Programmable Networks Lecture 6 – T4P4S & Traffic Management

Sándor Laki, PhD

Communication Networks Laboratory

Dept. of Information Systems, Faculty of Informatics

ELTE Eötvös Loránd University

lakis@elte.hu

http://lakis.web.elte.hu

this week

- T4P4S – a multi target P4 compiler
- Traffic Management – AQM – Drop policies in P4
- Traffic Management – Per Packet Value Core Stateless Resource Sharing

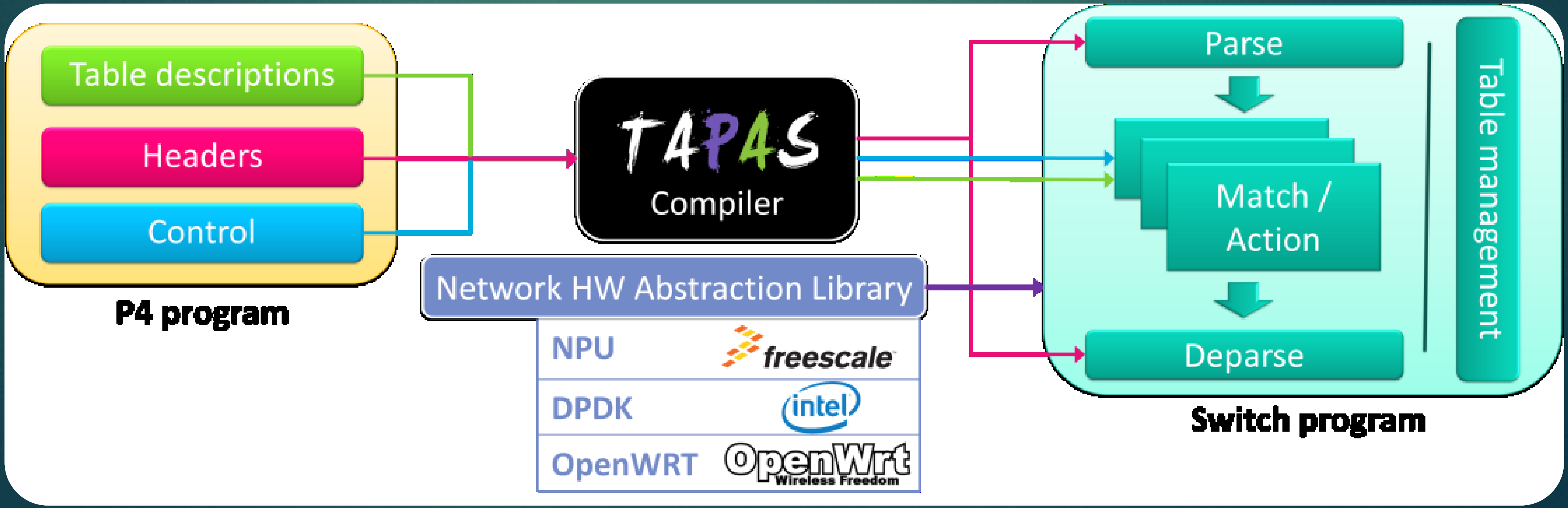
T4P4S

P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, S. Laki: „T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors”, Proceedings of IEEE International Conference on High Performance Switching and Routing (HPSR 2018), 17-20 June, 2018 – Bucharest, Romania

Goals of T4P4S

- Extended data plane programmability
  - P4 code as a high level abstraction
- Support of different hardware targets
  - CPUs, NPUs, FPGA, etc.

- Create a compiler that separates hardware
  dependent and independent parts
  - Easily retargetable P4 compiler

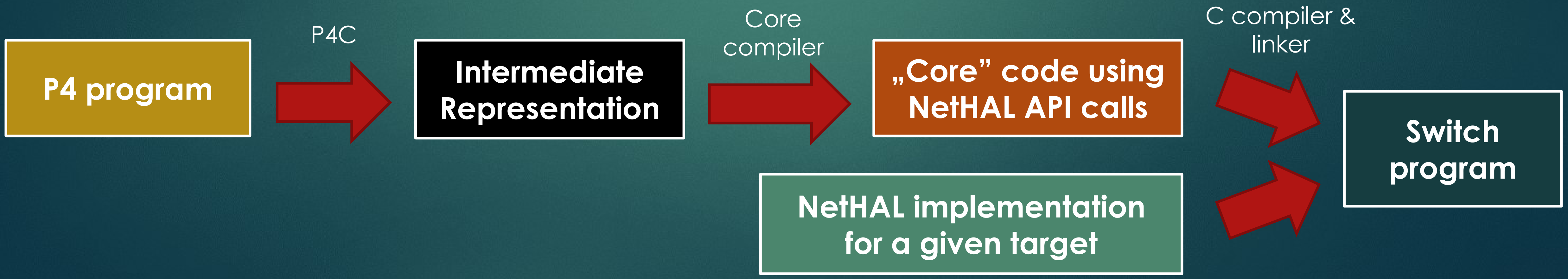


Multi-target Compiler Architecture for P4

1. **Hardware-independent „Core”**
   - Using an Intermediate Representation (IR)
   - Compiling IR to a hardware independent C code with NetHAL calls
2. **Hardware-dependent „Network Hardware Abstraction Layer” (NetHAL)**
   - Implementing primitives that fulfill the requirements of most hardware
   - A static and thin library
   - Written by a hardware expert (currently available for DPDK, ODP, native Linux)

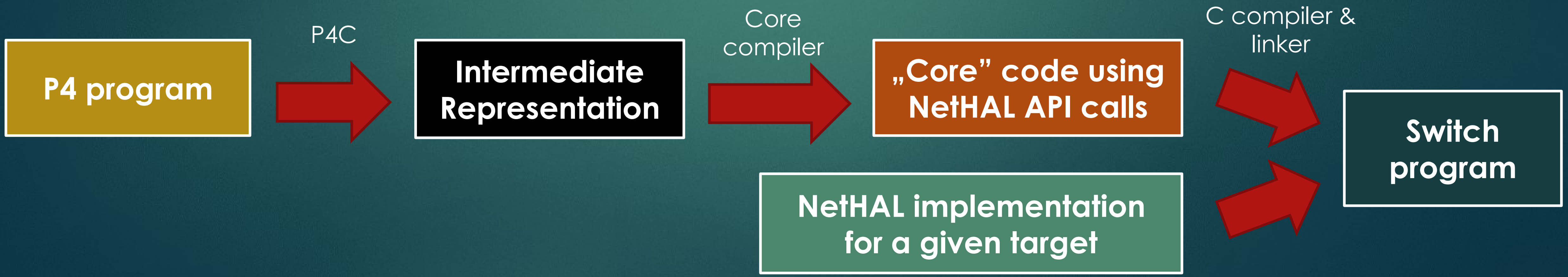
3. Switch program

- Compiled from the hardware-independent C code of the „Core” and the target-specific HAL
- Resulting in a hardware dependent switch program

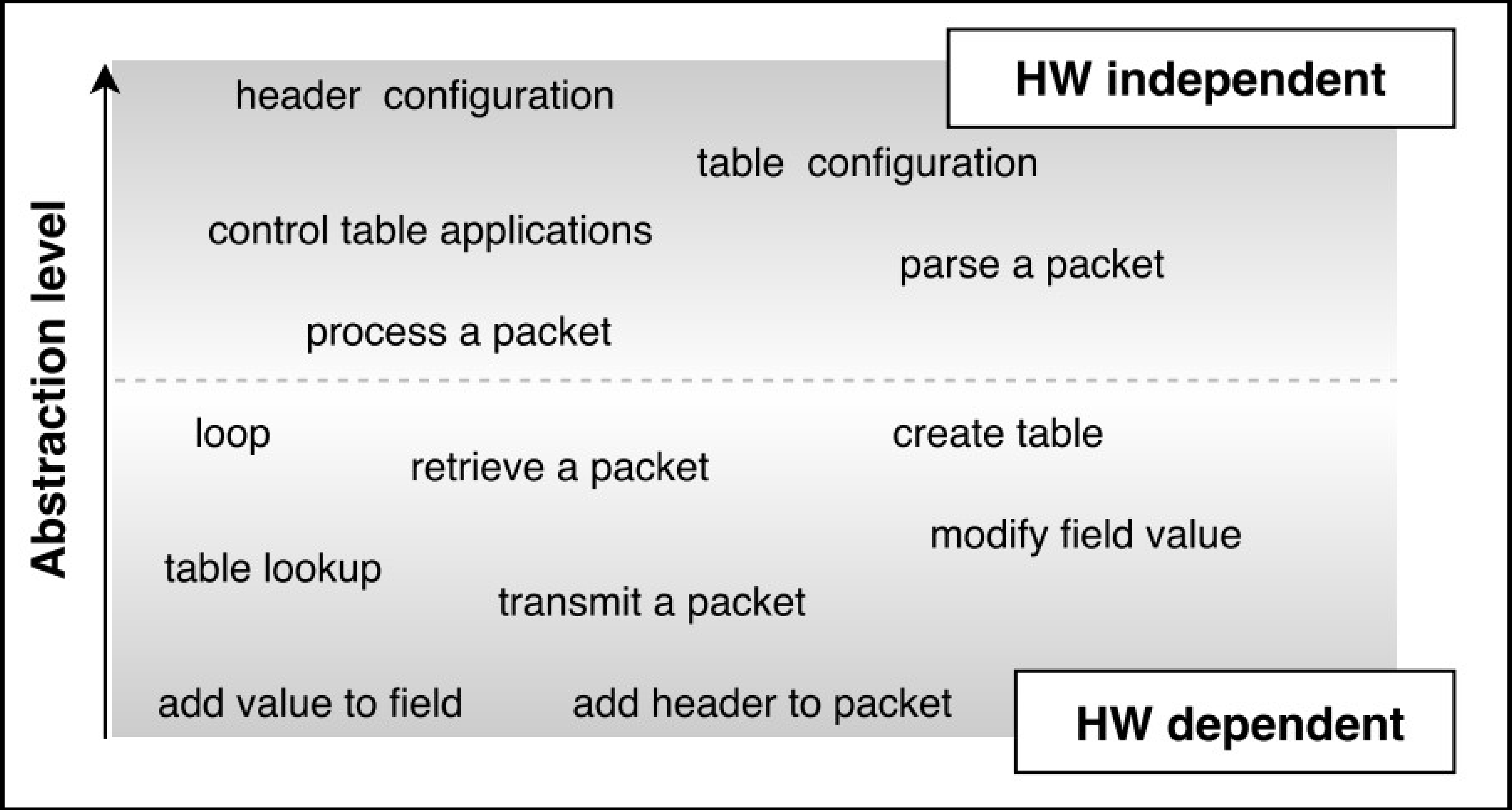


Multi-target Compiler Architecture for P4

- **PROs**
  - Much simpler compiler
  - Modularity = better maintainability
  - Exchangeable NetHAL = re-targetable switch (without rewriting a single line of code)
  - NetHAL is not affected by changes in the P4 program
- **CONs**
  - Potentially lower performance
  - Difficulties with protocol/hardware-dependent optimization
  - Communication overhead between the components (C function calls)
  - Too general vs too detailed NetHAL API



Multi-target Compiler Architecture for P4



The „core”

Run to completion model

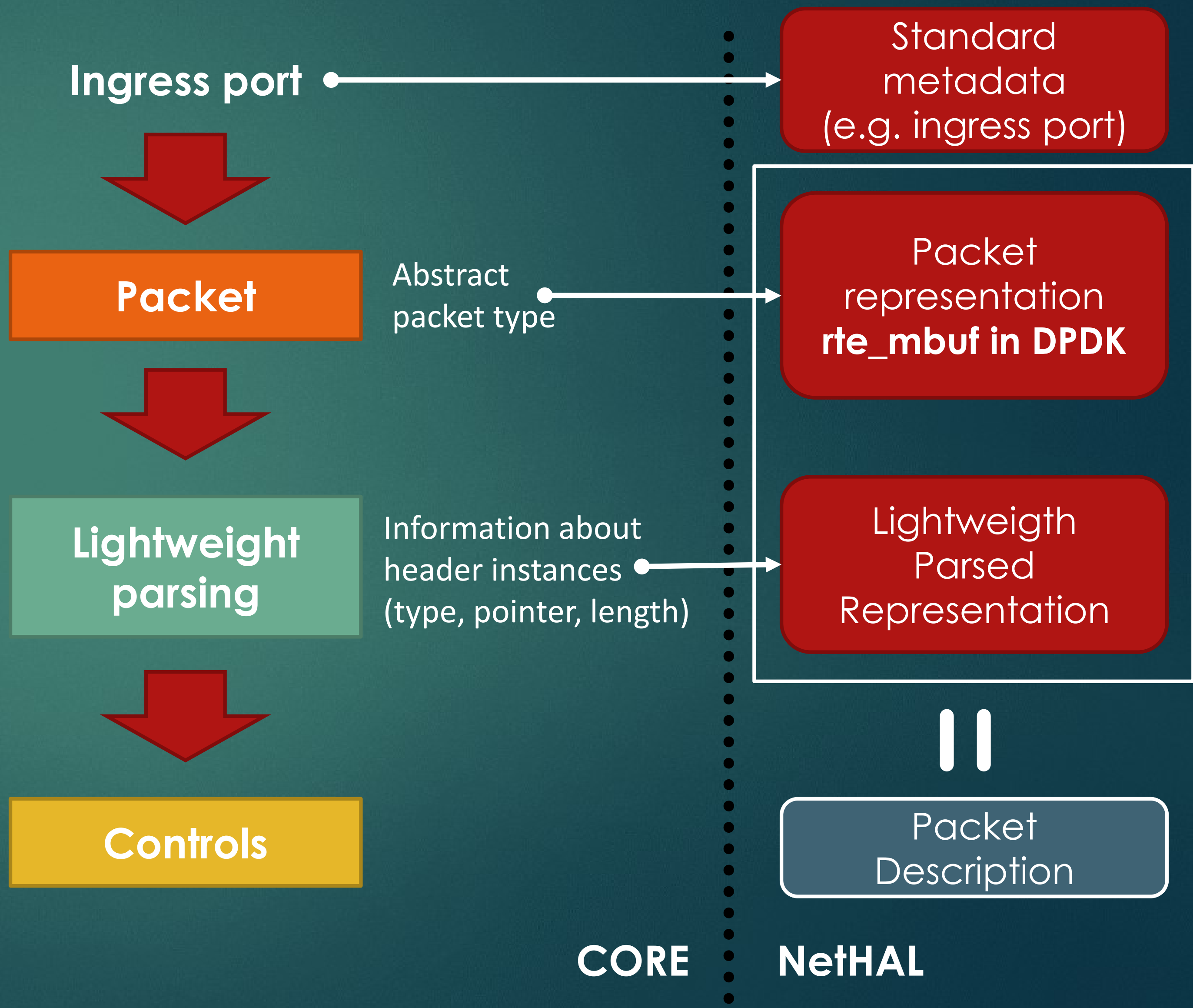
- Plans to move to a pipeline model

The core implements

- Packet „parsing”
- Control programs
- Actions
- Key calculations for lookup tables

Packet parsing

- Lightweight Parsed Representation
- Determining the positions and types of headers in the packet
- No "real" parsing or field extraction
  - lazy evaluation



The „core"

