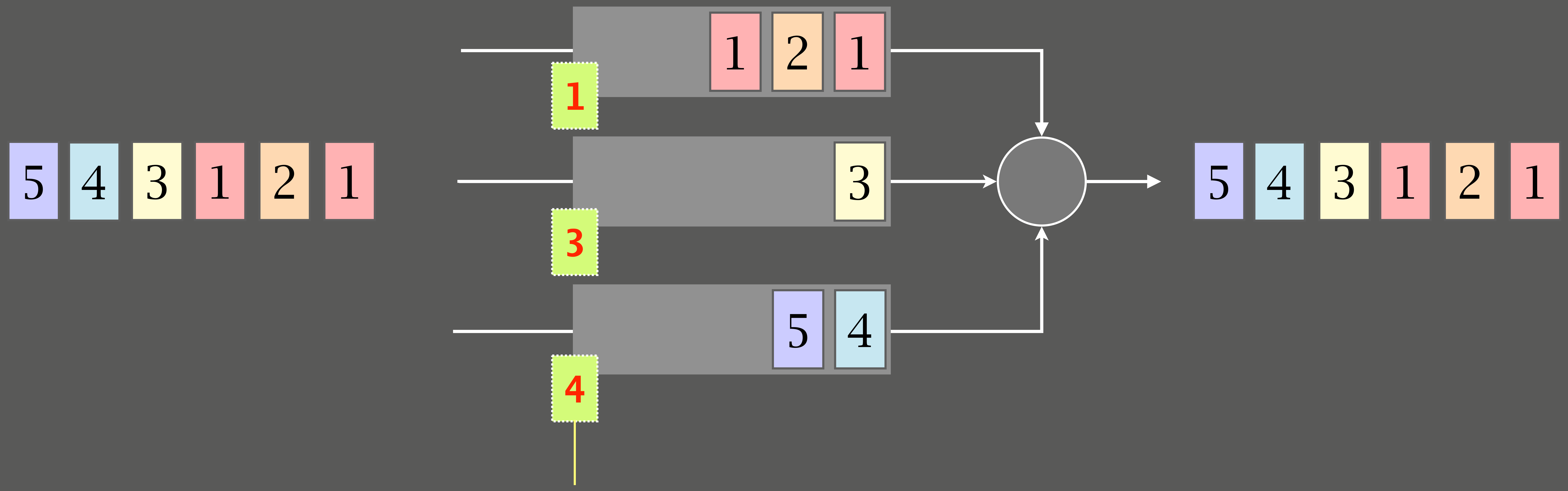
by dynamically adapting the mapping policy



mapping policy q* = [1,3,4]

We can minimize the number of scheduling errors

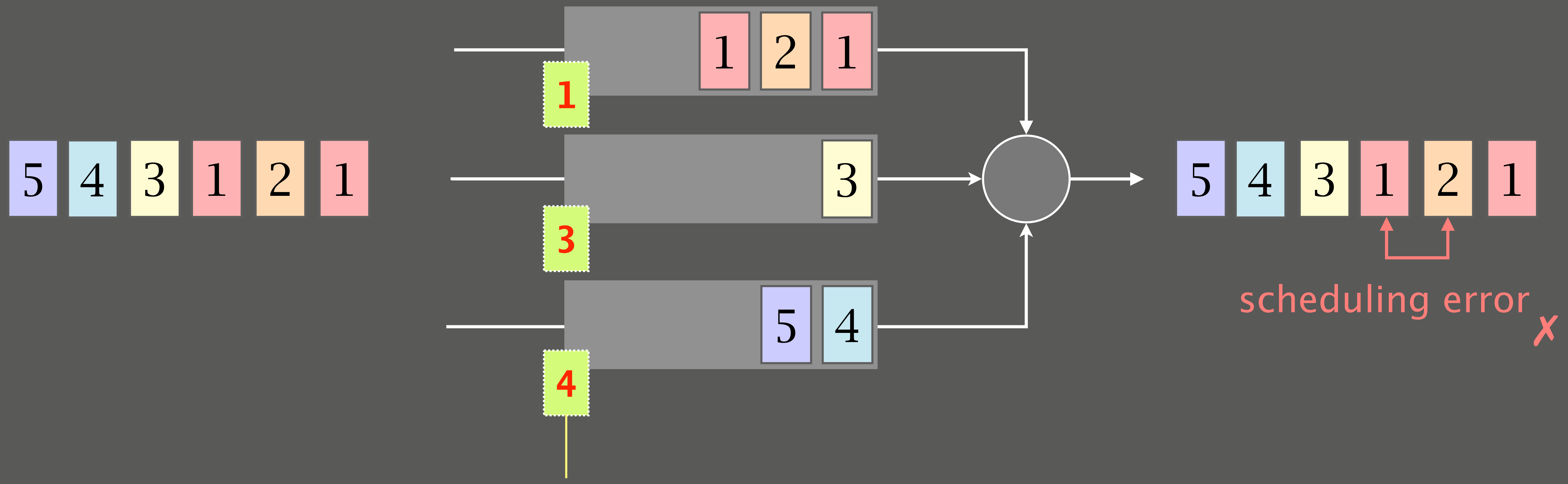
by dynamically adapting the mapping policy



mapping policy q* = [1,3,4]

We can minimize the number of scheduling errors

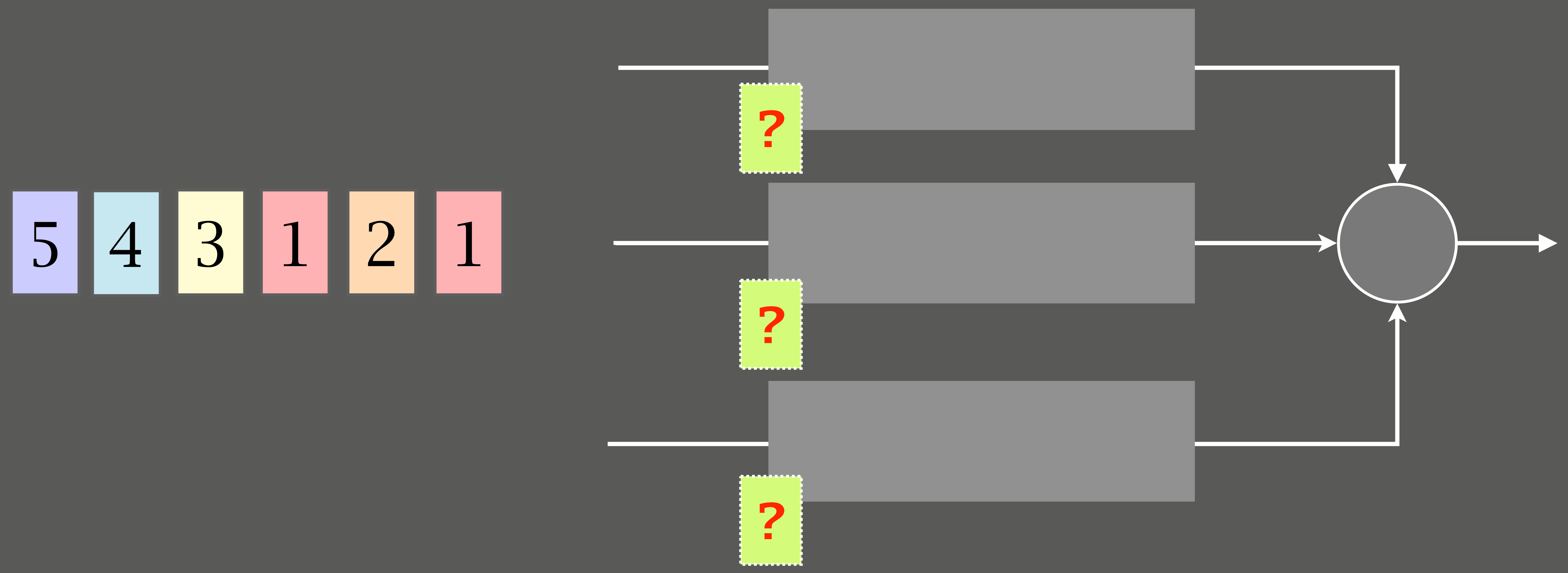
by dynamically adapting the mapping policy



mapping policy q* = [1,3,4]

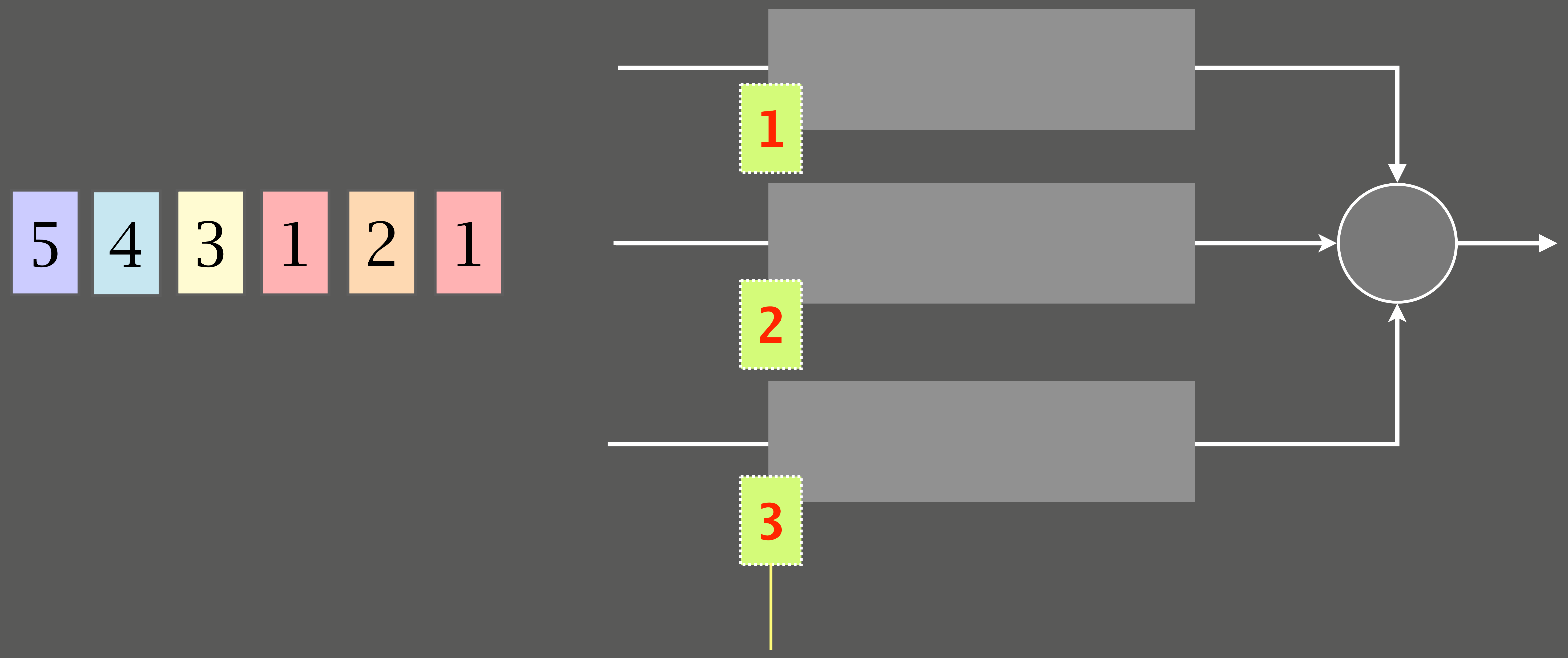
We can minimize the number of scheduling errors

by dynamically adapting the mapping policy



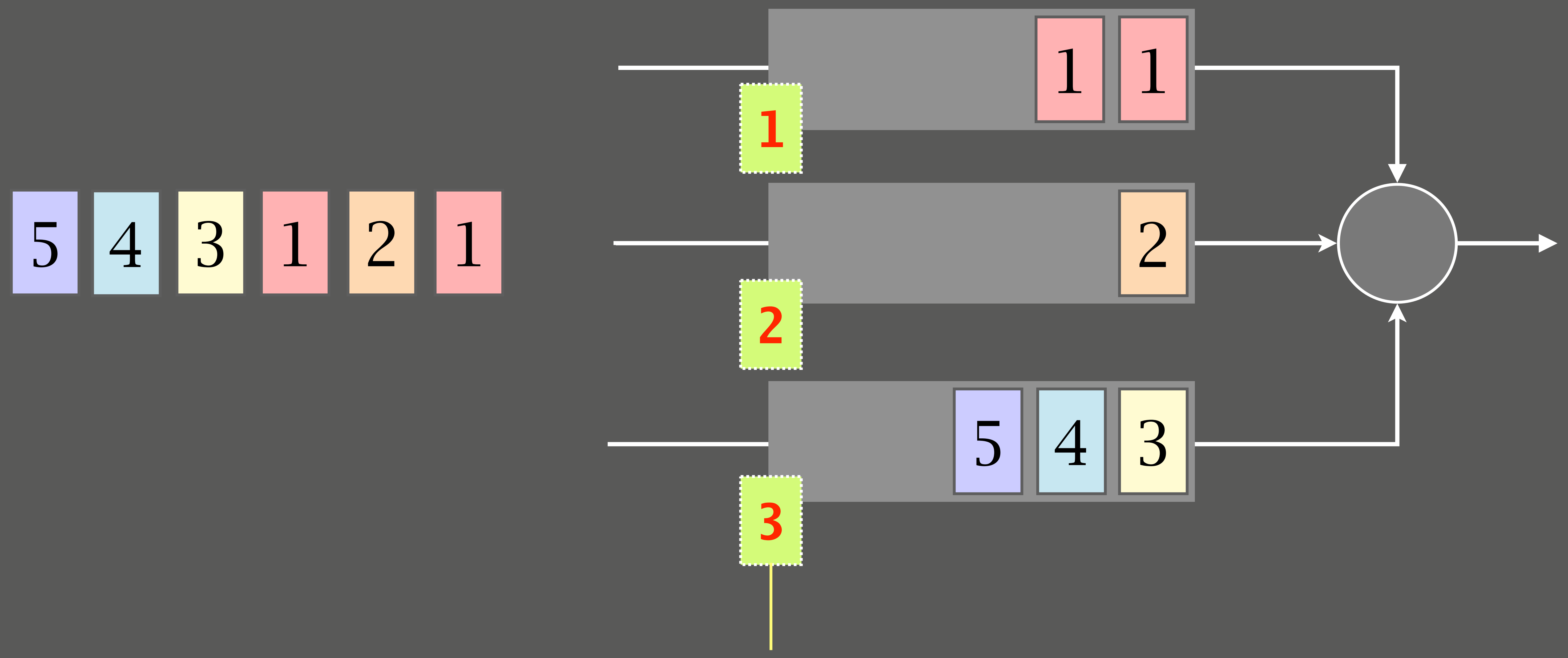
We can minimize the number of scheduling errors

by dynamically adapting the mapping policy



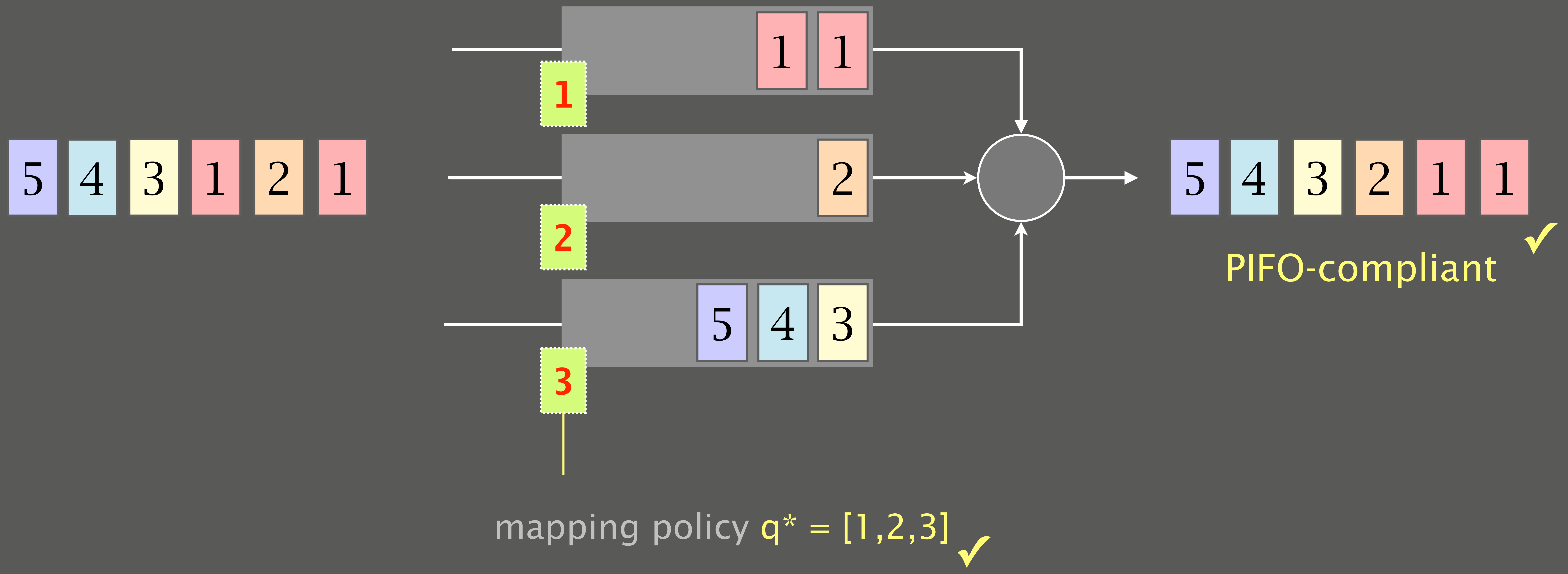
mapping policy q* = [1,2,3]

We can minimize the number of scheduling errors by dynamically adapting the mapping policy



mapping policy q* = [1,2,3]

We can minimize the number of scheduling errors by dynamically adapting the mapping policy



SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

1 Adaptation strategy

how does it work?

