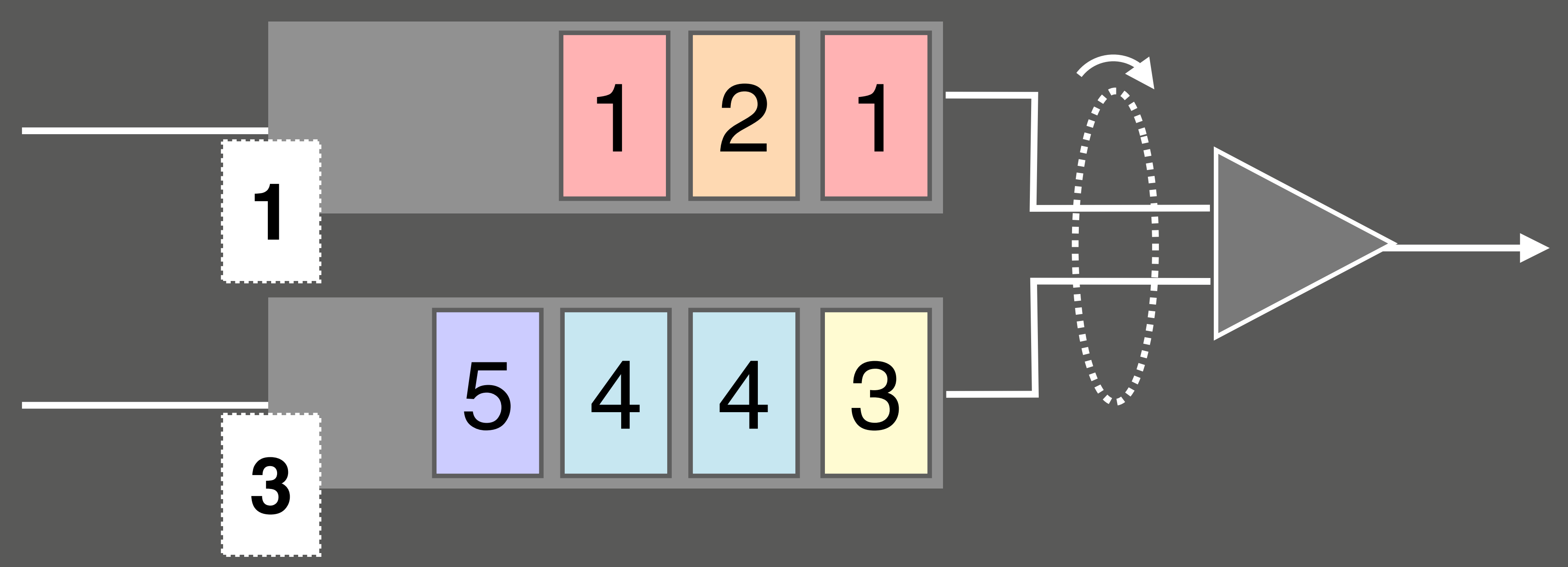
SP-PIFO: Programmable packet scheduling on existing hardware



Albert Gran Alcoz

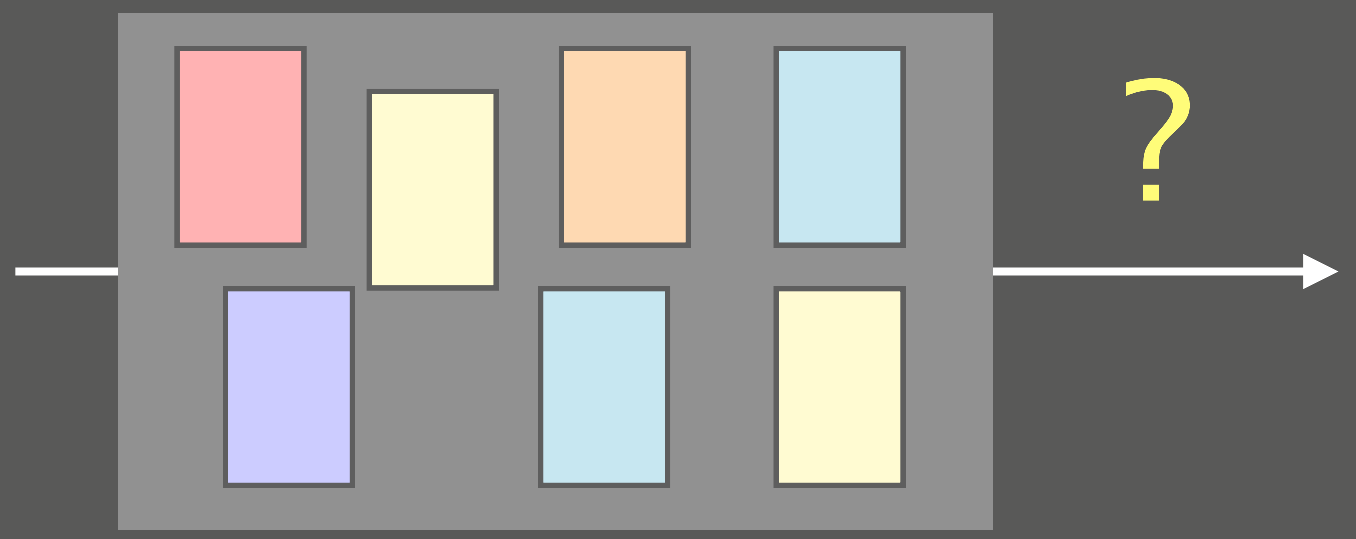
sp-pifo.ethz.ch

Princeton University

September 14 2022

Packet scheduling

When and *in which order* should we forward buffered packets?



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

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

+ *many* more

Minimize tail latency

FIFO+ — Prioritize packets with higher queuing time

LSTF

Minimize flow completion times

SRPT — Prioritize packets from short flows

PIAS

pFabric

Enforce max-min fairness

WRR — Send one packet from each class at a time

(S)FQ

WFQ

+ *many* more

Is there a universal packet scheduler?

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).

"You can't have *everything* you want,

Generality

Universal packet scheduler

“You can’t have *everything* you want,

but you can have *anything* you want”

Generality

Universal packet scheduler

Flexibility

Customized algorithms

“You can’t have *everything* you want,

but you can have *anything* you want”

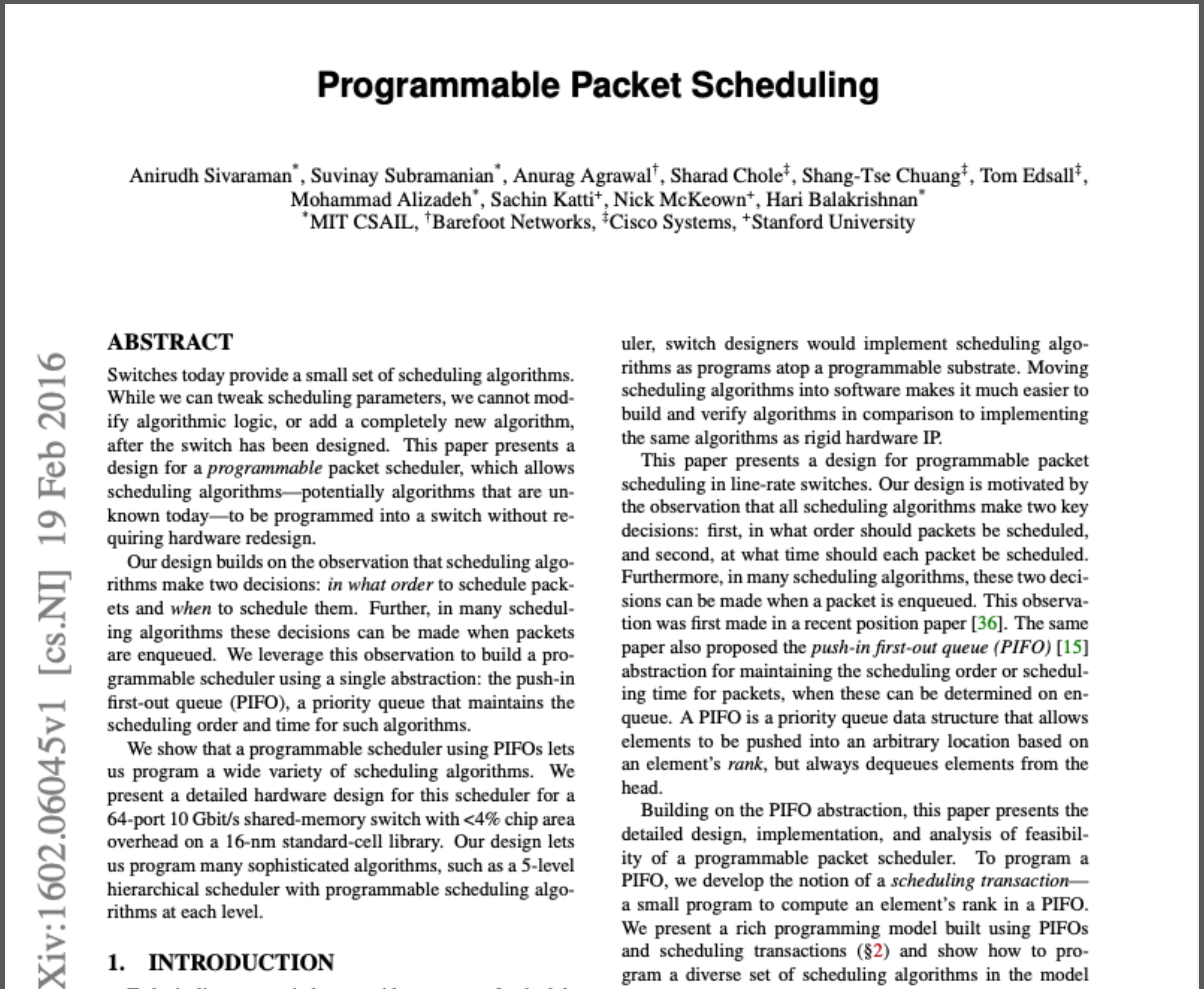
Generality

Universal packet scheduler

Programmable 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…

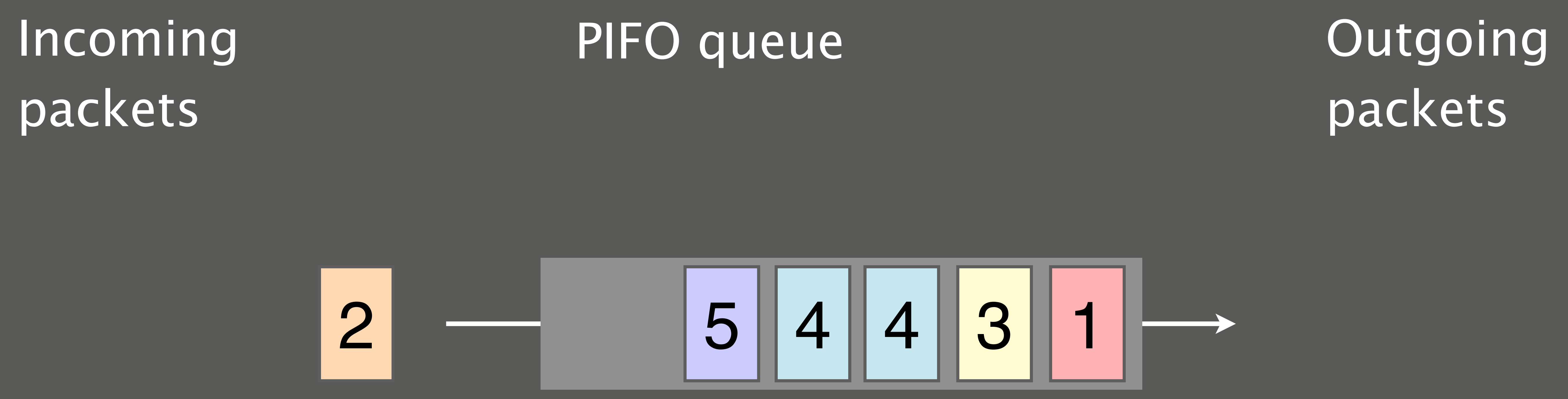
- pushes packets to arbitrary positions, based on their ranks
- drains packets from the head

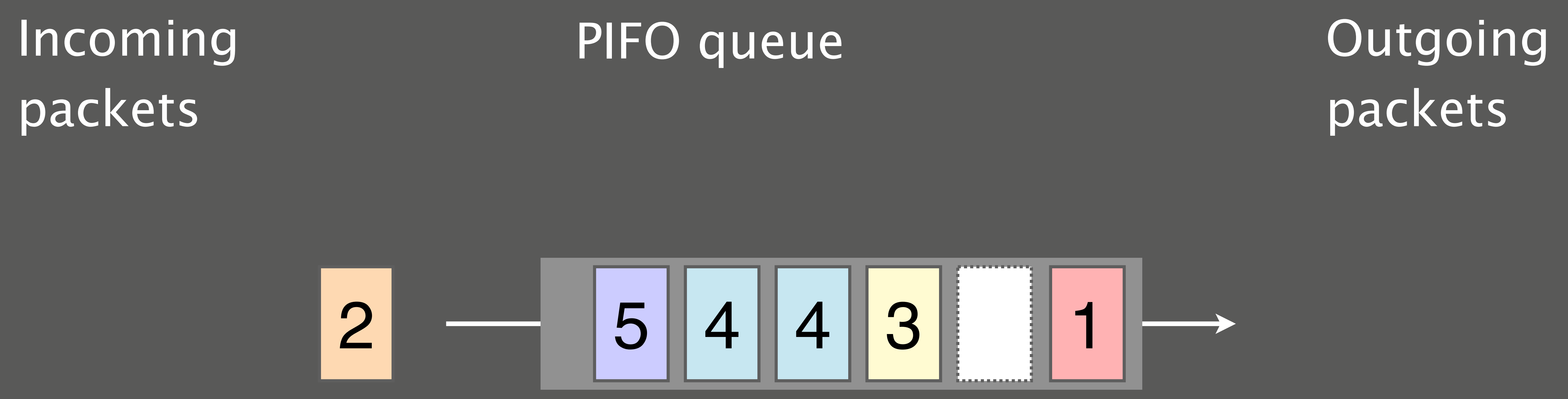
Push-In First-Out Queue (PIFO) is a data structure that enables programmable packet scheduling

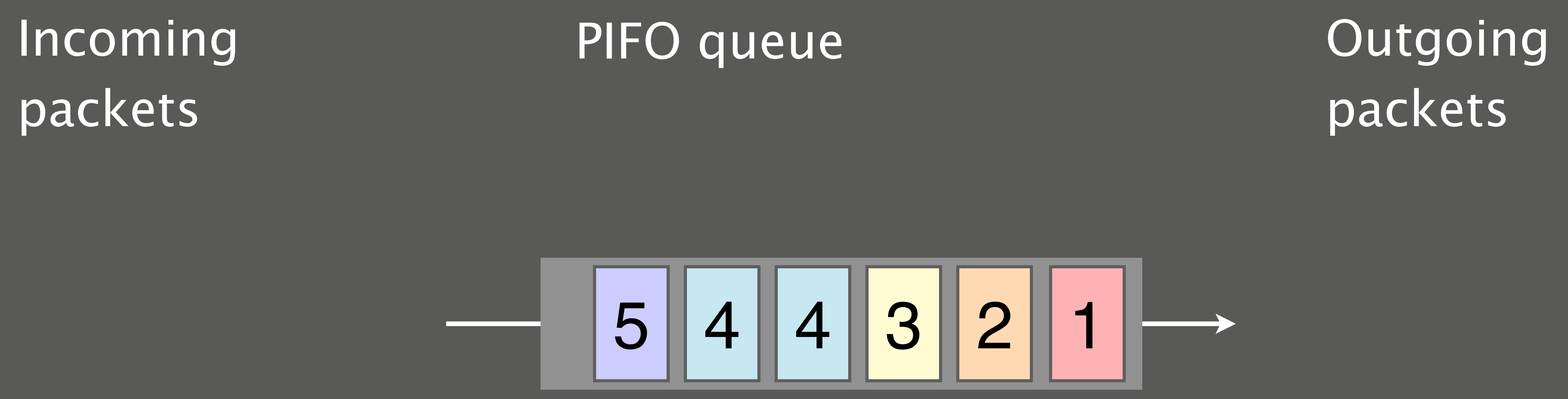
A PIFO queue…

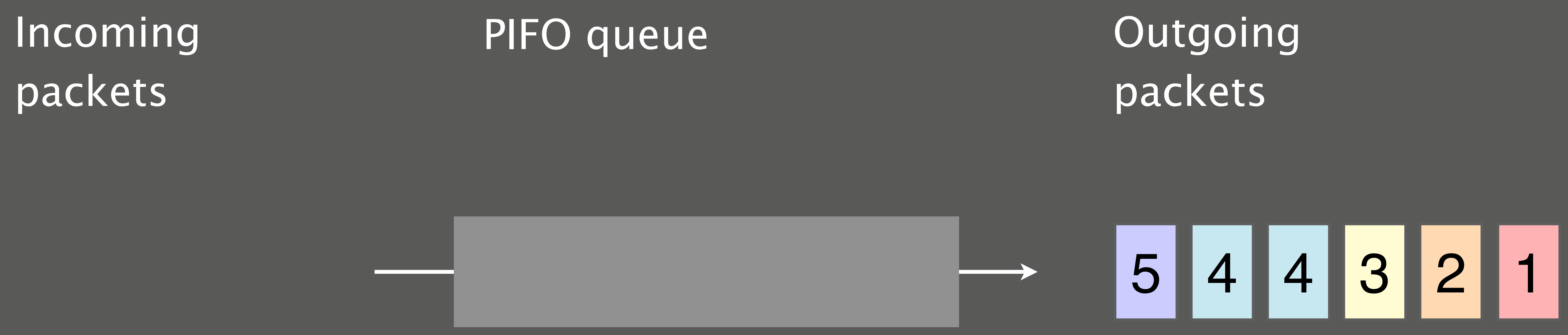
- pushes packets to arbitrary positions, based on their ranks
- drains packets from the head