Run to completion model

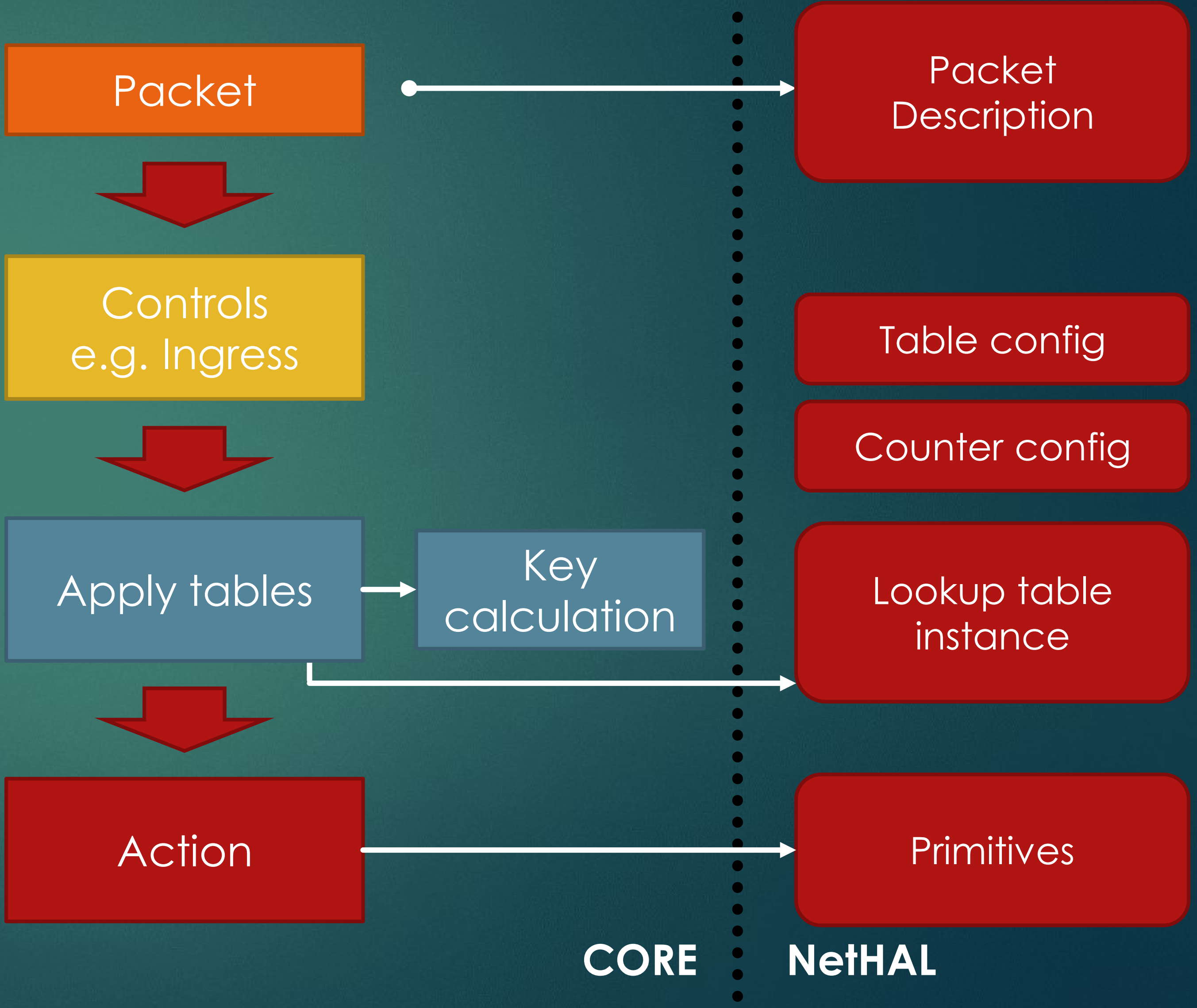
- Plans to move to a pipeline model

The core implements

- Packet „parsing"
- Control programs
- Actions
- Key calculations for lookup tables

Controls and actions

- Controls and actions are translated to C functions
- Key calculation for lookup tables
- Fields are extracted when needed
- In-place field modifications



Network Hardware Abstraction Library

Low-level generic C API

- For networking hardwares

Hardware specific implementations of

- States/settings (tables, counters, meters etc.)
- Related operations (table insert/delete/lookup, counter increment, etc.)
- Packet RX and TX operations
- Primitive actions (header-related + digests)
- Helpers for primitive actions (field-related)
  - Implemented as macros for performance reasons

Add and remove headers

add_header(packet_descriptor_t* p, header_reference_t h)
push(packet_descriptor_t* p, header_stack_t h)
remove_header(packet_descriptor_t* p, header_reference_t h)
pop(packet_descriptor_t* p, header_stack_t h)

Field modification & extraction

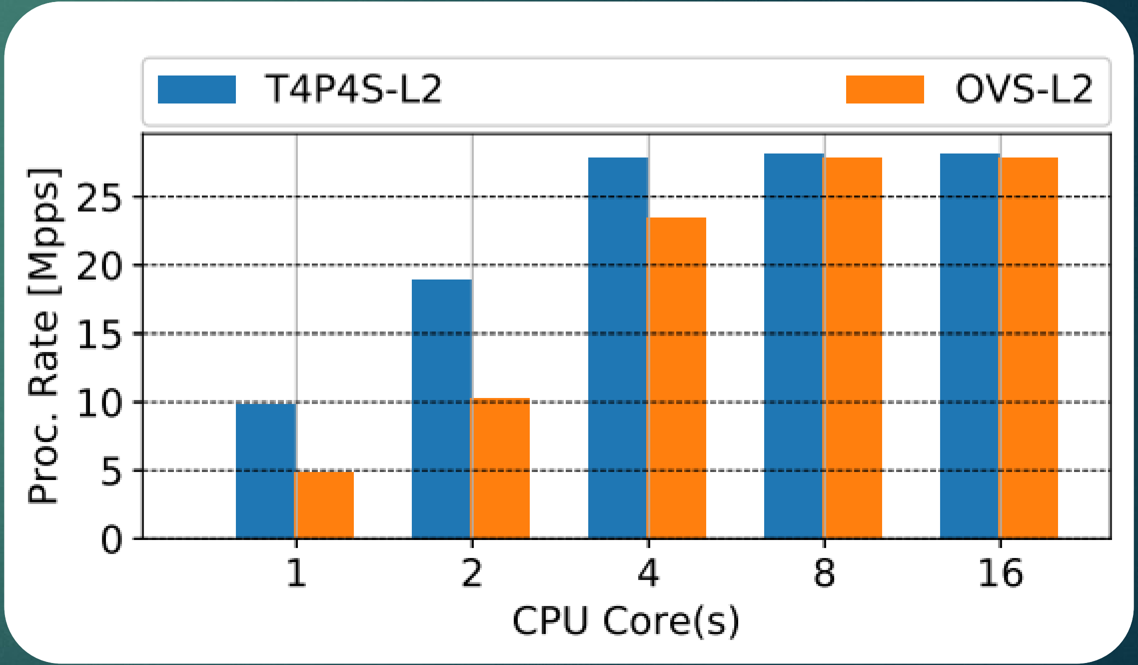
MODIFY_BYTEBUF_BYTEBUF(pd, dstfield, src, srclen)
MODIFY_INT32_BYTEBUF(pd, dstfield, src, srclen)
MODIFY_INT32_INT32(pd, dstfield, value32)
EXTRACT_INT32(pd, field, dst)

Table & counter operations

exact_lookup(lookup_table_t* t, uint8_t* key)
lpm_lookup(lookup_table_t* t, uint8_t* key)
ternary_lookup(lookup_table_t* t, uint8_t* key)
exact_add(lookup_table_t* t, uint8_t* key, uint8_t* value)
lpm_add(lookup_table_t* t, uint8_t* key, uint8_t depth, uint8_t* value)
ternary_add(lookup_table_t* t, uint8_t* key, uint8_t* mask, uint8_t* value)
increase_counter(int counterid, int index)
read_counter(int counterid, int index)

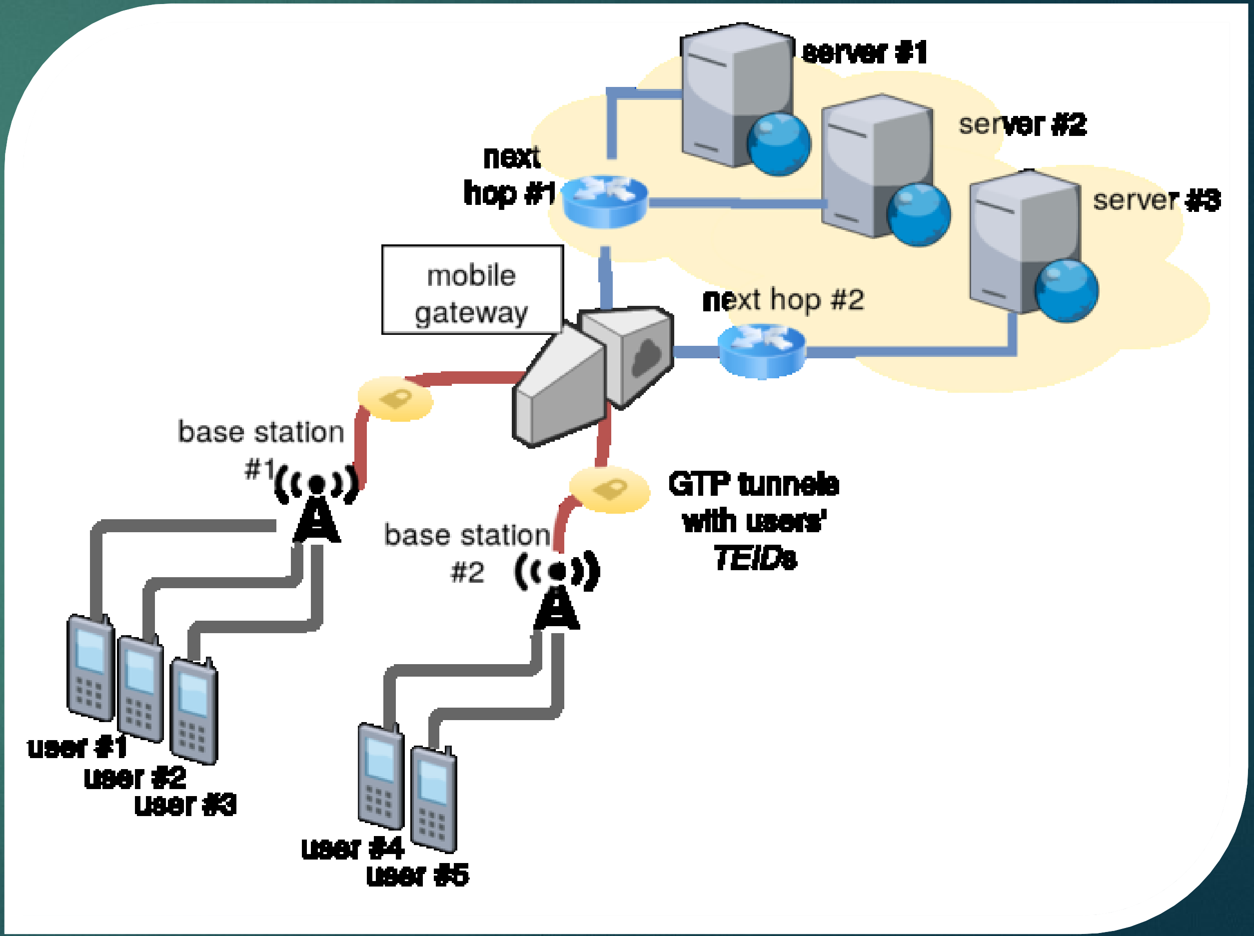
Evaluation - L2 forwarding

- L2 forwarding
  - Source mac learning
    - Two exact match tables: src mac + dst mac
- Testbed setup
  - Intel(R) Xeon(R) CPU E5-1660 v4 @ 8c 16t 3.20GHz, 8x8GB DDR4 SDRAM
  - Dual port 100 Gbps NIC
    - Mellanox MT27700 Family [ConnectX-4]
  - T4P4S performance is compared to OVS
    - Identical implementations in OpenFlow and P4
  - Pseudo random test traffic generated
    - A few hundred flows



Evaluation – Mobile Gateway

- Uplink:
  - L2, L3 and L4 check (gateway MAC/IP and UDP port destination 2152)
  - GTP decap, save TEID
  - -- Rate limit per bearer (TEID)
  - L3 routing towards the Internet + L2 fwd
- Downlink:
  - L2 and L3 check (check if destination IP is in the UE range)
  - -- Per user rate limiting
  - GTP encap (set bearer in TEID)
  - Set destination IP of the base station of the UE
  - L3 routing towards BSTs + L2 fwd

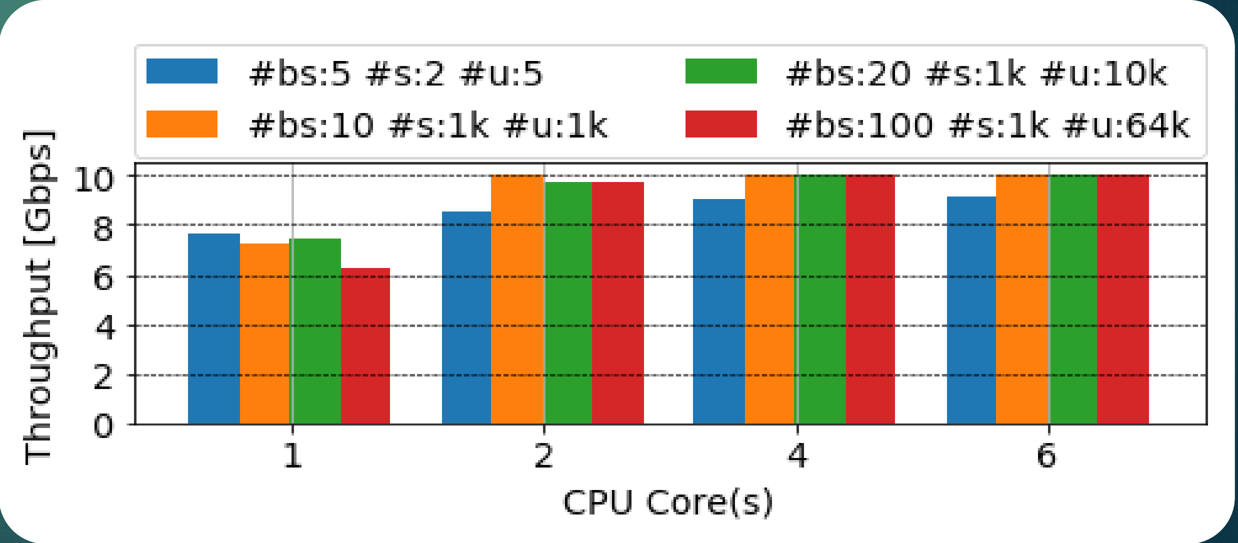


Evaluation – Mobile Gateway

- Uplink:
  - L2, L3 and L4 check (gateway MAC/IP and UDP port destination 2152)
  - GTP decap, save TEID
  - -- Rate limit per bearer (TEID)
  - L3 routing towards the Internet + L2 fwd
- Downlink:
  - L2 and L3 check (check if destination IP is in the UE range)
  - -- Per user rate limiting
  - GTP encap (set bearer in TEID)
  - Set destination IP of the base station of the UE
  - L3 routing towards BSTs + L2 fwd

Testbed setup

- AMD Ryzen Threadripper 1900X
- Intel Corporation 82599ES 10-Gigabit Dual port NIC



T4P4S

- A translator for P4 Switches
  - **Open source** (on GitHub)
    - Visit our site: http://p4.elte.hu
    - Or the GitHub repository: https://github.com/P4ELTE/t4p4s
  - **P4-14** and **P4-16** language support
  - Support of multiple targets
    - by the **Hardware Independent Core** and **Network Hardware Abstraction Libraries**
    - NetHALs for **Intel** (DPDK), **Freescale** (ODP SDK), **OpenWRT** (Native Linux) platforms

Traffic Management - AQM

Active Queue Management in general

Based on course at CMU: 15-441 Computer Networking

Active Queue Management (AQM)

- **Problem**: Standard loss-based TCP's congestion control plus large unmanaged buffers in Internet routers, switches, device drivers,... (a.k.a Bufferbloat)
- **Cause**: Latency issues for interactive/multimedia applications
- **Solution**: AQM tries to signal the onset of congestion by (randomly?) dropping/marking packets