2 Implementation

how can it be deployed?

3 Evaluation

how well does it perform?

SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

1 Adaptation strategy

how does it work?

2 Implementation

how can it be deployed?

3 Evaluation

how well does it perform?

Finding an optimal mapping policy is an optimization problem

$$\underbrace{\boldsymbol{q}^*}_{\substack{\text{optimal}\\ \text{mapping policy}}} = \underset{\boldsymbol{q}\in\mathcal{Q}}{\arg\min} \; \underbrace{\mathop{\mathbb{E}}_{r\sim\mathcal{R}} [\; \mathcal{U}(\boldsymbol{q}, r) \;]}_{\substack{\text{expected loss across all ranks}\\ \text{"unpifoness"}}}$$

Solving this optimization problem exactly is intractable unfortunately

unknown packet rank distributions

$$q^* = \arg\min_{q \in \mathcal{Q}} \mathop{\mathbb{E}}_{r \sim \mathcal{R}} [\, \mathcal{U}(q, r) \,]$$

optimal mapping policy

expected loss across all ranks
"unpifoness"

We can approximate the solution by turning the problem into an online empirical risk minimization problem

We can approximate the solution by turning the problem into an online empirical risk minimization problem

$$\underset{\text{online mapping policy}}{q^*} = \arg\min_{q\in\mathcal{Q}} \underset{\text{estimated unpifoness}}{\mathcal{U}(\overset{\text{enqueued packets}}{\mathcal{P}}, q)}$$

SP-PIFO dynamically adapts the mapping policy on a per-packet basis, in two phases

SP-PIFO dynamically adapts the mapping policy on a per-packet basis, in two phases

phase 1
push-up

gradually map higher-priority packets to higher-priority queues

concentrates scheduling errors in the highest-priority queue

SP-PIFO dynamically adapts the mapping policy on a per-packet basis, in two phases

phase 1
push-up

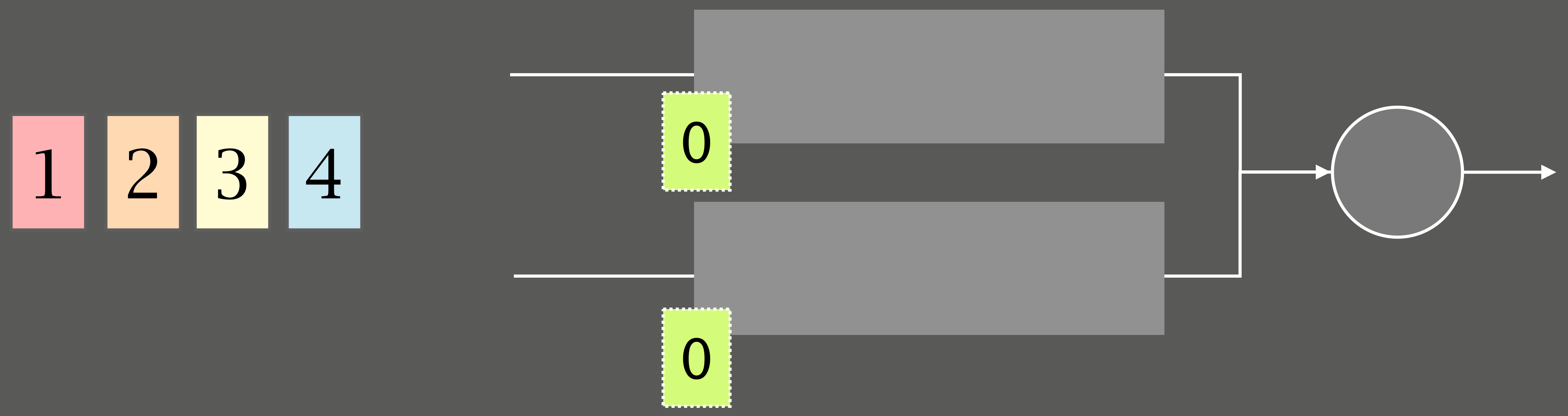
gradually map higher-priority packets
to higher-priority queues

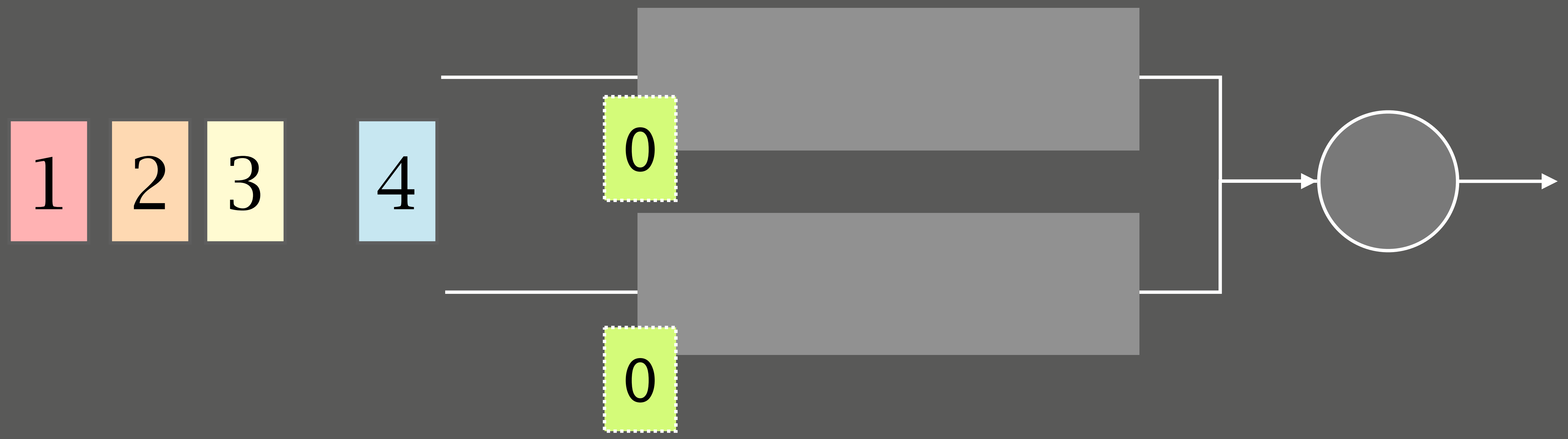
concentrates scheduling errors
in the highest-priority queue

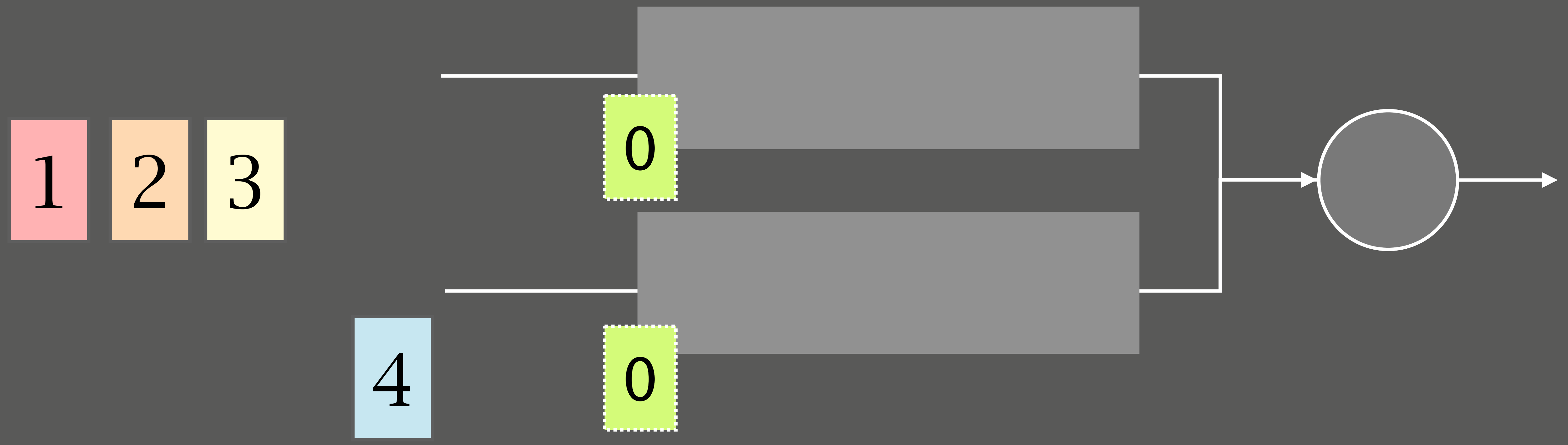
upon scheduling error…

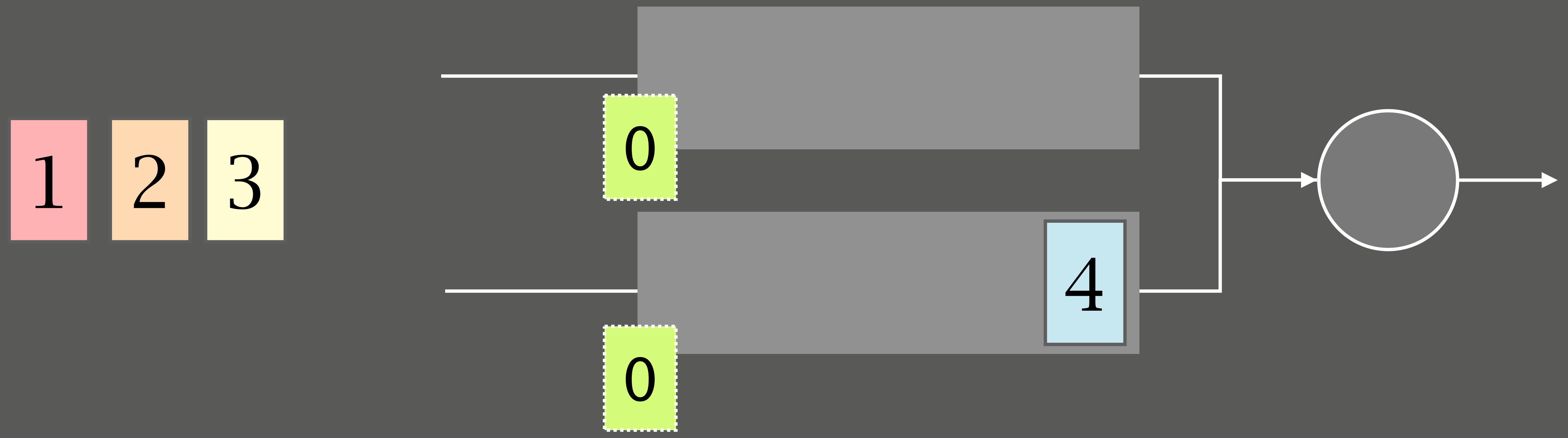
phase 2
push-down

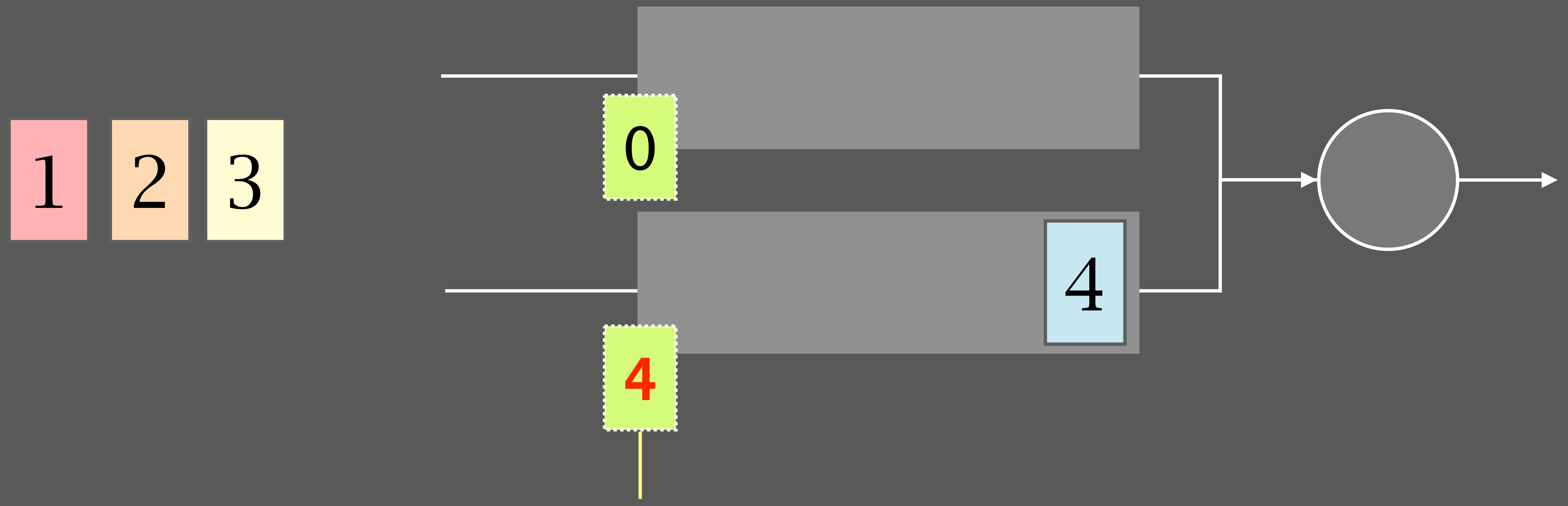
shift lower-priority packets
to lower-priority queues