Sorts packets perfectly by increasing rank order


Incoming
packets
PIFO queue
Outgoing
packets
2
5
4
4
3
1


Incoming
packets
PIFO queue
Outgoing
packets
2
5
4
4
3
1

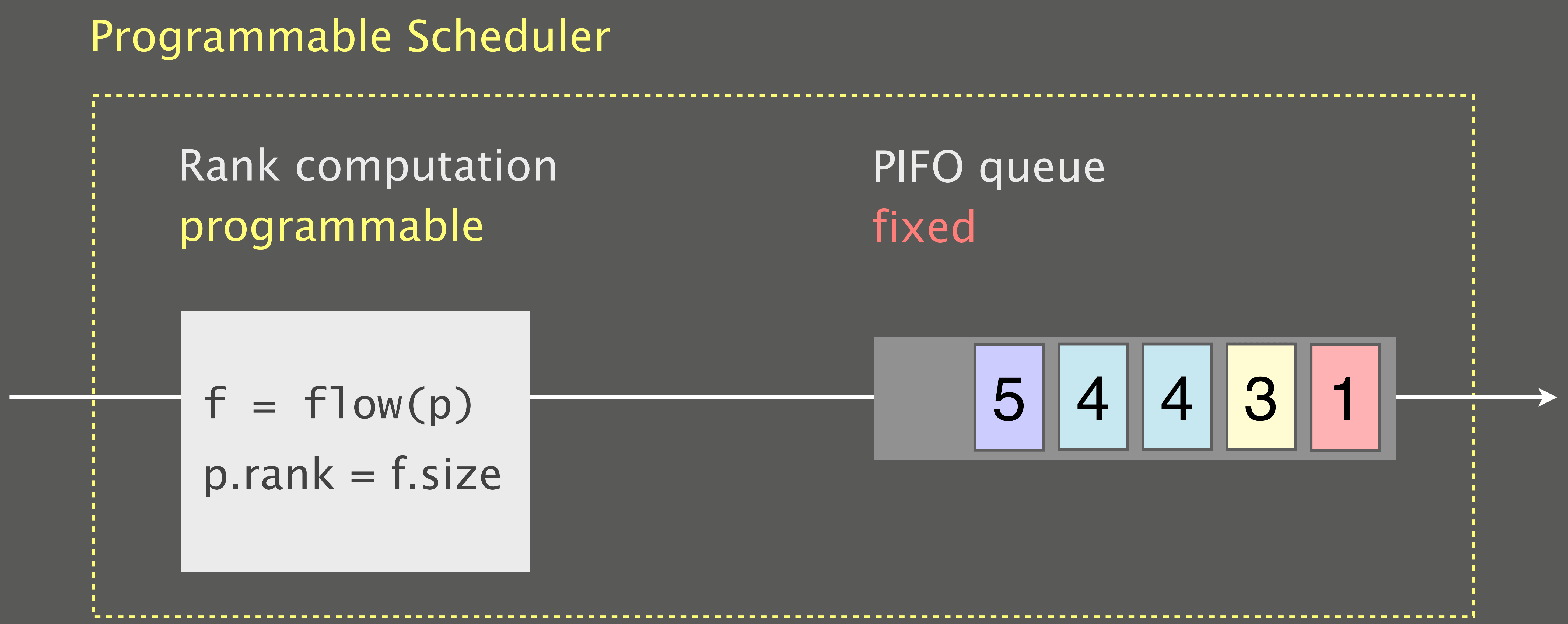

Incoming
packets
PIFO queue
Outgoing
packets
5
4
4
3
2
1


Incoming packets
PIFO queue
Outgoing packets
5
4
4
3
2
1

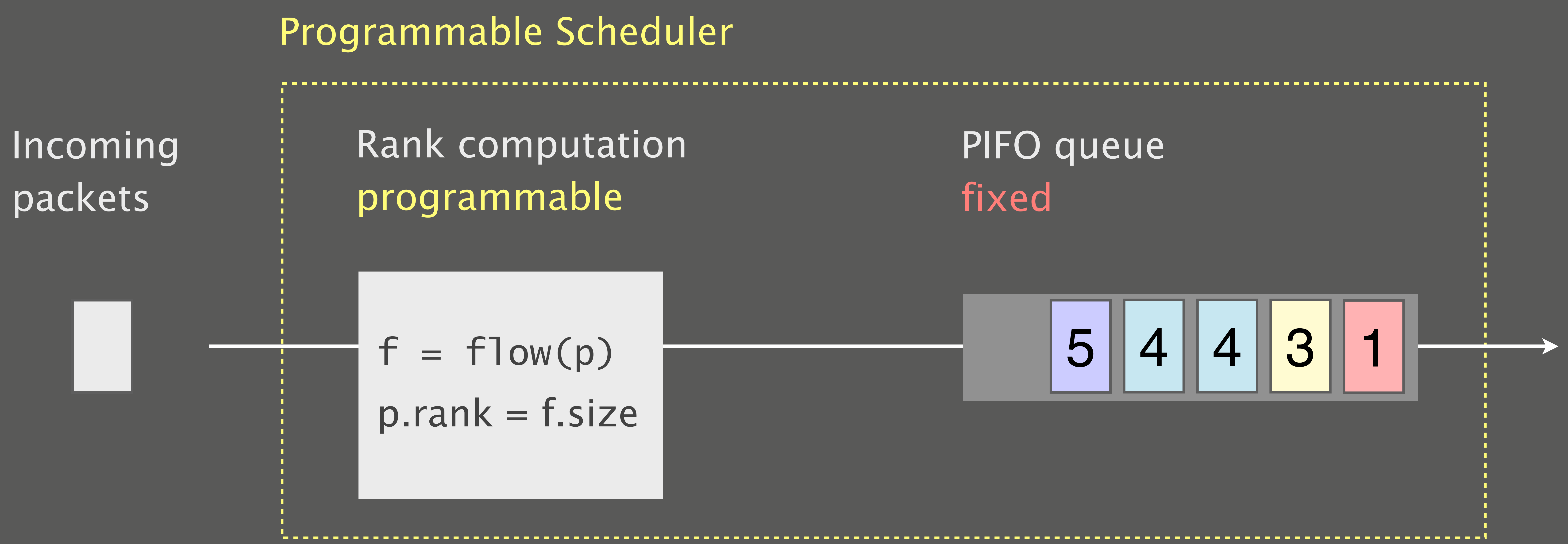
Push-In First-Out Queue (PIFO) is a data structure that enables programmable packet scheduling

How exactly?

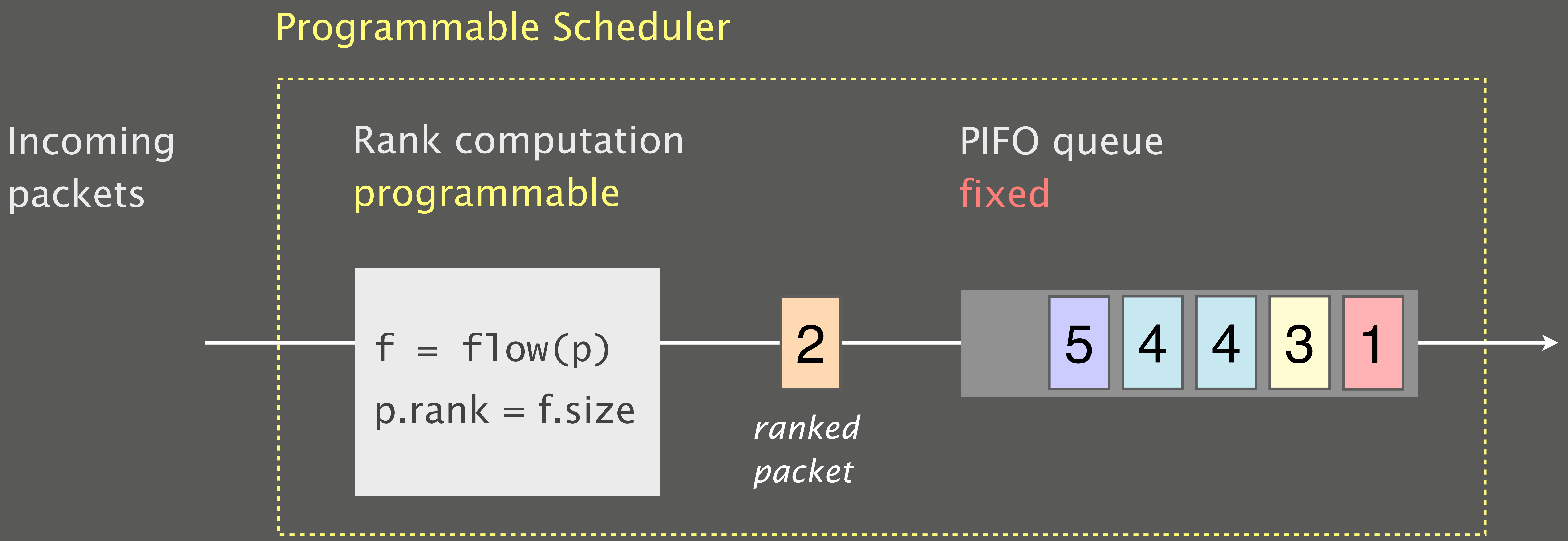
Implementing a new algorithm simply requires to adapt the rank computation logic



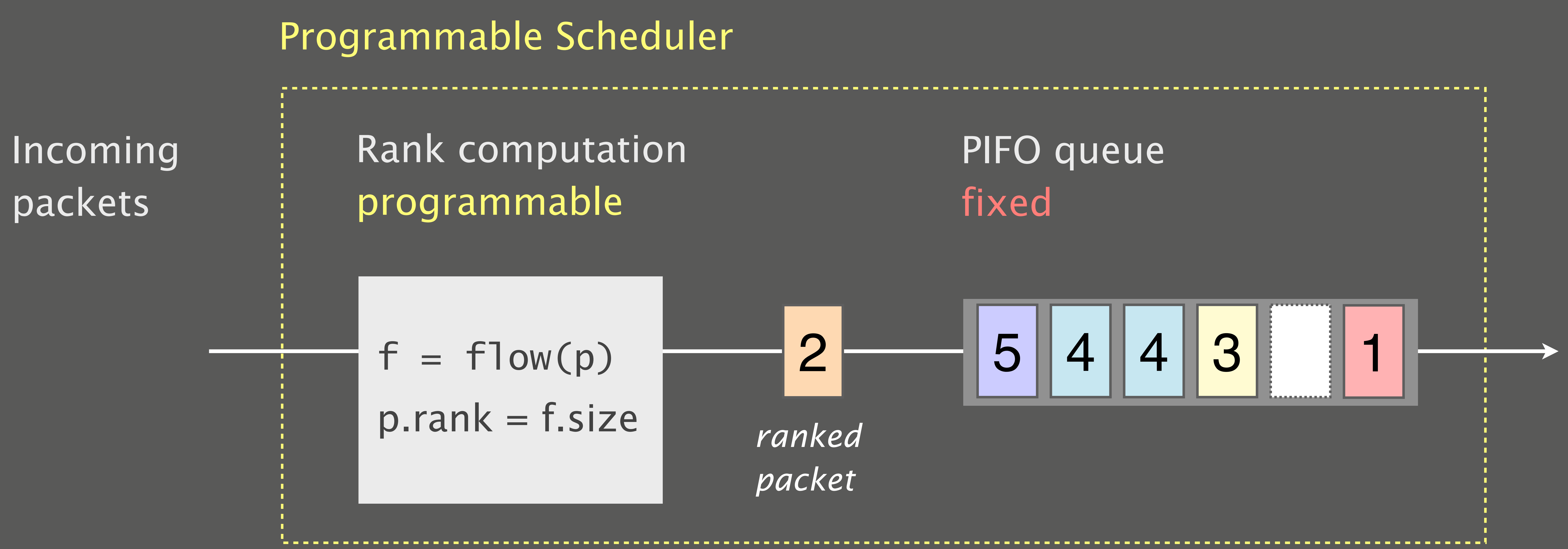
Implementing a new algorithm simply requires to adapt the rank computation logic



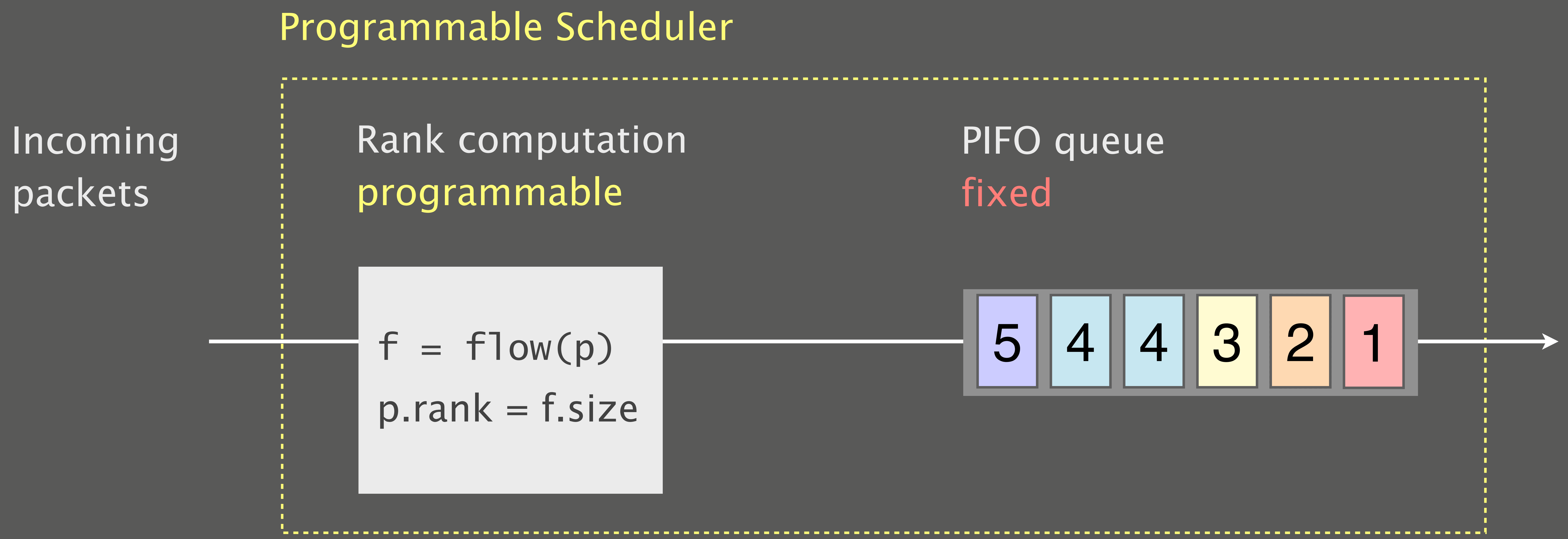
Implementing a new algorithm simply requires to adapt the rank computation logic



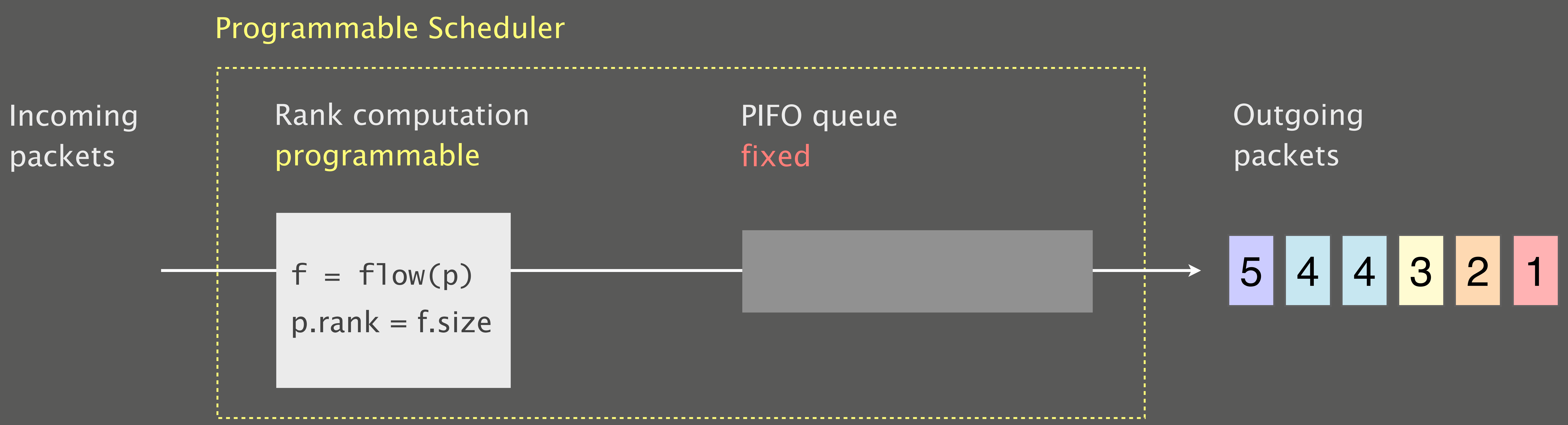
Implementing a new algorithm simply requires to adapt the rank computation logic



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



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



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…

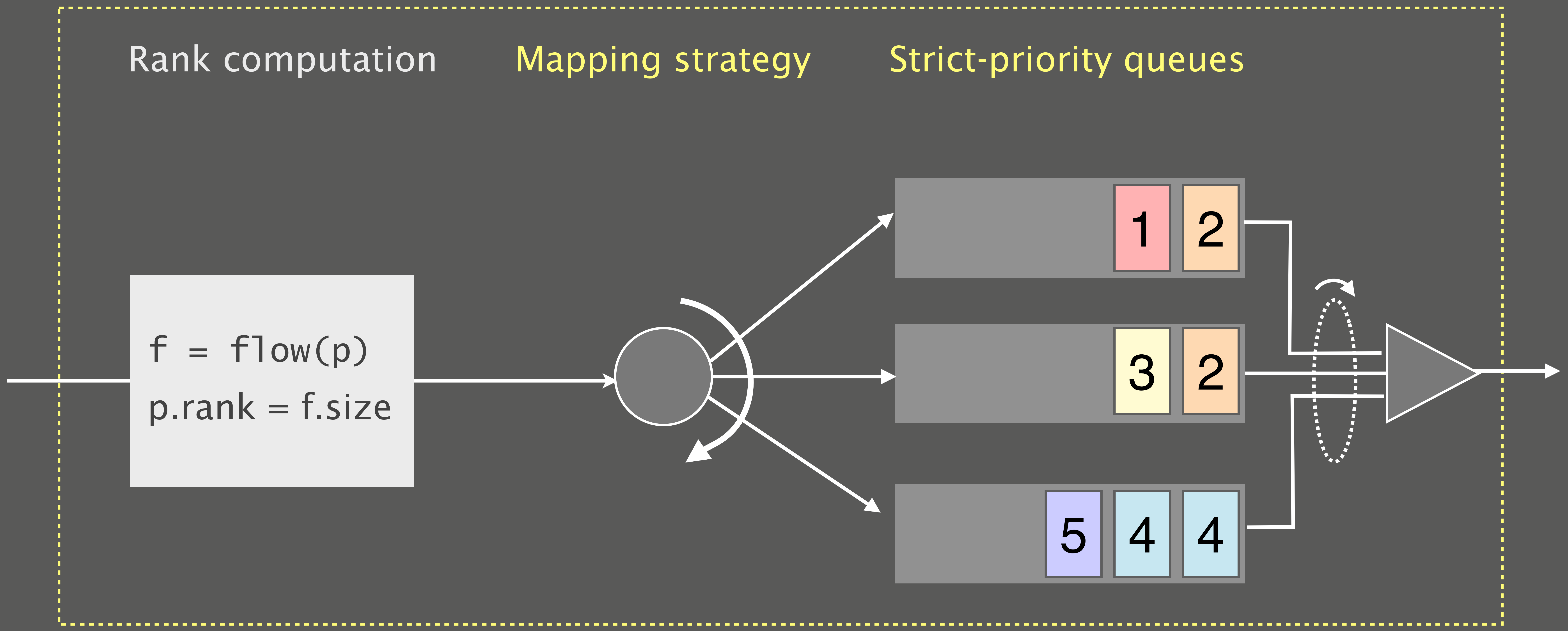
- at line rate;
- at scale;
- on existing devices?

Yep!

