Chair of Communication Networks
Departement of Electrical and Computer Engineering
Technical University of Munich

Loko: Predictable Latency in Small Networks

Amaury Van Bemten*, Nemanja Deric*, Johannes Zerwas*, Andreas Blenk*, Stefan Schmid° and Wolfgang Kellerer*
amaury.van-bemten@tum.de

*Technical University of Munich (Munich, Germany)
°University of Vienna (Vienna, Austria)

December, 12 2019 – CoNEXT, Orlando, FL (USA)

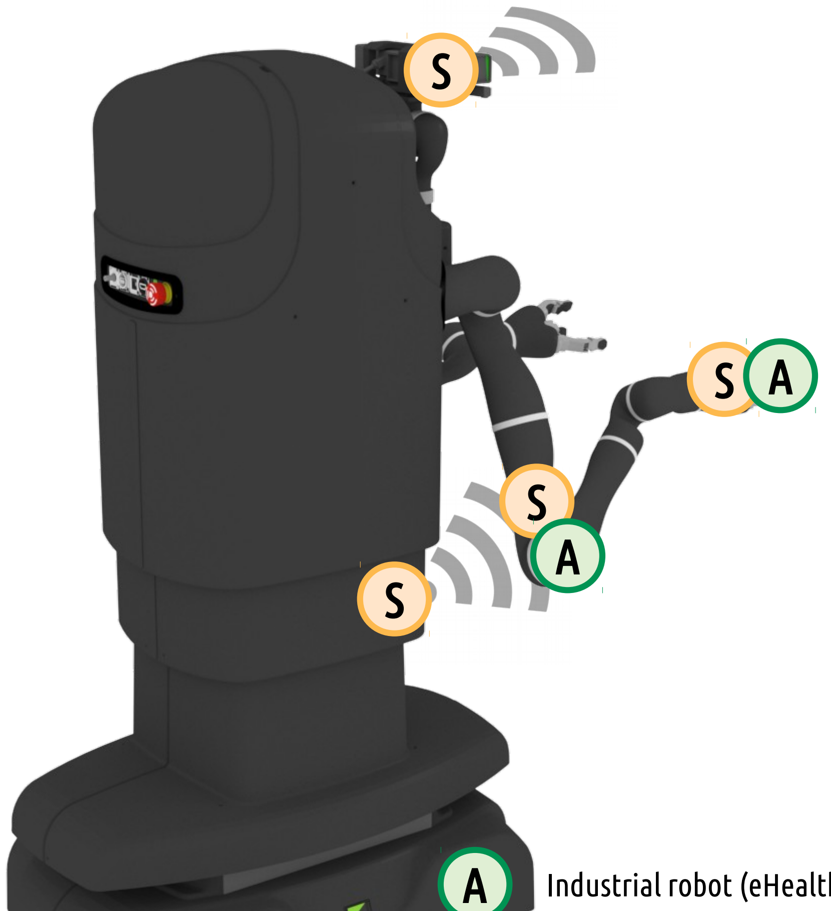
Loko: Predictable Latency in Small Networks