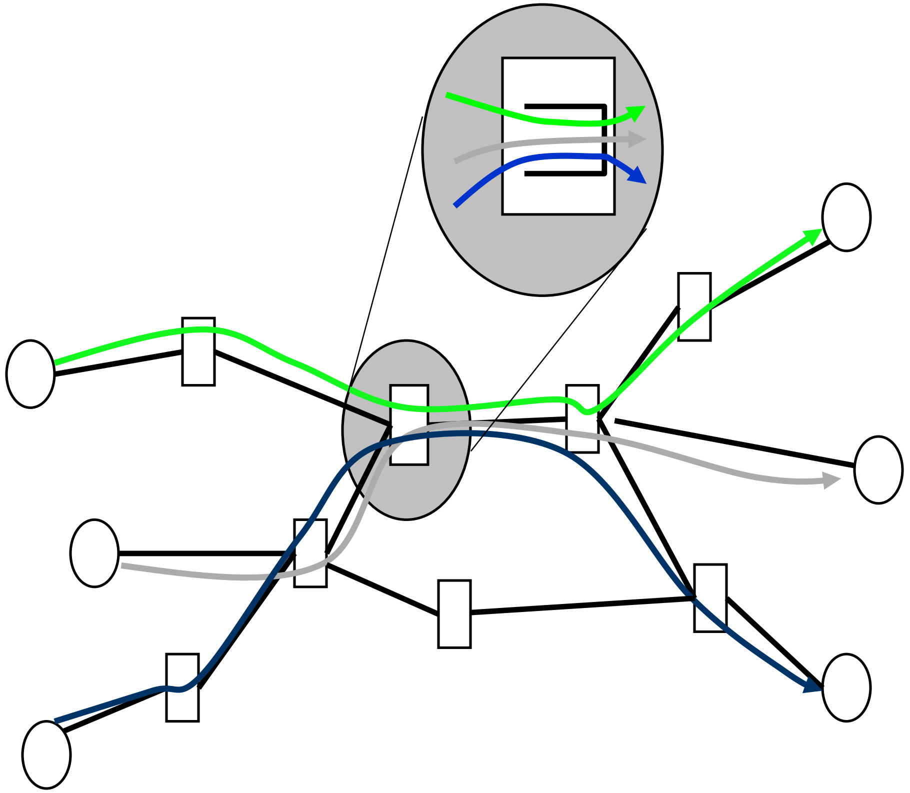
- AQM Goals
  - Maintain low average queue/latency
  - Allow occasional packet bursts
  - Break synchronization among TCP flows

Traffic and Resource Management

- Resources statistically shared

$$\sum \mathbf{Demand_i(t) > Resource(t)}$$

- Overload causes congestion
  - packet delayed or dropped
  - application performance suffer
- Local vs. network wide
- Transient vs. persistent
- Challenge
  - high resource utilization
  - high application performance



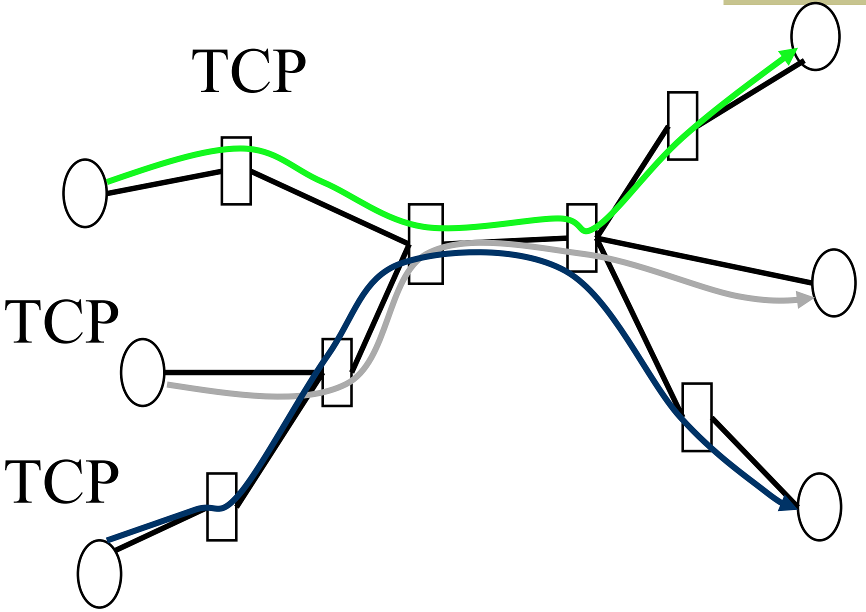
Resource Management Approaches

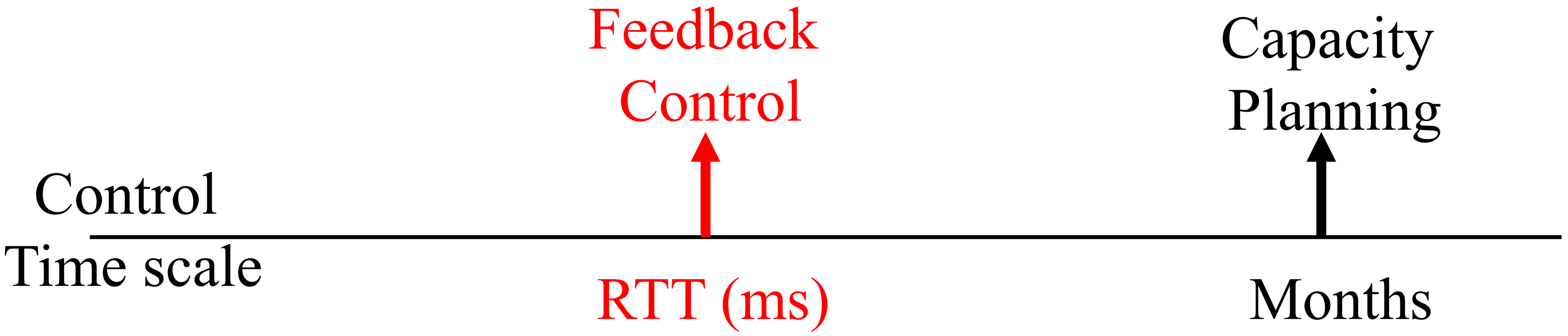
$$\sum \mathbf{Demand}_i(\mathbf{t}) > \mathbf{Resource}(\mathbf{t})$$

- Increase resources
  - install new links, faster routers
  - capacity planning, provisioning, traffic engineering
  - happen at longer timescale
- Reduce or delay demand
  - Reactive approach: encourage everyone to reduce or delay demand
  - Reservation approach: some requests will be rejected by the network

Congestion Control in Today's Internet

- End-system-only solution (TCP)
  - dynamically estimates network state
  - packet loss signals congestion
  - reduces transmission rate in presence of congestion
  - routers play little role



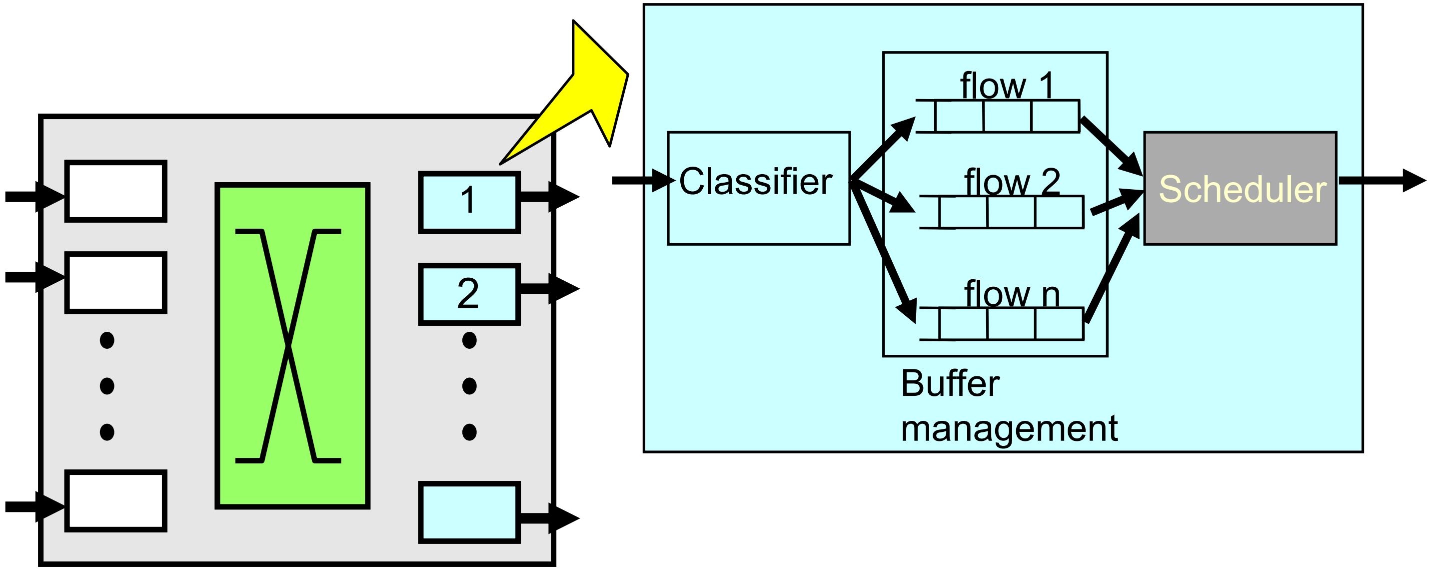


More Ideas on Traffic Management

- Improve TCP
  - Stay with end-point only architecture
- Enhance routers to help TCP
  - Random Early Discard
- Enhance routers to control traffic
  - Rate limiting
  - Fair Queueing
- Provide QoS by limiting congestion

Router Mechanisms

- Buffer management: when and which packet to drop?
- Scheduling: which packet to transmit next?



Overview

- Queue management & RED
- Fair-queuing
- Why QOS?
- Integrated services

Queuing Disciplines

- Each router must implement some queuing discipline
- Queuing allocates both bandwidth and buffer space:
  - Bandwidth: which packet to serve (transmit) next
  - Buffer space: which packet to drop next (when required)
- Queuing also affects latency

Typical Internet Queuing

- FIFO + drop-tail
  - Simplest choice
  - Used widely in the Internet
- FIFO (first-in-first-out)
  - Implies single class of traffic
- Drop-tail
  - Arriving packets get dropped when queue is full regardless of flow or importance
- Important distinction:
  - FIFO: scheduling discipline
  - Drop-tail: drop policy

FIFO + Drop-tail Problems

- Leaves responsibility of congestion control completely to the edges (e.g., TCP)
- Does not separate between different flows
- No policing: send more packets → get more service
- Synchronization: end hosts react to same events

FIFO + Drop-tail Problems

- Full queues
  - Routers are forced to have have large queues to maintain high utilizations
  - TCP detects congestion from loss
    - Forces network to have long standing queues in steady-state
- Lock-out problem
  - Drop-tail routers treat bursty traffic poorly
  - Traffic gets synchronized easily → allows a few flows to monopolize the queue space

Active Queue Management

- Design active router queue management to aid congestion control
- Why?
  - Router has unified view of queuing behavior
  - Routers see actual queue occupancy (distinguish queue delay and propagation delay)
  - Routers can decide on transient congestion, based on workload

Design Objectives

- Keep throughput high and delay low
  - High power (throughput/delay)
- Accommodate bursts
- Queue size should reflect ability to accept bursts rather than steady-state queuing
- Improve TCP performance with minimal hardware changes

Lock-out Problem

- Random drop
  - Packet arriving when queue is full causes some random packet to be dropped
- Drop front
  - On full queue, drop packet at head of queue
- Random drop and drop front solve the lock-out problem but not the full-queues problem

Full Queues Problem

- Drop packets before queue becomes full (early drop)
- Intuition: notify senders of incipient congestion
  - Example: early random drop (ERD):
    - If qlen > drop level, drop each new packet with fixed probability *p*
    - Does not control misbehaving users

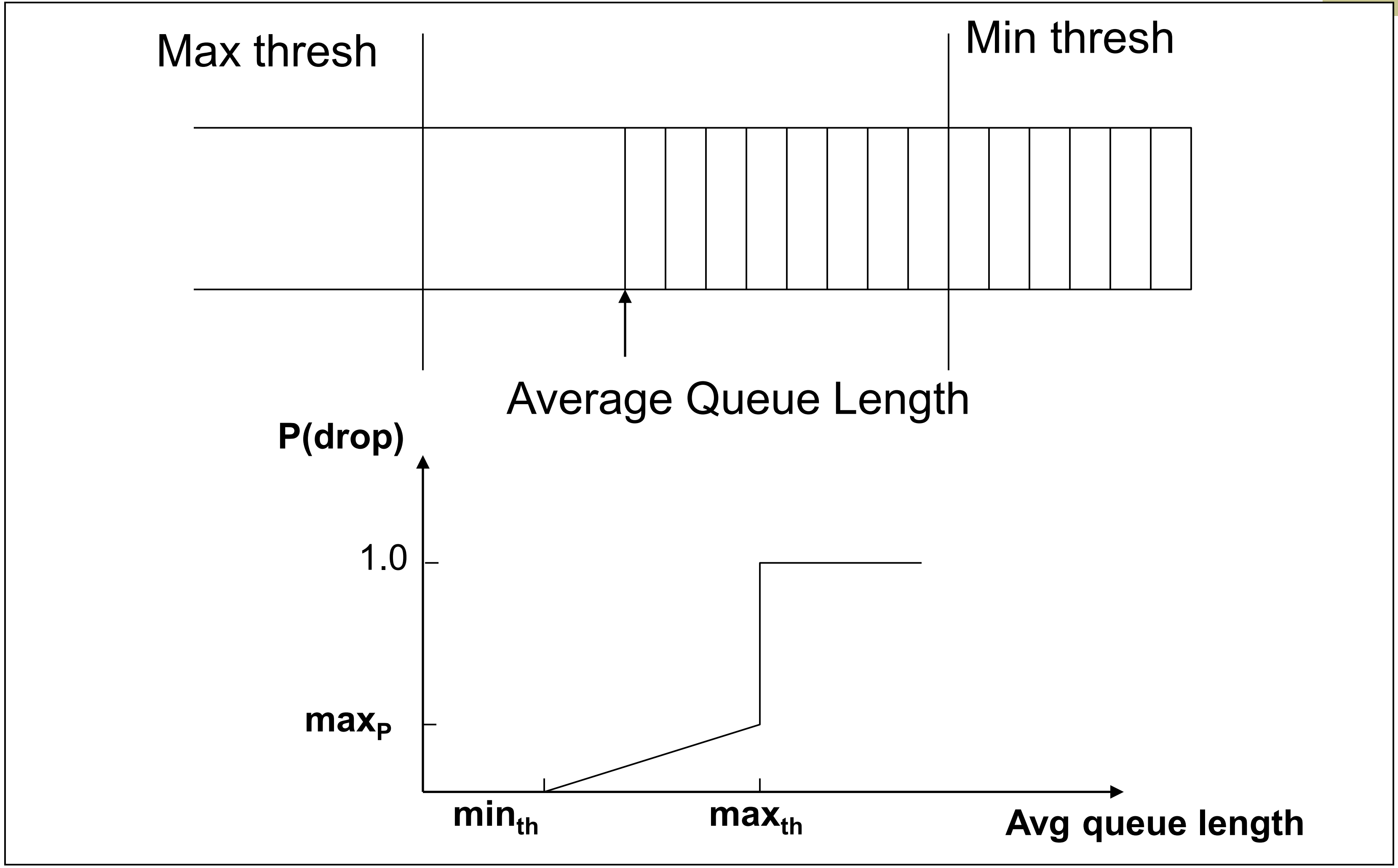
Random Early Detection (RED)

- Detect incipient congestion
- Assume hosts respond to lost packets
- Avoid window synchronization
  - Randomly mark packets
- Avoid bias against bursty traffic

RED Algorithm

- Maintain running average of queue length
- If avg < min_{th} do nothing
  - Low queuing, send packets through
- If avg > max_{th}, drop packet
  - Protection from misbehaving sources
- Else mark packet in a manner proportional to queue length
  - Notify sources of incipient congestion

RED Operation



Explicit Congestion Notification (ECN) [Floyd and Ramakrishnan 98]