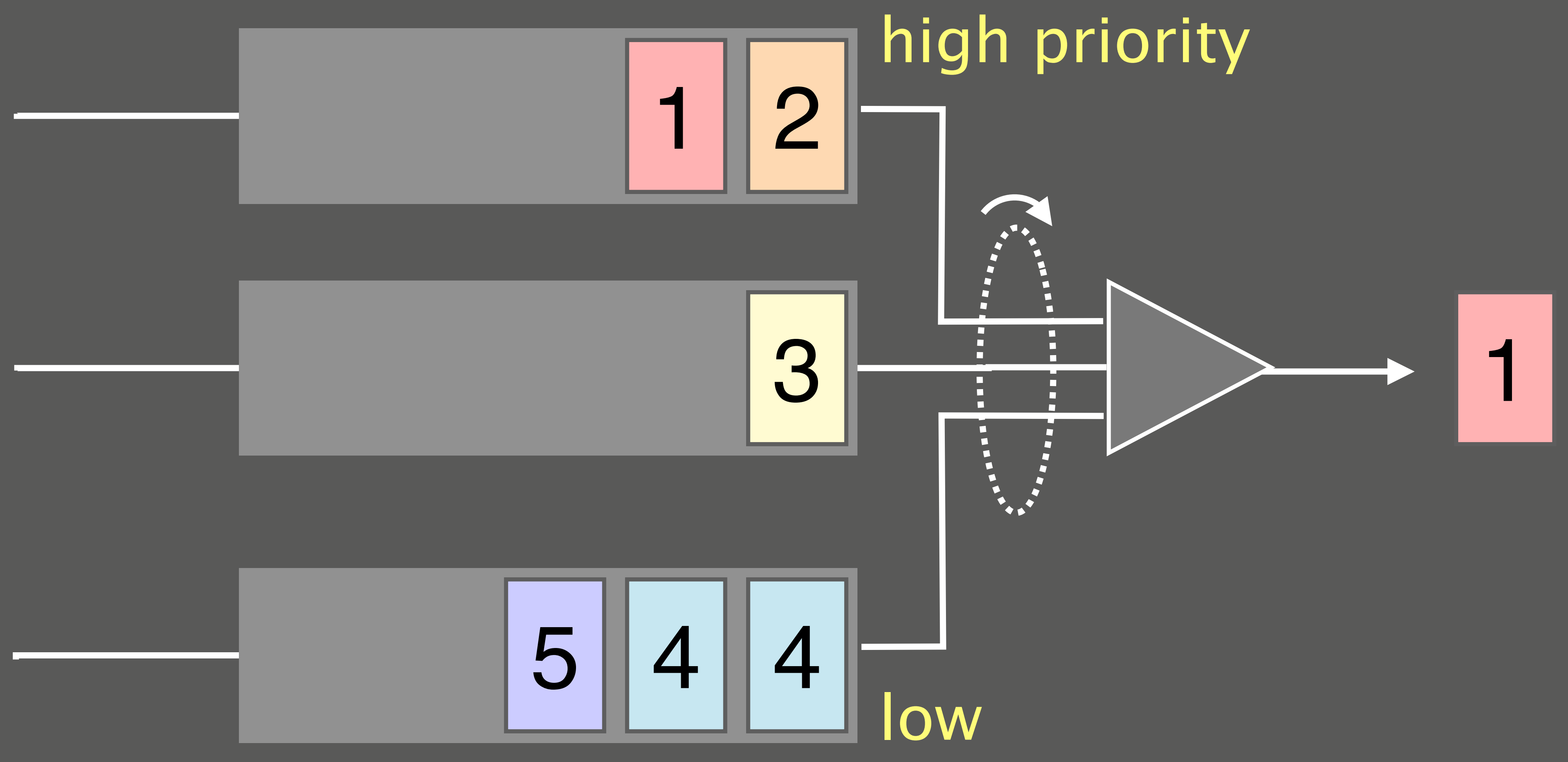
Introducing **SP-PIFO**

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

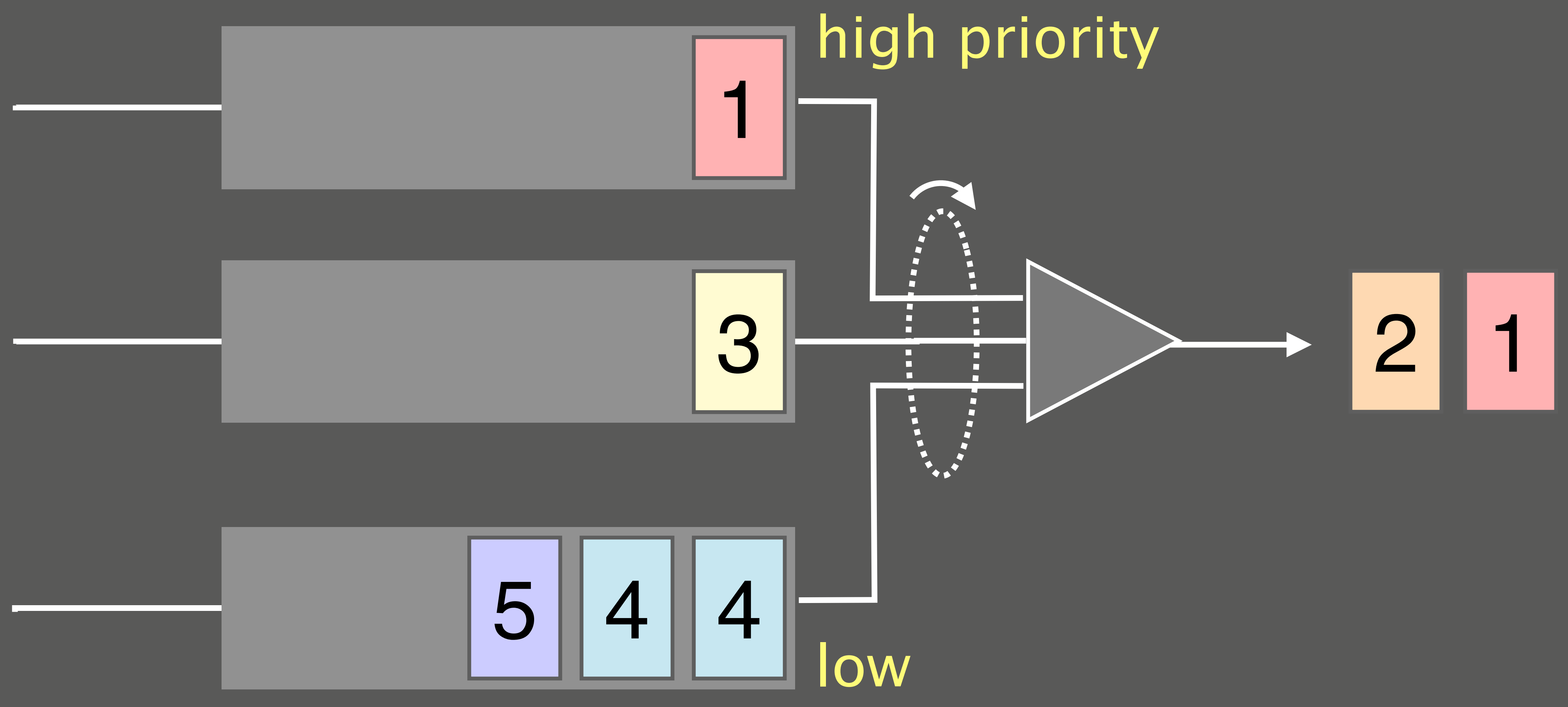
SP-PIFO Programmable Scheduler



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

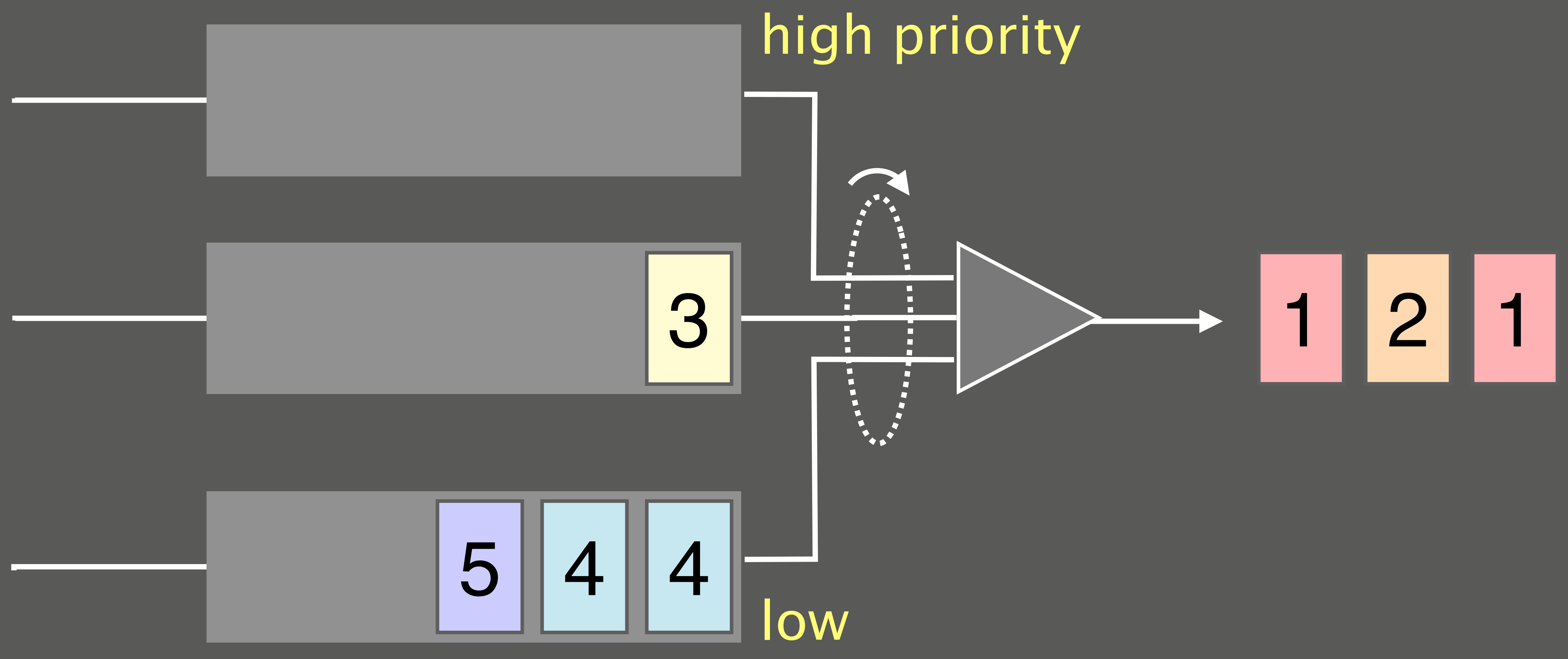


SP-PIFO approximates PIFO queues using strict-priority queues and a dynamic mapping strategy



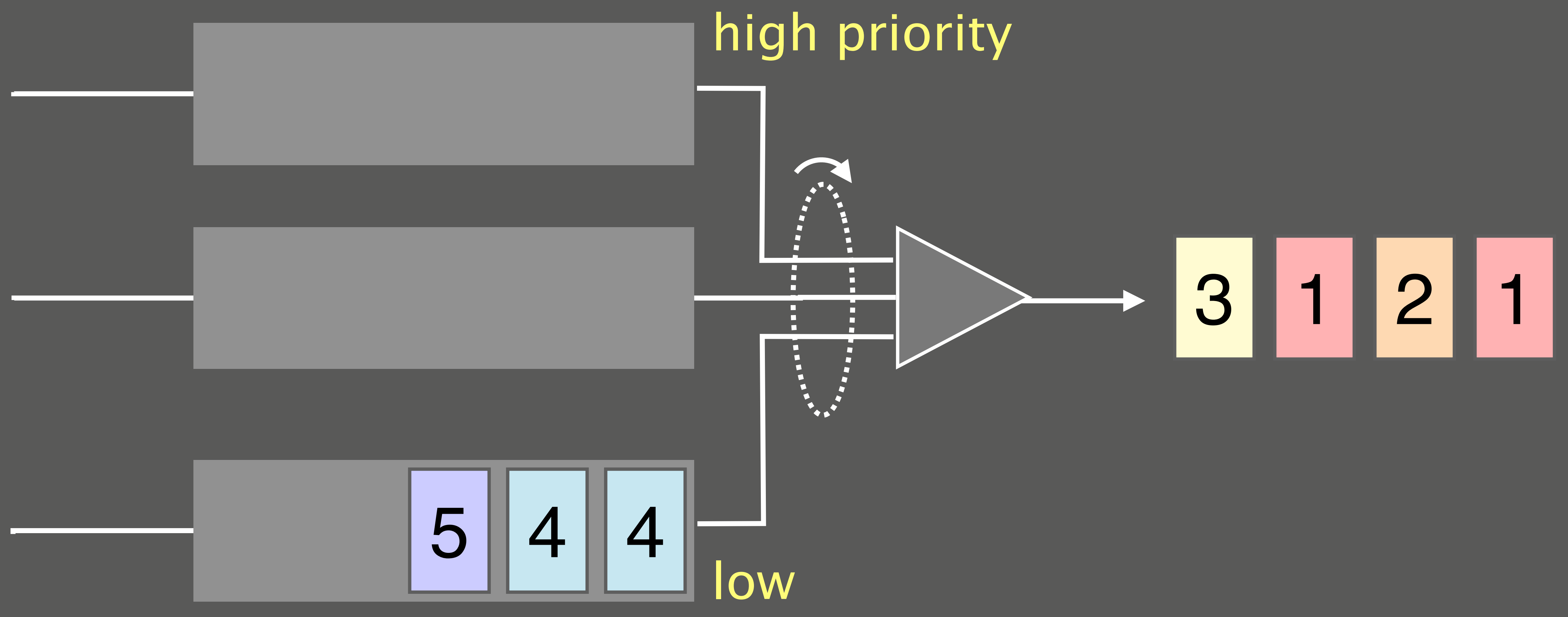
SP-PIFO approximates PIFO queues using

strict-priority queues and a dynamic mapping strategy

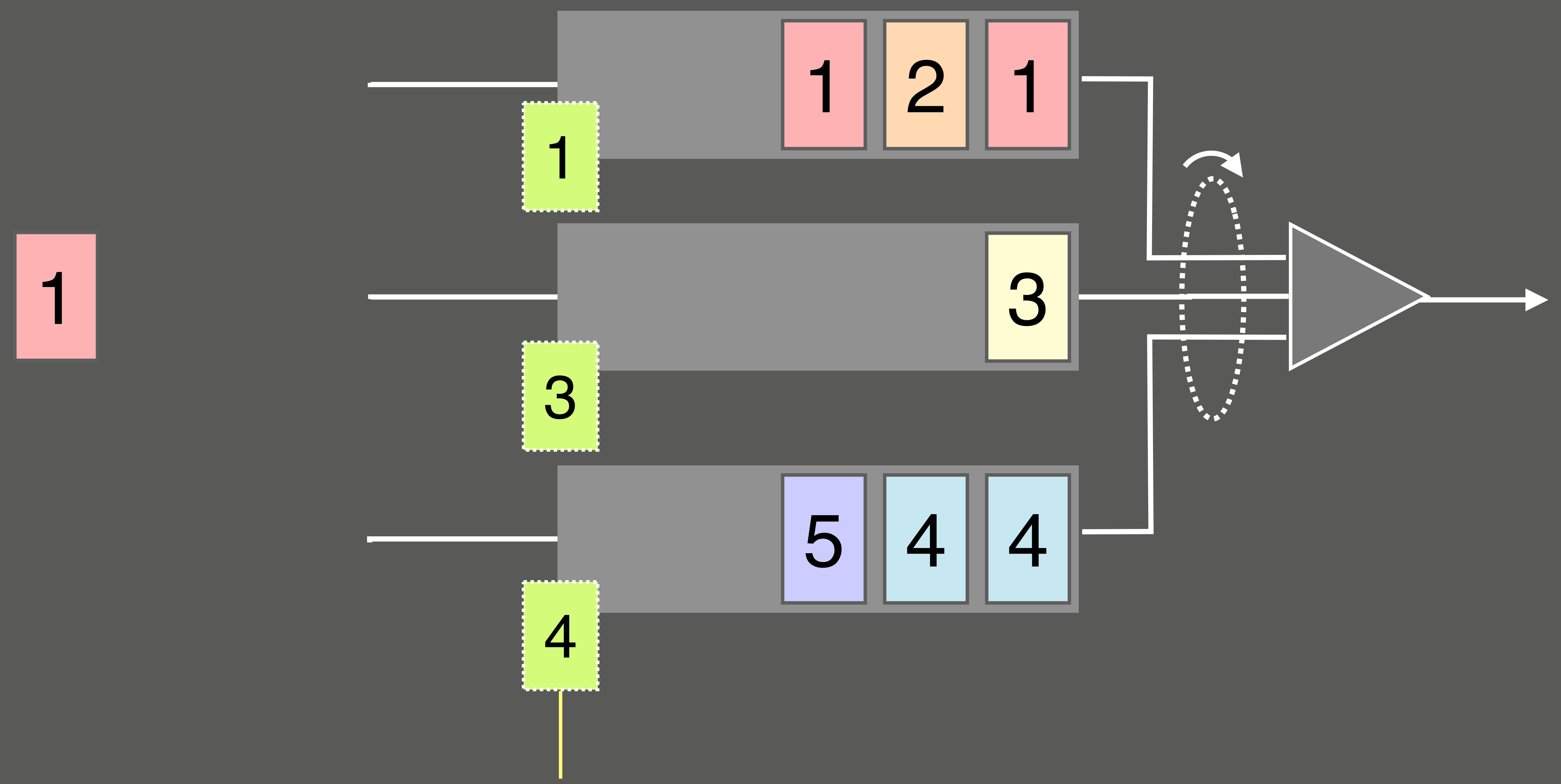


SP-PIFO approximates PIFO queues using

strict-priority queues and a dynamic mapping strategy



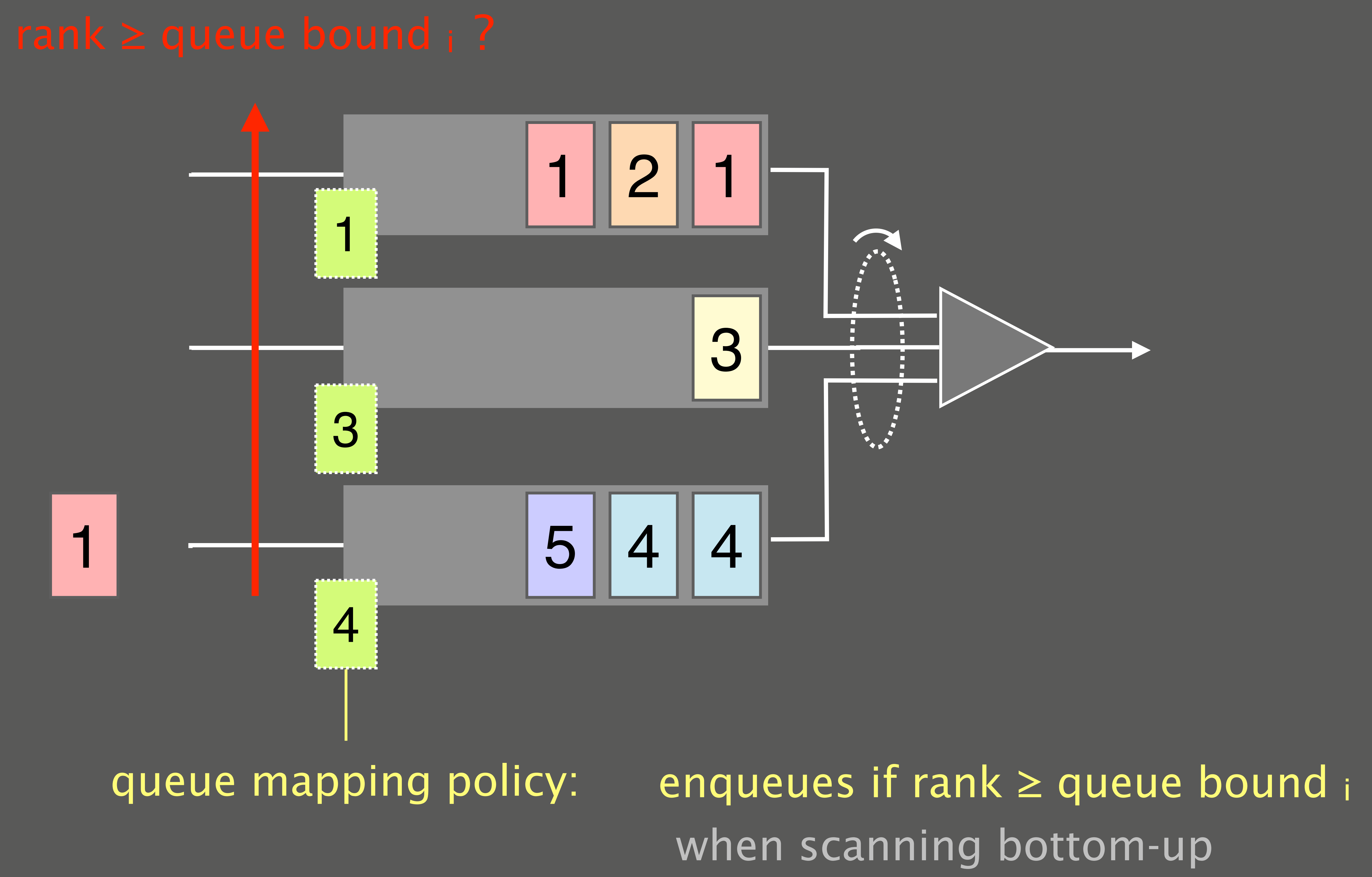
SP-PIFO approximates PIFO queues using strict-priority queues and a dynamic mapping strategy