1
2
3
4
0
0

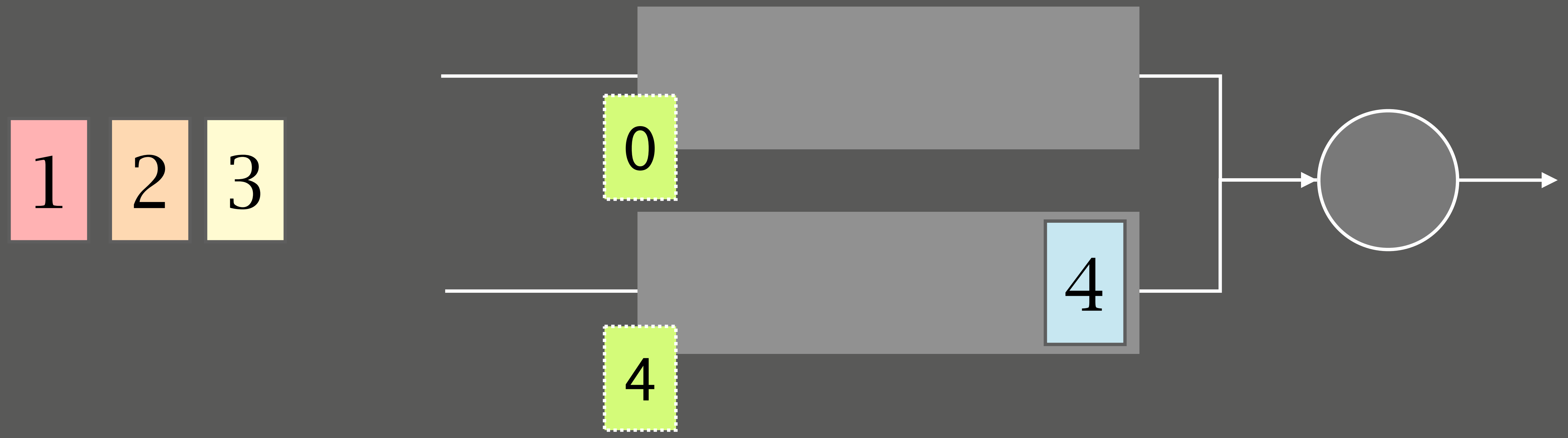

1
2
3
4
0
0

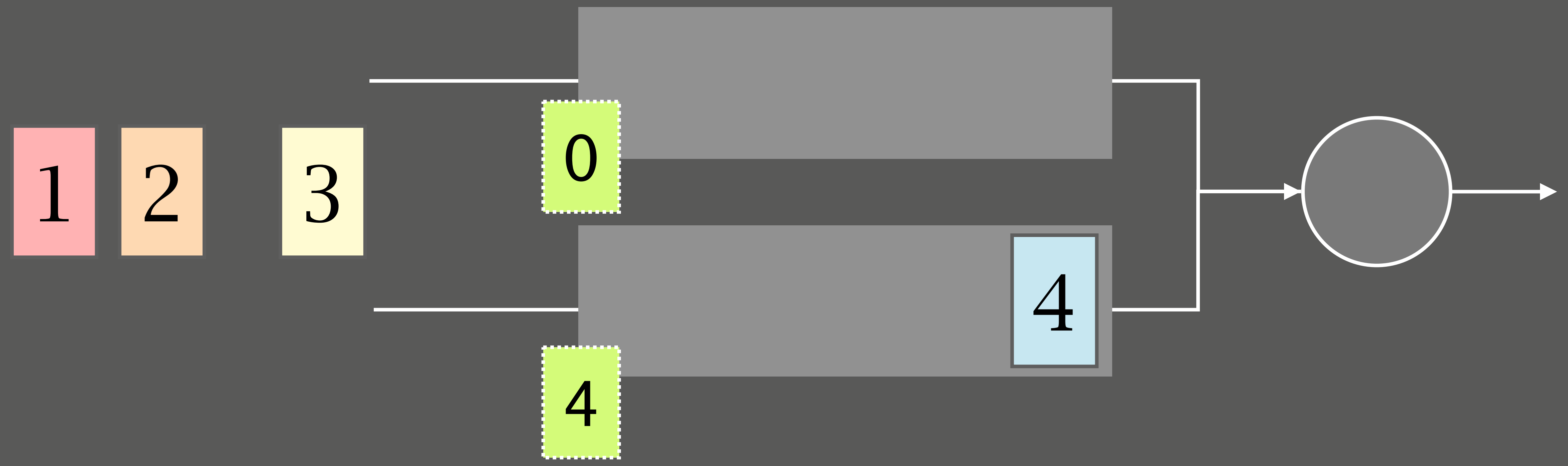

1
2
3
0
4
0

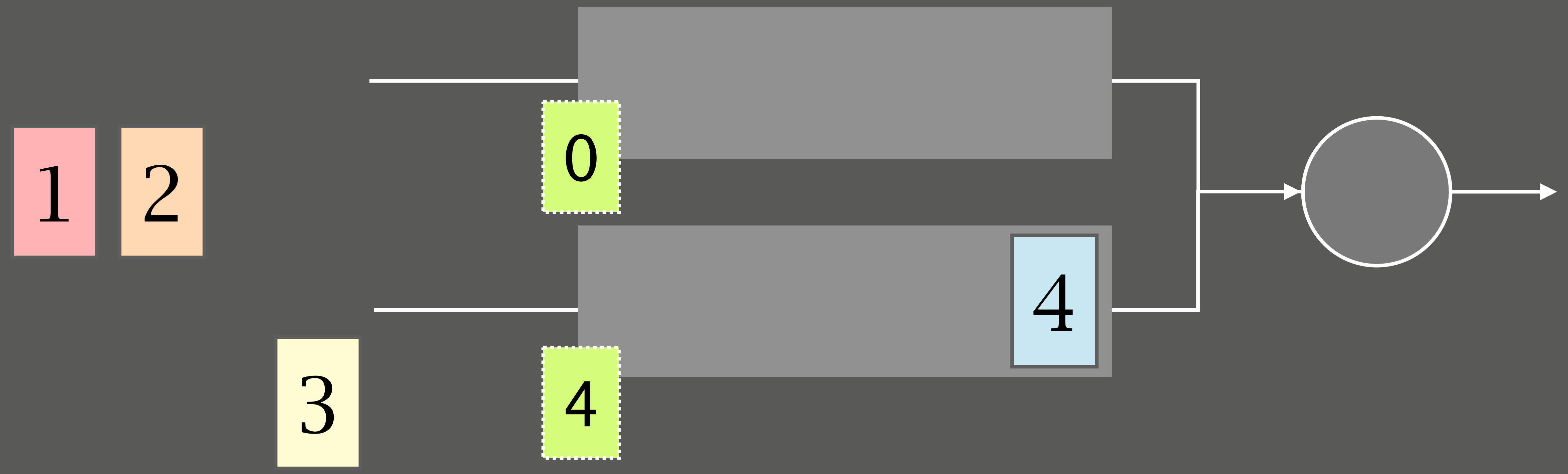

1
2
3
0
0
4

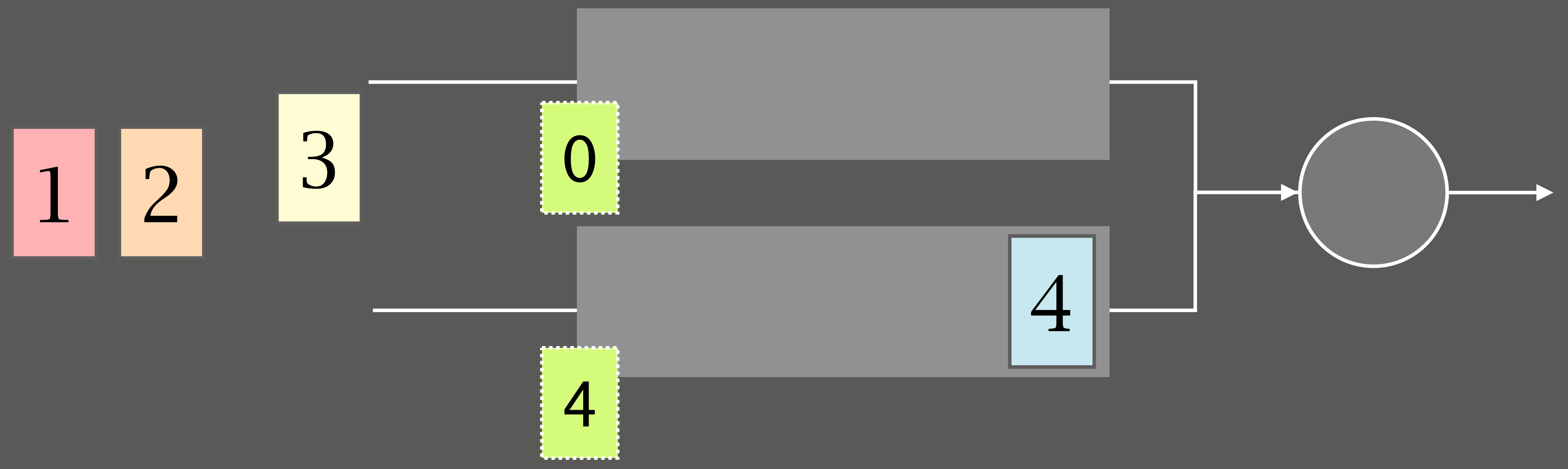


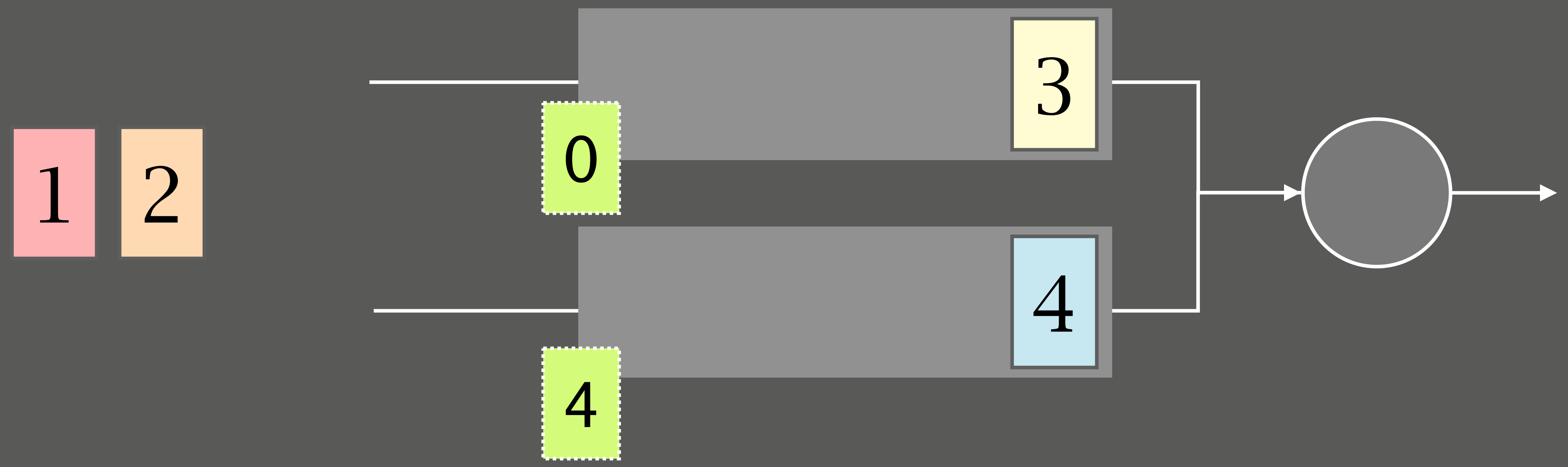
"push-up" increase queue$_{idx}$ to rank(enqueued packet)