- Traditional mechanism
  - packet drop as implicit congestion signal to end systems
  - TCP will slow down
- Works well for bulk data transfer
- Does not work well for delay sensitive applications
  - audio, WEB, telnet
- Explicit Congestion Notification (ECN)
  - borrow ideas from DECBit
  - use two bits in IP header
    - ECN-Capable Transport (ECT) bit set by sender
    - Congestion Experienced (CE) bit set by router

Congestion Control Summary

- Architecture: end system detects congestion and slow down
- Starting point:
  - slow start/congestion avoidance
    - packet drop detected by retransmission timeout RTO as congestion signal
  - fast retransmission/fast recovery
    - packet drop detected by three duplicate acks
- TCP Improvement:
  - NewReno: better handle multiple losses in one round trip
  - SACK: better feedback to source
  - NetReno: reduce RTO in high loss rate, small window scenario
  - FACK, NetReno: better end system control law

Congestion Control Summary (II)

- Router support
  - RED: early signaling
  - ECN: explicit signaling

RED in P4

- RED: https://github.com/PIFO-TM/ns3-bmv2/blob/master/traffic-control/examples/p4-src/red/basic/red.p4

PIE AQM

- Uses a Proportional Integral (PI) controller to manage drop probability and keep the queue delay around a target value
- Lightweight as it uses delay estimation instead of timestamping
- Uses trend of latency (increasing or decreasing) over time to determine the congestion level

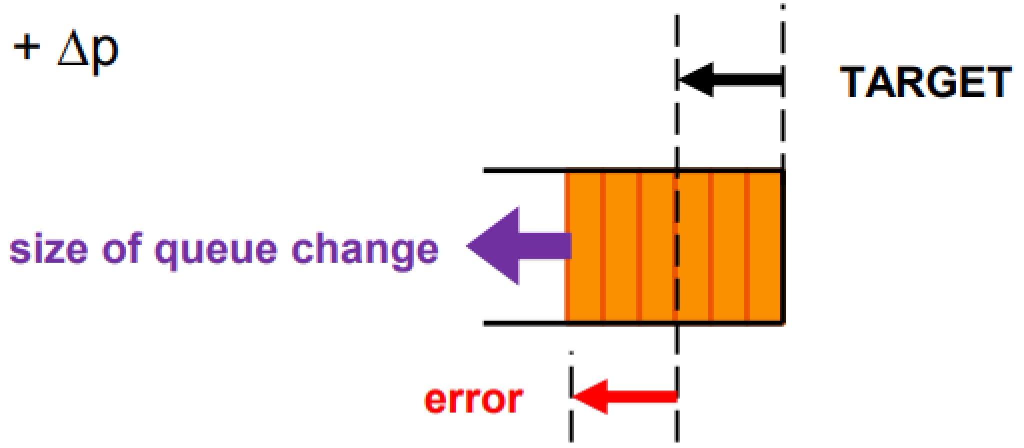
PI control

PI

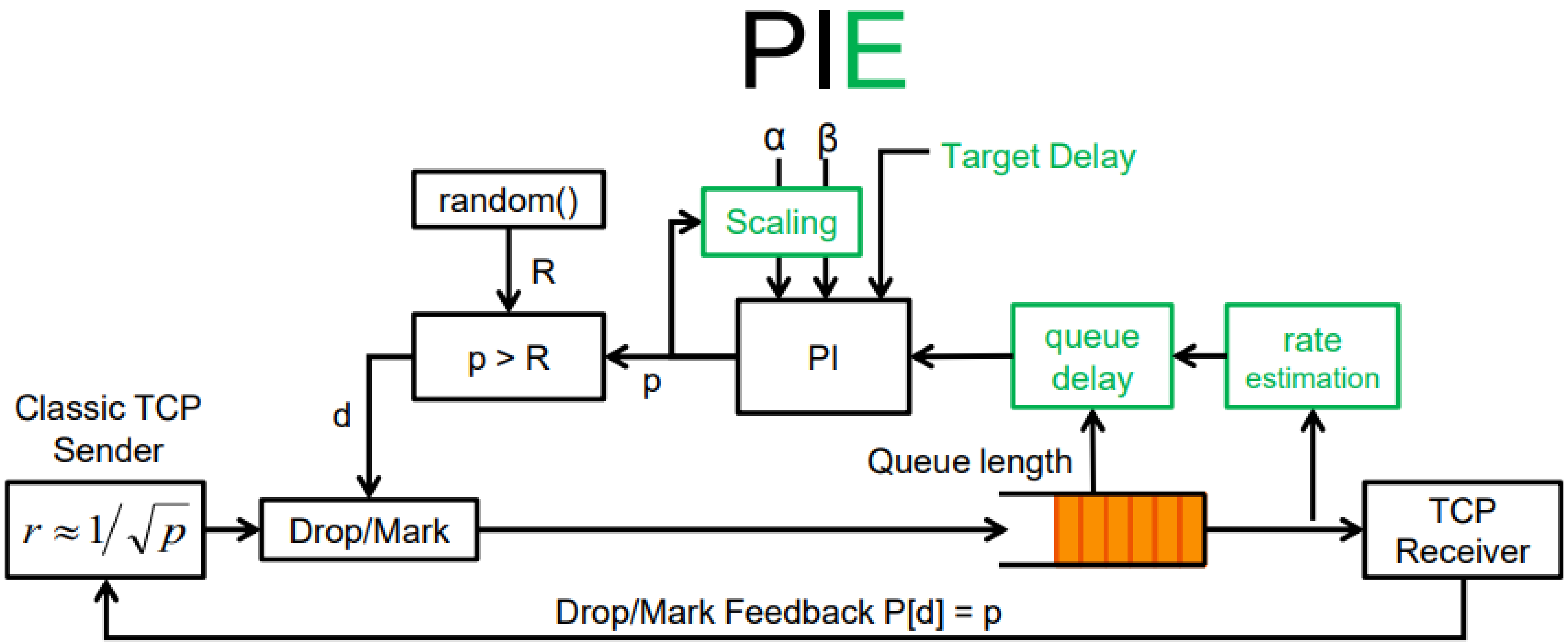
Every T_{update} interval do:

$$\Delta p = \alpha * (current\_queue - TARGET) + \beta * (current\_queue - prev\_queue)$$

$$p = p + \Delta p$$



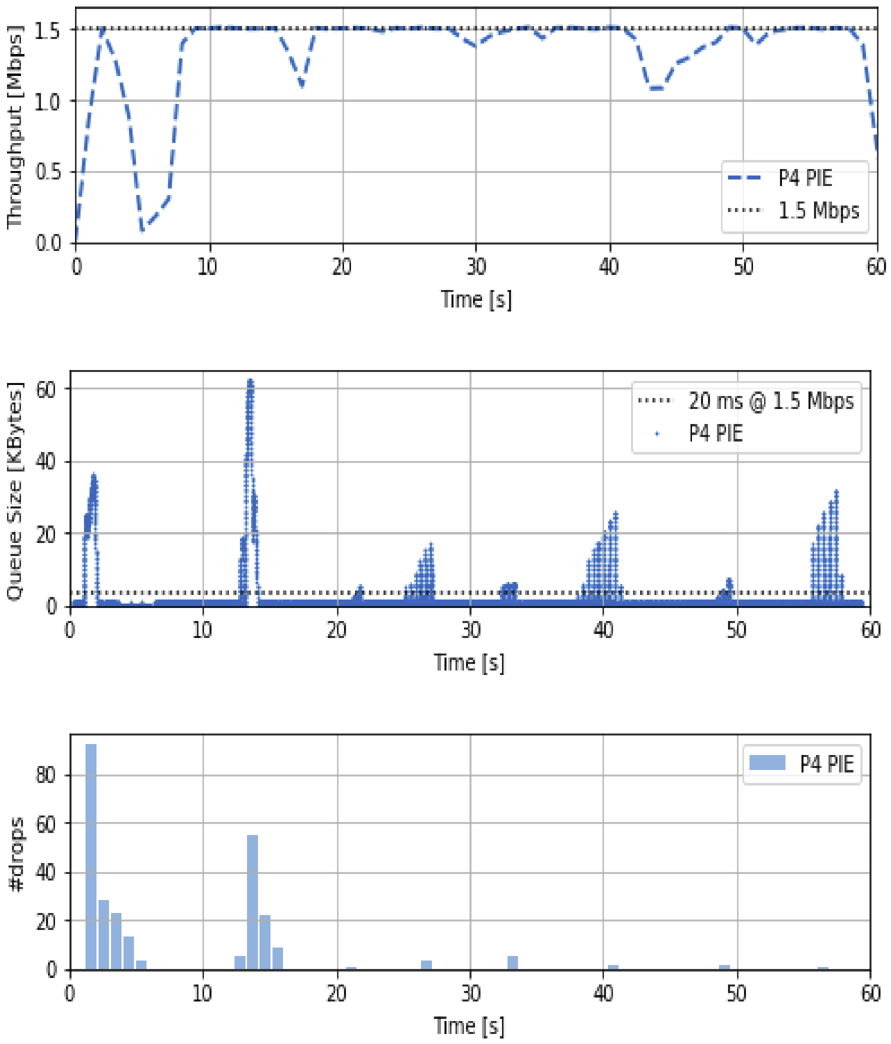
PIE AQM



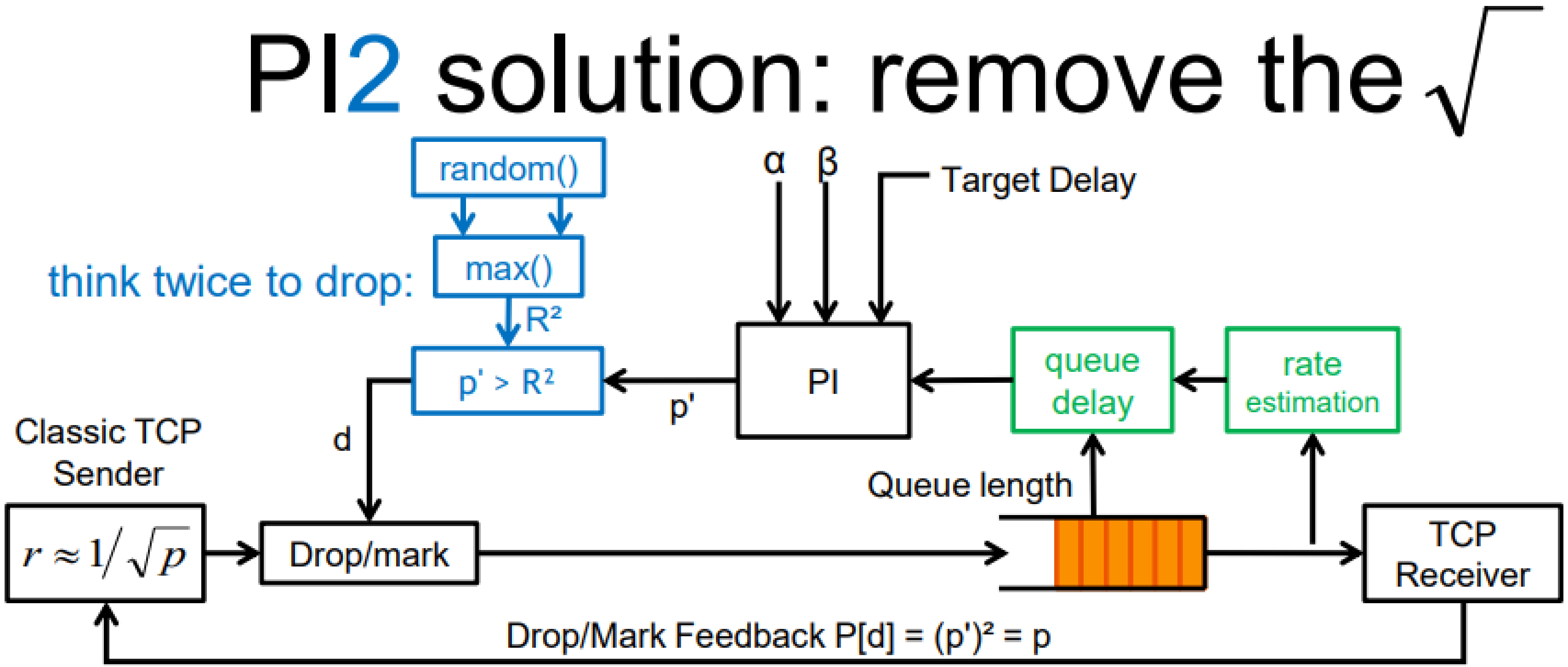
- Enhancements are: rate estimation, queue delay and gain scaling

PIE in P4

- https://github.com/PIFO-TM/ns3-bmv2/blob/master/traffic-control/examples/p4-src/pie/pie.p4



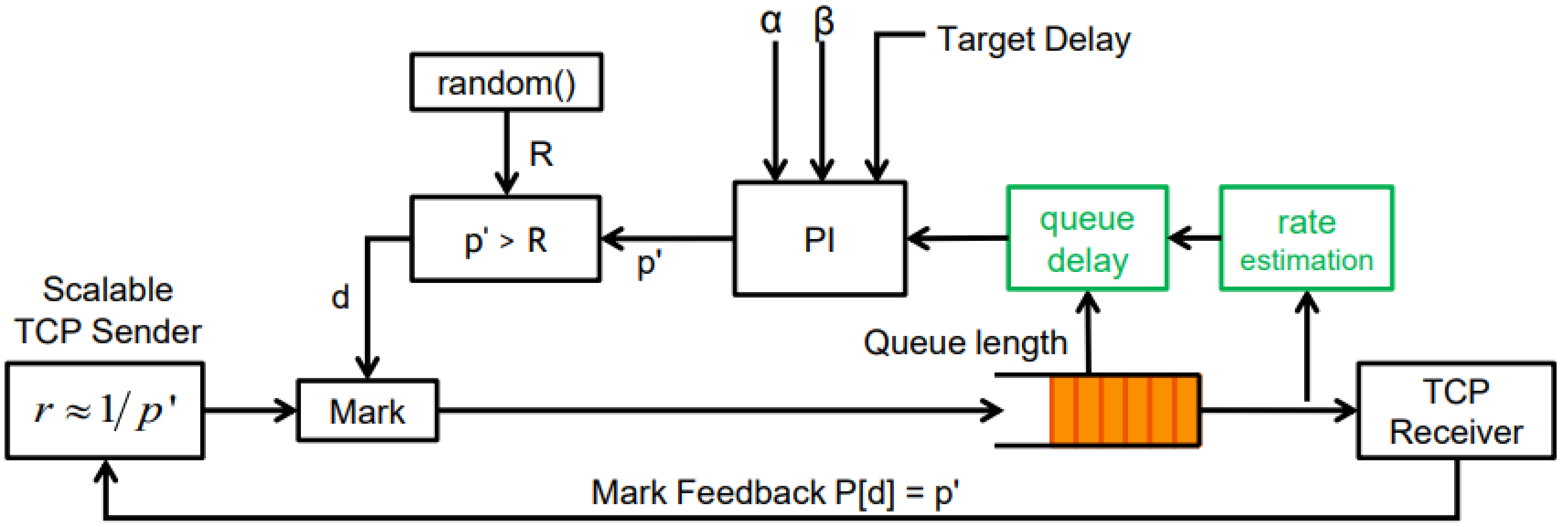
PI2 – for classic TCP



- Replaces gain scaling with a square: $P[d] = (p')^2 = p$
- PI2 controls p‘ which is actually $\sqrt{p}$ so $r \approx 1/p'$

PI2 – for scalable TCP

PI2 also supports scalable TCP



- Scalable TCP needs no scaling, nor squaring
- Can use the same parameters as PI2 for Reno or Cubic