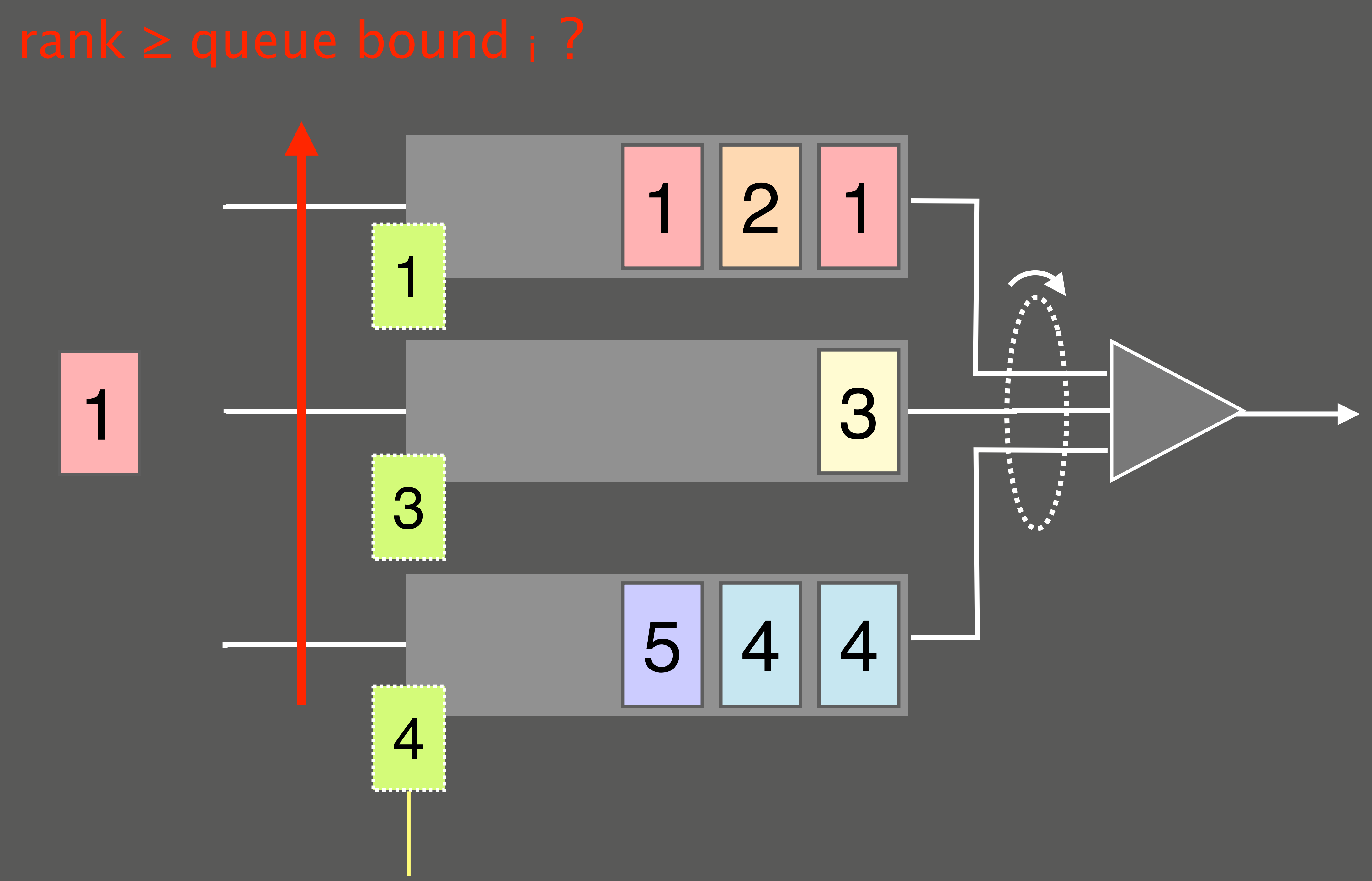
queue mapping policy: enqueues if rank ≥ queue bound $_i$

when scanning bottom-up

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



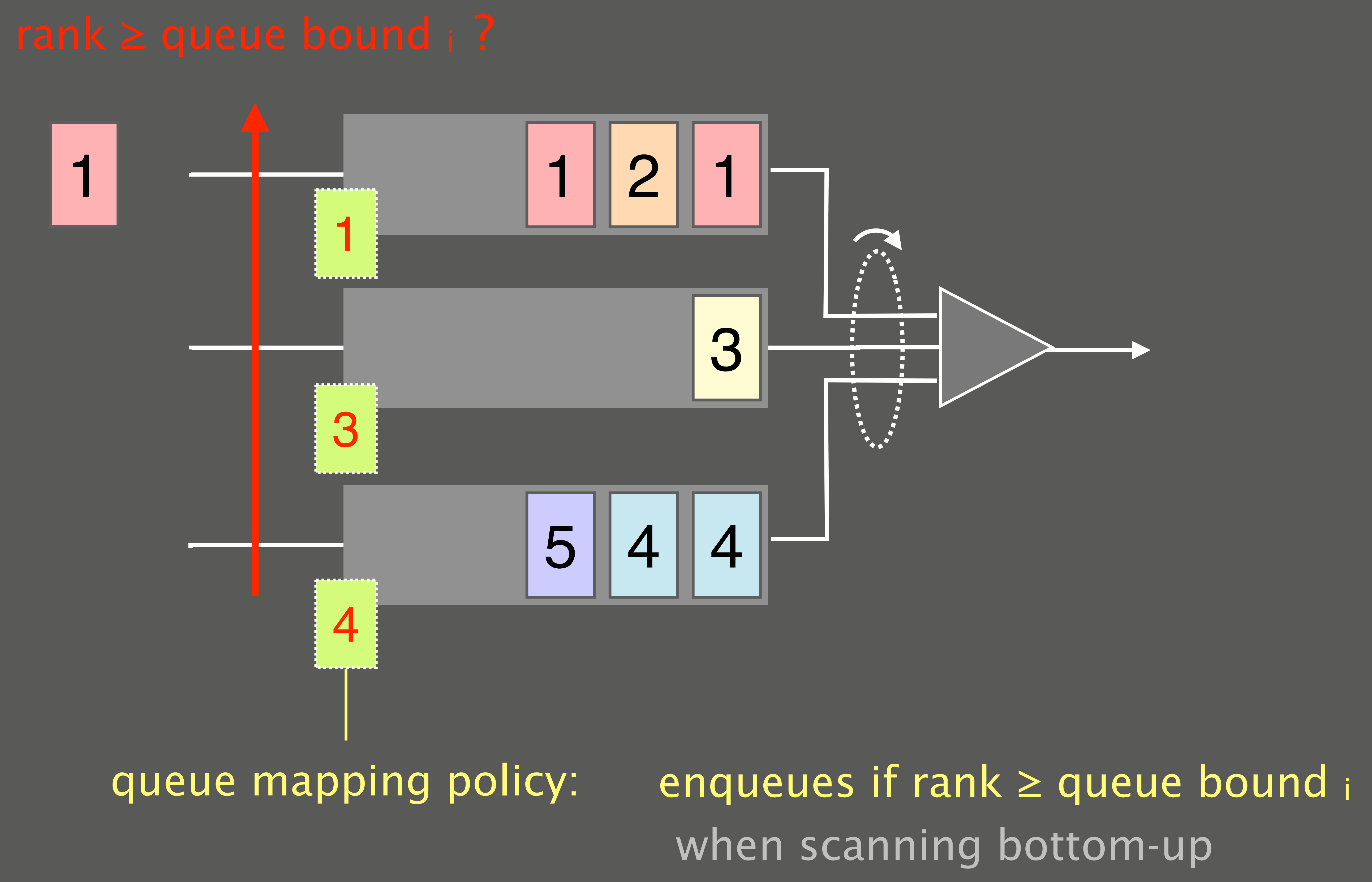
SP-PIFO approximates PIFO queues using strict-priority queues and a dynamic mapping strategy



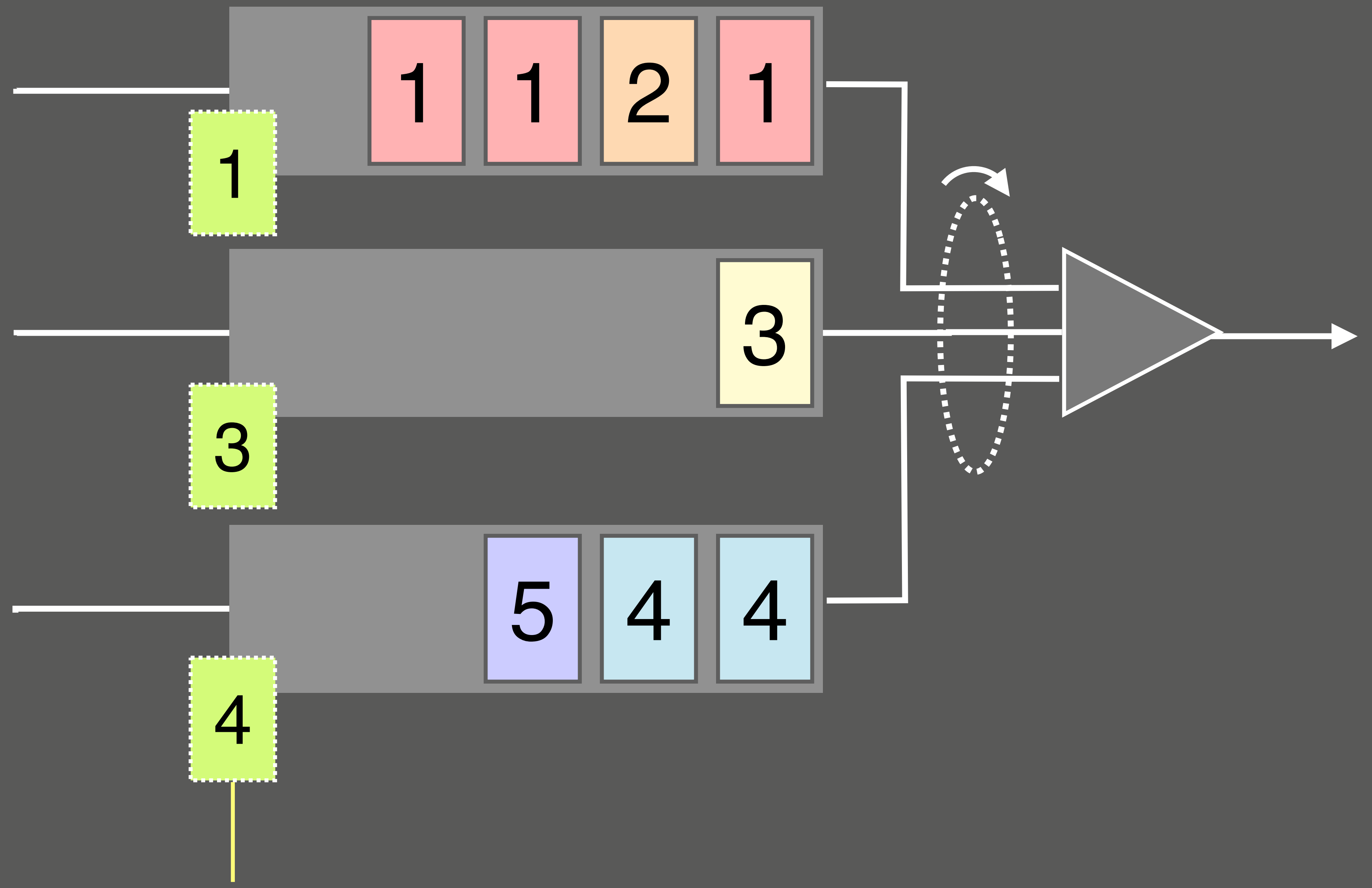
queue mapping policy: enqueues if rank ≥ queue bound $_i$

when scanning bottom-up

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

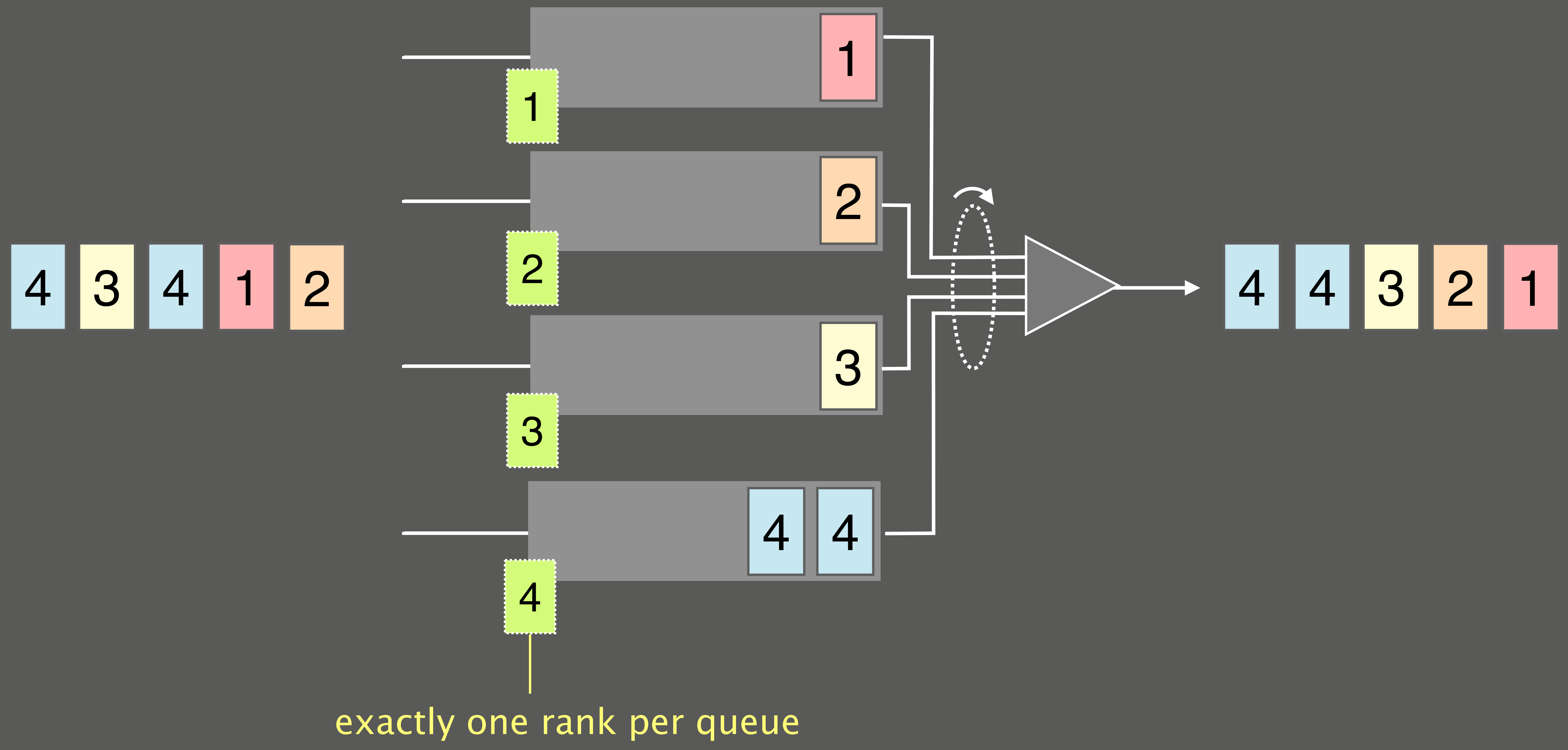


SP-PIFO approximates PIFO queues using strict-priority queues and a dynamic mapping strategy



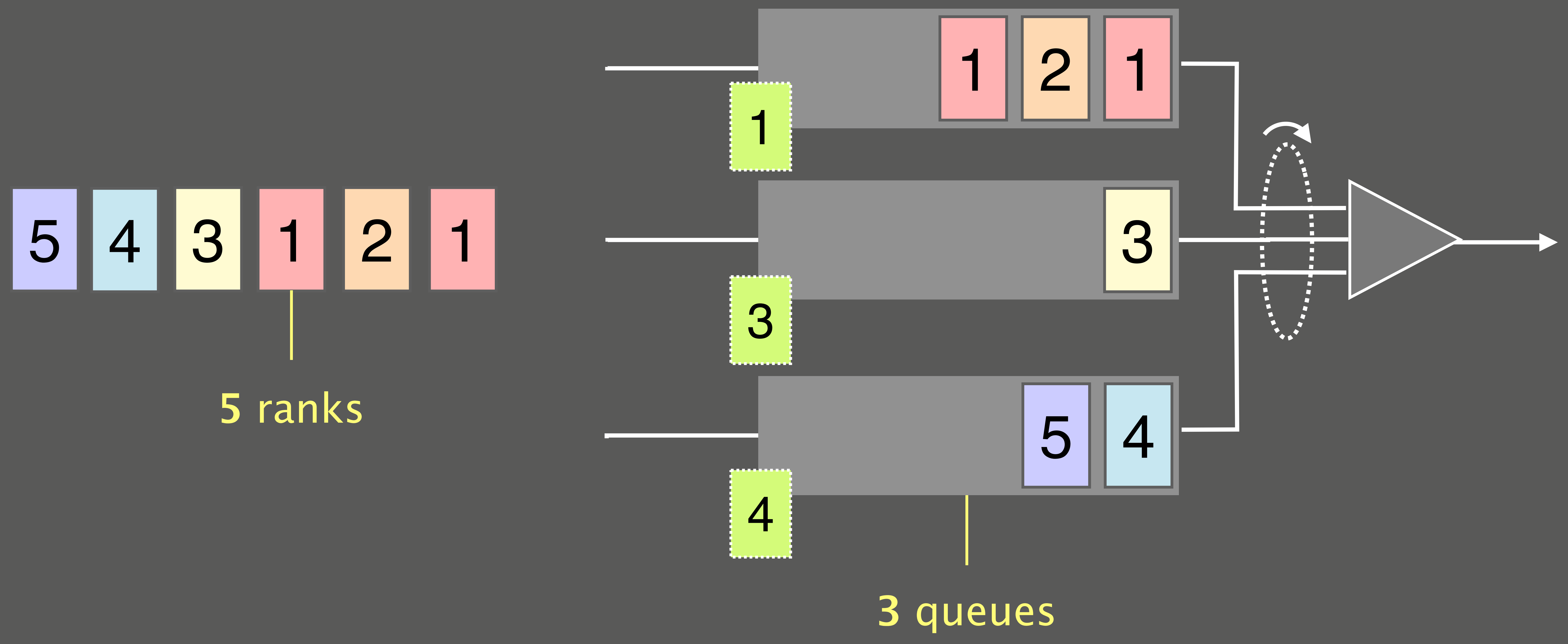
queue mapping policy: enqueues if rank ≥ queue bound $_i$ when scanning bottom-up

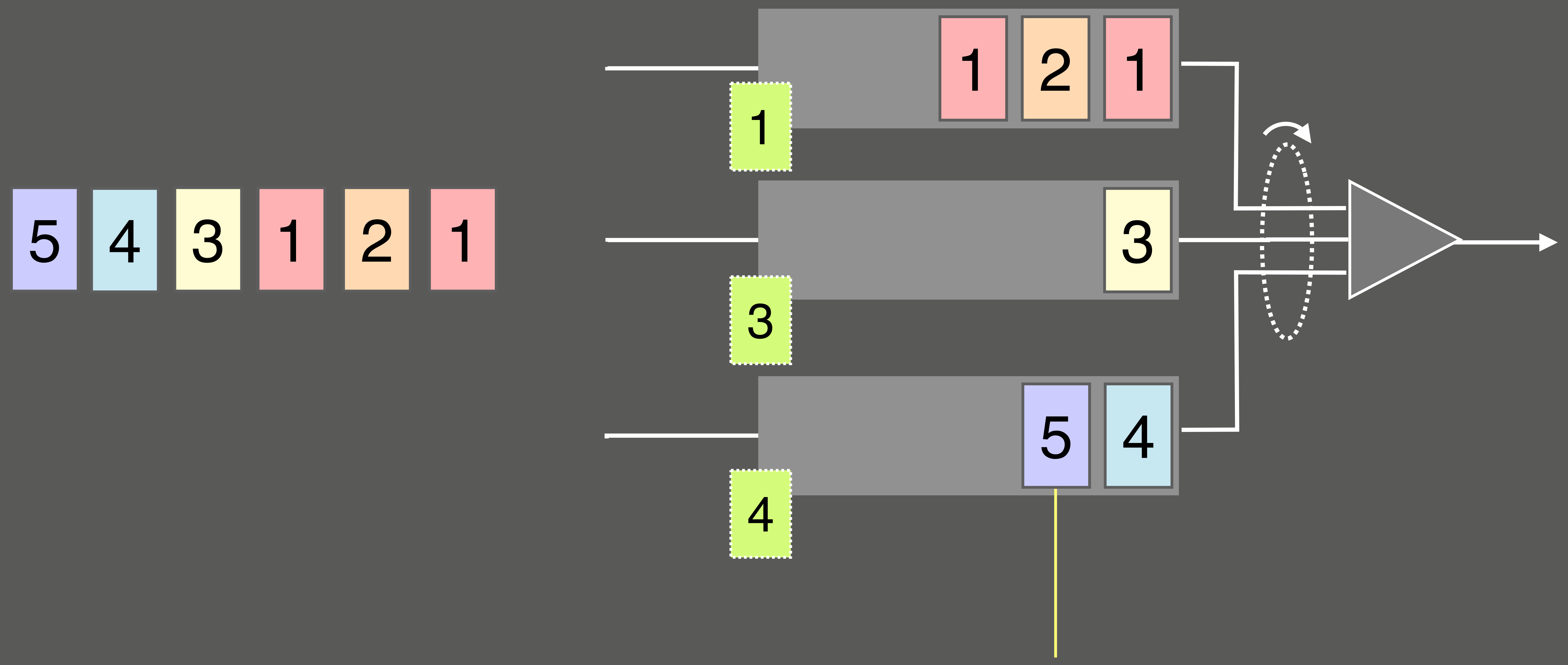
If there are as many queues as ranks,
SP-PIFO is equivalent to PIFO