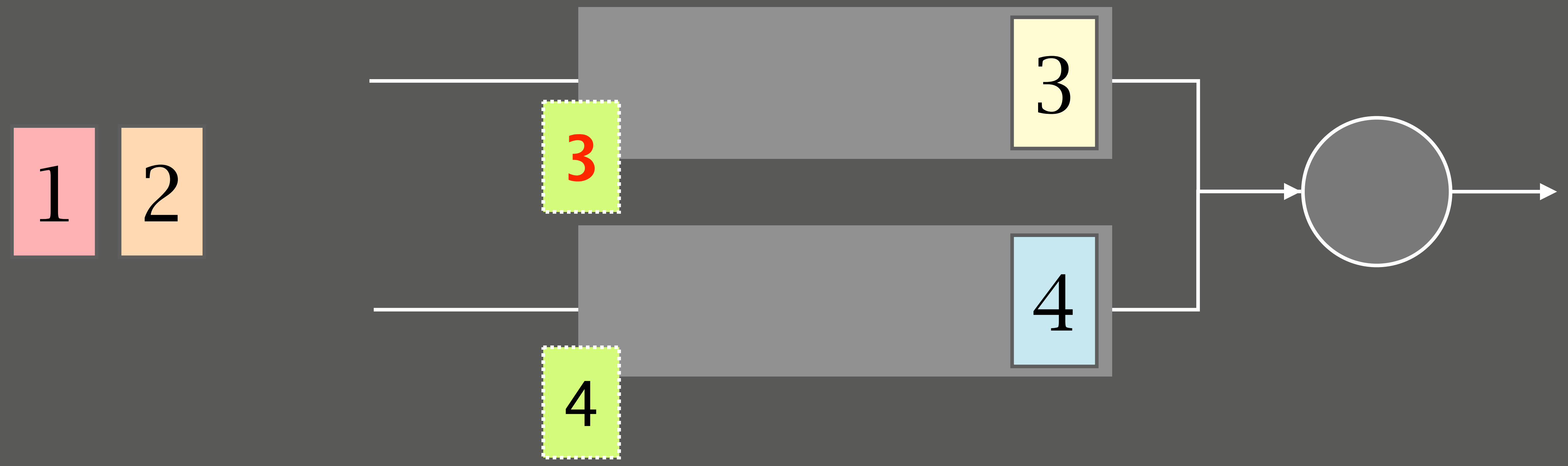

1
2
3
0
4
4

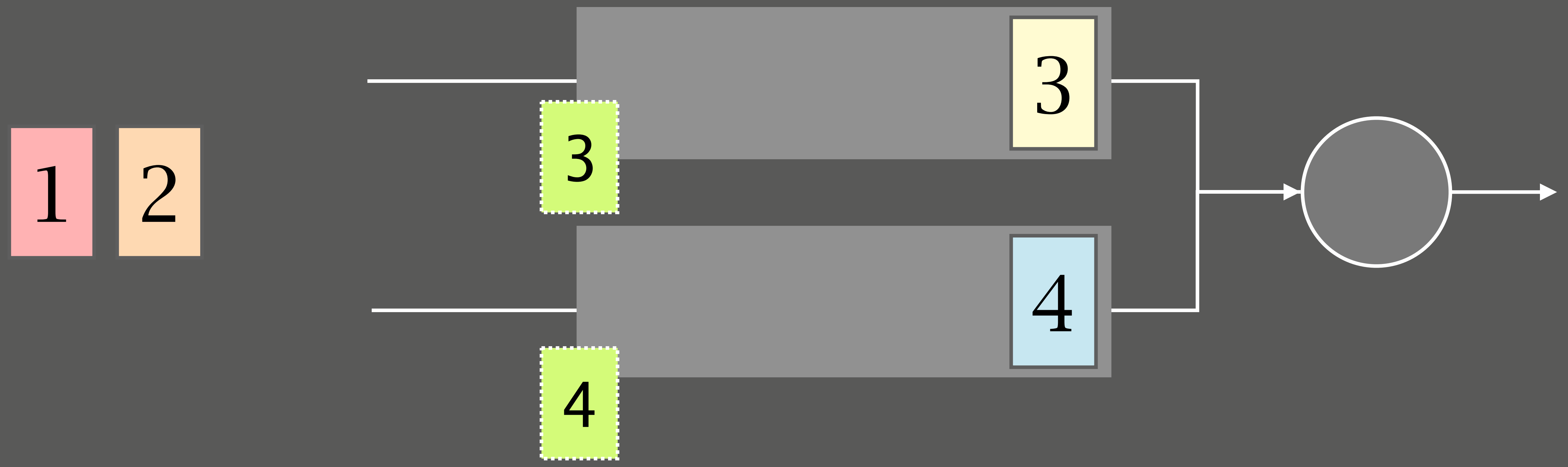

1
2
3
0
4
4

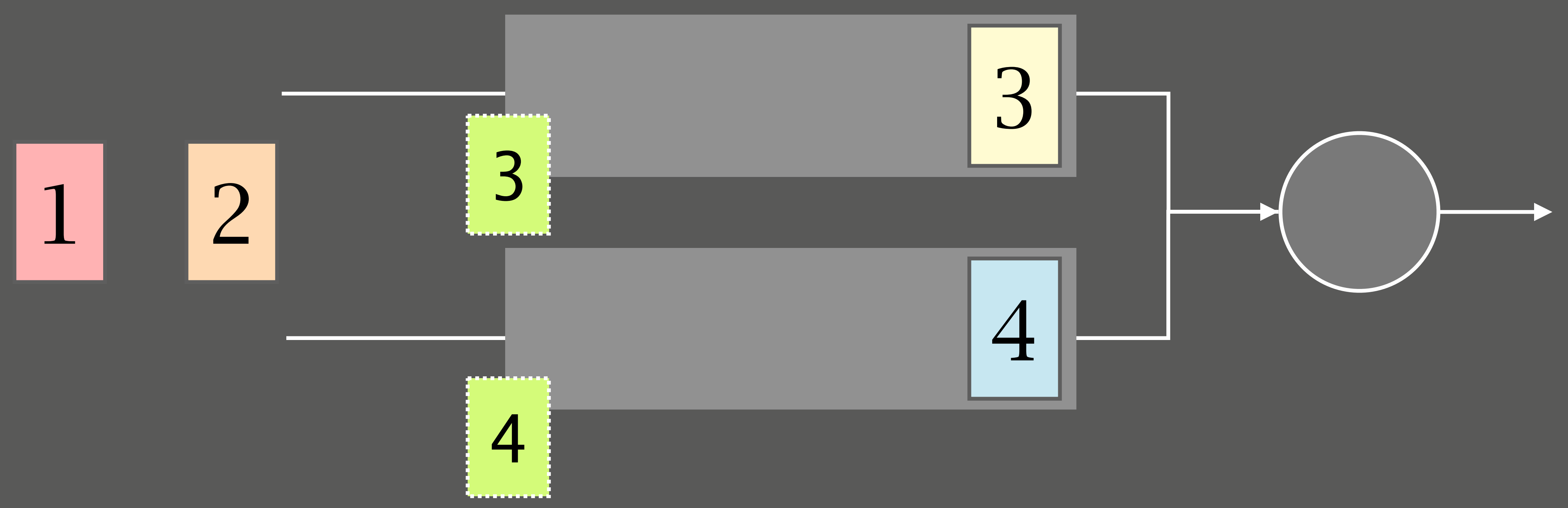

1
2
3
0
4
4

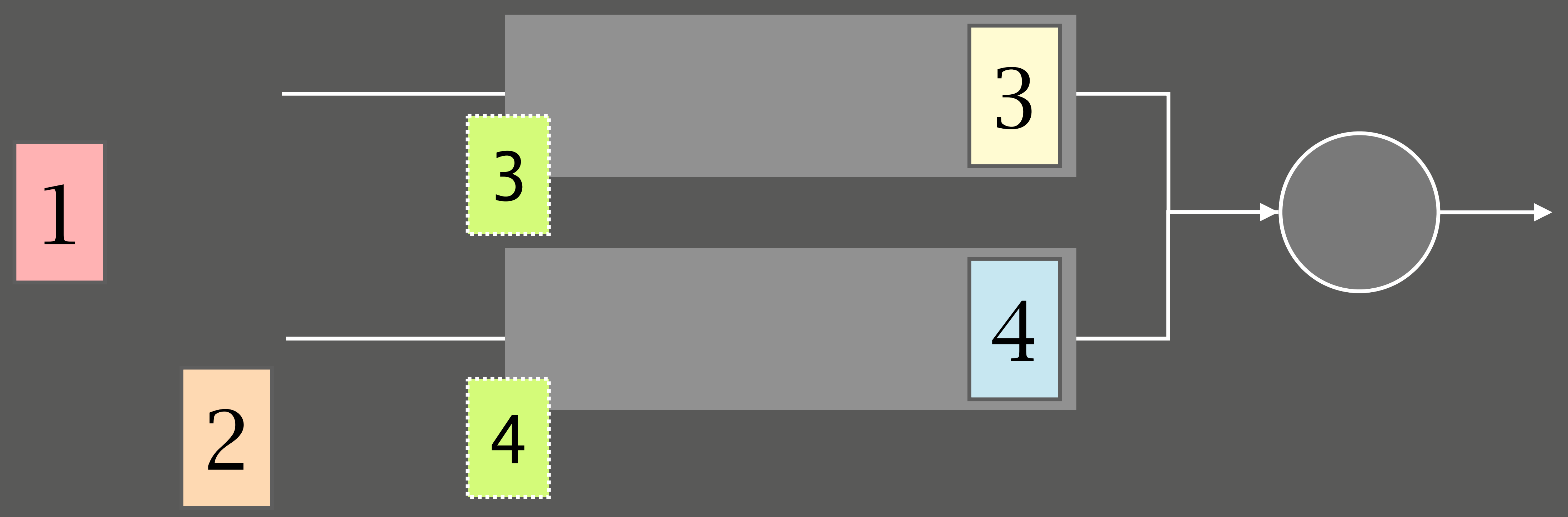

1
2
3
0
4
4

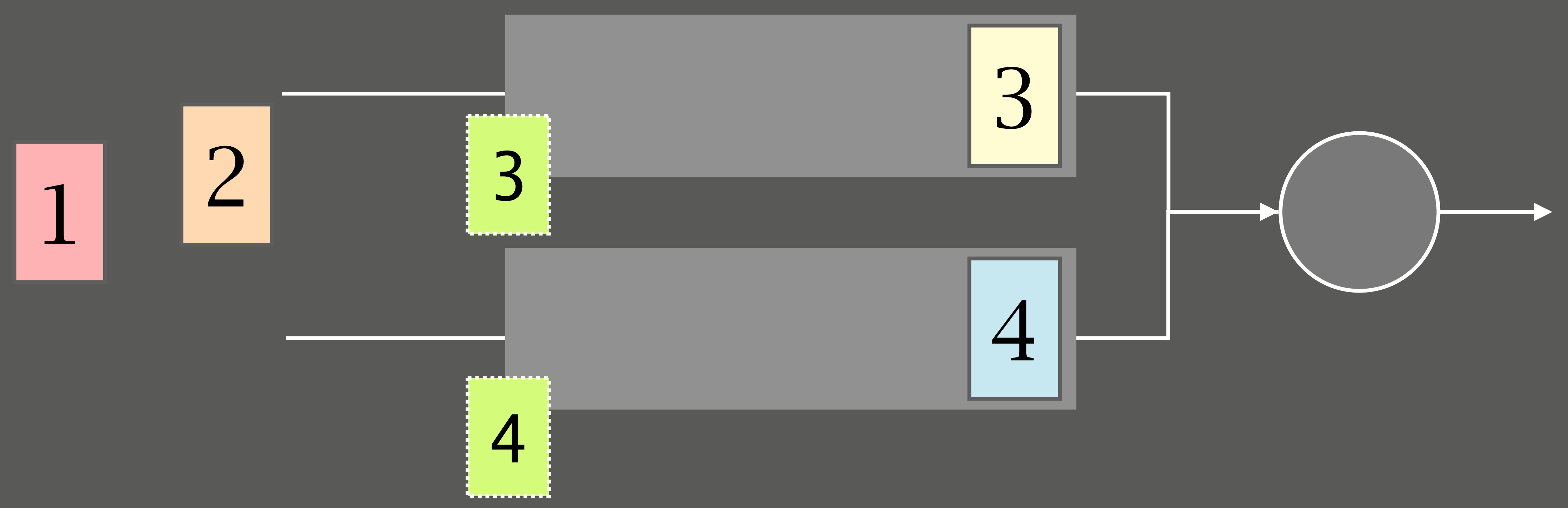

1
2
0
3
4
4


1
2
3
3
4
4

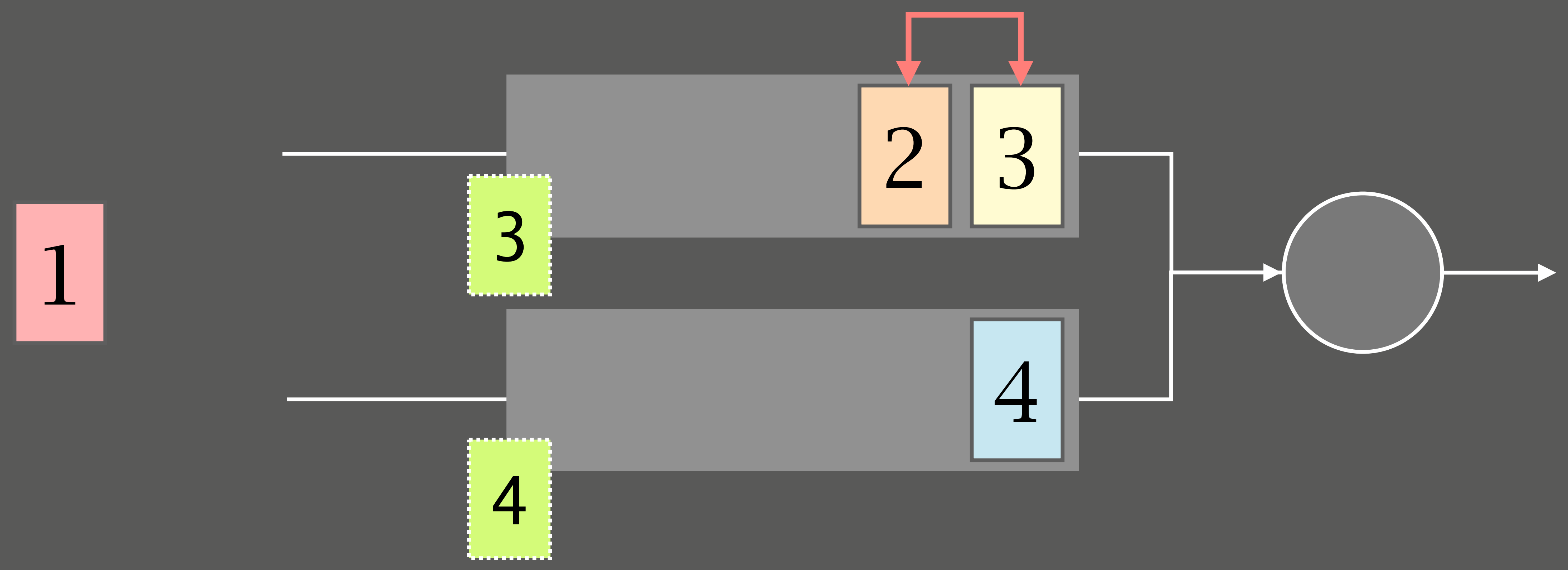

1
2
3
3
4
4

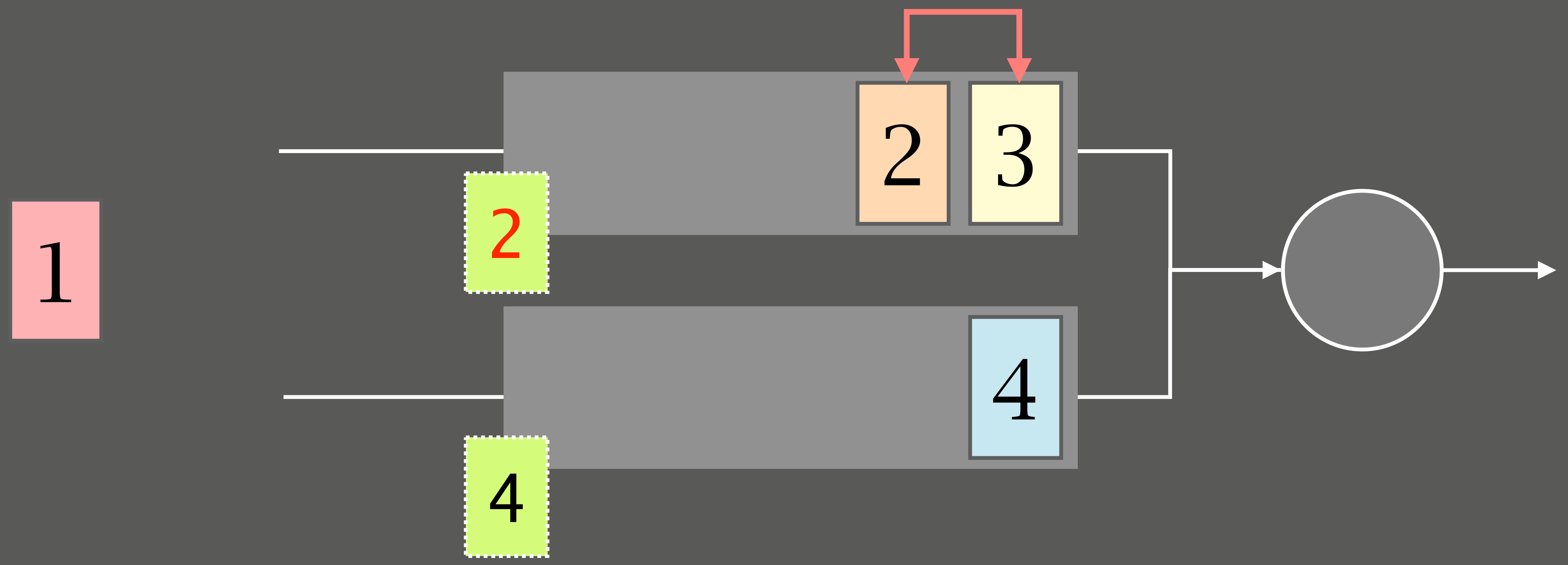

1
2
3
3
4
4


1
2
3
4
3
4


1
2
3
4
3
4

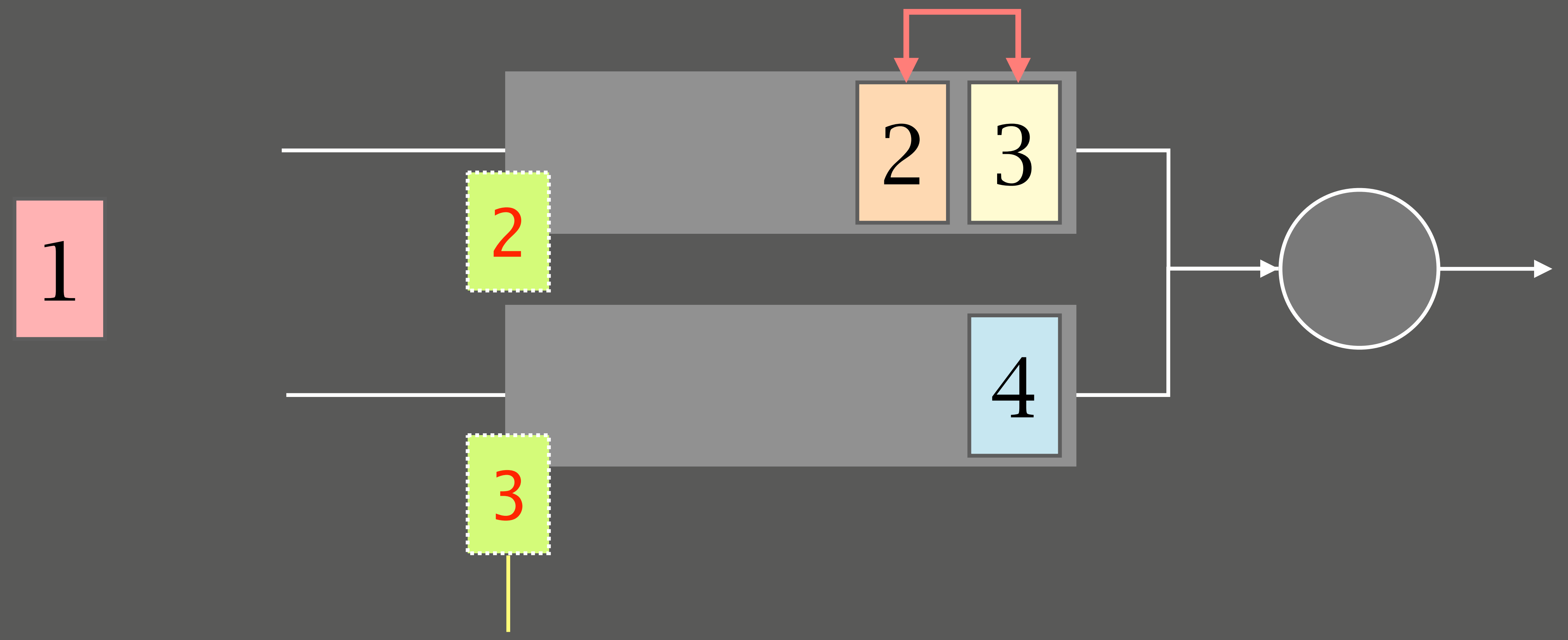
scheduling error of cost 3-2=1





scheduling error of cost 3-2=1

scheduling error of cost 3-2=1



"push-down"

decrease all queue$_{idx}$ by cost

SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

Adaptation strategy

how does it work?