PI2 needs no α and β scaling

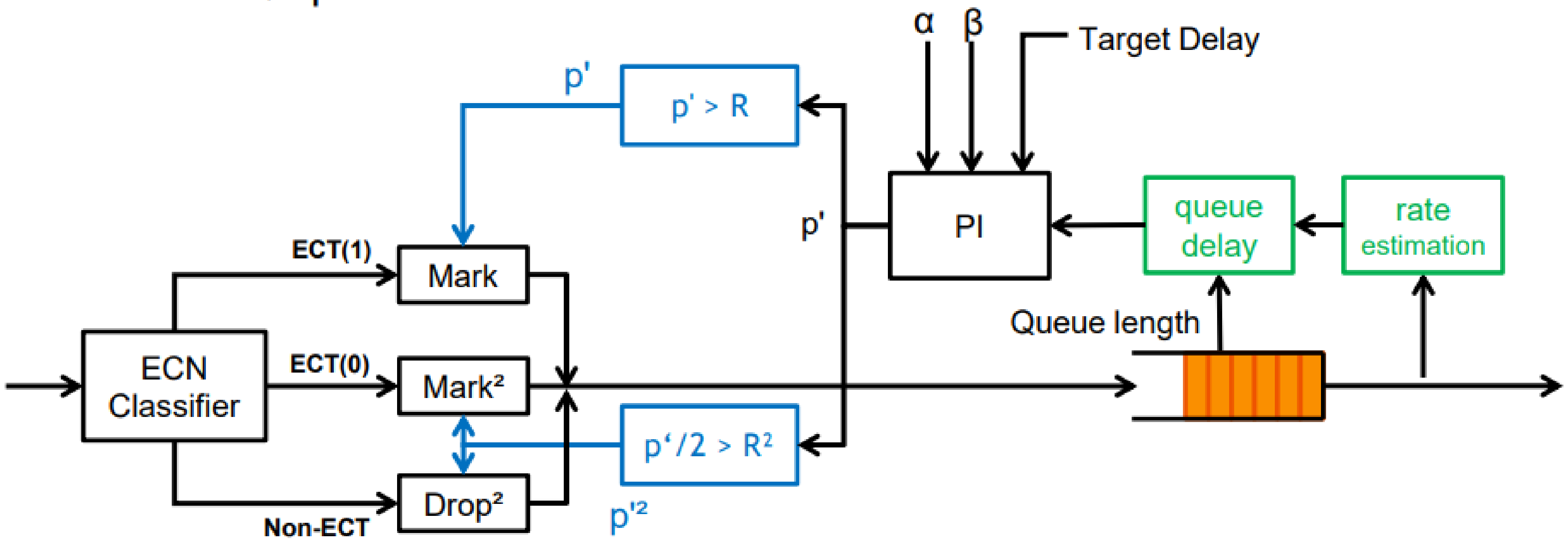
- By squaring at the end, Reno can be controlled like a Scalable TCP
- Models used for:
  - TCP Reno on PI: $$\frac{dW(t)}{dt} = \frac{1}{R(t)} - 0.5\frac{W(t)W(t-R(t))}{R(t-R(t))}p(t-R(t))$$ [1][2]
  - TCP Reno on PI2: $$\frac{dW(t)}{dt} = \frac{1}{R(t)} - 0.5\frac{W(t)W(t-R(t))}{R(t-R(t))}(p'(t-R(t)))^2$$
  - Scalable TCP on PI2: $$\frac{dW(t)}{dt} = \frac{1}{R(t)} - 0.5\frac{W(t-R(t))}{R(t-R(t))}p'(t-R(t))$$

[1] V. Misra, W.-B. Gong, and D. Towsley, "Fluid-based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED," SIGCOMM Computer Comms.. Review, vol. 30, no. 4, pp. 151–160, Aug. 2000.

[2] C. V. Hollot, V. Misra, D. F. Towsley, and W. Gong, "A Control Theoretic Analysis of RED," in Proc. INFOCOM 2001. 20th Annual Joint Conf. of the IEEE Computer and Communications Societies., vol. 3, 2001, pp. 1510—19.

Single Q PI2 experiments

- Lunix implementation
- DualQ option not used here



CoDel – controlling delay

- Tries to detect the standing queue by measuring minimum sojourn delay ($delay_{min}$) over a fixed-duration interval (default 100 ms)
- Uses timestamping
- If $delay_{min}$ > target for at least one interval, enters dropping mode and a packet is dropped from the tail (deque)
- **Next dropping time**: Dropping interval decreases in inverse proportion to the square root of the number of drops since the dropping mode was entered
- Exits dropping mode if $delay_{min} \leq$ target
- No drop when queue is less than 1 MTU

CoDel Assumptions

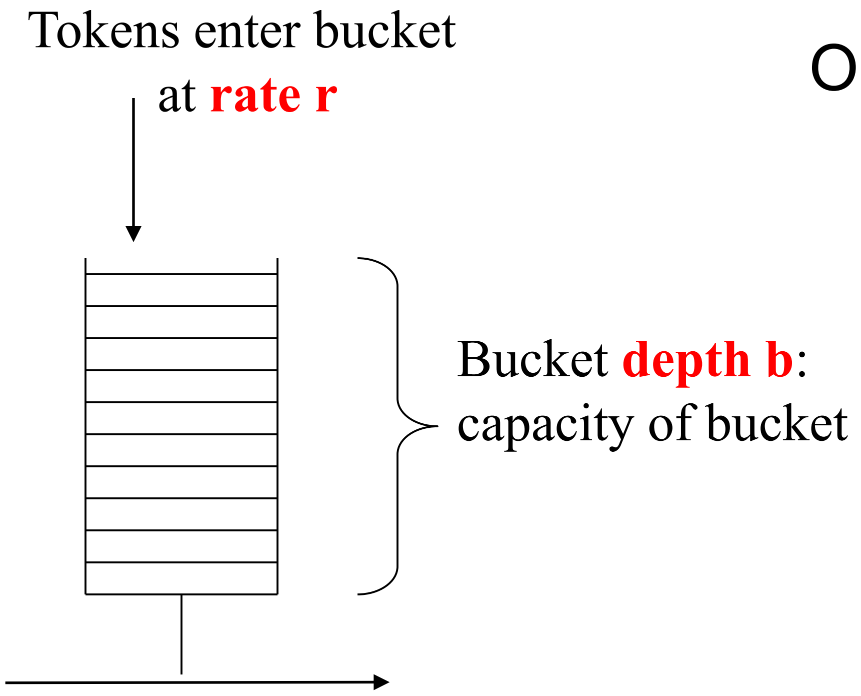
- 100 ms is nominal RTT assumed typical on the Internet paths
- interval = 100 ms; assures protection of normal packet bursts
- A small target standing queue (5% of nominal RTT) is tolerable for achieving better link utilization

CoDel in P4

- https://github.com/ralfkundel/p4-codel/blob/master/srcP4/codel.p4

Traffic Management – Token Bucket

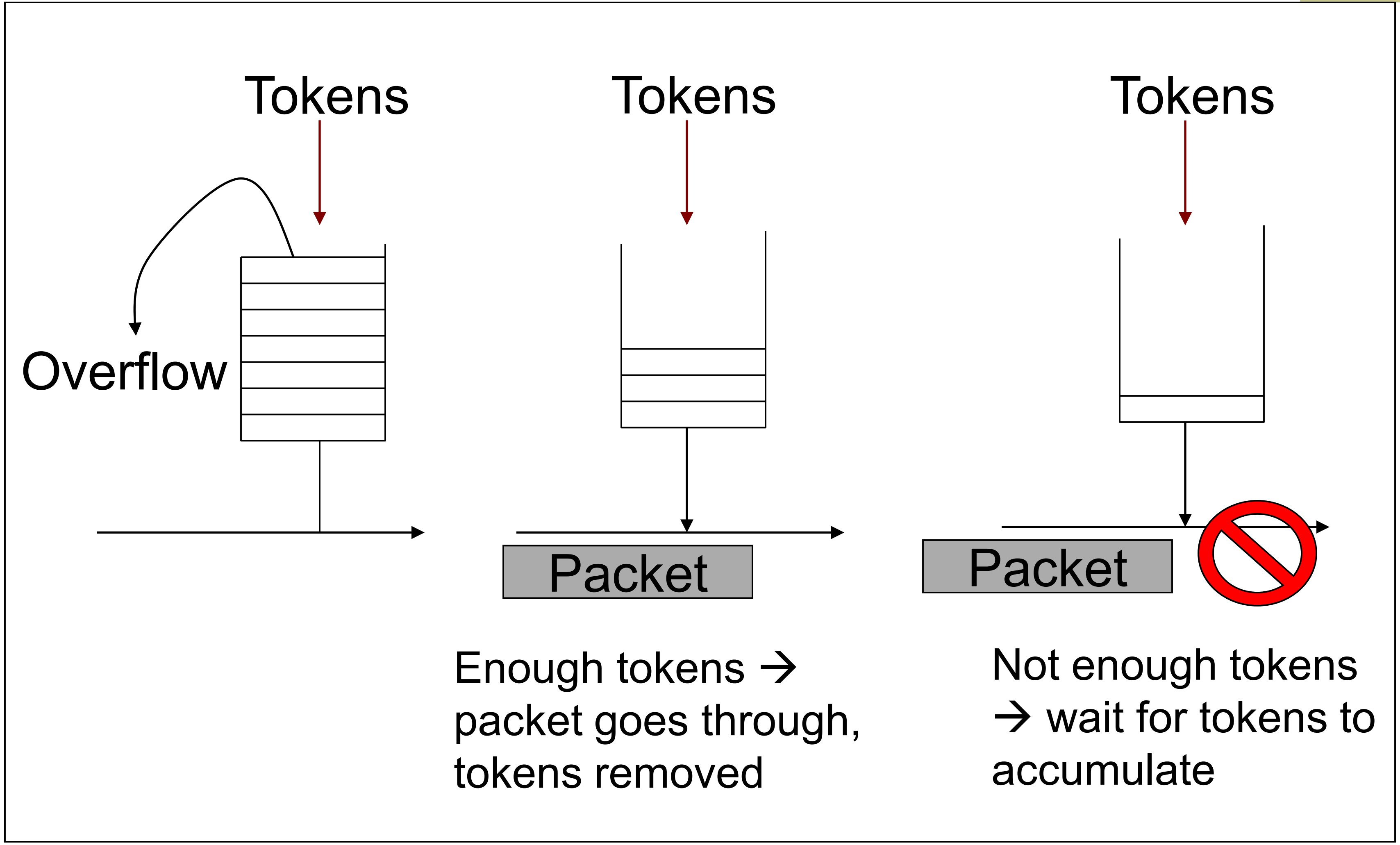
Token Bucket Filter



Operation:

- If bucket fills, tokens are discarded
- Sending a packet of size P uses P tokens
- If bucket has P tokens, packet sent at max rate, else must wait for tokens to accumulate

Token Bucket Operation

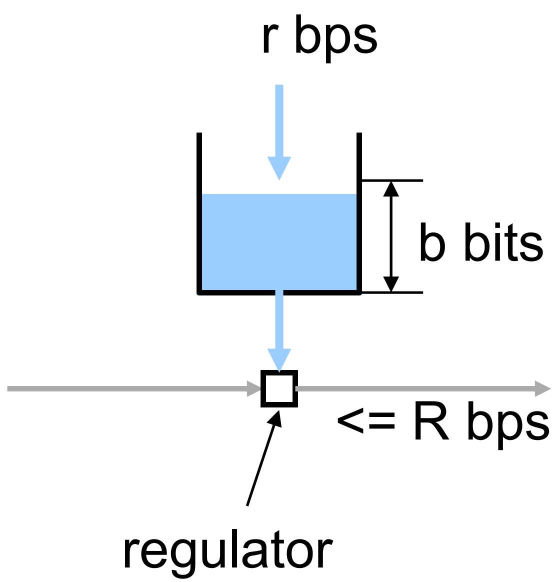


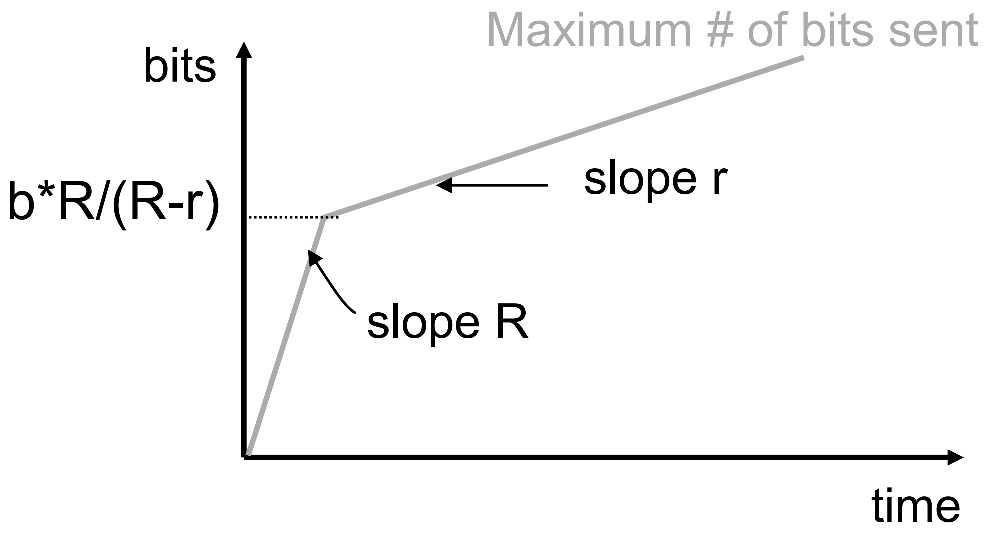
Token Bucket Characteristics

- On the long run, rate is limited to r
- On the short run, a burst of size b can be sent
- Amount of traffic entering at interval T is bounded by:
  - Traffic = b + r*T
- Information useful to admission algorithm

Token Bucket

- Parameters
  - r – average rate, i.e., rate at which tokens fill the bucket
  - b – bucket depth
  - R – maximum link capacity or peak rate (optional parameter)
- A bit is transmitted only when there is an available token



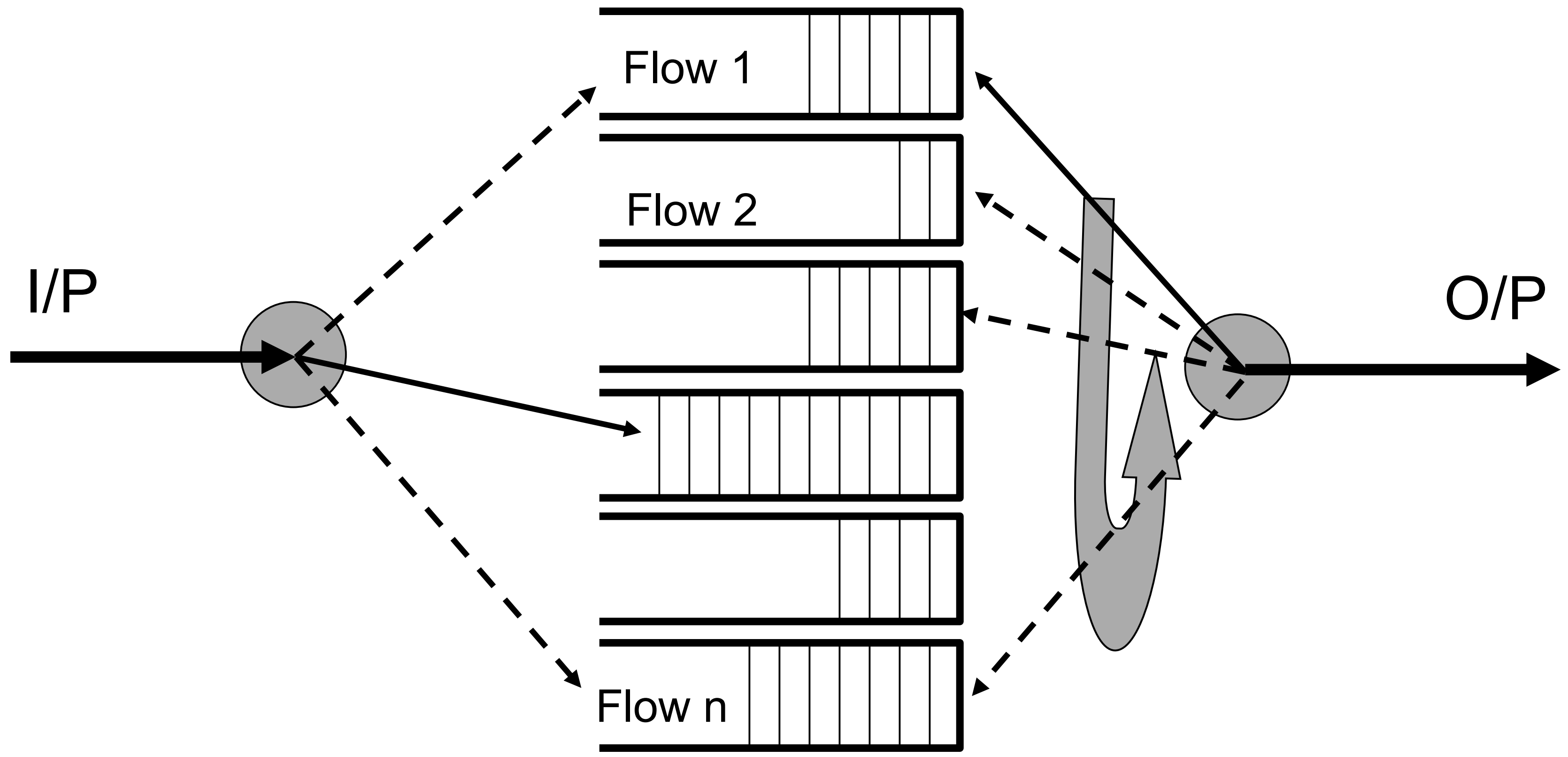


Token bucket in P4

- https://github.com/PIFO-TM/ns3-bmv2/tree/master/traffic-control/examples/p4-src/token-bucket