In practice though,

number of ranks >> number of queues



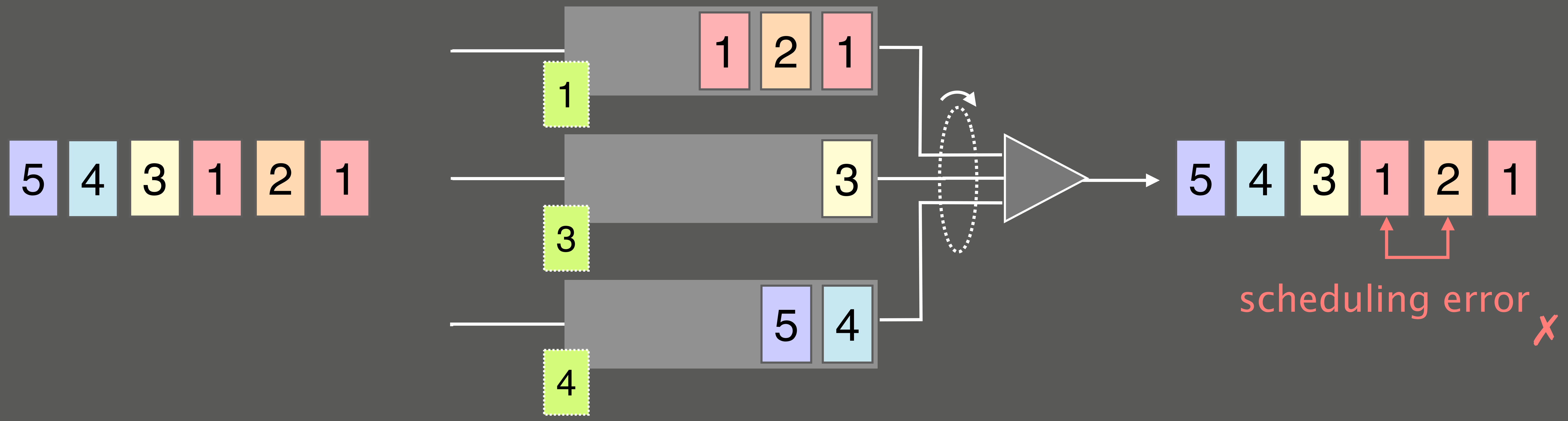


Different ranks share the same queues

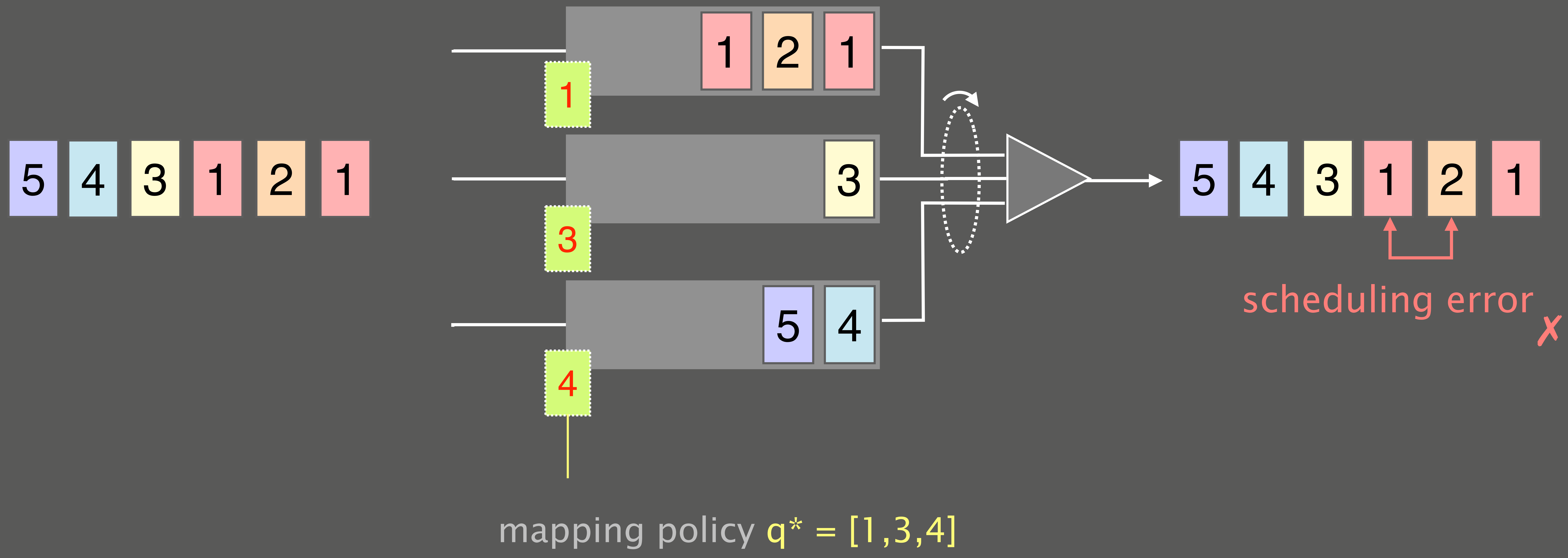
ranks {1,2} and ranks {4,5}

We can have

scheduling errors

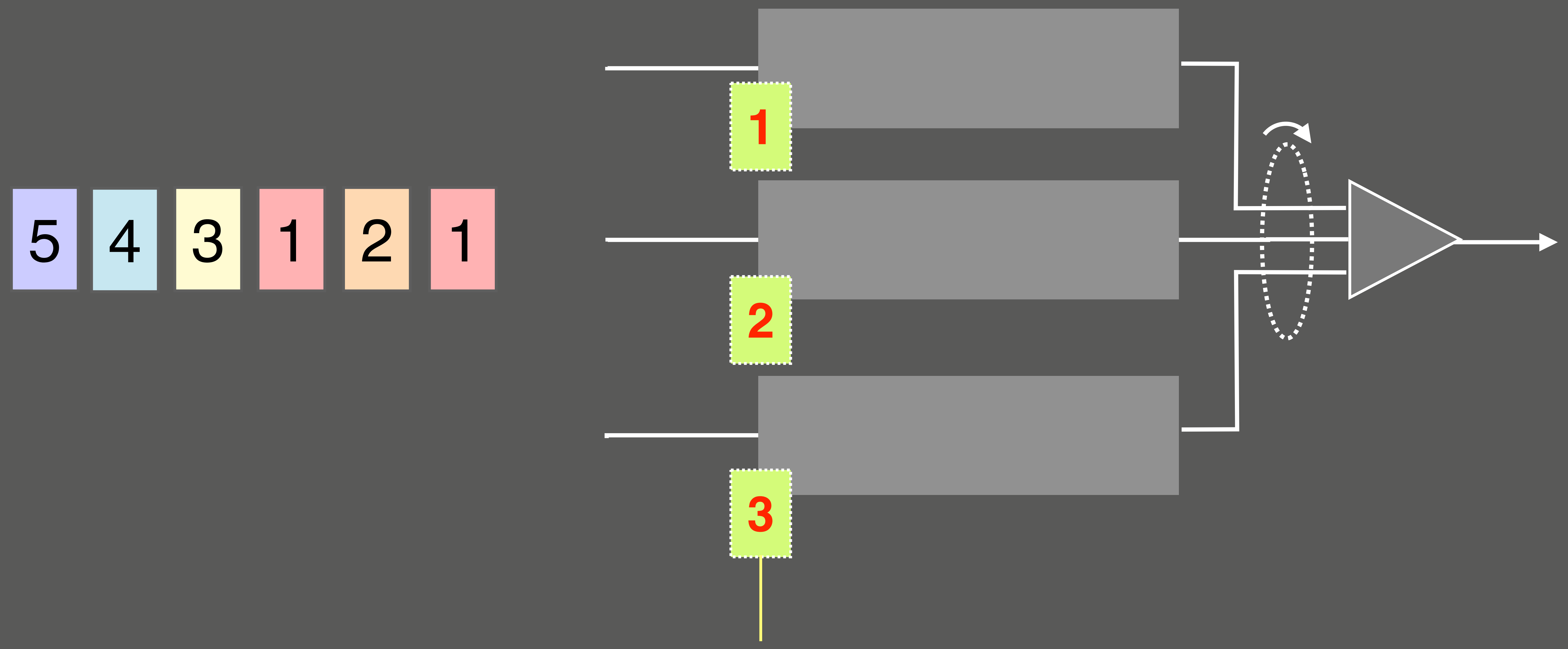


5 4 3 1 2 1
1 2 1
1
3
3
5 4
4
5 4 3 1 2 1
scheduling error
✗


5 4 3 1 2 1
1 2 1
1
3
3
5 4
4
5 4 3 1 2 1
scheduling error
mapping policy q* = [1,3,4]

We can minimize the number of scheduling errors

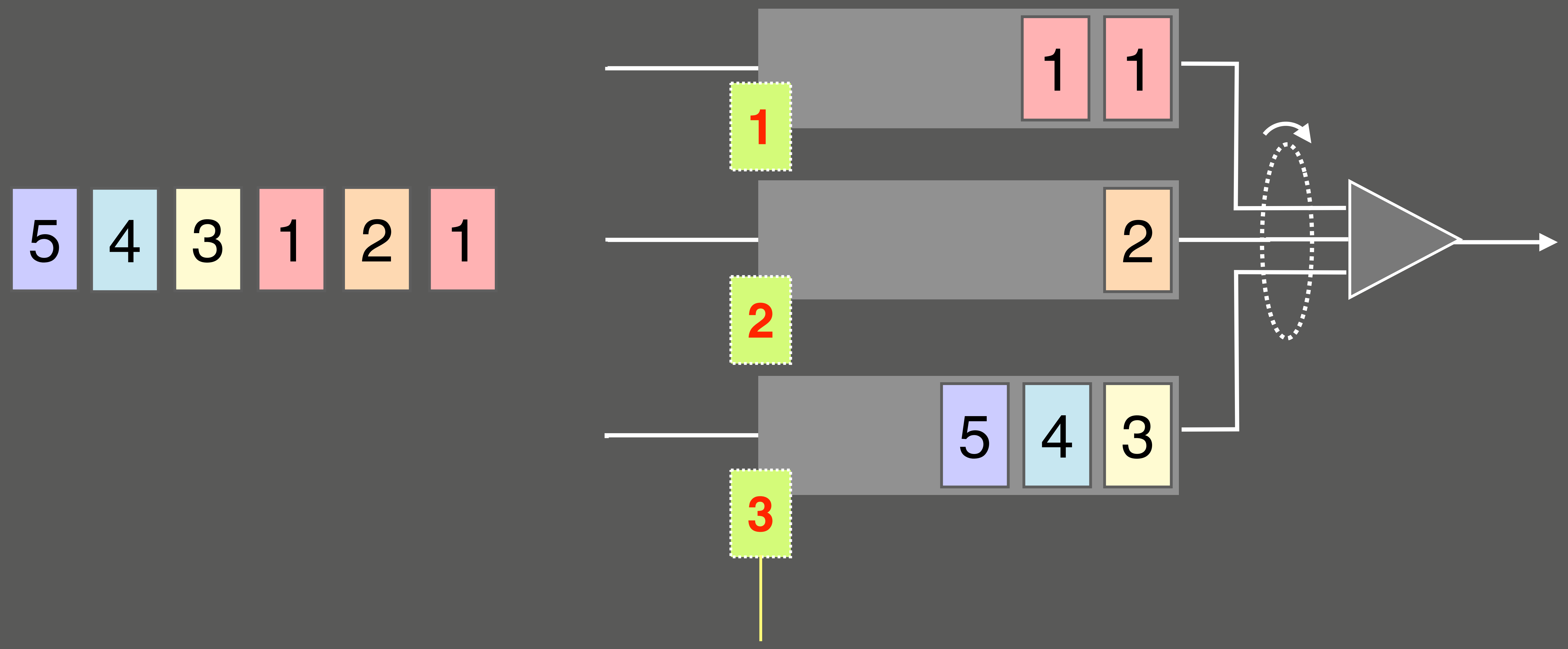
by dynamically adapting the mapping policy



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

We can minimize the number of scheduling errors

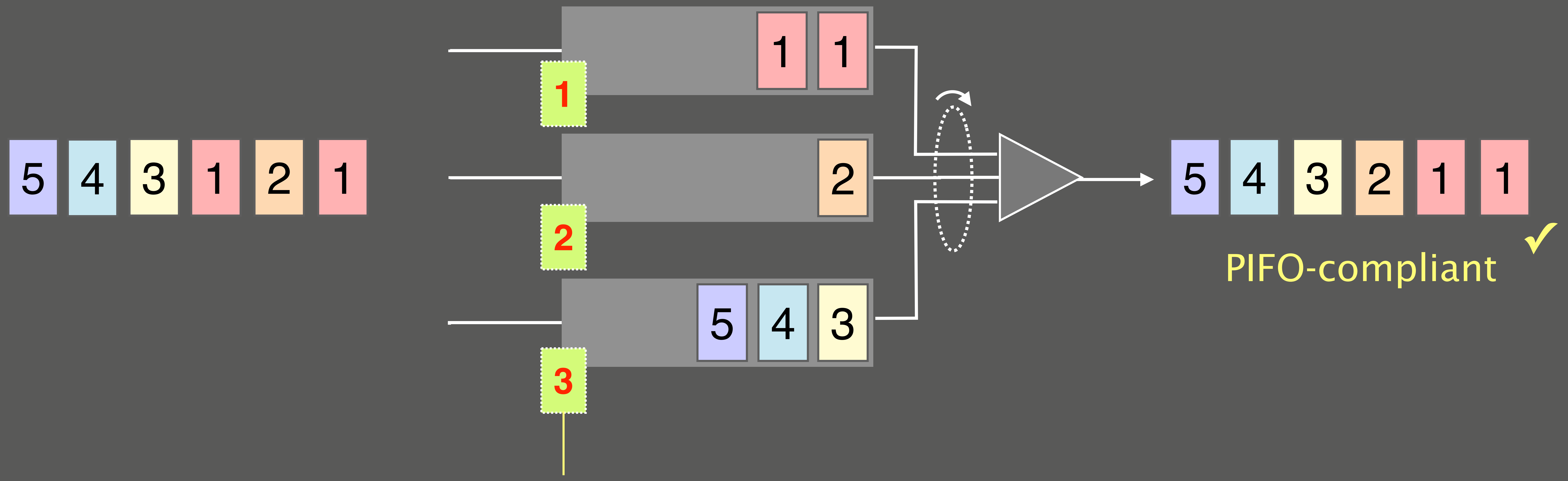
by dynamically adapting the mapping policy



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

We can minimize the number of scheduling errors

by dynamically adapting the mapping policy



mapping policy q* = [1,2,3] ✓

How can we design a mapping strategy that minimizes scheduling errors?

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

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

optimal
mapping policy

expected loss across all ranks
"unpifoness"

Solving this optimization problem exactly is intractable unfortunately

unknown packet rank distributions

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

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

$$q^* = \arg\min_{q \in \mathcal{Q}} \mathcal{U}(\mathcal{P}, q)$$

enqueued packets ($\mathcal{P}$)

online mapping policy (q^*)

estimated unpifoness ($\mathcal{U}$)

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

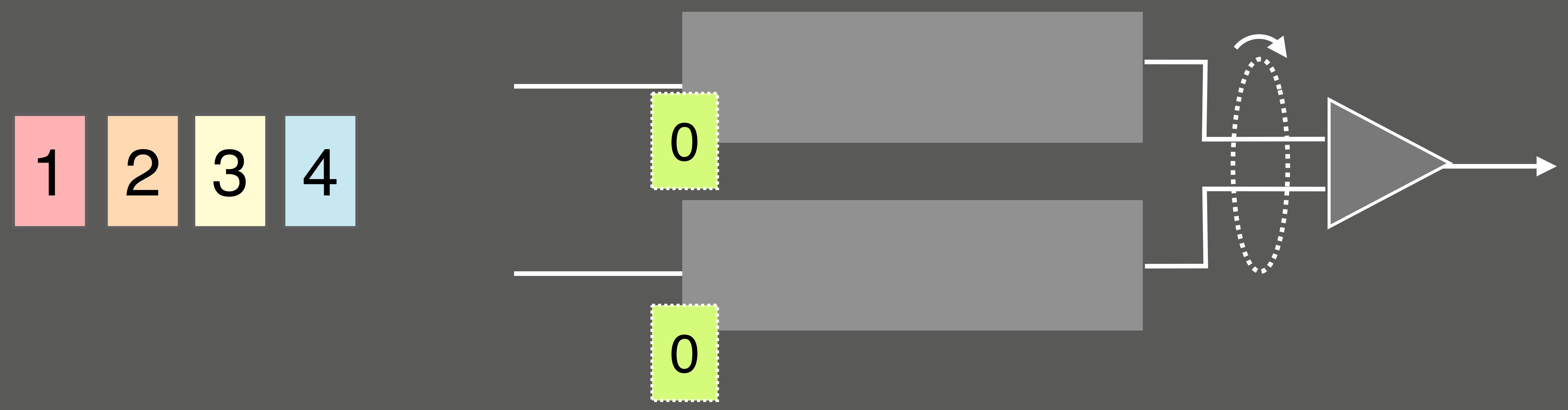
phase 1	gradually map higher-priority packets
push-up	to higher-priority queues
	concentrates scheduling errors
	in the highest-priority queue

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

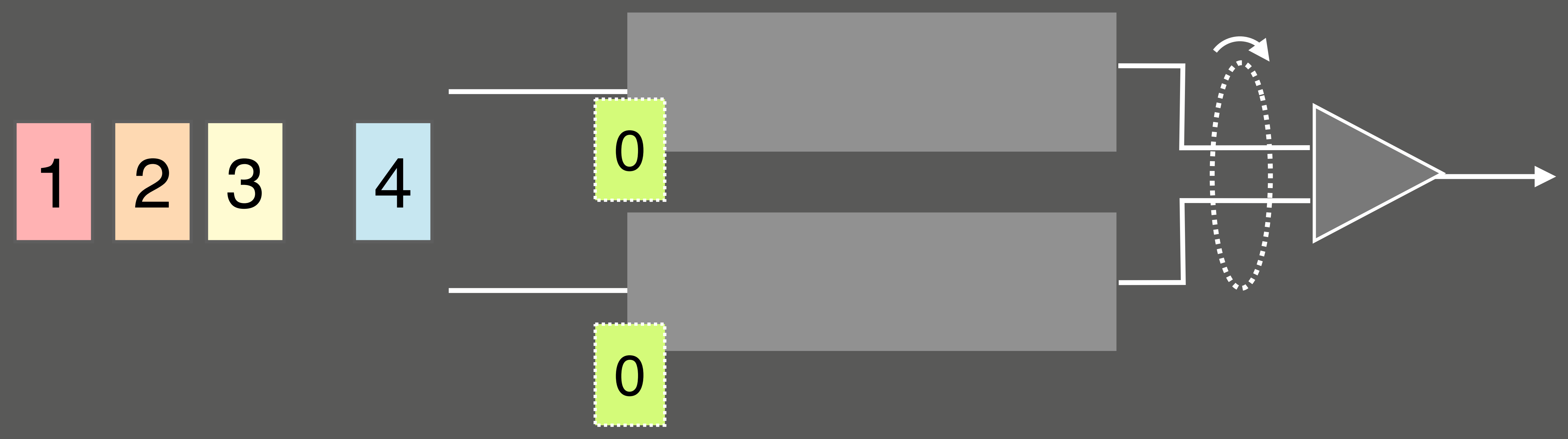
phase 1 push-up	gradually map higher-priority packets to higher-priority queues
	concentrates scheduling errors in the highest-priority queue