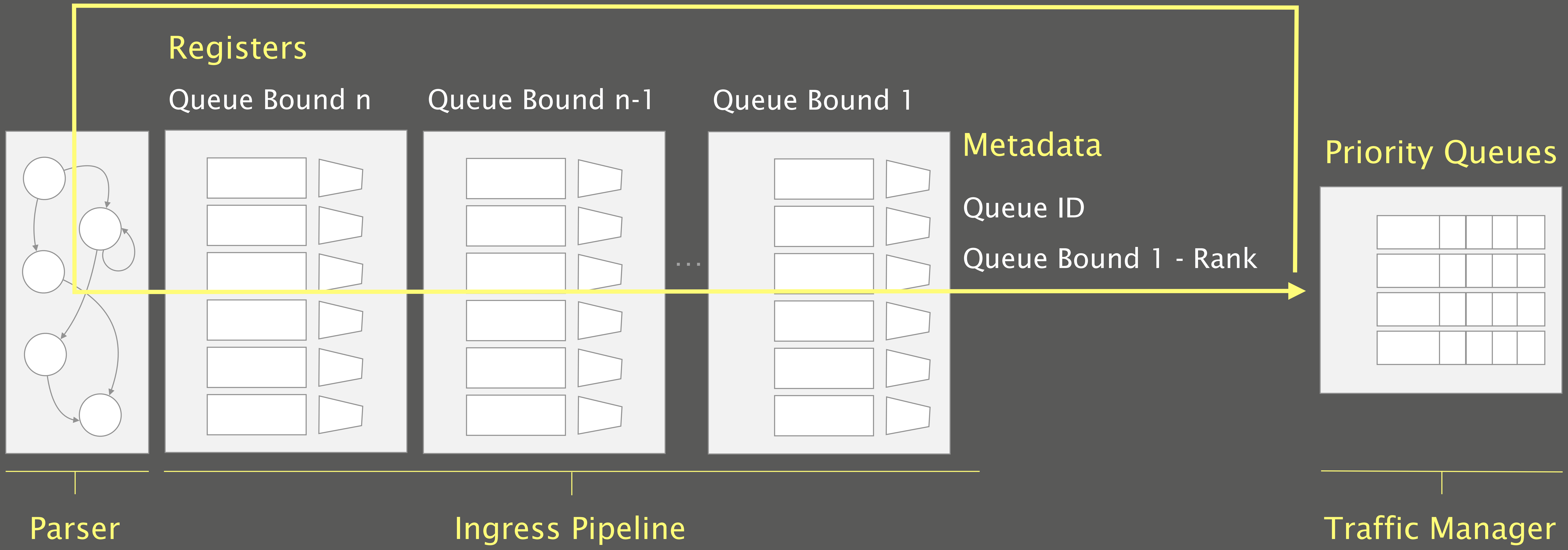
2 Implementation

how can it be deployed?

Evaluation

how well does it perform?

We managed to program SP-PIFO on existing programmable data planes (Barefoot Tofino)



SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

Adaptation strategy

how does it work?

Implementation

how can it be deployed?

3 Evaluation

how well does it perform?

How well can SP-PIFO approximate well-known scheduling objectives?

How well can SP-PIFO approximate well-known scheduling objectives?

Scheduling objectives

Minimize Flow Completion Time

pFabric (8 queues)

Ranks are set to the remaining flow size

Enforce max-min fairness

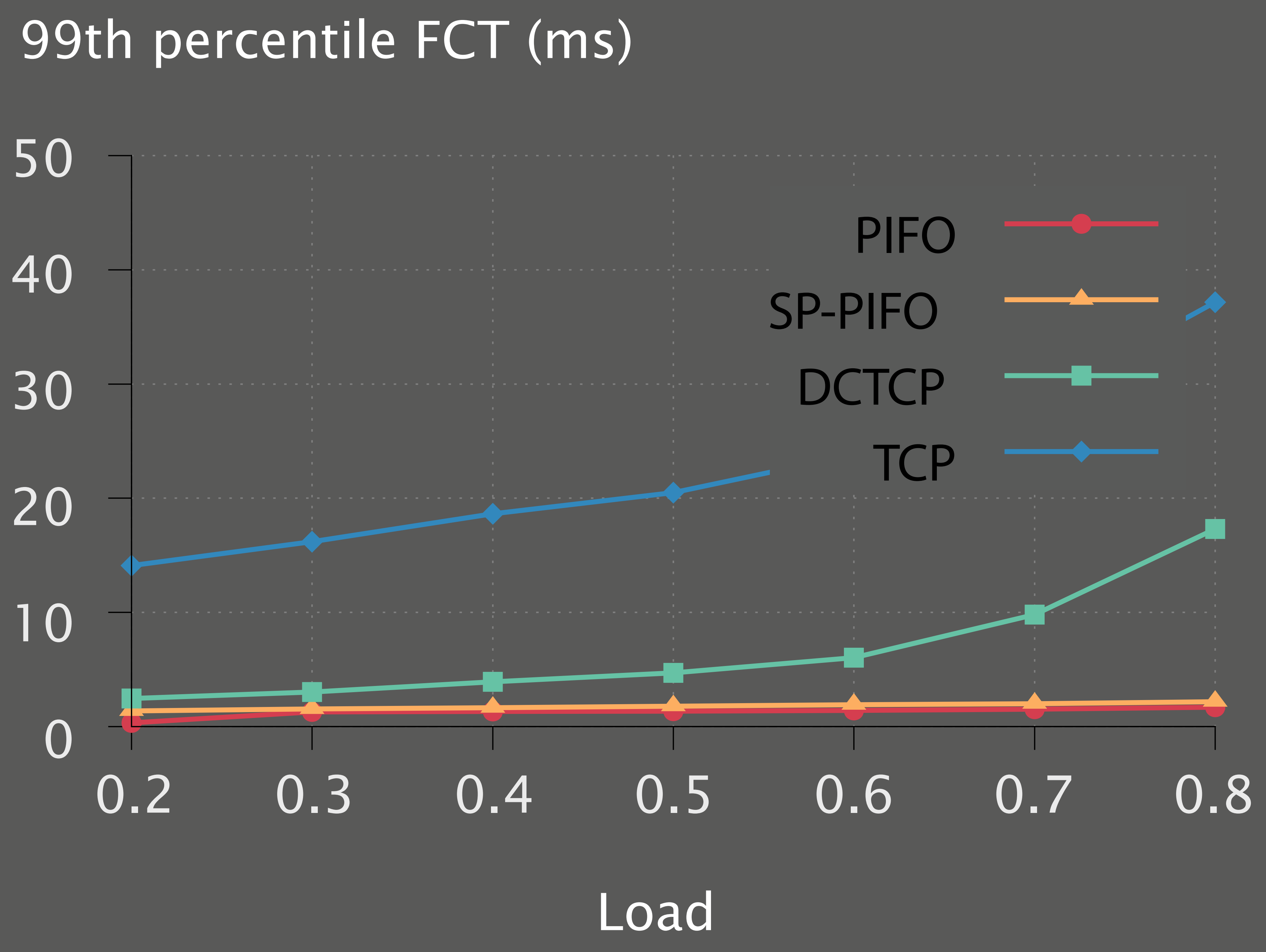
Start-Time Fair Queuing (32 queues)

Ranks based on a fluid model

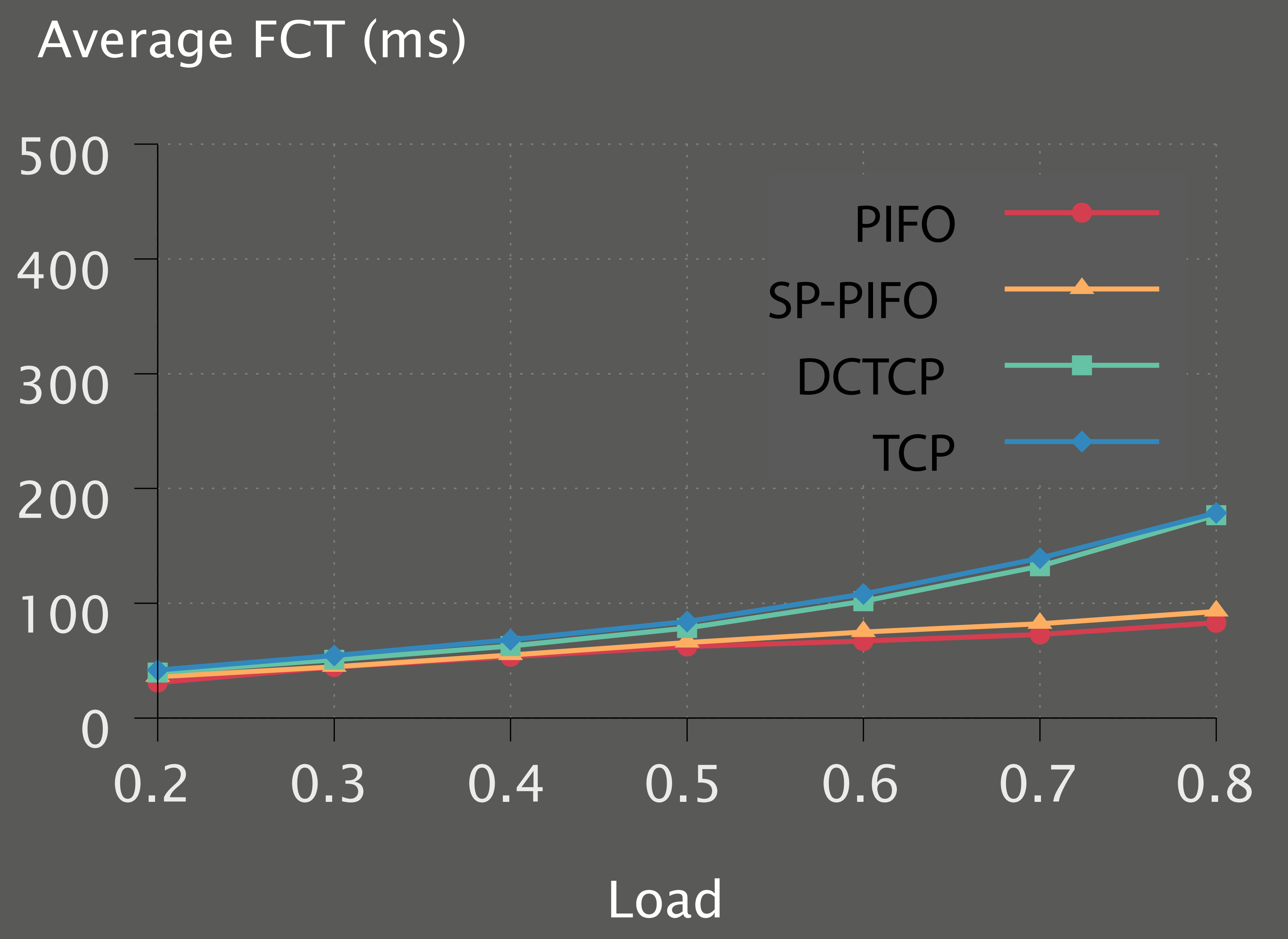
Packet-level simulator	Netbench [SIGCOMM 2017]
Topology	We use a leaf-spine topology with: 144 servers, 1/4 Gbps links
Realistic workloads	pFabric web-search workload