Loko: Predictable Latency in Small Networks

Loko: Predictable Latency in Small Networks

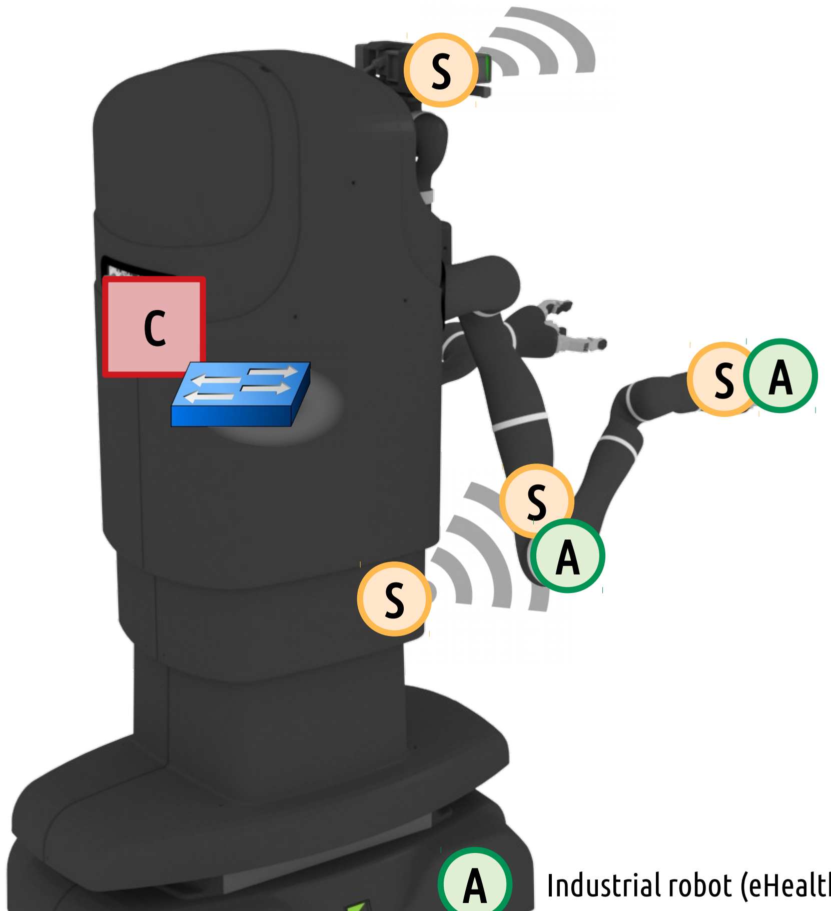
Industrial robot (eHealth 5G Research Hub Munich)

Loko: Predictable Latency in Small Networks



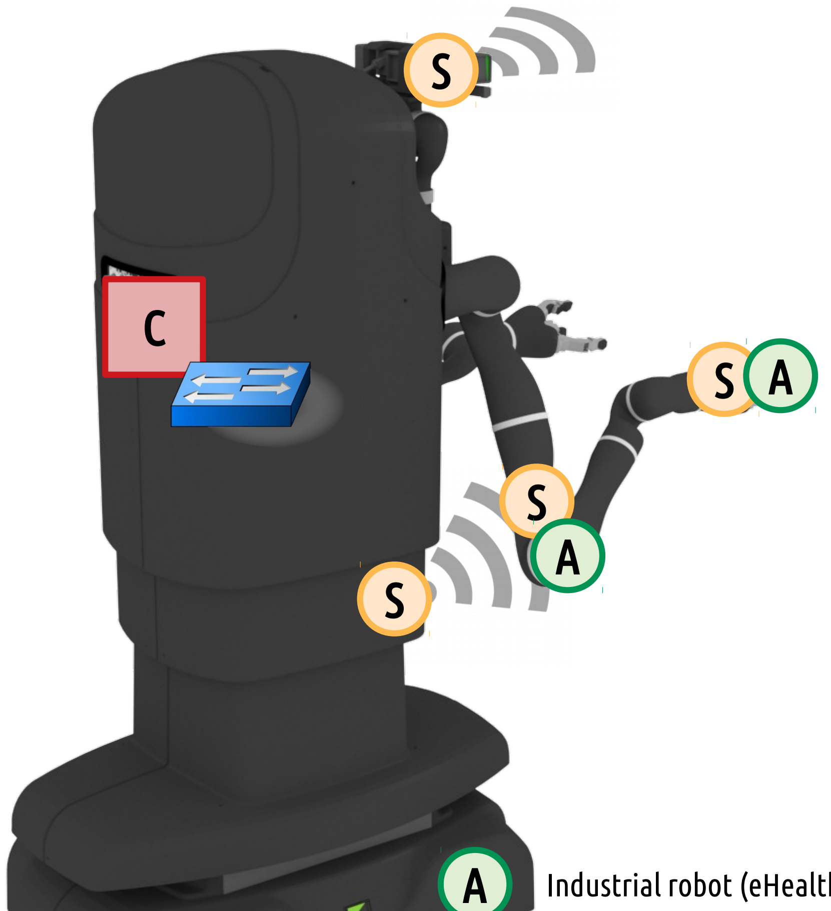
Industrial robot (eHealth 5G Research Hub Munich)

Loko: Predictable Latency in Small Networks



Industrial robot (eHealth 5G Research Hub Munich)