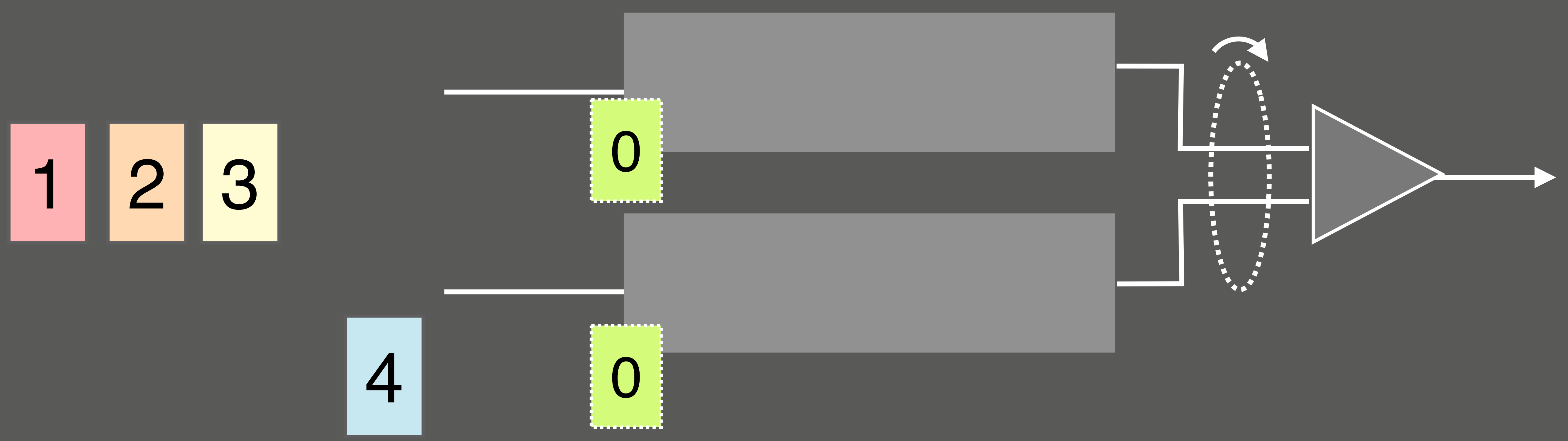
upon scheduling error…

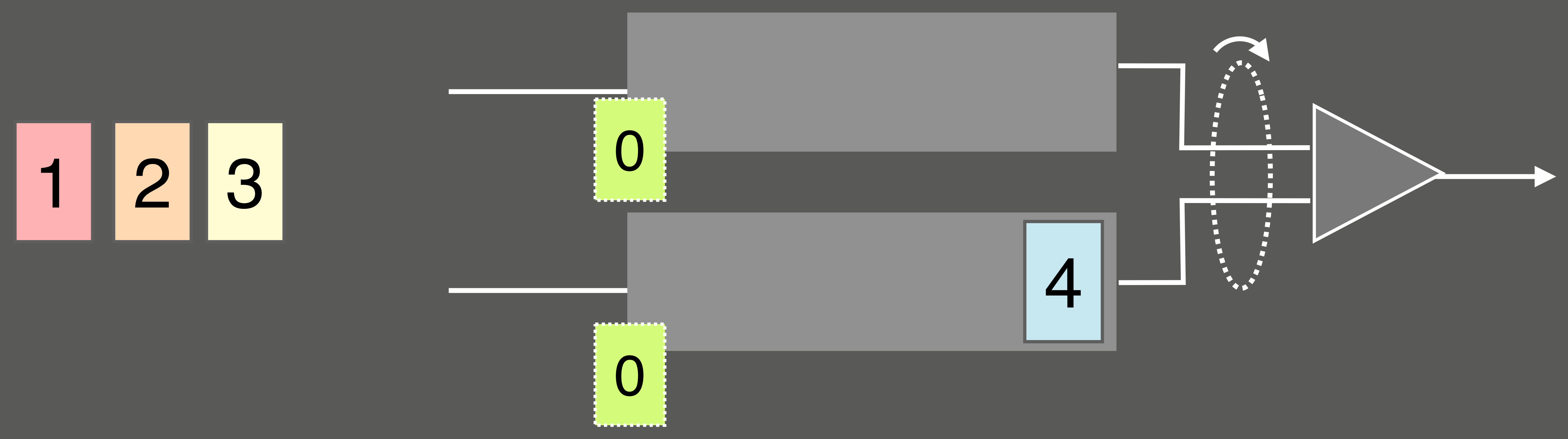
phase 2 push-down	shift lower-priority packets to lower-priority queues

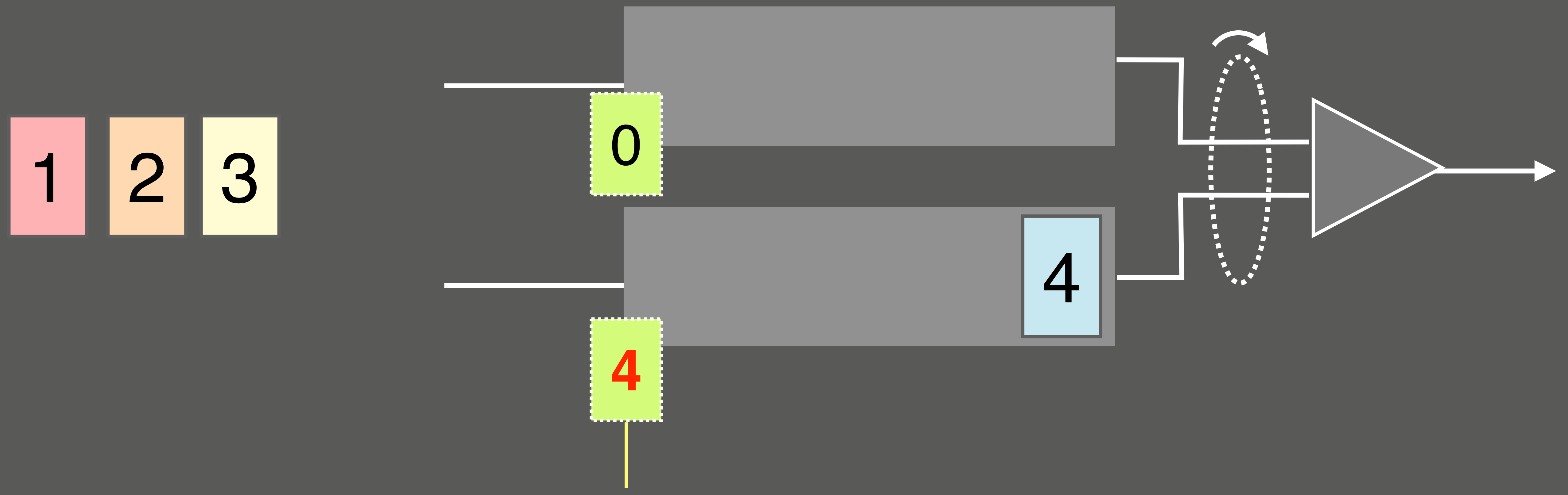


1
2
3
4
0
0

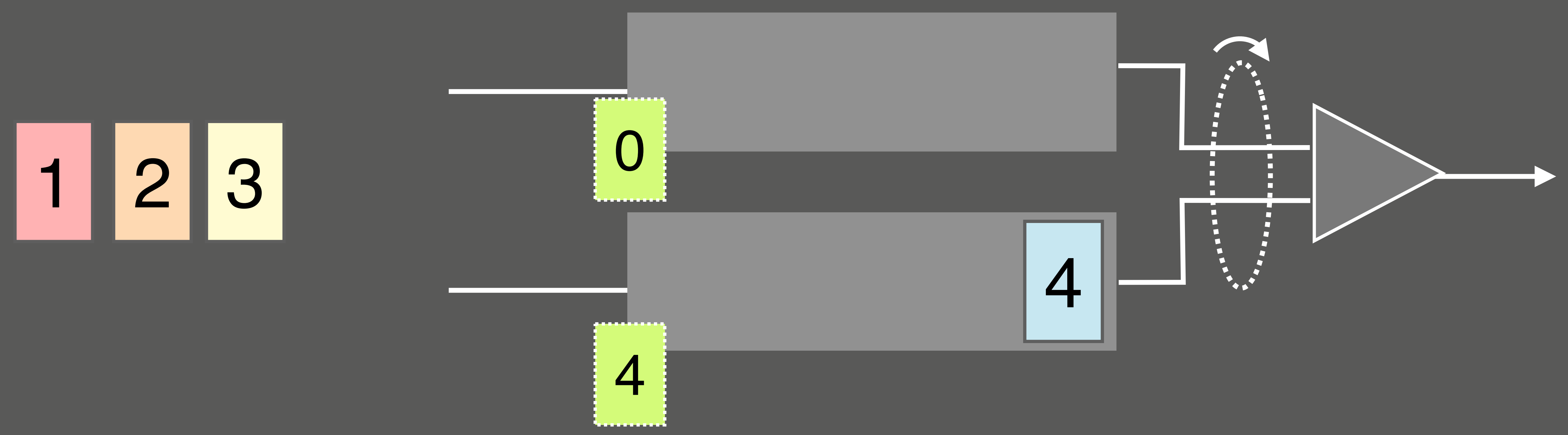

1
2
3
4
0
0

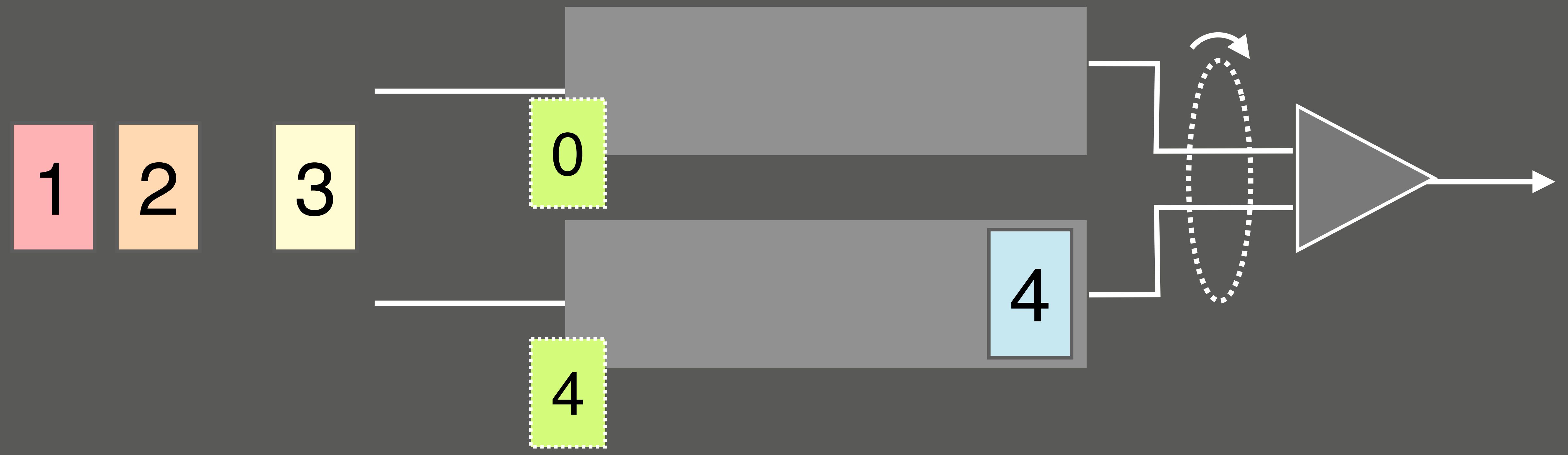

1
2
3
4
0
0

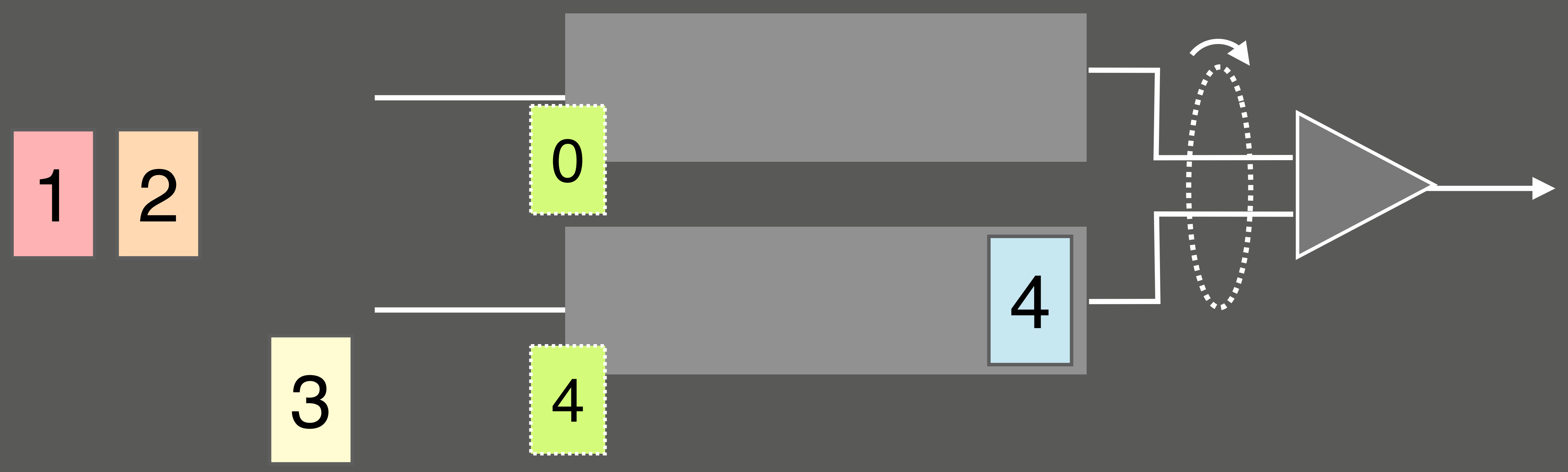

1
2
3
0
0
4

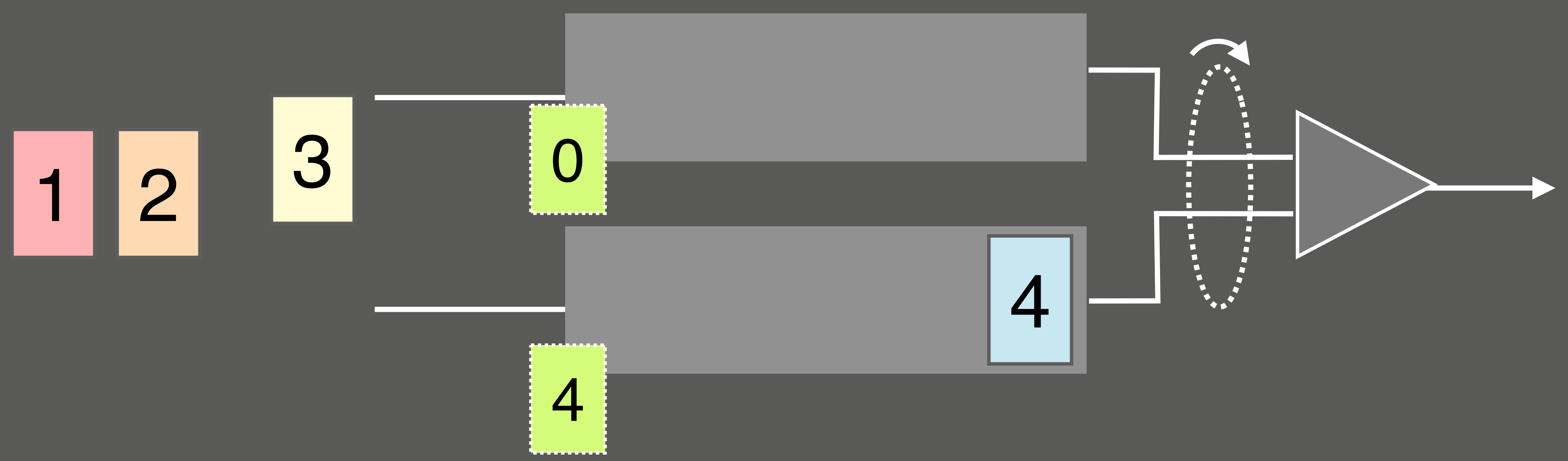


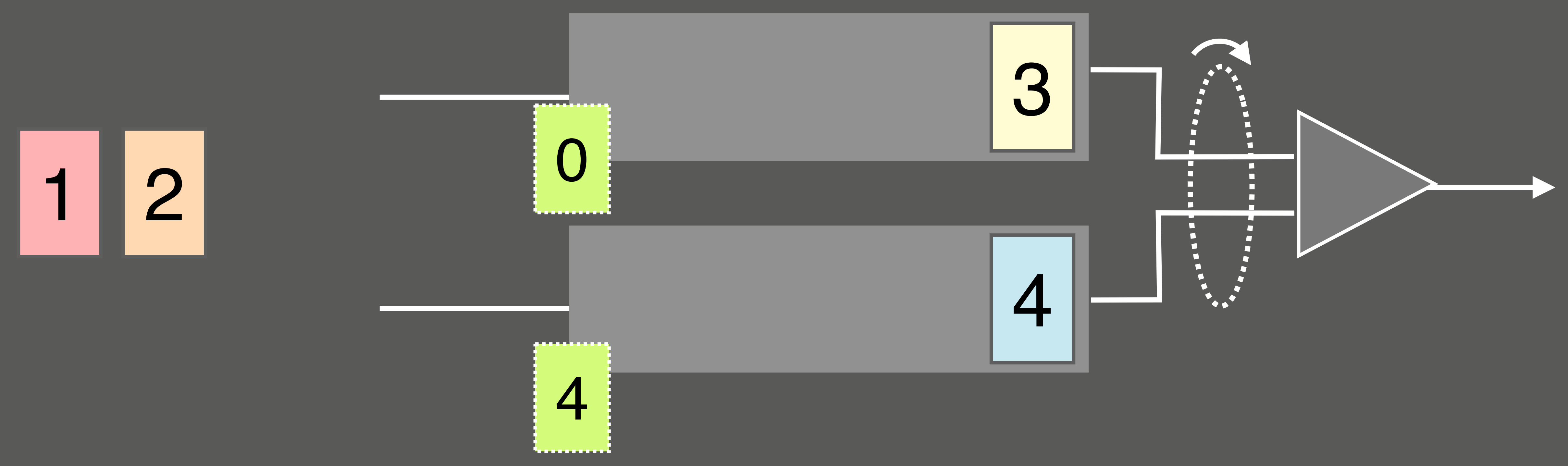
"push-up"   increase queue bound $_i$ to rank(enqueued packet)