SP-PIFO closely approximates pFabric

minimizing FCTs for both small and big flows

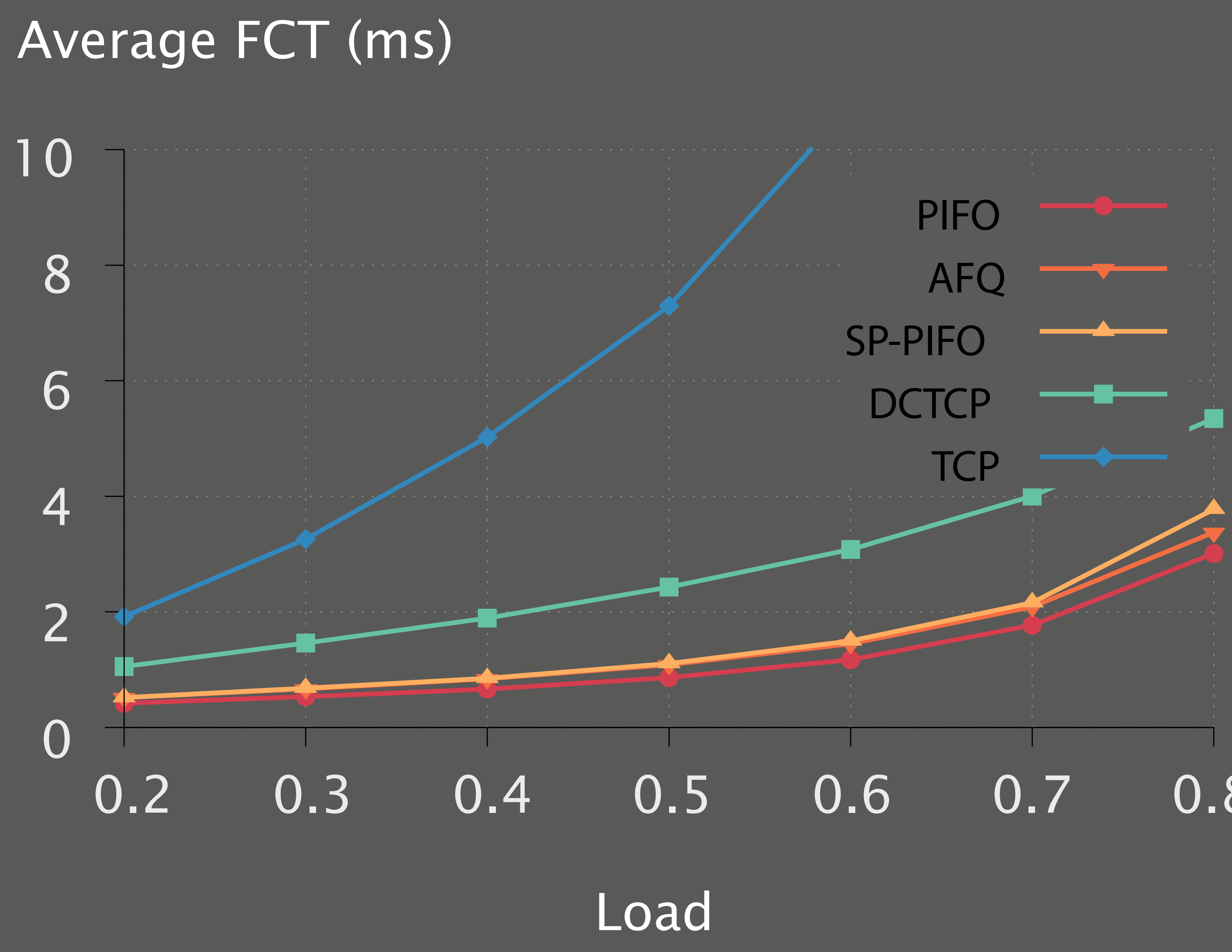


Small flows <100KB

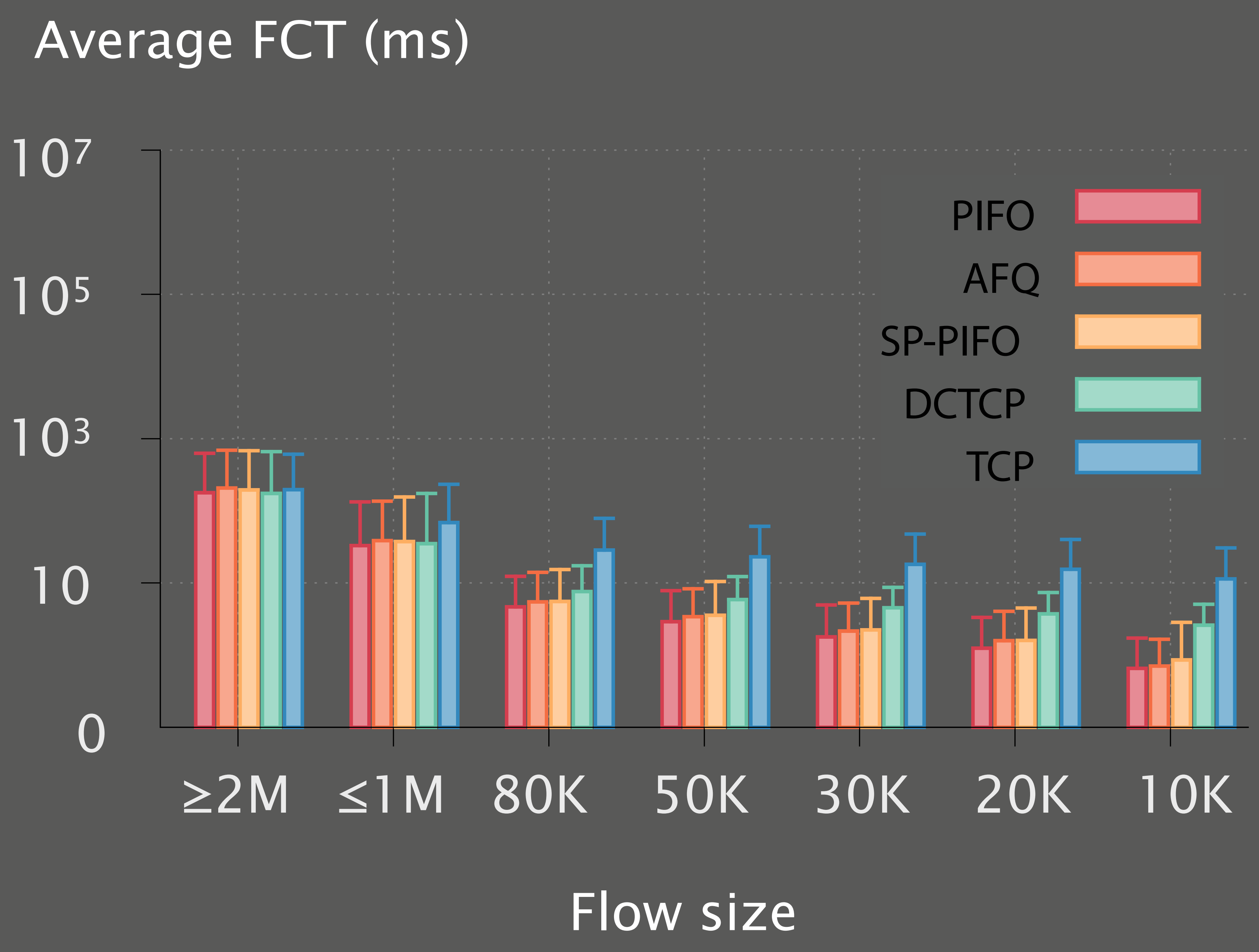


Big flows ≥1MB

SP-PIFO closely approximates fair-queueing algorithms



Small flows <100KB



All flows @ Load 0.7

SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

Adaptation strategy

how does it work?

Implementation

how can it be deployed?

Evaluation

how well does it perform?

SP-PIFO makes packet scheduling programmable… today!

SP-PIFO approximates the behavior of PIFO queues
at line rate, at scale and on existing devices

SP-PIFO dynamically maps packets to queues
so as to minimize scheduling errors

SP-PIFO automatically reacts to traffic variations
without requiring any traffic knowledge