Per Packet Value
Core stateless resource sharing

Resource sharing **nowadays** - FQ Illustration



Variation: Weighted Fair Queuing (WFQ)

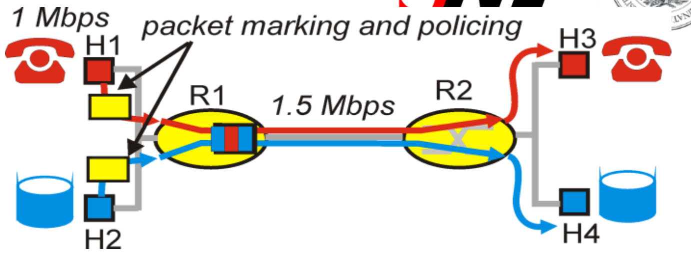
Problem

- **High speed access**
  - Mobile Access Networks, Residental Access Networks, Multi-tenant Data Centers, etc.

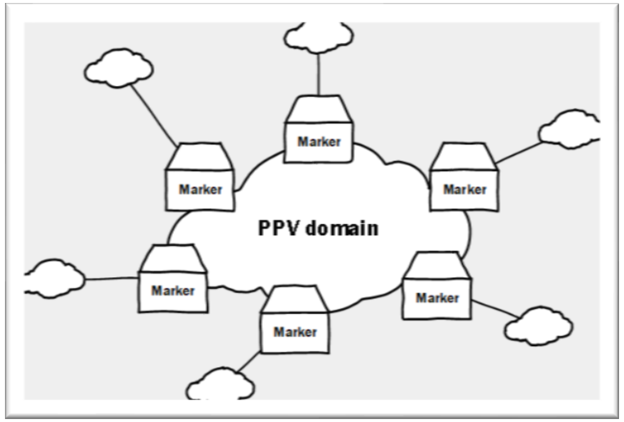
- Appropriate **overprovisioning** of backhaul networks
  - **Difficult & Costly**

- **Scalable** bandwidth sharing supporting **QoS** is needed in **congestion situations**
  - Simple network nodes, no per-user states, service differentiation, rich set of policies, etc.

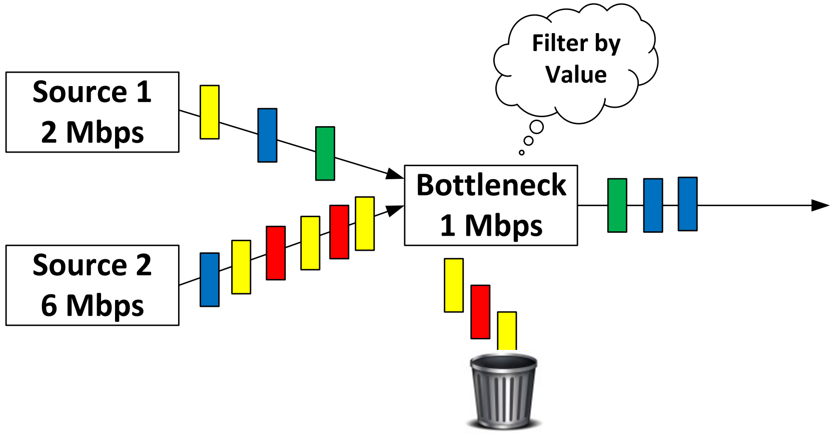
Per Packet Value (PPV) Resource Sharing



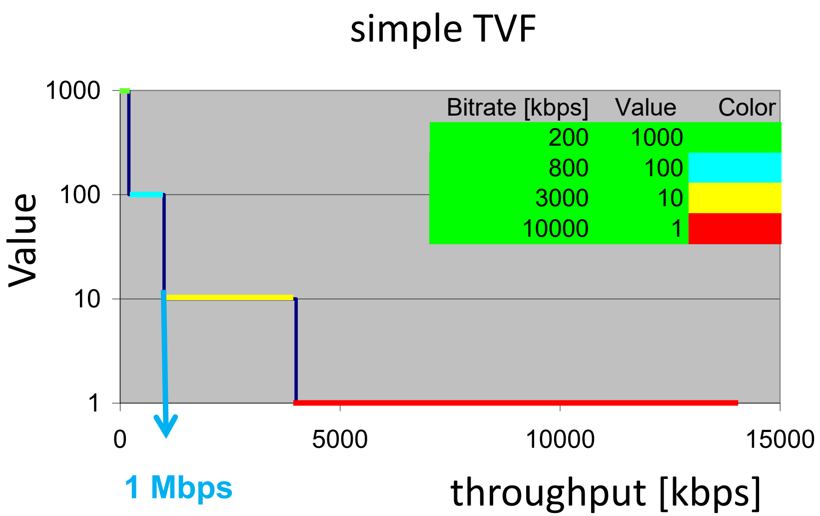
- Resource sharing policies for all congestion situations by **Throughput-Value Functions** (TVF)

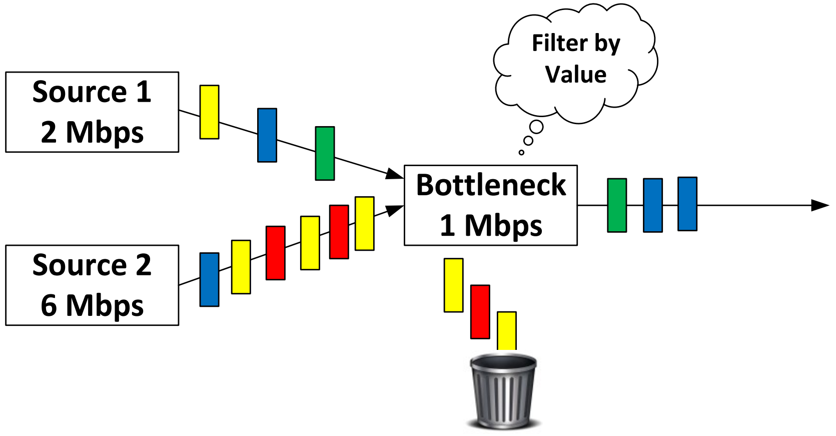


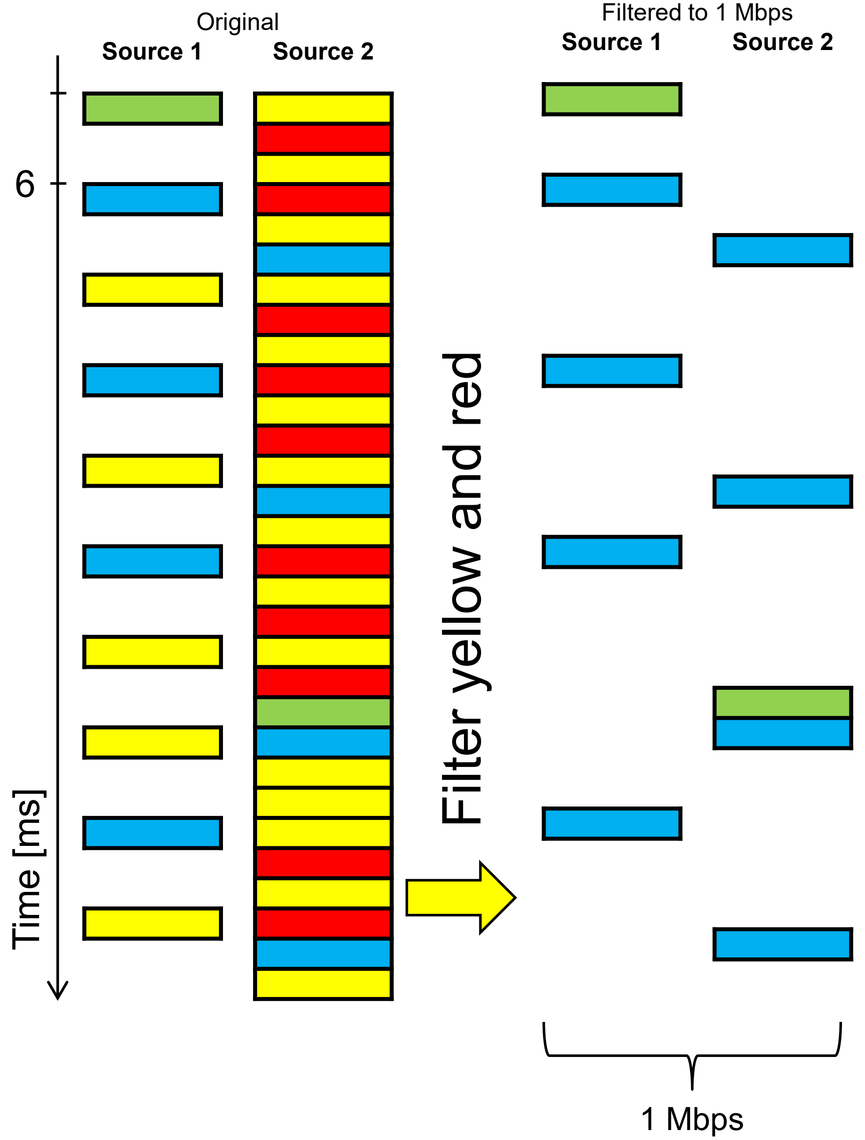
- **Packet Marker** at the edge of the network
  - **Stateful, but highly *distributed***
- **Resource Nodes** (e.g. routers) aim at maximizing the total transmitted Packet Value.
  - **Stateless and *simple***