1
2
3
0
4
4

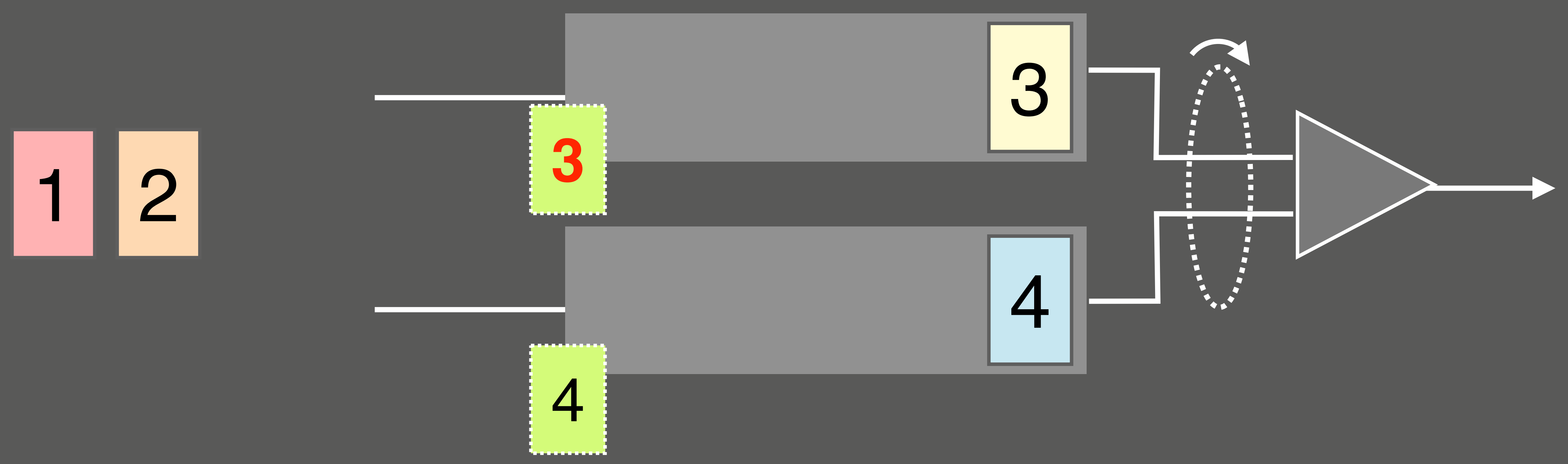

1
2
3
0
4
4

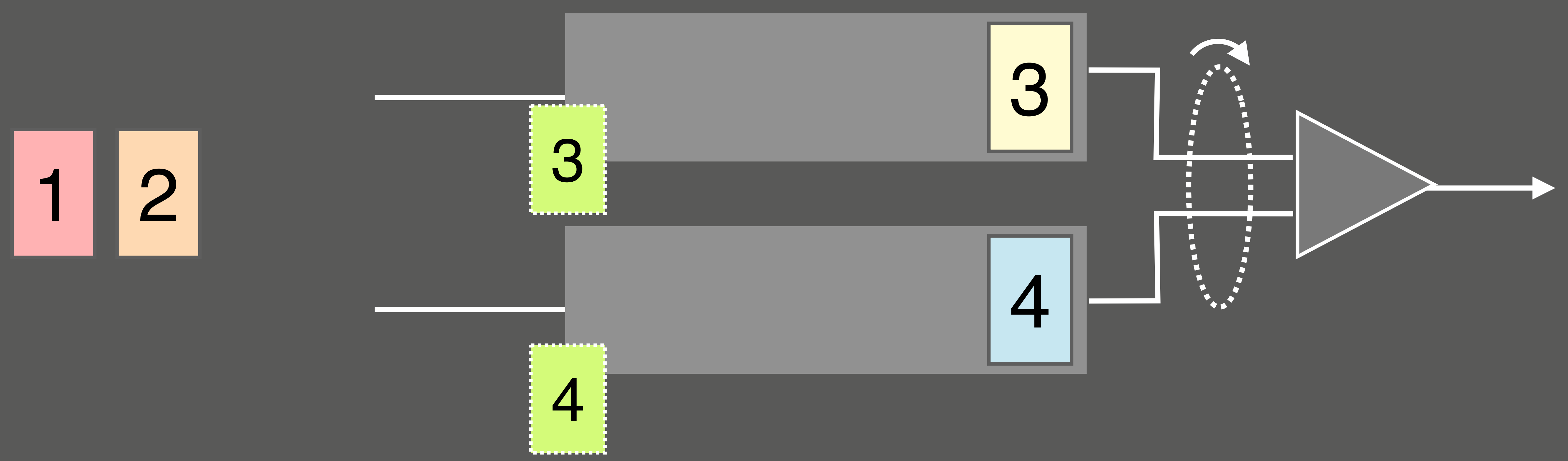

1
2
3
0
4
4

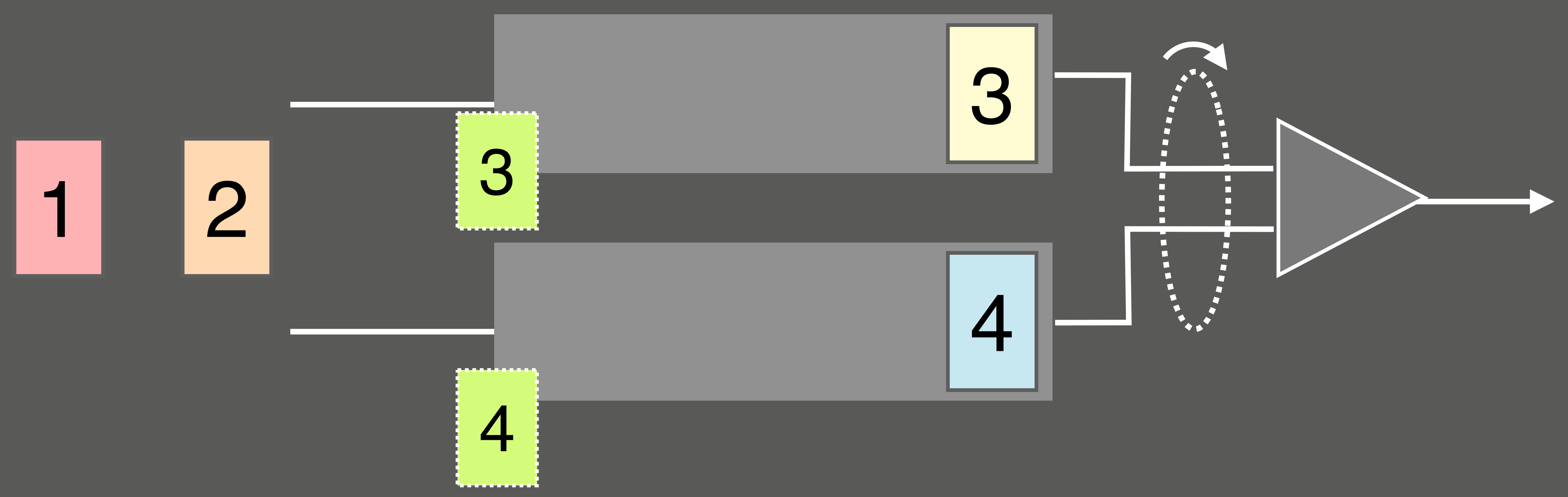

1
2
3
0
4
4

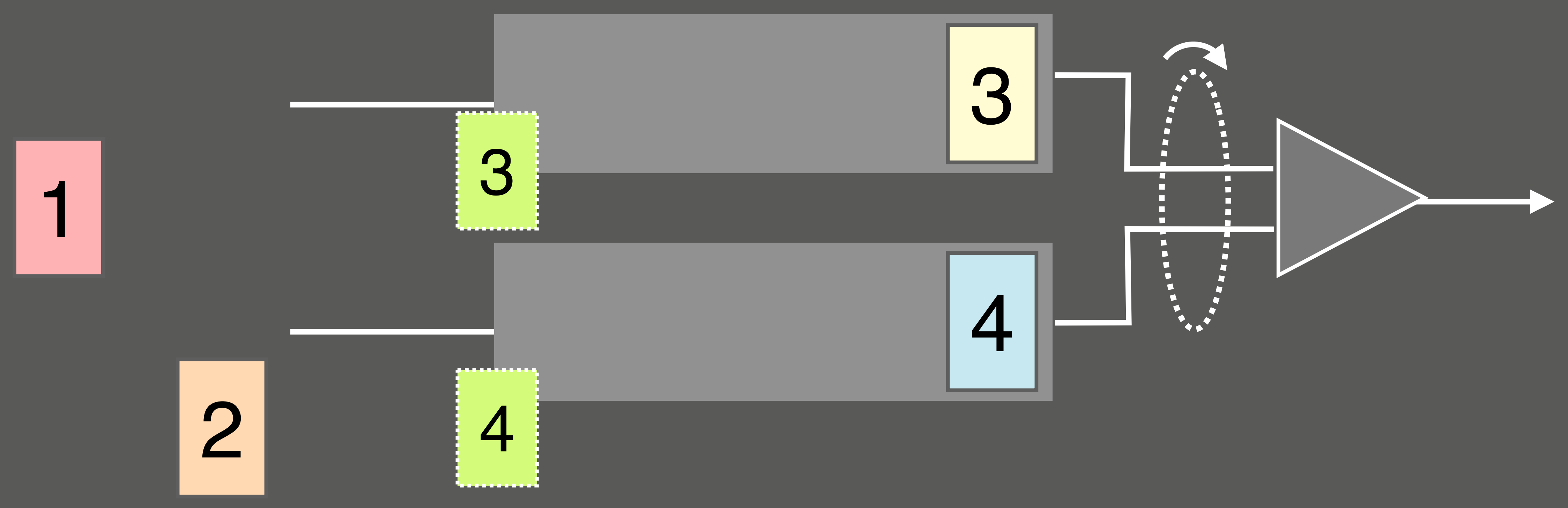


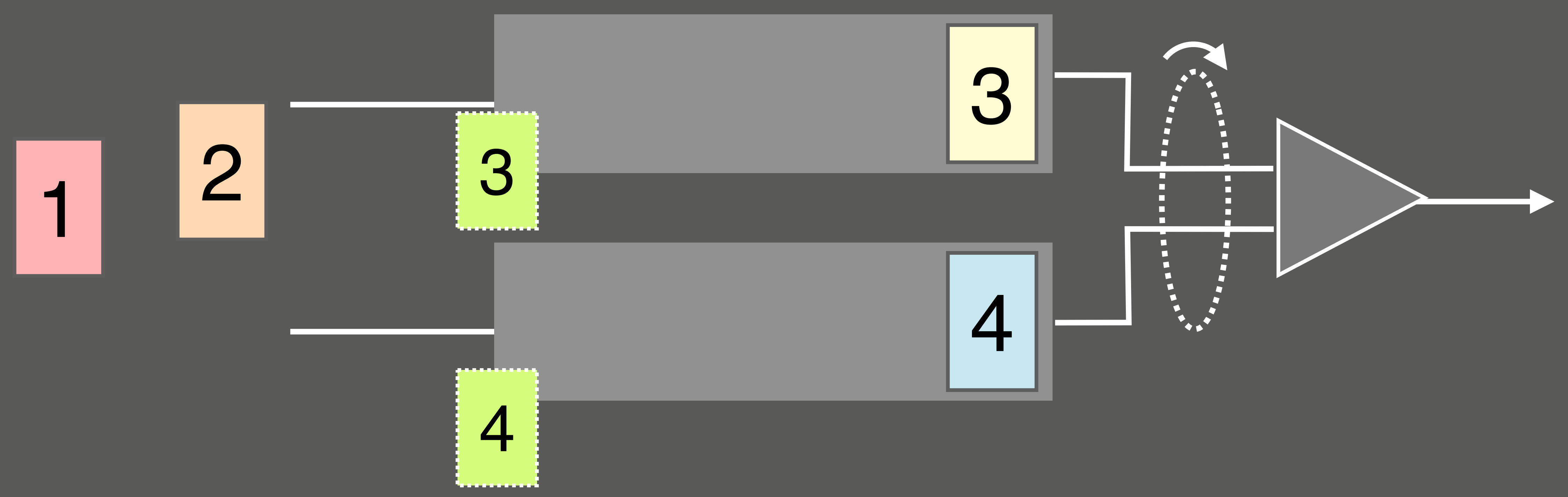
1
2
0
4
3
4


1
2
3
3
4
4

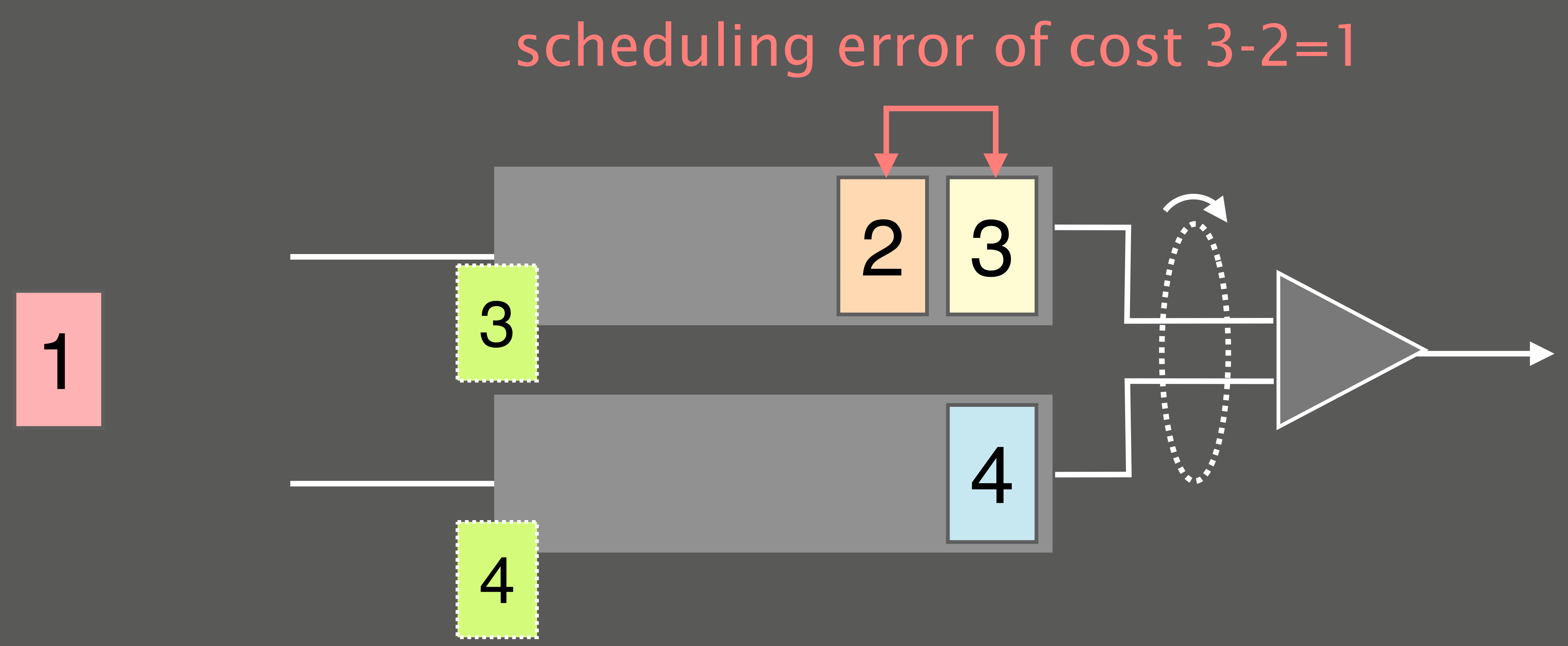

1
2
3
3
4
4

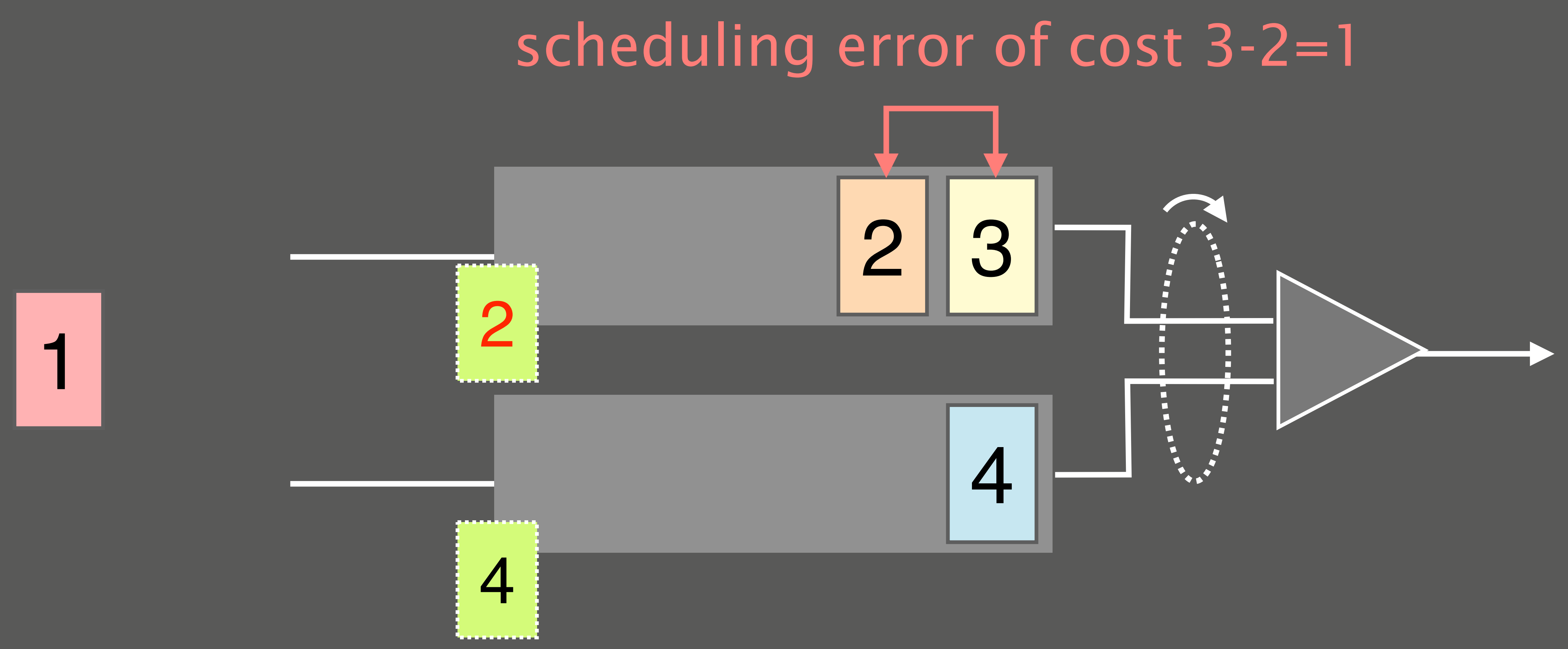

1
2
3
4
3
4

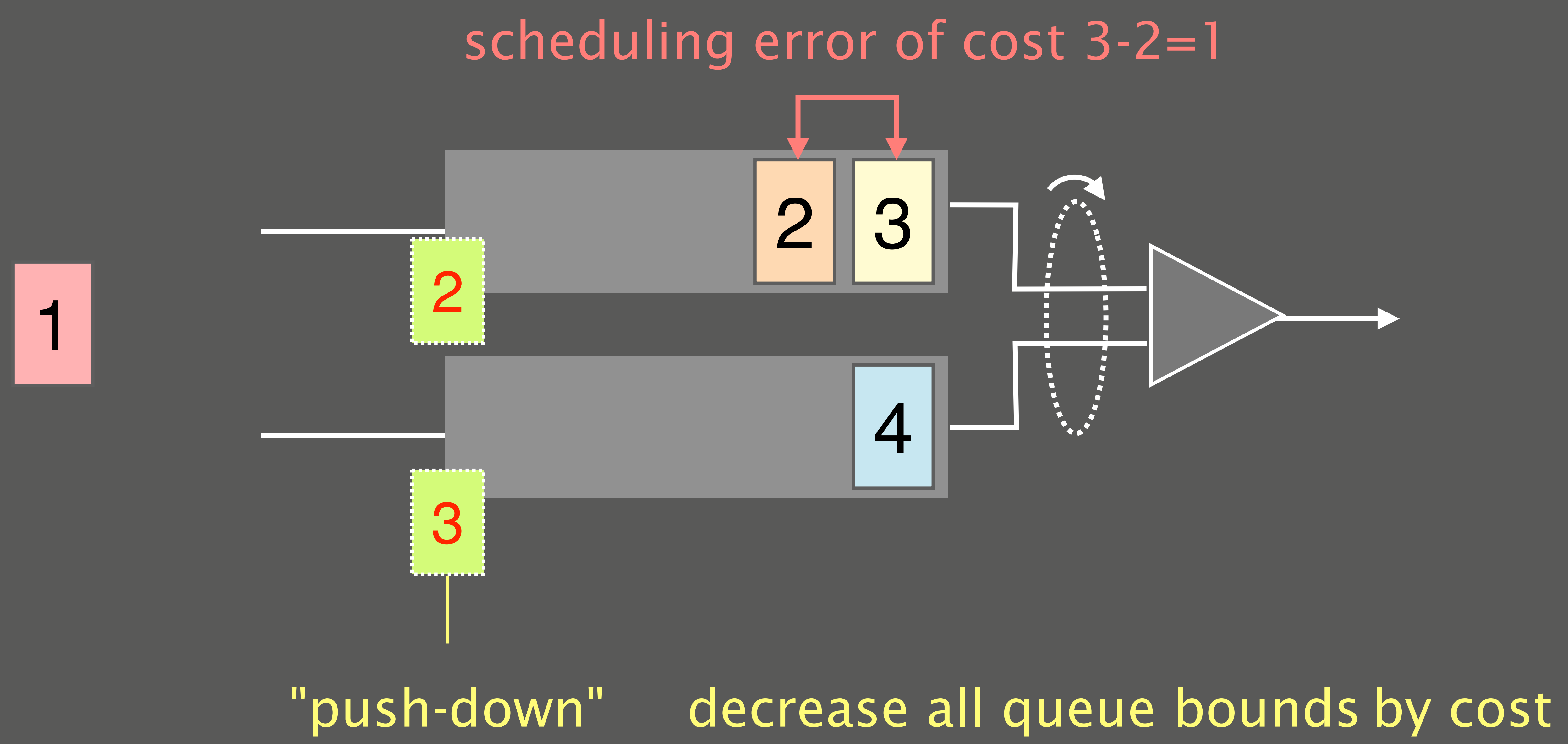

1
2
3
4
3
4



1
2
3
4
3
4


scheduling error of cost 3-2=1
1
3
4
2
3
4


scheduling error of cost 3-2=1
1
2
2
3
4
4


scheduling error of cost 3-2=1
2
3
1
2
4
3
"push-down"
decrease all queue bounds by cost

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

Adaptation strategy

how does it work?

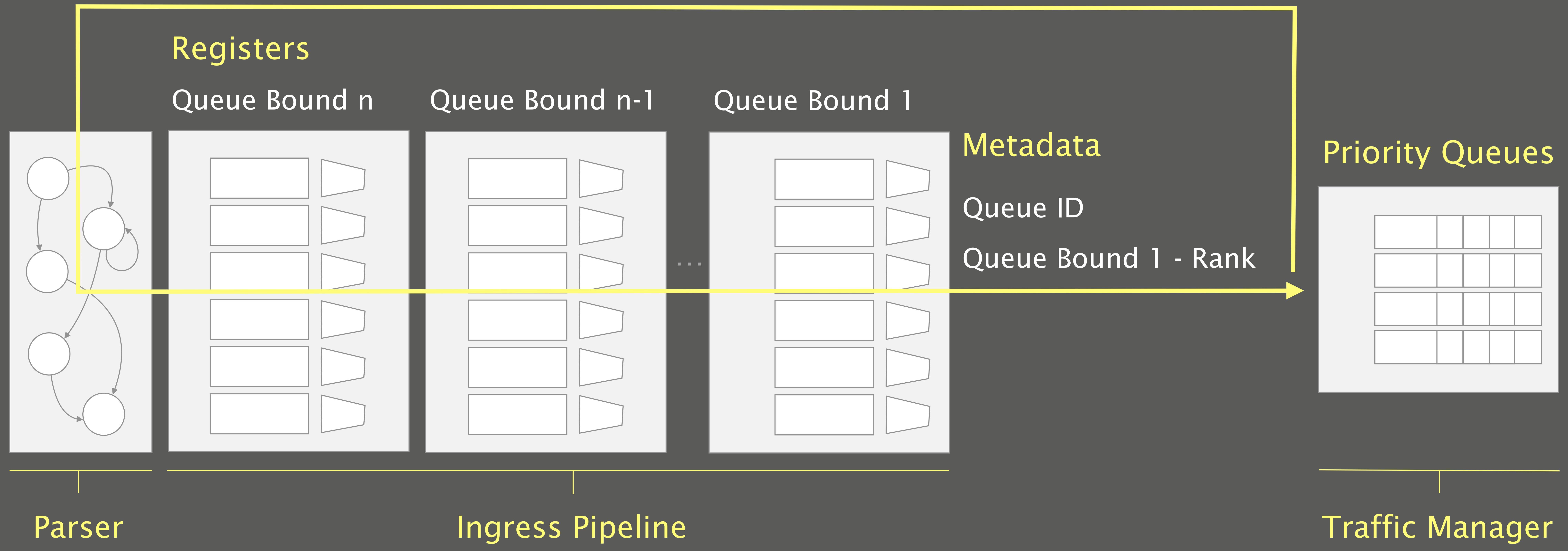
2 Implementation

how can it be deployed?