Loko: Predictable Latency in Small Networks

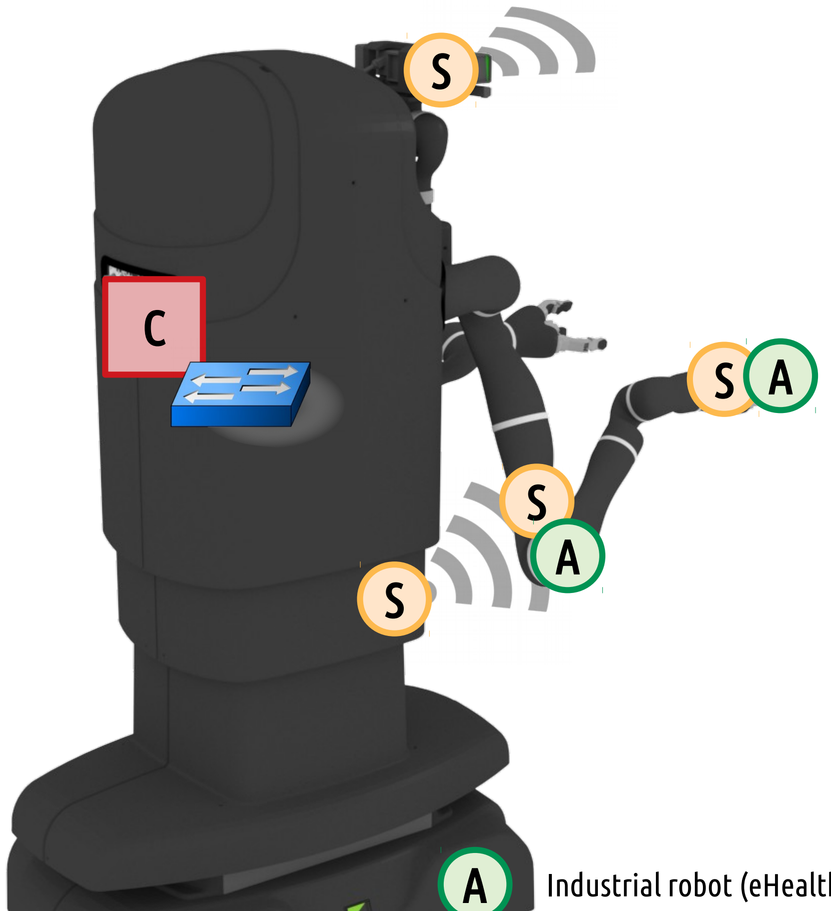


Industrial robot (eHealth 5G Research Hub Munich)

Low-capacity

~kbps, up to few Mbps, predictable traffic patterns

Loko: Predictable Latency in Small Networks



Industrial robot (eHealth 5G Research Hub Munich)