Programmable, hardware-based

Routing and Scheduling

Fast routing

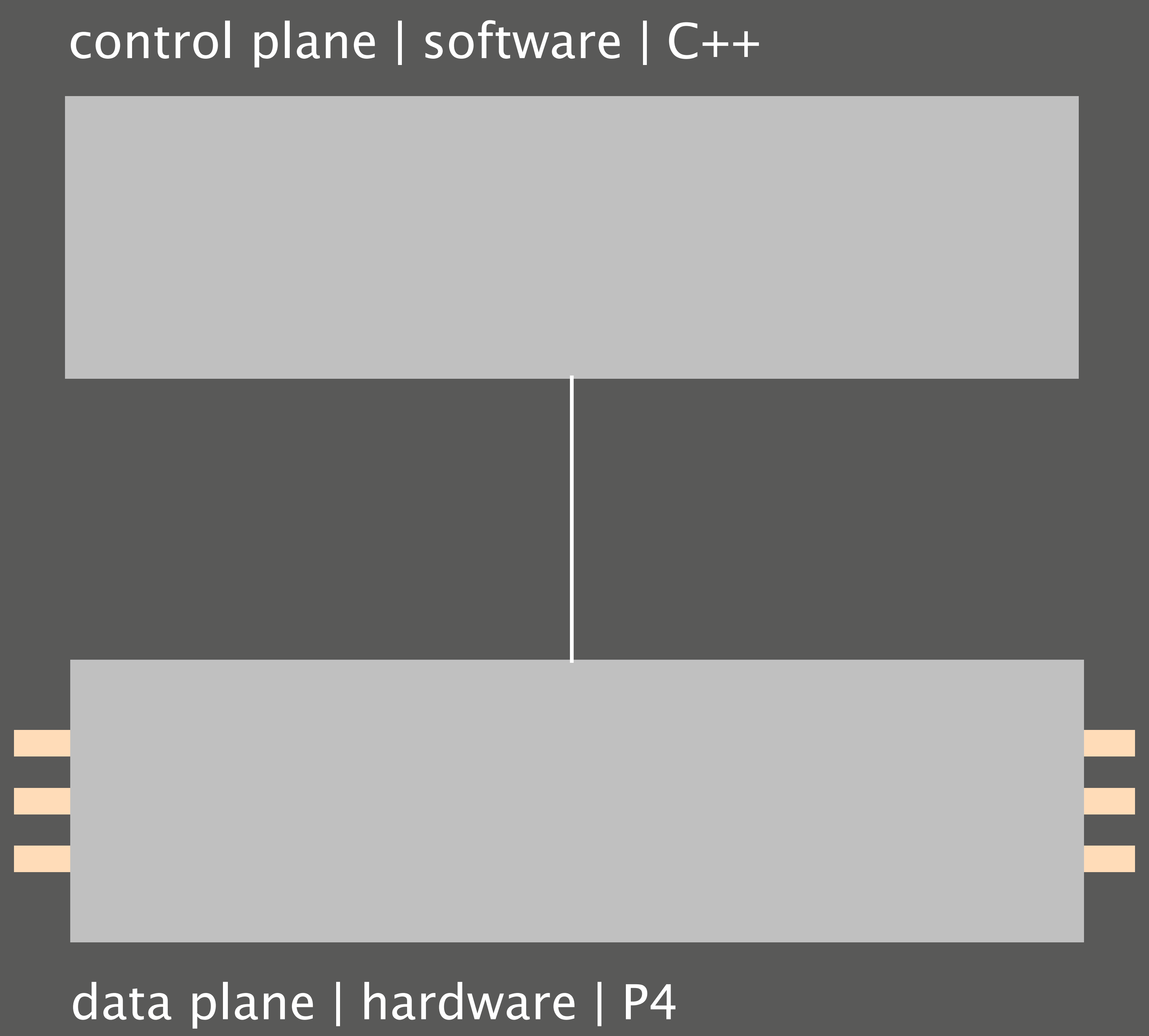
data-plane offloading

Flexible scheduling

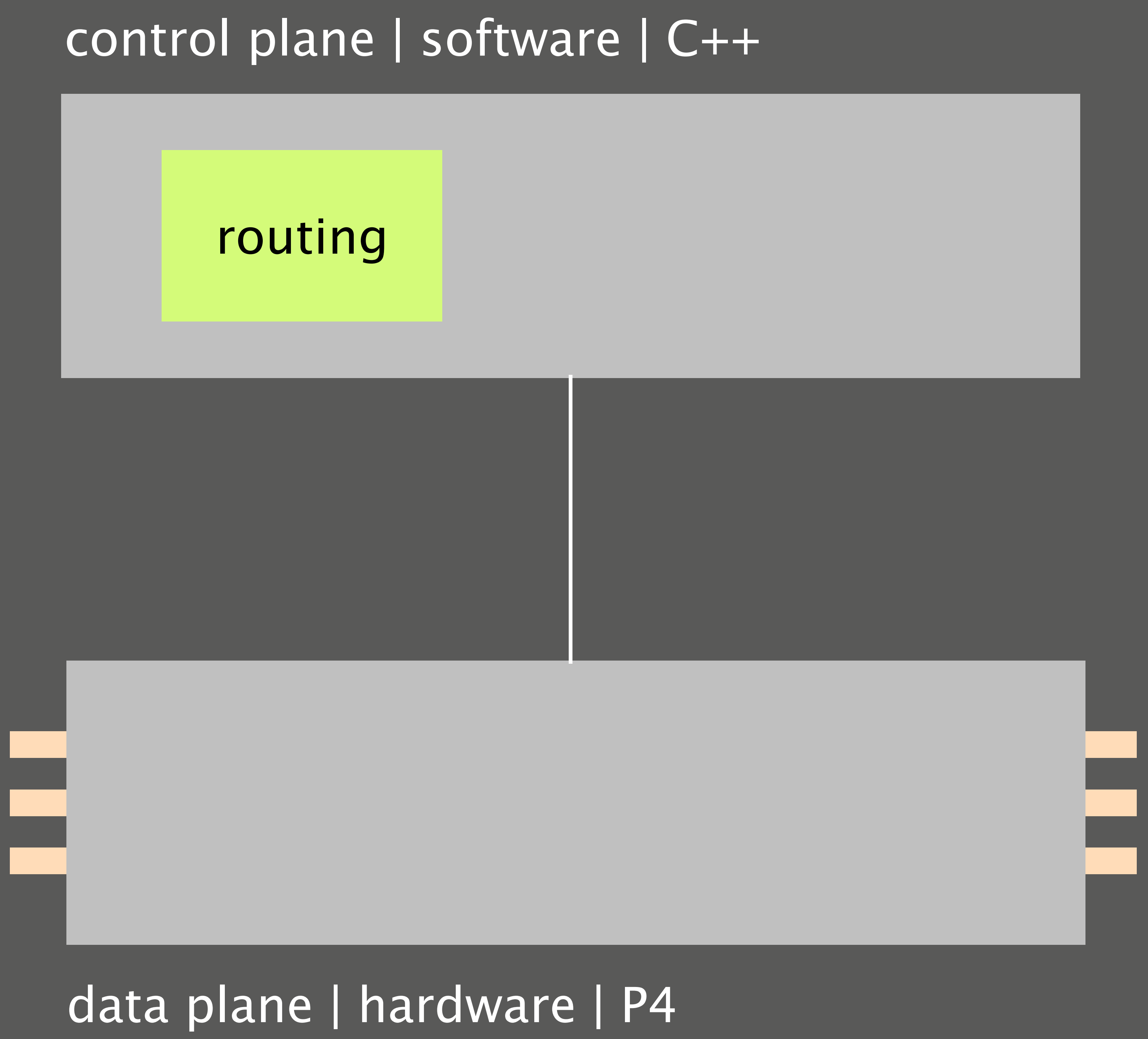
dynamic packet adaptation

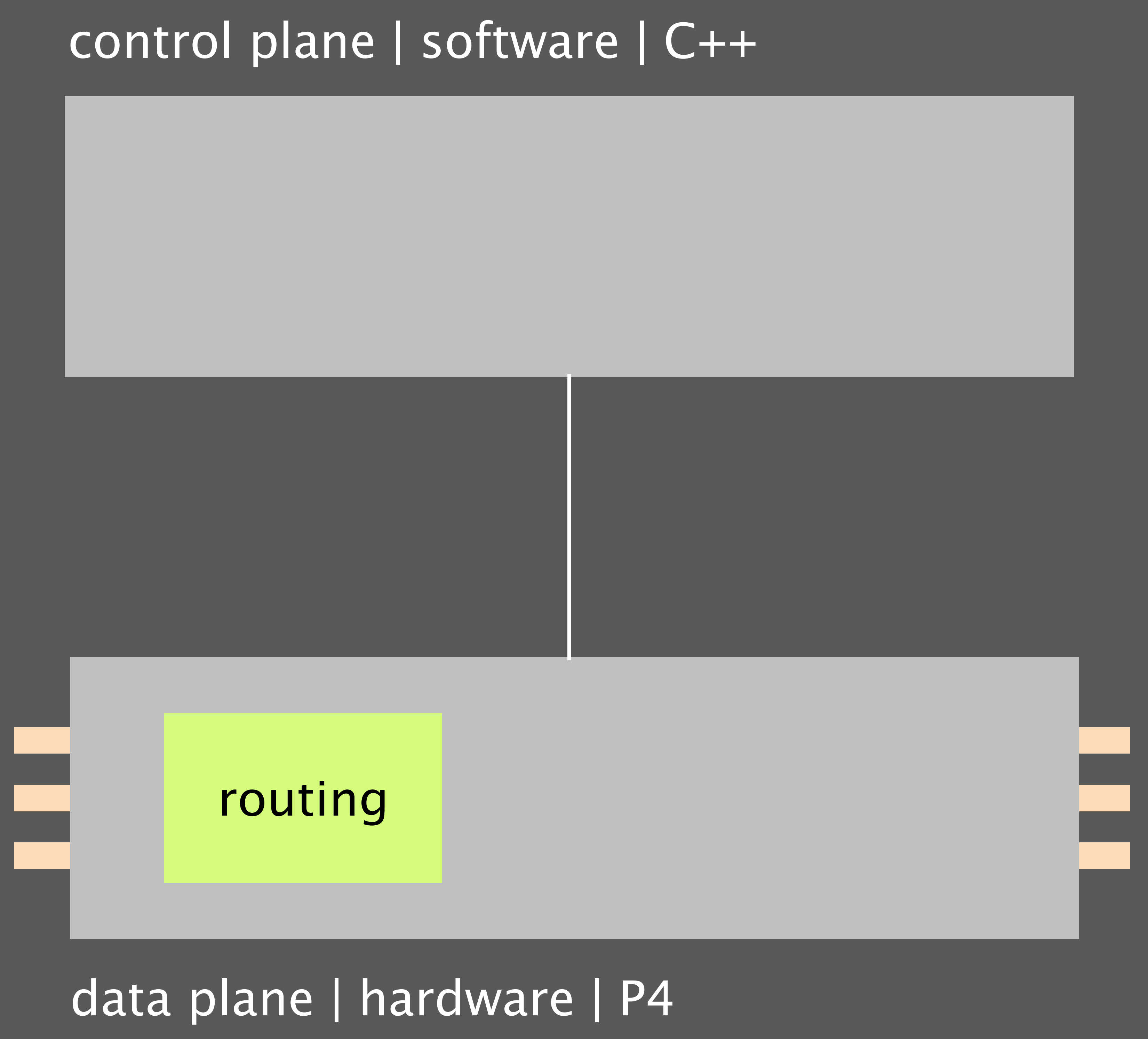
3 Looking forward

routing | forwarding

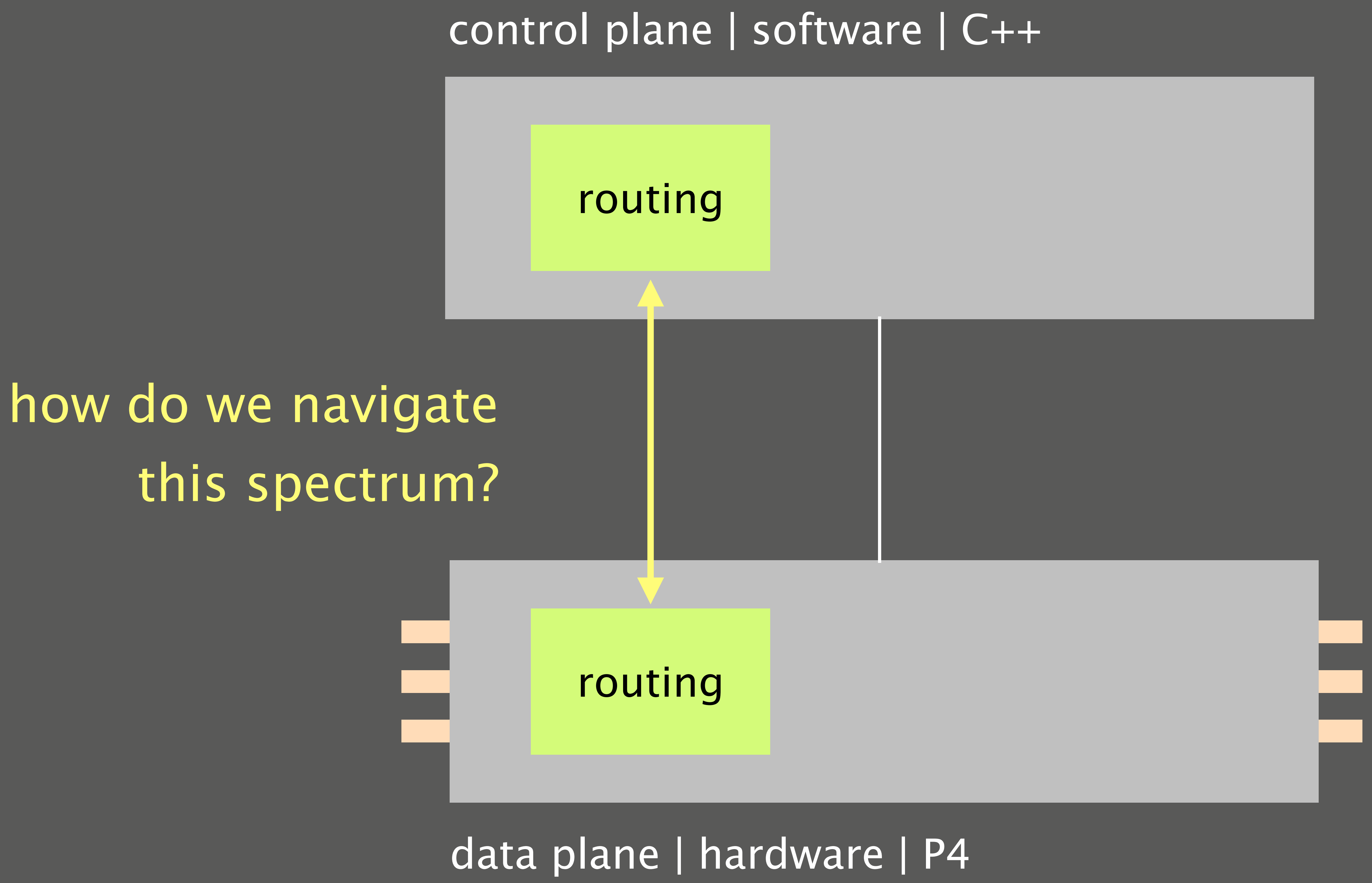

control plane | software | C++
data plane | hardware | P4

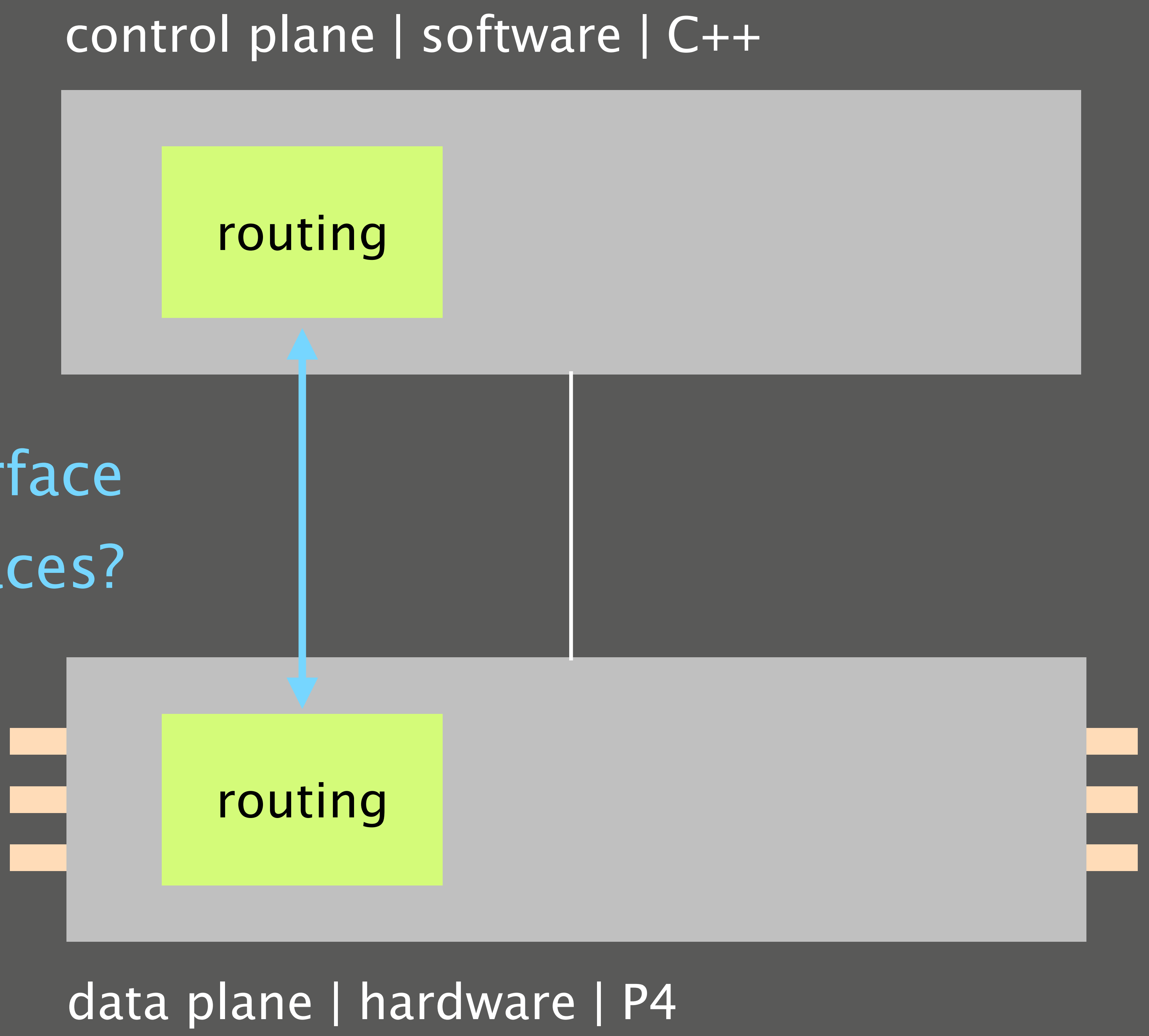
definitely not great





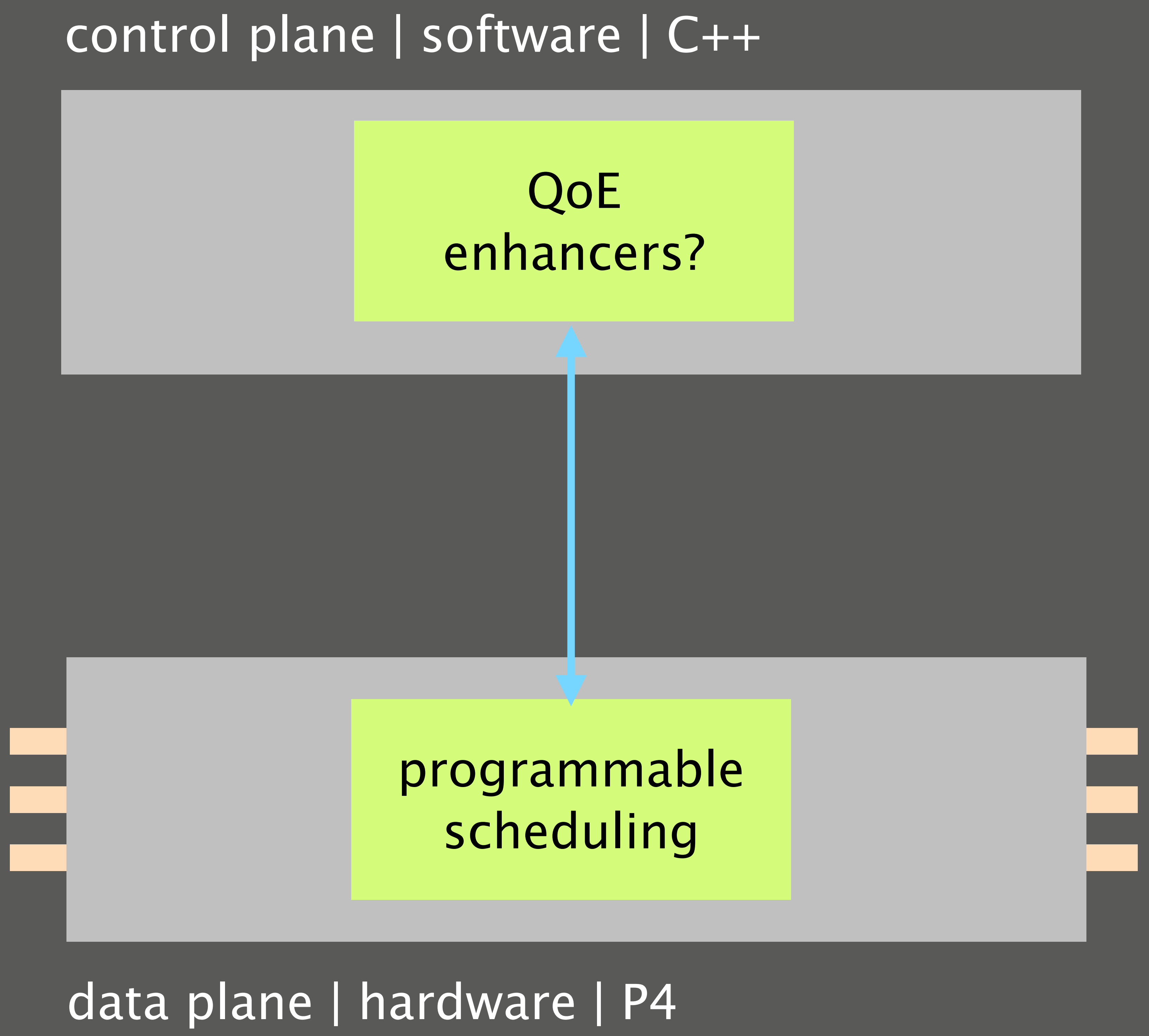
likely an overkill


control plane | software | C++
routing
how do we navigate
this spectrum?
routing
data plane | hardware | P4



what about we revisit the interface
between the two places?

what cool applications
can we build on top of
programmable schedulers?



control plane | software | C++

QoE
enhancers?

programmable
scheduling

data plane | hardware | P4

can we design better
adaptation strategies?
traffic-awareness?

Programmable, hardware-based

Routing and Scheduling

Fast routing

data plane offloading

Flexible scheduling

dynamic packet adaptation

Looking forward

routing | forwarding

All kudos go to my research group!

Alex Rüdiger Albert

Roland

Ahmed

Edgar

Mirja

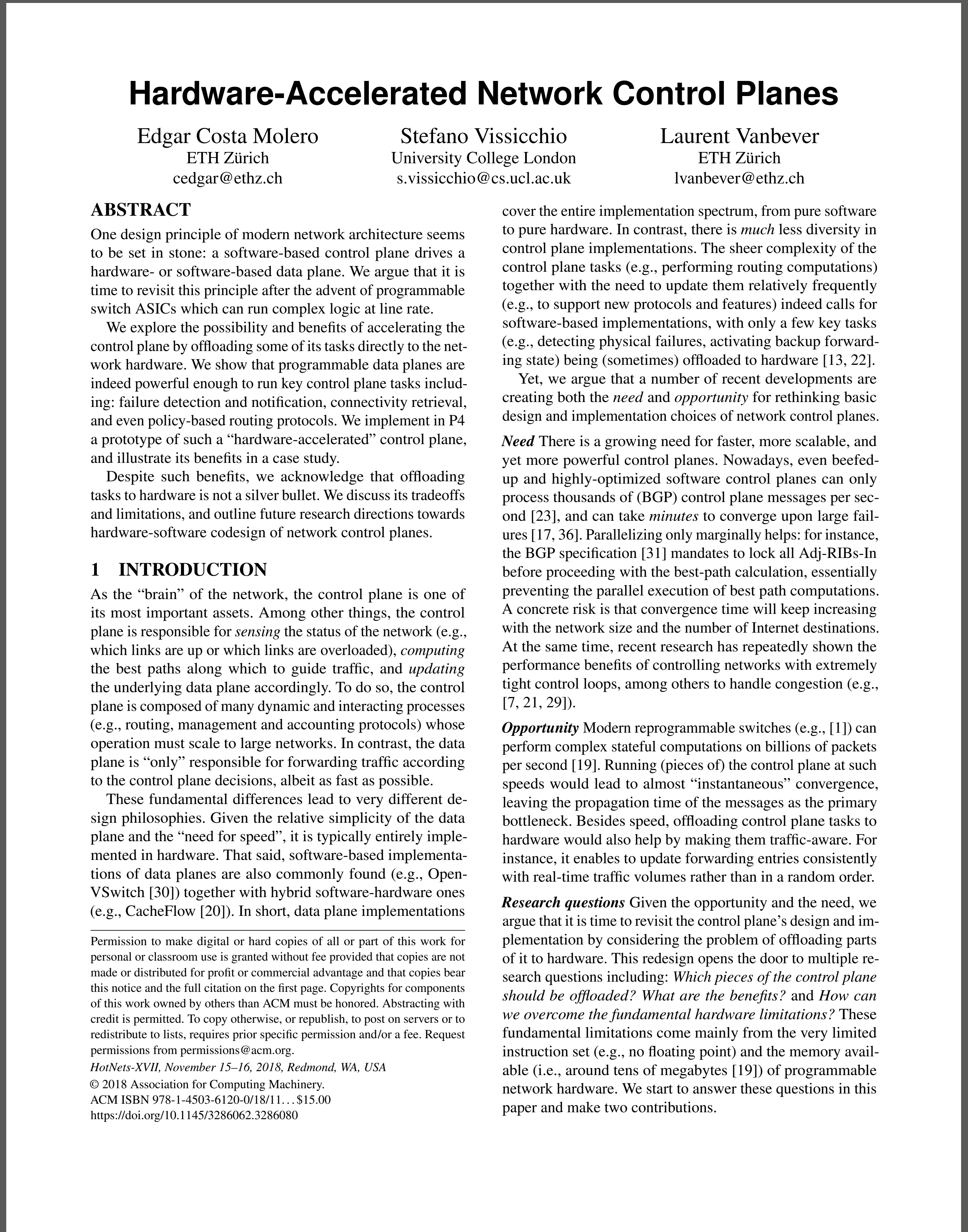
Thomas

Tobias

Coralie Romain Ege Roland Rui Maria

Check our papers out for *much* more info…

HotNets'18



Hardware-Accelerated Network Control Planes

Edgar Costa Molero
ETH Zürich
cedgar@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@cs.ucl.ac.uk

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

ABSTRACT

One design principle of modern network architecture seems to be set in stone: a software-based control plane drives a hardware- or software-based data plane. We argue that it is time to revisit this principle after the advent of programmable switch ASICs which can run complex logic at line rate.

We explore the possibility and benefits of accelerating the control plane by offloading some of its tasks directly to the network hardware. We show that programmable data planes are indeed powerful enough to run key control plane tasks including: failure detection and notification, connectivity retrieval, and even policy-based routing protocols. We implement in P4 a prototype of such a "hardware-accelerated" control plane, and illustrate its benefits in a case study.

Despite such benefits, we acknowledge that offloading tasks to hardware is not a silver bullet. We discuss its tradeoffs and limitations, and outline future research directions towards hardware-software codesign of network control planes.

1 INTRODUCTION

As the "brain" of the network, the control plane is one of its most important assets. Among other things, the control plane is responsible for *sensing* the status of the network (e.g., which links are up or which links are overloaded), *computing* the best paths along which to guide traffic, and *updating* the underlying data plane accordingly. To do so, the control plane is composed of many dynamic and interacting processes (e.g., routing, management and accounting protocols) whose operation must scale to large networks. In contrast, the data plane is "only" responsible for forwarding traffic according to the control plane decisions, albeit as fast as possible.