Evaluation

how well does it perform?

We managed to program SP-PIFO on existing programmable data planes (Intel 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

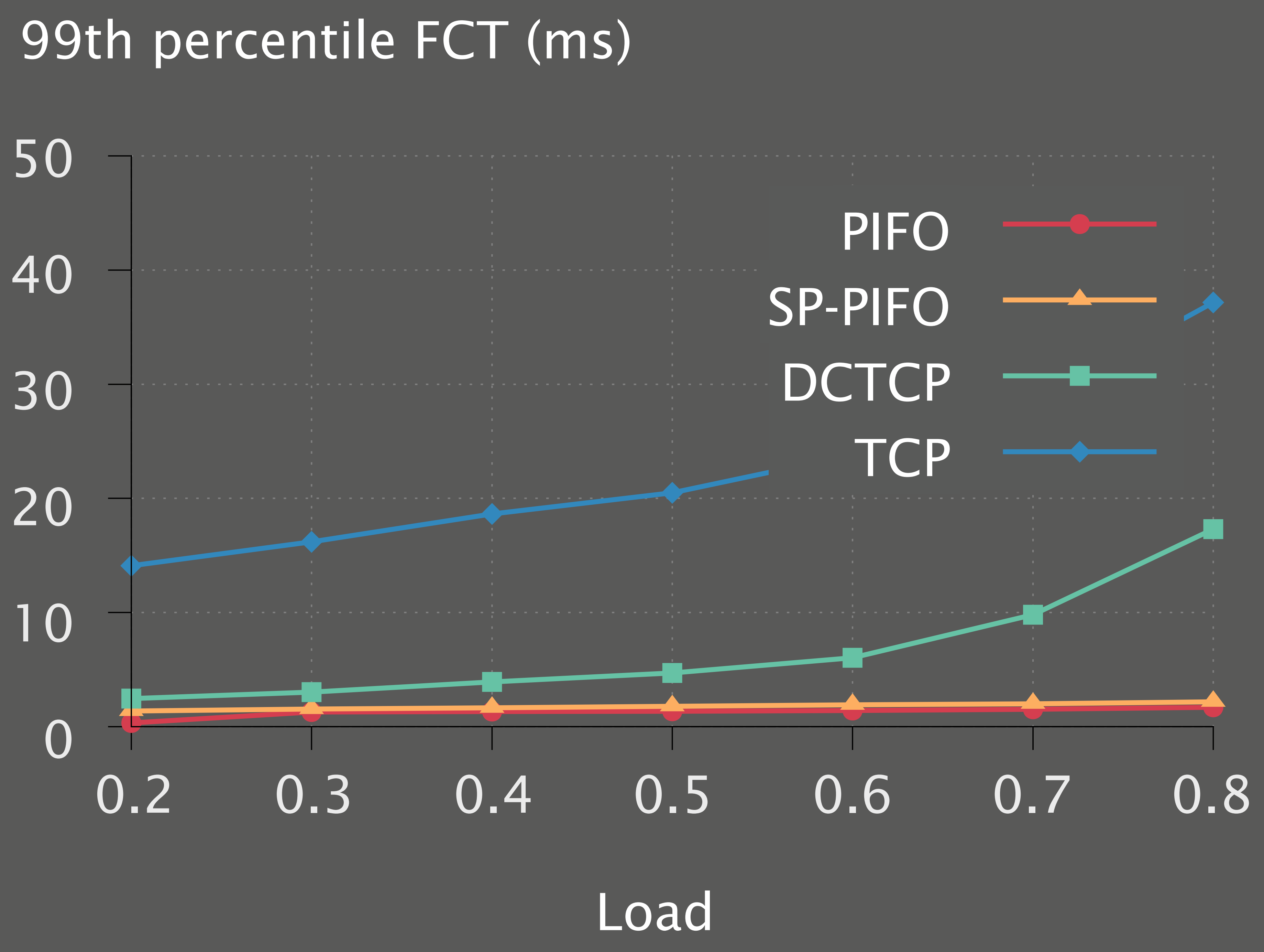
Start-Time Fair Queuing (32 queues)

Ranks based on a fluid model

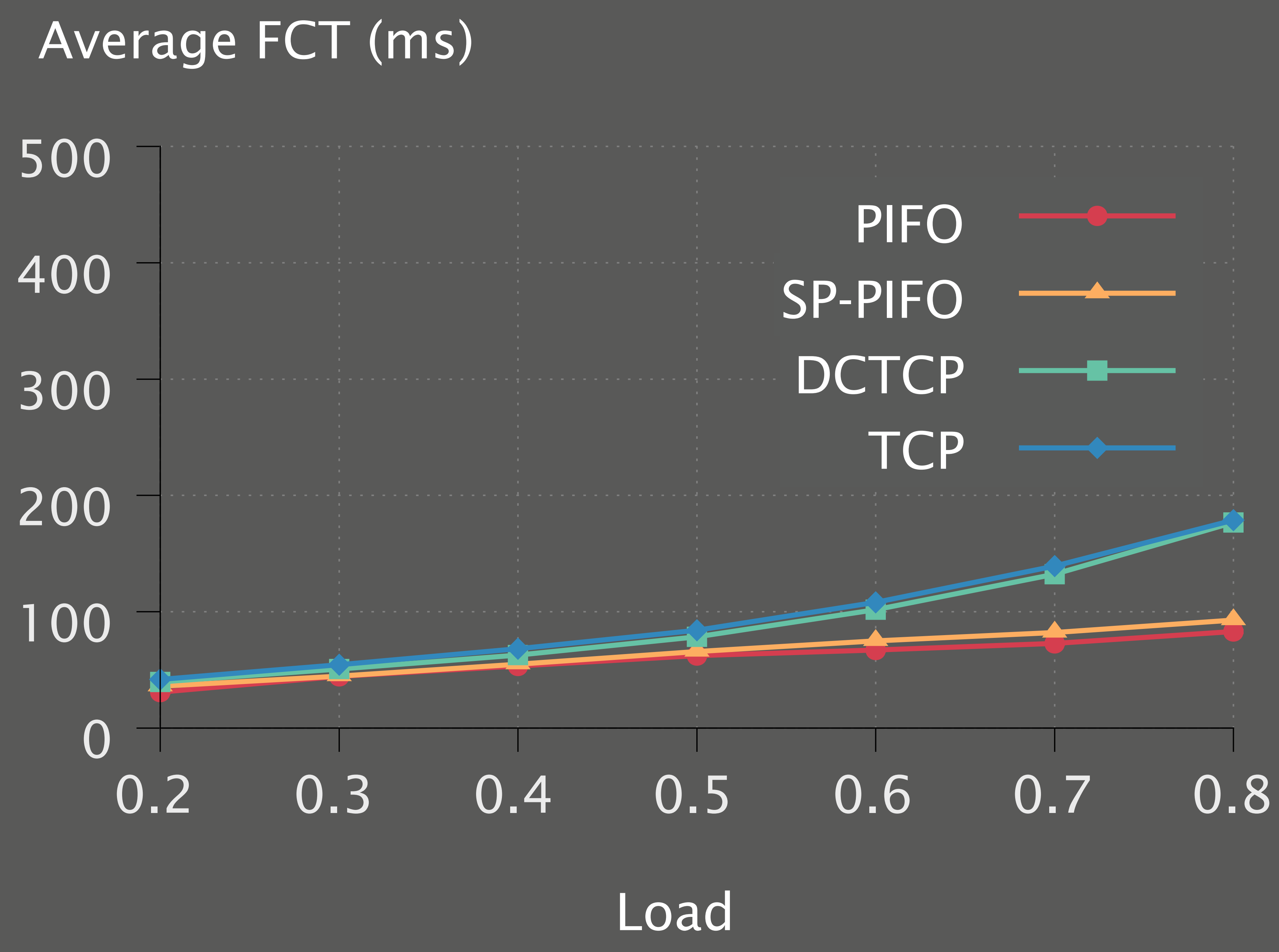
Packet-level simulator	Netbench [SIGCOMM 2017]
Topology	We use a leaf-spine topology with: 144 servers, 1/4 Gbps links
Realistic workloads	pFabric web-search workload

SP-PIFO closely approximates pFabric

minimizing FCTs for both small and big flows

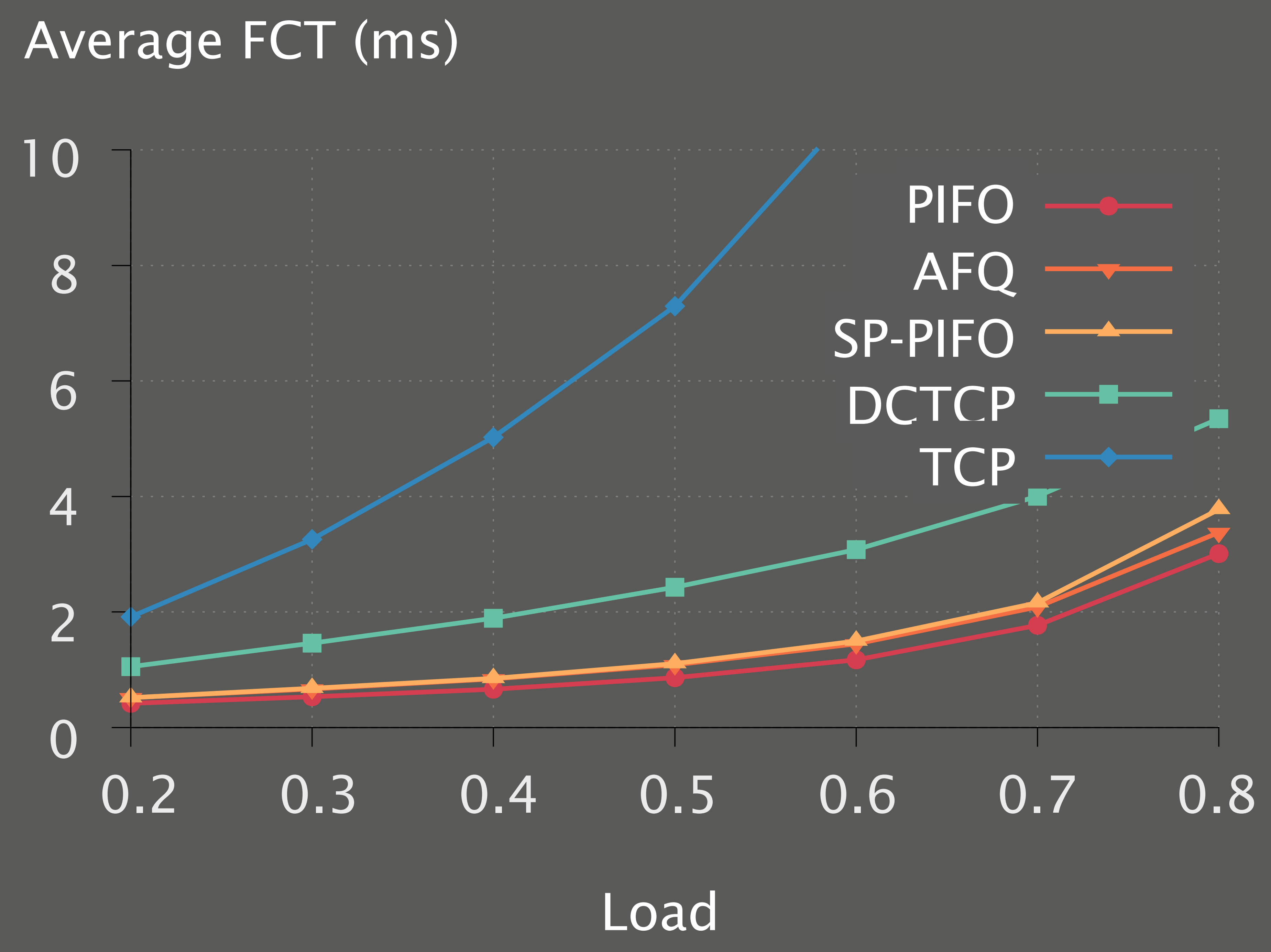


Small flows <100KB

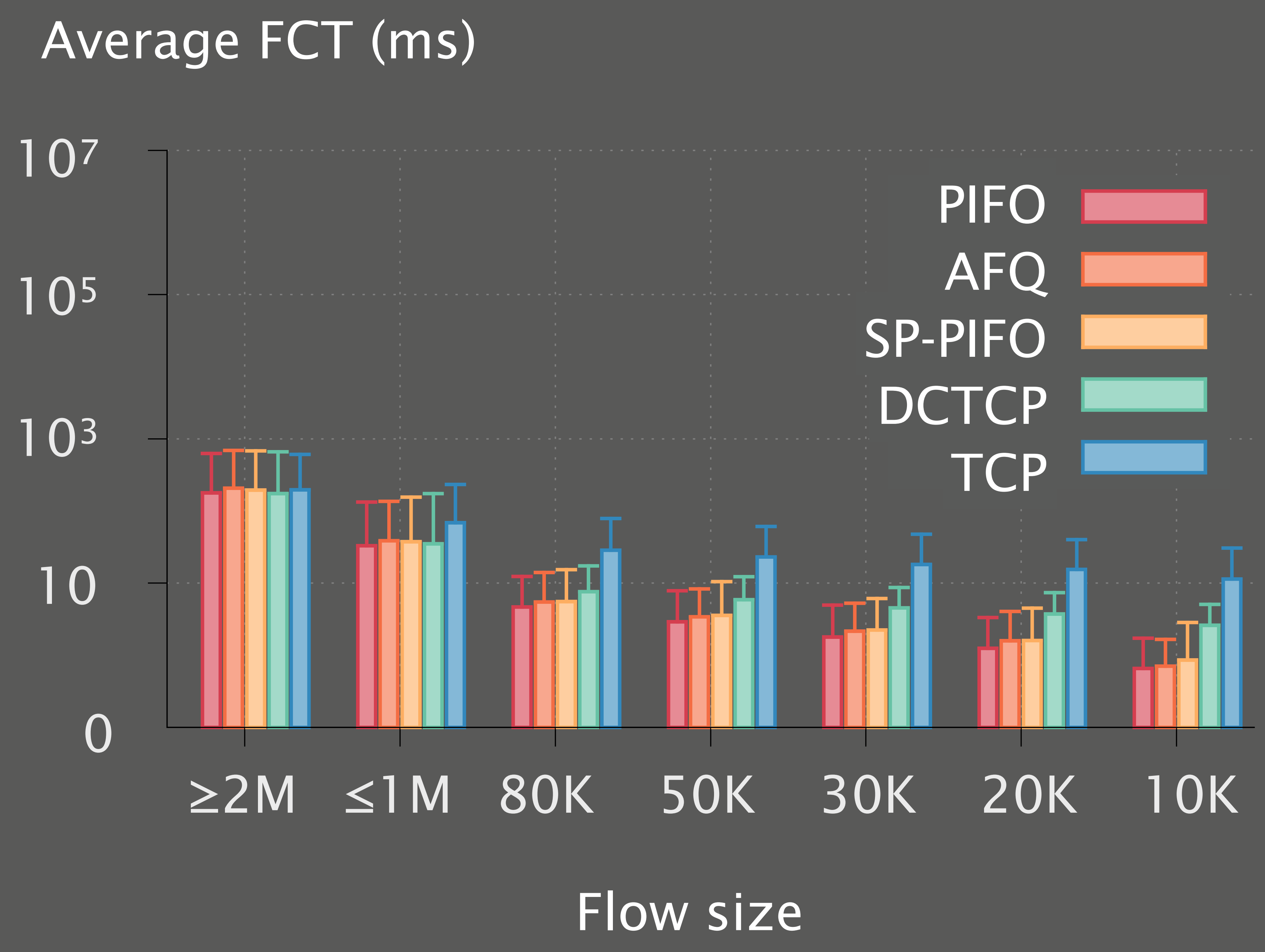


Big flows ≥1MB

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?

Check our paper out for *much* more info…

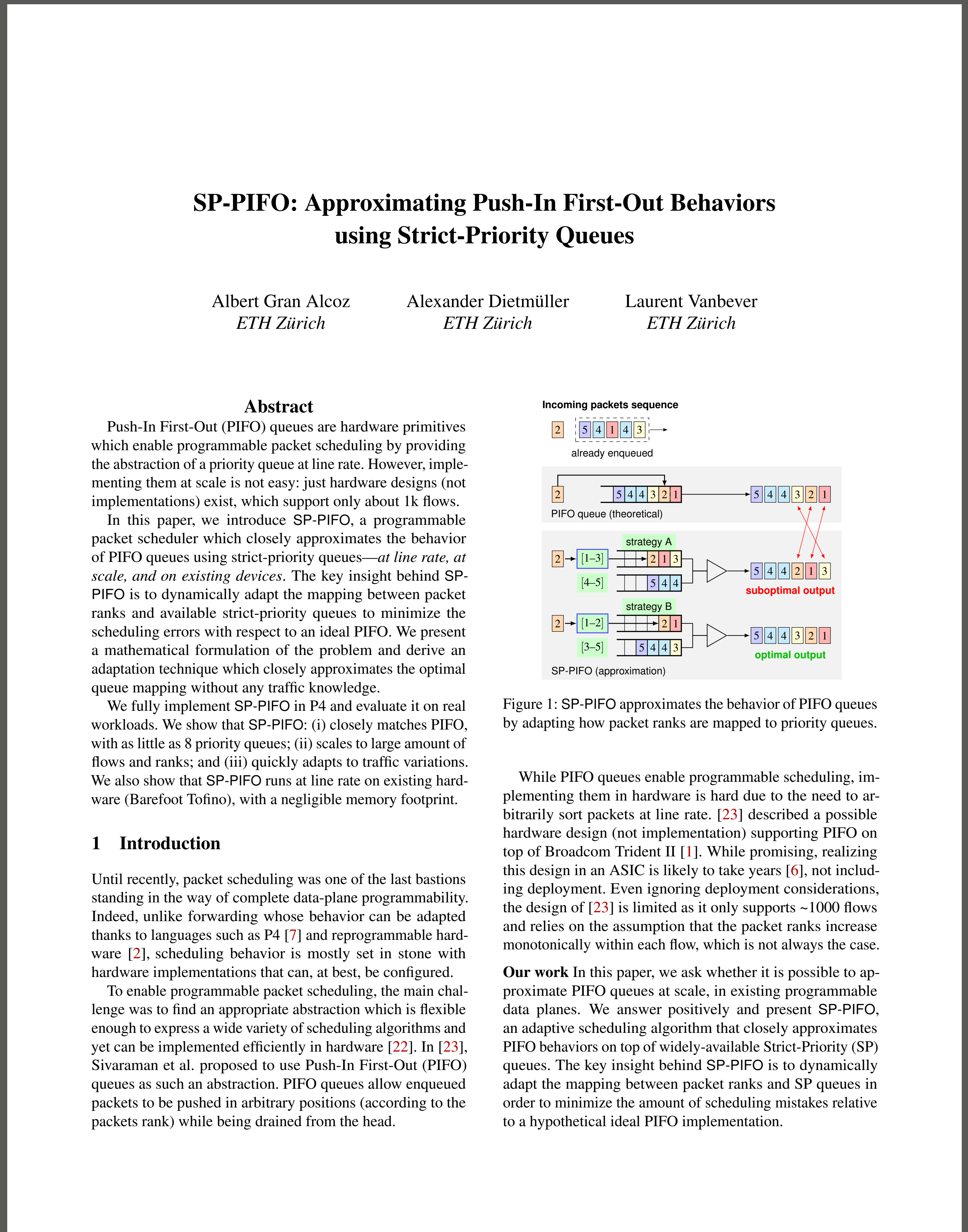
SP-PIFO characterization, comparison with gradient

Hardware evaluation on Intel Tofino

Limitations and future improvements

sp-pifo.ethz.ch

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.

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