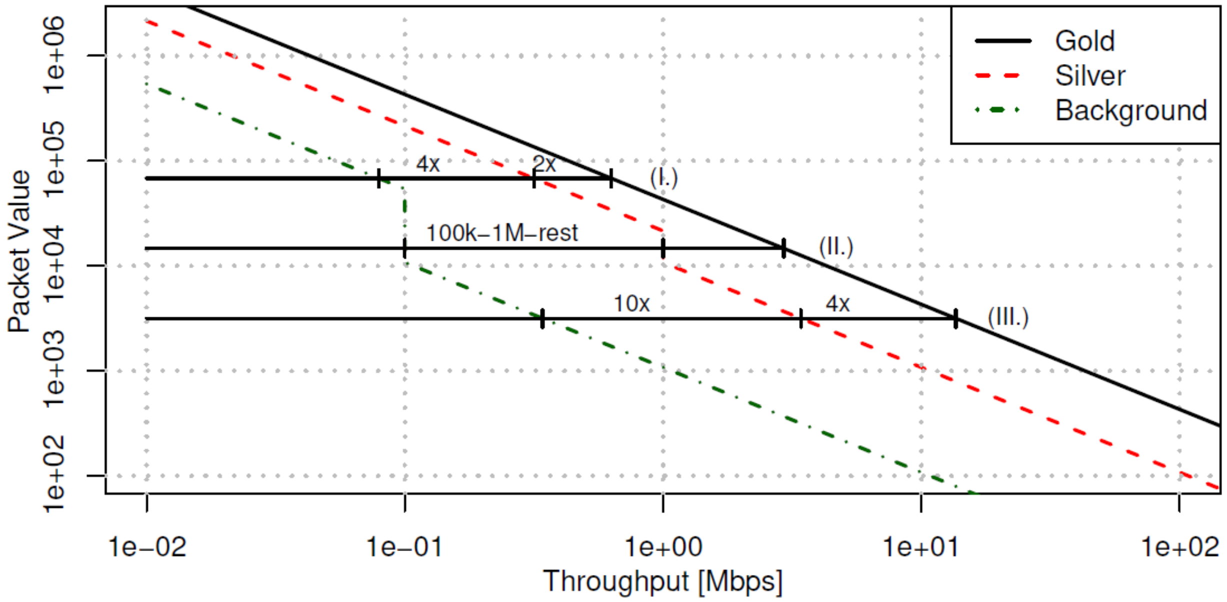
PPV – Packet Marking

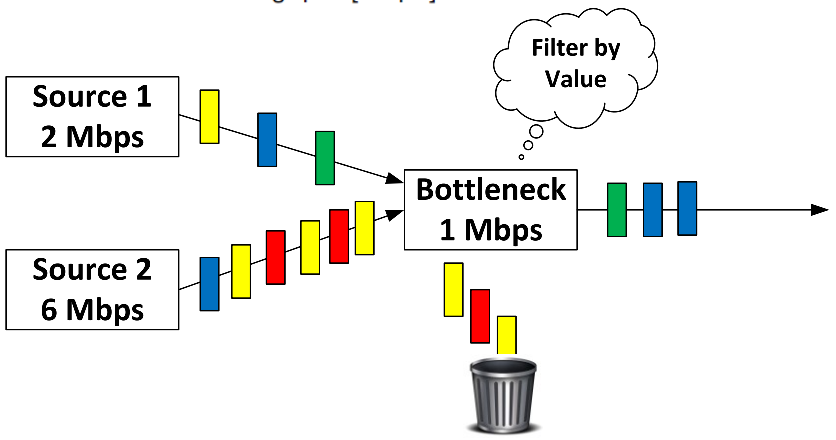


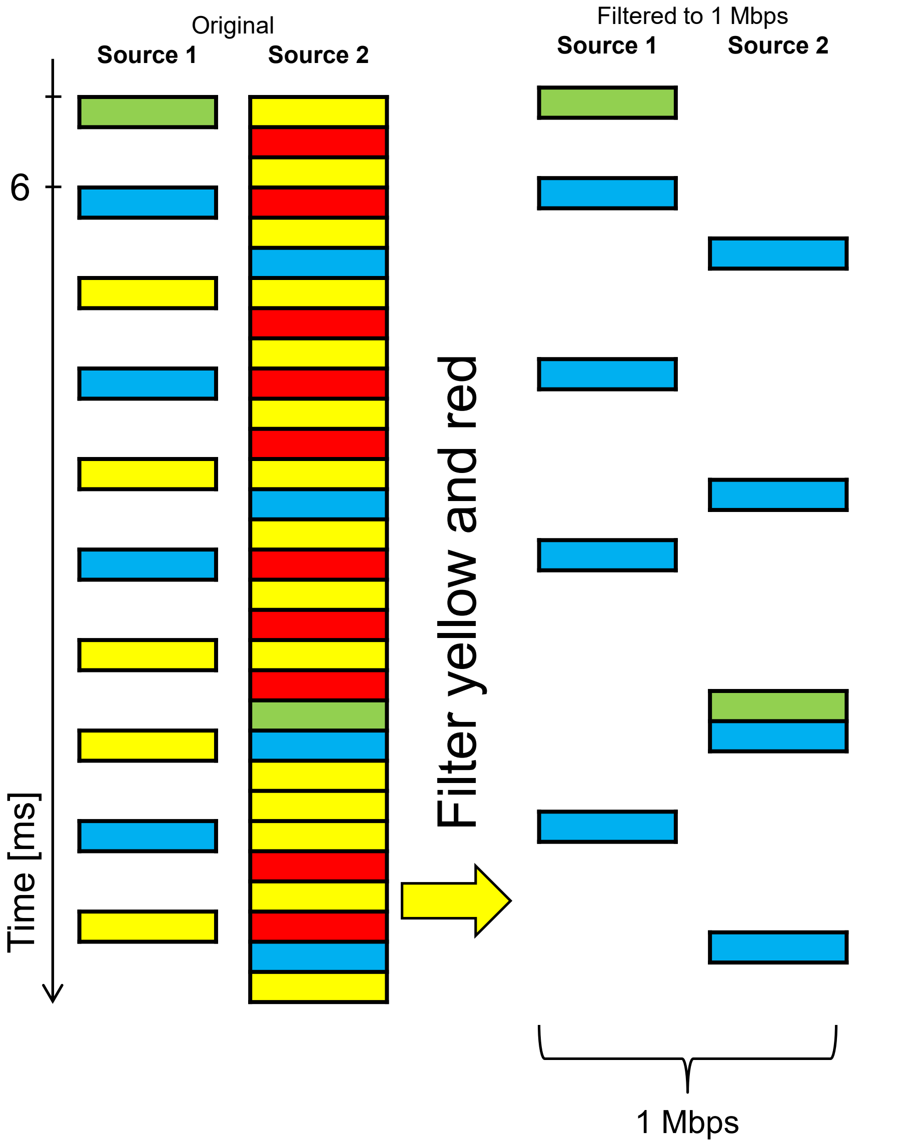




PPV – Packet Marking

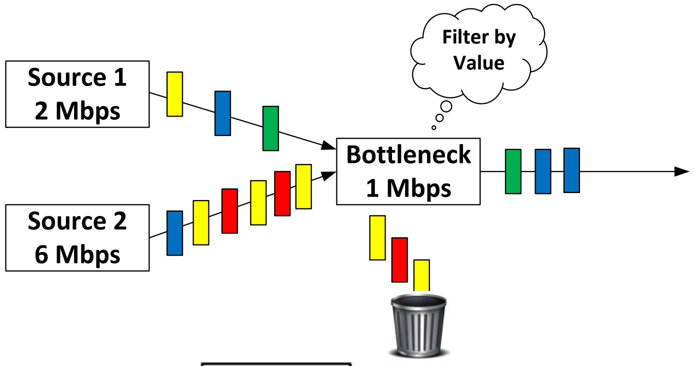


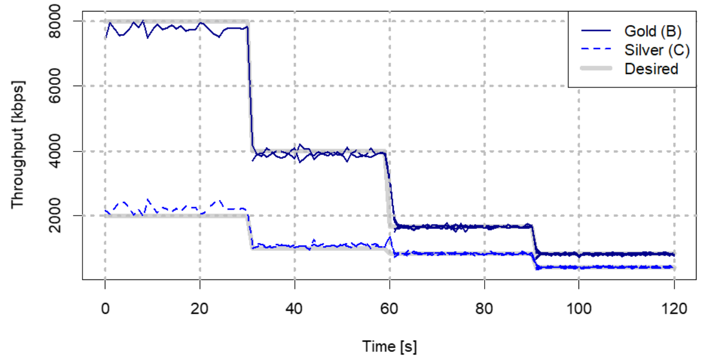




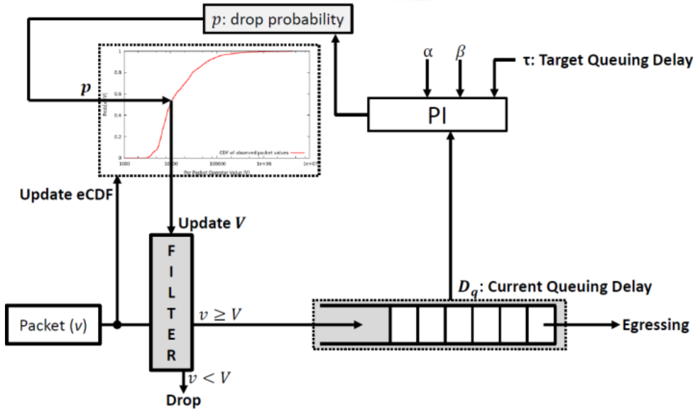
PPV – Resource node proposals

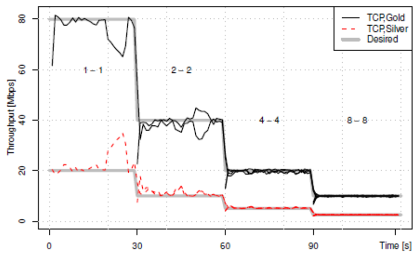
Drop minPPV first scheduling [1]

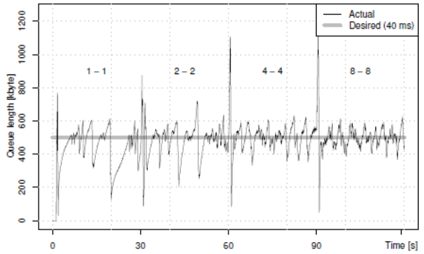




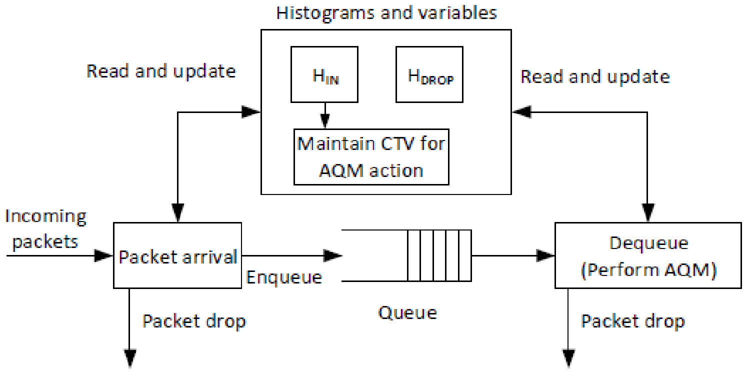
PVPIE – PPV with PIE AQM [2]

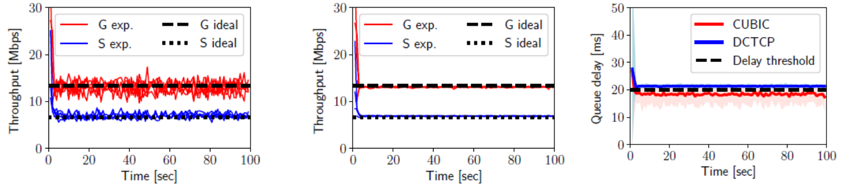






CSAQM – PPV + CC indep. AQM [3]





(a) Throughput - CUBIC (b) Throughput - DCTCP (c) Queueing delay

More

Further readings

[1] Sz. Nadas et al., Per Packet Value: A Practical Concept for Network Resource Sharing. In proc. of IEEE Globecom 2016.

[2] S. Laki et al., Take Your Own Share of the PIE, In proc. of IRTF/ACM ANRW 2017

[3] Sz. Nadas et al., Towards a Congestion Control-Independent Core-Stateless AQM, In proc. of IRTF/ACM ANRW 2018

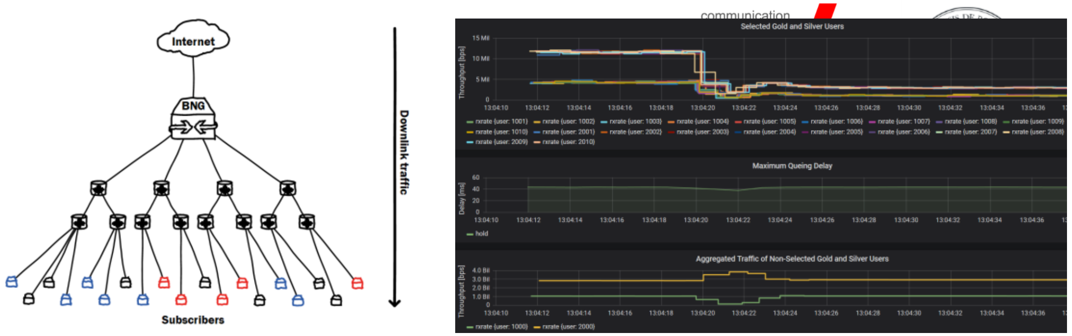
[4] S. Laki et al., Scalable Per Subscriber QoS with Core-Stateless Scheduling, Industrial demo at ACM SIGCOMM 2018

Similar approaches published recently

[5] M. Menth et al, Activity-based congestion management for fair bandwidth sharing in trusted packet networks, In proc. of IEEE/IFIP NOMS 2016

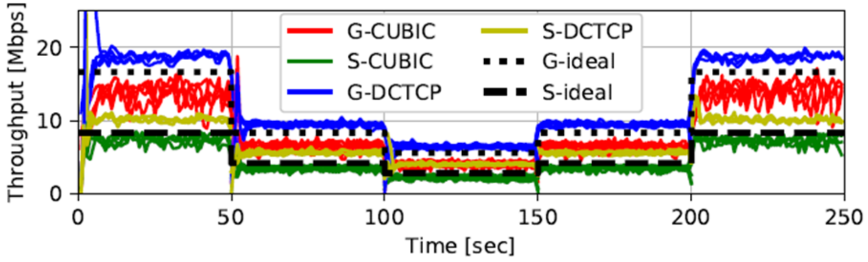
[6] M. Menth et al., Fair Resource Sharing for Stateless-Core Packet-Switched Networks with Prioritization, IEEE Access 2018.

[7] R. Bless et al., Policy-oriented AQM Steering, In proc. of IFIP Networking 2018.

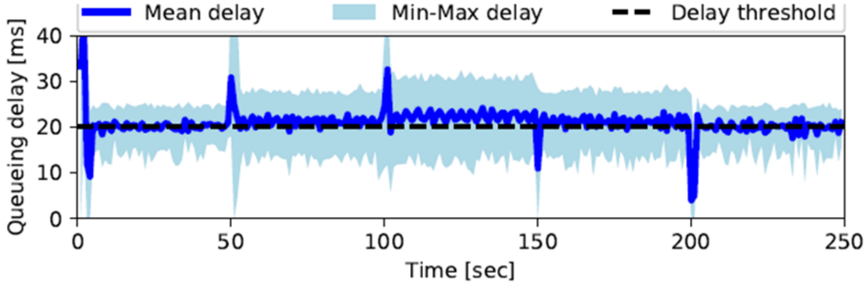


Industrial Demo at **SIGCOMM 2018**

PPV-based Core Stateless vBNG node implementation



(a) Throughput with CSAQM

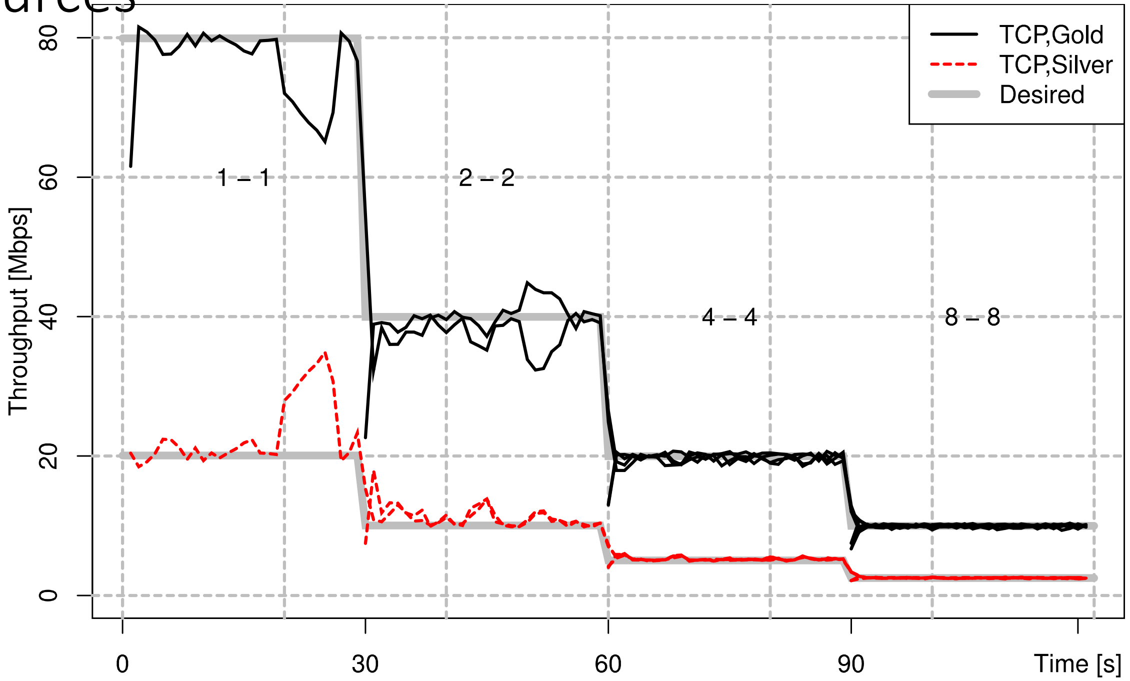


(c) Queueing delay with CSAQM

PVPIE results

Simulation Results

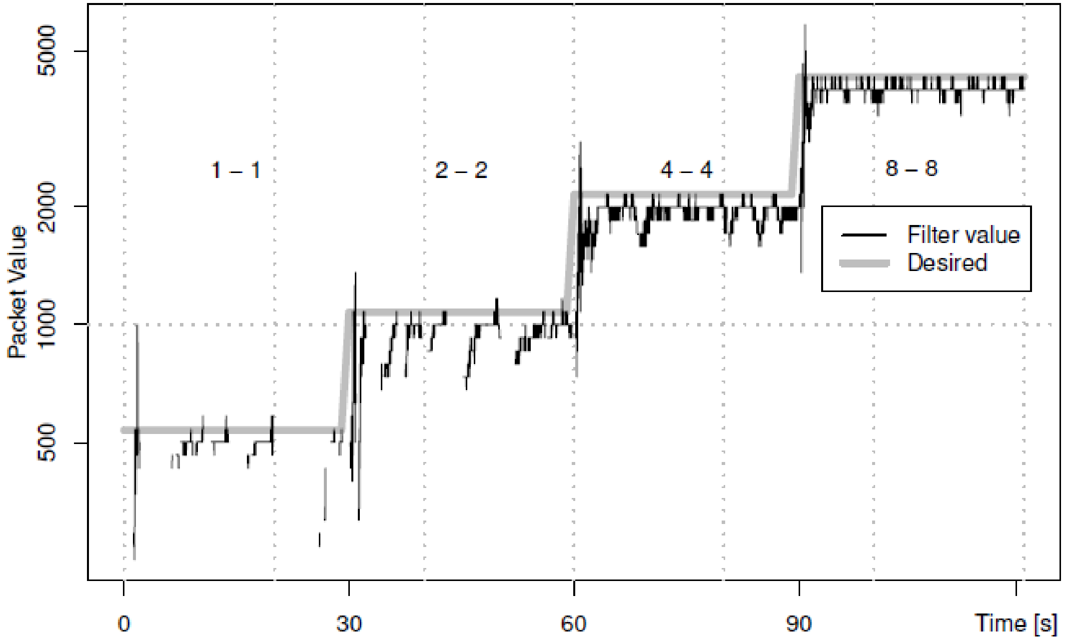
Gold and silver TCP sources



Scenario	1c	2	3a	4b
Bottleneck [Mbps]	**100**	50	10,50,100,50,10	10
Number of TCP flows (Gold-Silver)	**1-1, 2-2, 4-4, 8-8**	0-0, 1-1, 2-2, 4-4	1-1	5-0
Number of UDP flows	**0**	3 (Background)	0	2 (Silver)
Number of TCP connections / flow	**5**	1	5	5
Target Delay [ms]	**40**	40	40	20
round-trip propegation delay [ms]	**40**	40	40	100
ECDF window	**1 . T**	1 . T	1 . T	10 . T