These fundamental differences lead to very different design philosophies. Given the relative simplicity of the data plane and the "need for speed", it is typically entirely implemented in hardware. That said, software-based implementations of data planes are also commonly found (e.g., OpenVSwitch [30]) together with hybrid software-hardware ones (e.g., CacheFlow [20]). In short, data plane implementations cover the entire implementation spectrum, from pure software to pure hardware. In contrast, there is *much* less diversity in control plane implementations. The sheer complexity of the control plane tasks (e.g., performing routing computations) together with the need to update them relatively frequently (e.g., to support new protocols and features) indeed calls for software-based implementations, with only a few key tasks (e.g., detecting physical failures, activating backup forwarding state) being (sometimes) offloaded to hardware [13, 22].

Yet, we argue that a number of recent developments are creating both the *need* and *opportunity* for rethinking basic design and implementation choices of network control planes.

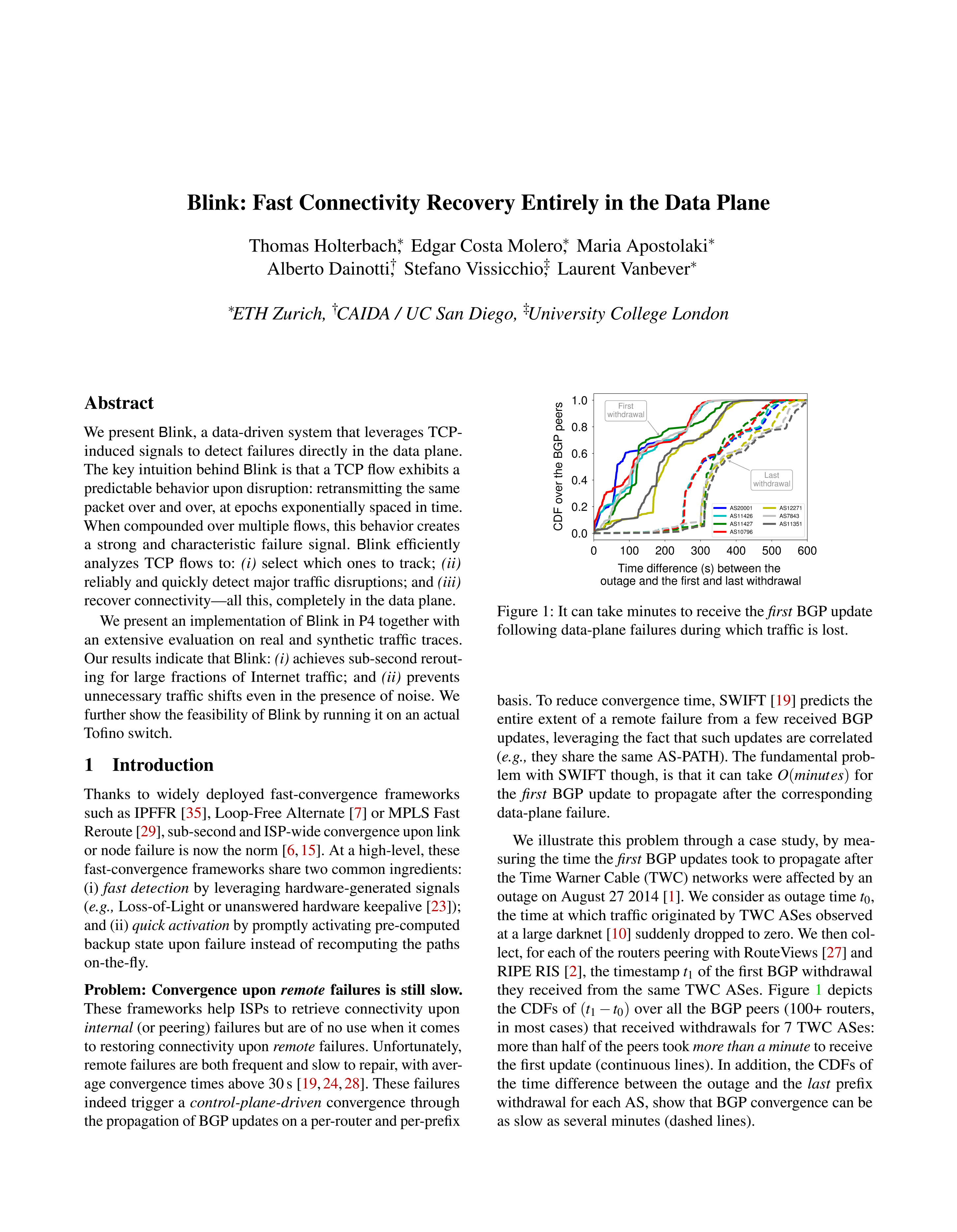
Need There is a growing need for faster, more scalable, and yet more powerful control planes. Nowadays, even beefed-up and highly-optimized software control planes can only process thousands of (BGP) control plane messages per second [23], and can take *minutes* to converge upon large failures [17, 36]. Parallelizing only marginally helps: for instance, the BGP specification [31] mandates to lock all Adj-RIBs-In before proceeding with the best-path calculation, essentially preventing the parallel execution of best path computations. A concrete risk is that convergence time will keep increasing with the network size and the number of Internet destinations. At the same time, recent research has repeatedly shown the performance benefits of controlling networks with extremely tight control loops, among others to handle congestion (e.g., [7, 21, 29]).

Opportunity Modern reprogrammable switches (e.g., [1]) can perform complex stateful computations on billions of packets per second [19]. Running (pieces of) the control plane at such speeds would lead to almost "instantaneous" convergence, leaving the propagation time of the messages as the primary bottleneck. Besides speed, offloading control plane tasks to hardware would also help by making them traffic-aware. For instance, it enables to update forwarding entries consistently with real-time traffic volumes rather than in a random order.

Research questions Given the opportunity and the need, we argue that it is time to revisit the control plane's design and implementation by considering the problem of offloading parts of it to hardware. This redesign opens the door to multiple research questions including: *Which pieces of the control plane should be offloaded? What are the benefits?* and *How can we overcome the fundamental hardware limitations?* These fundamental limitations come mainly from the very limited instruction set (e.g., no floating point) and the memory available (i.e., around tens of megabytes [19]) of programmable network hardware. We start to answer these questions in this paper and make two contributions.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286080

NSDI'19



Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach[*], Edgar Costa Molero[*], Maria Apostolaki[*]
Alberto Dainotti[†], Stefano Vissicchio[‡], Laurent Vanbever[*]

[*]*ETH Zurich,* [†]*CAIDA / UC San Diego,* [‡]*University College London*

Abstract

We present Blink, a data-driven system that leverages TCP-induced signals to detect failures directly in the data plane. The key intuition behind Blink is that a TCP flow exhibits a predictable behavior upon disruption: retransmitting the same packet over and over, at epochs exponentially spaced in time. When compounded over multiple flows, this behavior creates a strong and characteristic failure signal. Blink efficiently analyzes TCP flows to: *(i)* select which ones to track; *(ii)* reliably and quickly detect major traffic disruptions; and *(iii)* recover connectivity—all this, completely in the data plane.

We present an implementation of Blink in P4 together with an extensive evaluation on real and synthetic traffic traces. Our results indicate that Blink: *(i)* achieves sub-second rerouting for large fractions of Internet traffic; and *(ii)* prevents unnecessary traffic shifts even in the presence of noise. We further show the feasibility of Blink by running it on an actual Tofino switch.

1 Introduction

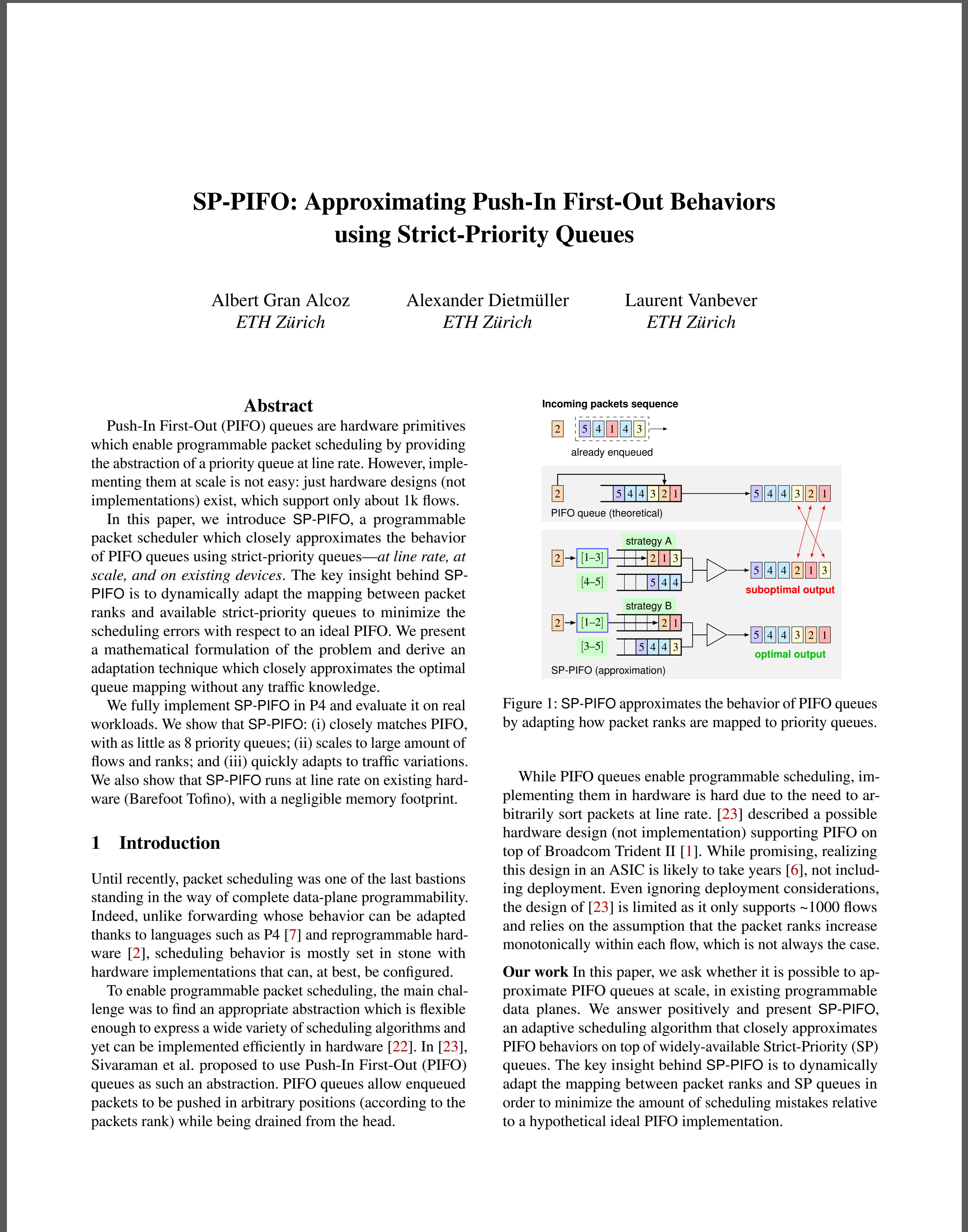
Thanks to widely deployed fast-convergence frameworks such as IPFFR [35], Loop-Free Alternate [7] or MPLS Fast Reroute [29], sub-second and ISP-wide convergence upon link or node failure is now the norm [6, 15]. At a high-level, these fast-convergence frameworks share two common ingredients: (i) *fast detection* by leveraging hardware-generated signals (*e.g.,* Loss-of-Light or unanswered hardware keepalive [23]); and (ii) *quick activation* by promptly activating pre-computed backup state upon failure instead of recomputing the paths on-the-fly.

Problem: Convergence upon *remote* failures is still slow. These frameworks help ISPs to retrieve connectivity upon *internal* (or peering) failures but are of no use when it comes to restoring connectivity upon *remote* failures. Unfortunately, remote failures are both frequent and slow to repair, with average convergence times above 30 s [19, 24, 28]. These failures indeed trigger a *control-plane-driven* convergence through the propagation of BGP updates on a per-router and per-prefix basis. To reduce convergence time, SWIFT [19] predicts the entire extent of a remote failure from a few received BGP updates, leveraging the fact that such updates are correlated (*e.g.,* they share the same AS-PATH). The fundamental problem with SWIFT though, is that it can take $O(minutes)$ for the *first* BGP update to propagate after the corresponding data-plane failure.

We illustrate this problem through a case study, by measuring the time the *first* BGP updates took to propagate after the Time Warner Cable (TWC) networks were affected by an outage on August 27 2014 [1]. We consider as outage time t_0, the time at which traffic originated by TWC ASes observed at a large darknet [10] suddenly dropped to zero. We then collect, for each of the routers peering with RouteViews [27] and RIPE RIS [2], the timestamp t_1 of the first BGP withdrawal they received from the same TWC ASes. Figure 1 depicts the CDFs of $(t_1 - t_0)$ over all the BGP peers (100+ routers, in most cases) that received withdrawals for 7 TWC ASes: more than half of the peers took *more than a minute* to receive the first update (continuous lines). In addition, the CDFs of the time difference between the outage and the *last* prefix withdrawal for each AS, show that BGP convergence can be as slow as several minutes (dashed lines).

Figure 1: It can take minutes to receive the *first* BGP update following data-plane failures during which traffic is lost.

NSDI'20



SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues

Albert Gran Alcoz
ETH Zürich

Alexander Dietmüller
ETH Zürich

Laurent Vanbever
ETH Zürich

Abstract

Push-In First-Out (PIFO) queues are hardware primitives which enable programmable packet scheduling by providing the abstraction of a priority queue at line rate. However, implementing them at scale is not easy: just hardware designs (not implementations) exist, which support only about 1k flows.

In this paper, we introduce SP-PIFO, a programmable packet scheduler which closely approximates the behavior of PIFO queues using strict-priority queues—*at line rate, at scale, and on existing devices*. The key insight behind SP-PIFO is to dynamically adapt the mapping between packet ranks and available strict-priority queues to minimize the scheduling errors with respect to an ideal PIFO. We present a mathematical formulation of the problem and derive an adaptation technique which closely approximates the optimal queue mapping without any traffic knowledge.

We fully implement SP-PIFO in P4 and evaluate it on real workloads. We show that SP-PIFO: (i) closely matches PIFO, with as little as 8 priority queues; (ii) scales to large amount of flows and ranks; and (iii) quickly adapts to traffic variations. We also show that SP-PIFO runs at line rate on existing hardware (Barefoot Tofino), with a negligible memory footprint.

1 Introduction

Until recently, packet scheduling was one of the last bastions standing in the way of complete data-plane programmability. Indeed, unlike forwarding whose behavior can be adapted thanks to languages such as P4 [7] and reprogrammable hardware [2], scheduling behavior is mostly set in stone with hardware implementations that can, at best, be configured.

To enable programmable packet scheduling, the main challenge was to find an appropriate abstraction which is flexible enough to express a wide variety of scheduling algorithms and yet can be implemented efficiently in hardware [22]. In [23], Sivaraman et al. proposed to use Push-In First-Out (PIFO) queues as such an abstraction. PIFO queues allow enqueued packets to be pushed in arbitrary positions (according to the packets rank) while being drained from the head.

Figure 1: SP-PIFO approximates the behavior of PIFO queues by adapting how packet ranks are mapped to priority queues.

While PIFO queues enable programmable scheduling, implementing them in hardware is hard due to the need to arbitrarily sort packets at line rate. [23] described a possible hardware design (not implementation) supporting PIFO on top of Broadcom Trident II [1]. While promising, realizing this design in an ASIC is likely to take years [6], not including deployment. Even ignoring deployment considerations, the design of [23] is limited as it only supports ~1000 flows and relies on the assumption that the packet ranks increase monotonically within each flow, which is not always the case.

Our work In this paper, we ask whether it is possible to approximate PIFO queues at scale, in existing programmable data planes. We answer positively and present SP-PIFO, an adaptive scheduling algorithm that closely approximates PIFO behaviors on top of widely-available Strict-Priority (SP) queues. The key insight behind SP-PIFO is to dynamically adapt the mapping between packet ranks and SP queues in order to minimize the amount of scheduling mistakes relative to a hypothetical ideal PIFO implementation.

Programmable, hardware-based

Routing and Scheduling

Speed × Flexibility

Laurent Vanbever

nsg.ee.ethz.ch

EuroP4

Tue Dec 1 2020