Simulation Results

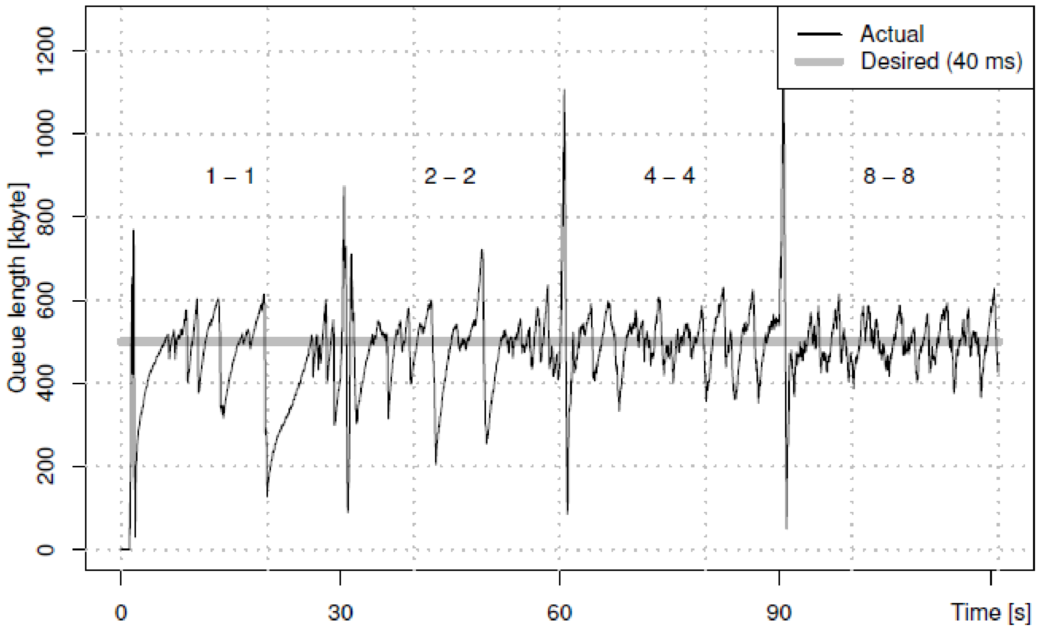
Gold and silver TCP sources



Scenario	1c	2	3a	4b
Bottleneck [Mbps]	**100**	50	10,50,100,50,10	10
Number of TCP flows (Gold-Silver)	**1-1, 2-2, 4-4, 8-8**	0-0, 1-1, 2-2, 4-4	1-1	5-0
Number of UDP flows	**0**	3 (Background)	0	2 (Silver)
Number of TCP connections / flow	**5**	1	5	5
Target Delay [ms]	**40**	40	40	20
round-trip propegation delay [ms]	**40**	40	40	100
ECDF window	**1 . T**	1 . T	1 . T	10 . T

Simulation Results

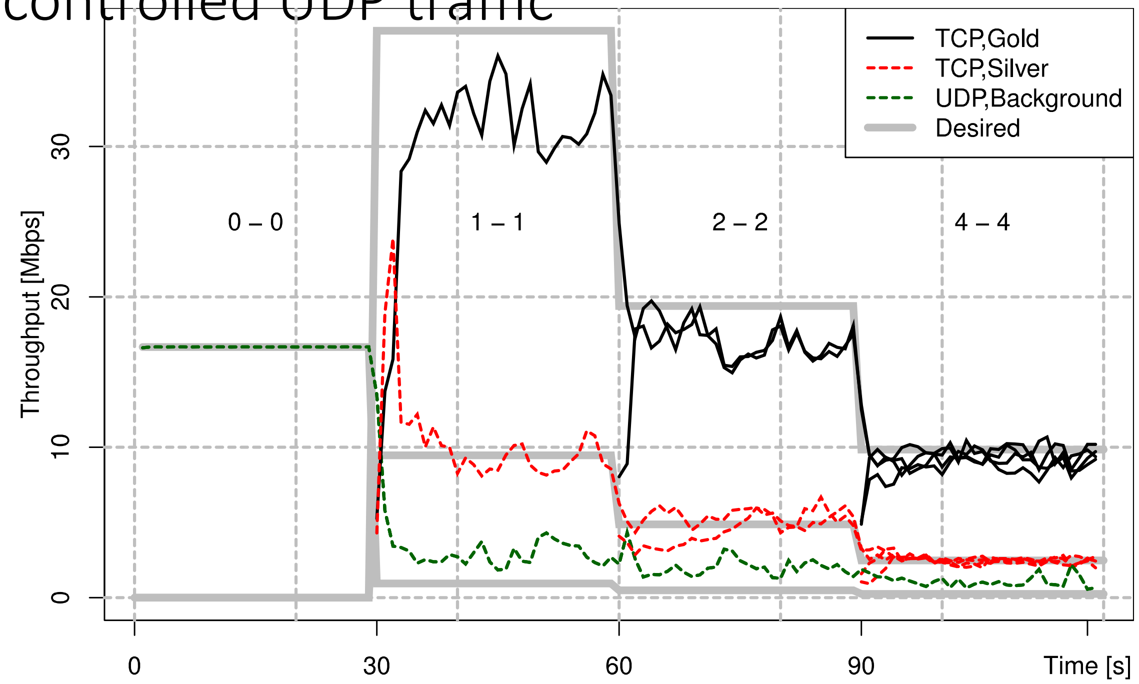
Gold and silver TCP sources



Scenario	1c	2	3a	4b
Bottleneck [Mbps]	**100**	50	10,50,100,50,10	10
Number of TCP flows (Gold-Silver)	**1-1, 2-2, 4-4, 8-8**	0-0, 1-1, 2-2, 4-4	1-1	5-0
Number of UDP flows	**0**	3 (Background)	0	2 (Silver)
Number of TCP connections / flow	**5**	1	5	5
Target Delay [ms]	**40**	40	40	20
round-trip propegation delay [ms]	**40**	40	40	100
ECDF window	**1 . T**	1 . T	1 . T	10 . T

Simulation Results

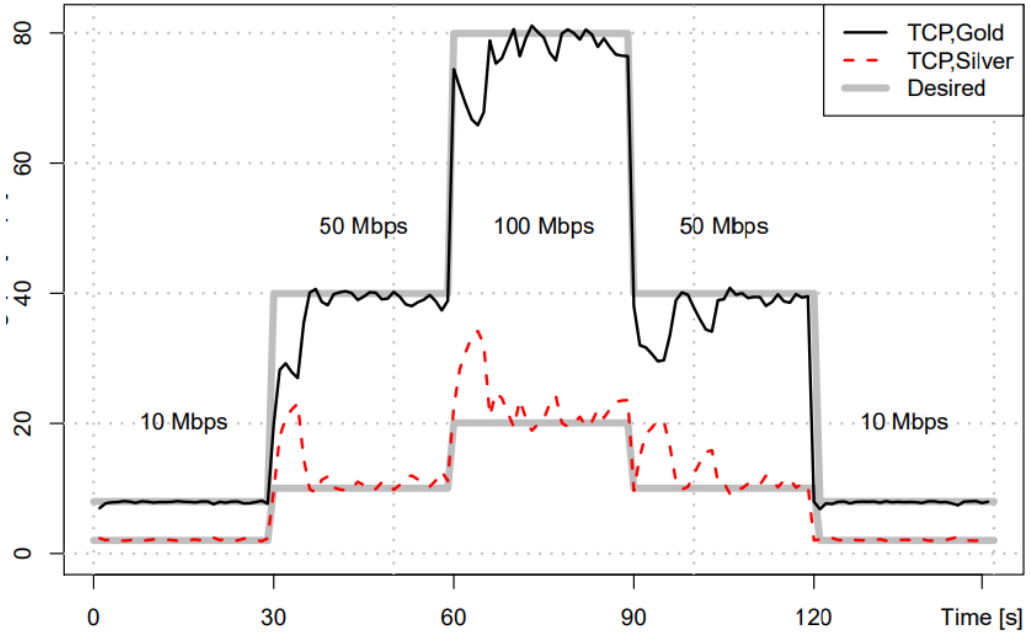
with nON-congestion controlled UDP traffic



Scenario	1c	2	3a	4b
Bottleneck [Mbps]	100	**50**	10,50,100,50,10	10
Number of TCP flows (Gold-Silver)	1-1, 2-2, 4-4, 8-8	**0-0, 1-1, 2-2, 4-4**	1-1	5-0
Number of UDP flows	0	**3 (Background)**	0	2 (Silver)
Number of TCP connections / flow	5	**1**	5	5
Target Delay [ms]	40	**40**	40	20
round-trip propegation delay [ms]	40	**40**	40	100
ECDF window	1 . T	**1 . T**	1 . T	10 . T

Simulation Results

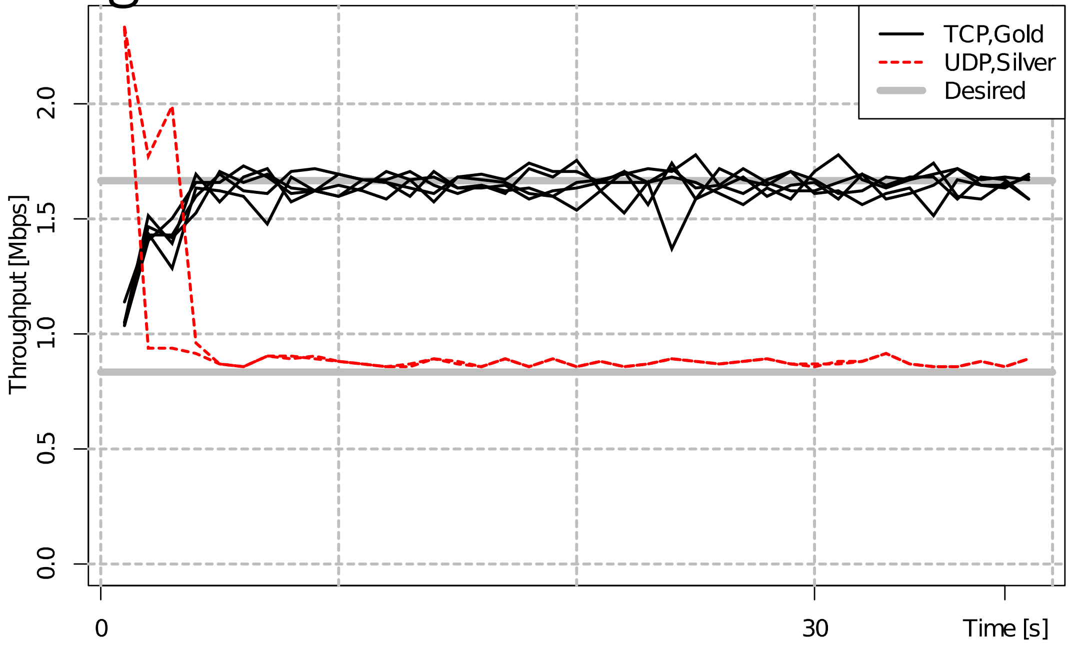
Dynamic bottleneck



Scenario	1c	2	3a	4b
Bottleneck [Mbps]	100	50	**10,50,100,50,10**	10
Number of TCP flows (Gold-Silver)	1-1, 2-2, 4-4, 8-8	0-0, 1-1, 2-2, 4-4	**1-1**	5-0
Number of UDP flows	0	3 (Background)	**0**	2 (Silver)
Number of TCP connections / flow	5	1	**5**	5
Target Delay [ms]	40	40	**40**	20
round-trip propegation delay [ms]	40	40	**40**	100
ECDF window	1 . T	1 . T	**1 . T**	10 . T

Simulation Results

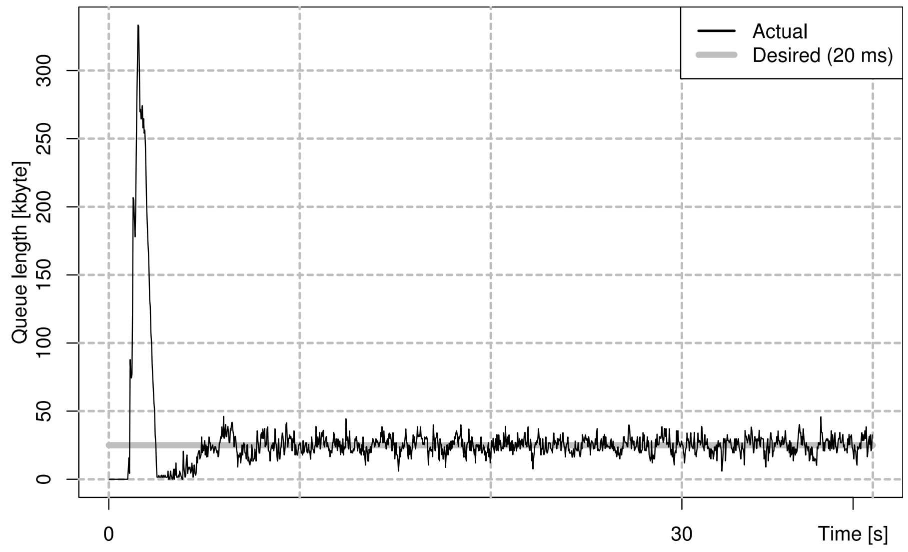
PIE with Resource sharing

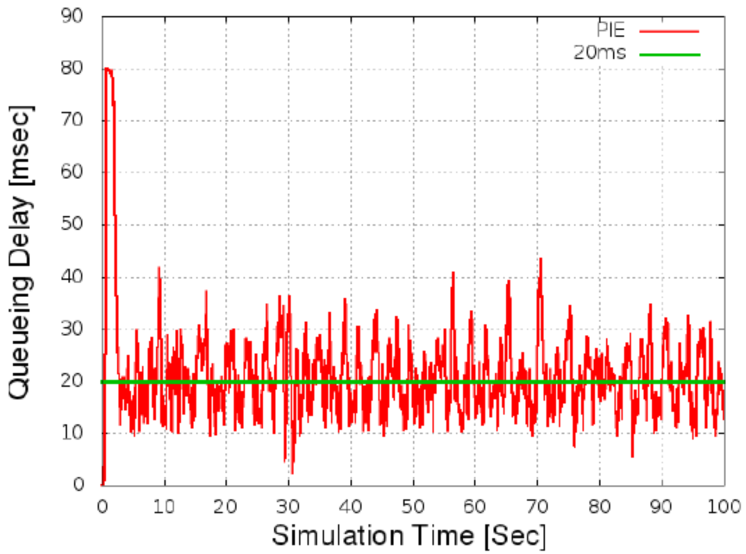


Scenario	1c	2	3a	4b
Bottleneck [Mbps]	100	50	10,50,100,50,10	**10**
Number of TCP flows (Gold-Silver)	1-1, 2-2, 4-4, 8-8	0-0, 1-1, 2-2, 4-4	1-1	**5-0**
Number of UDP flows	0	3 (Background)	0	**2 (Silver)**
Number of TCP connections / flow	5	1	5	**5**
Target Delay [ms]	40	40	40	**20**
round-trip propegation delay [ms]	40	40	40	**100**
ECDF window	1 . T	1 . T	1 . T	**10 . T**

Simulation Results

PIE with Resource sharing





(c) Mix 5TCP + 2UDP Flows

Scenario	1c	2	3a	4b
Bottleneck [Mbps]	100	50	10,50,100,50,10	**10**
Number of TCP flows (Gold-Silver)	1-1, 2-2, 4-4, 8-8	0-0, 1-1, 2-2, 4-4	1-1	**5-0**
Number of UDP flows	0	3 (Background)	0	**2 (Silver)**
Number of TCP connections / flow	5	1	5	**5**
Target Delay [ms]	40	40	40	**20**
round-trip propegation delay [ms]	40	40	40	**100**
ECDF window	1 . T	1 . T	1 . T	**10 . T**