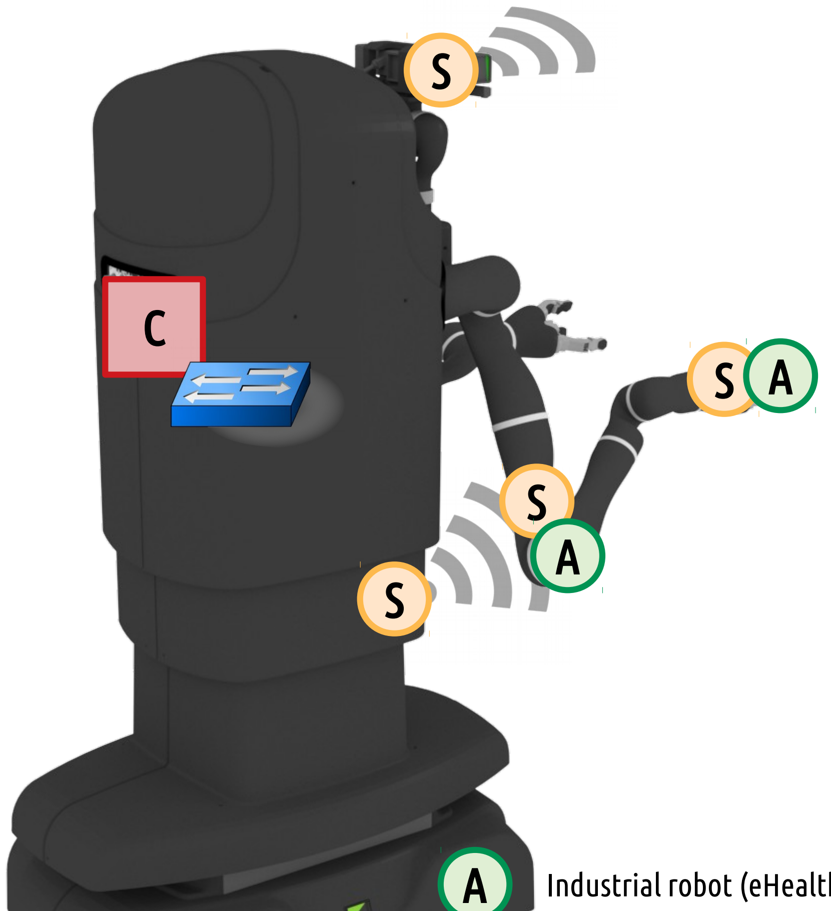
Low-capacity

~kbps, up to few Mbps, predictable traffic patterns

Small devices

Devices have to fit in small (~cm²) areas

Loko: Predictable Latency in Small Networks



Industrial robot (eHealth 5G Research Hub Munich)

Low-capacity

~kbps, up to few Mbps, predictable traffic patterns

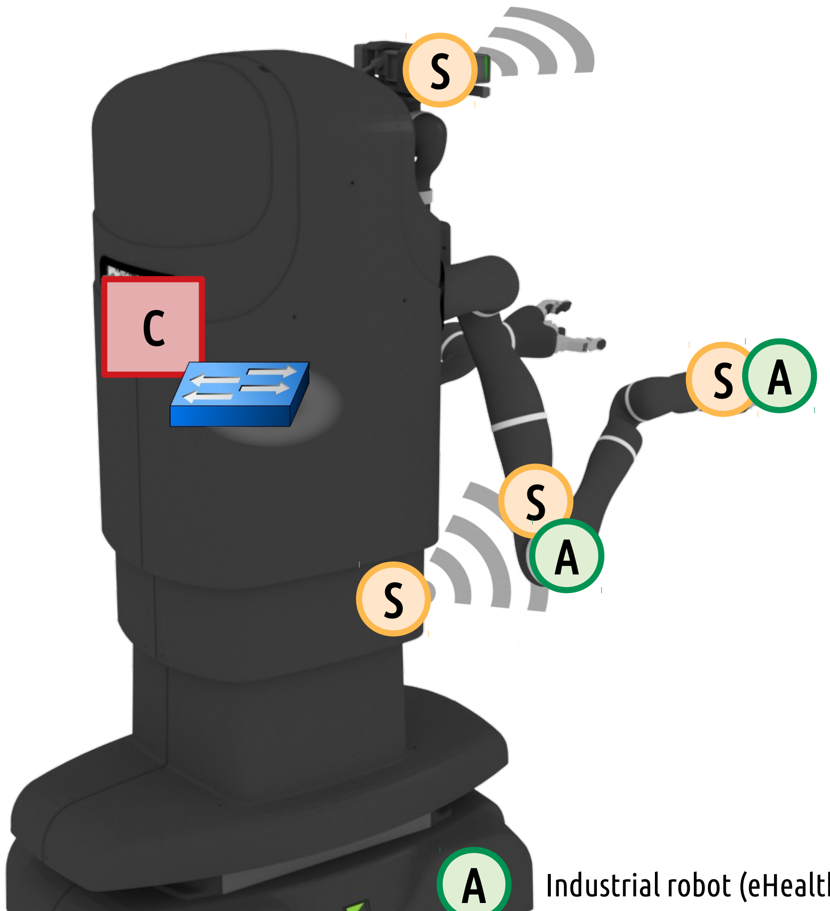
Small devices

Devices have to fit in small (~cm²) areas

Lightweight

Power consumption and physical constraints

Loko: Predictable Latency in Small Networks



Industrial robot (eHealth 5G Research Hub Munich)

Low-capacity

~kbps, up to few Mbps, predictable traffic patterns

Small devices

Devices have to fit in small (~cm²) areas

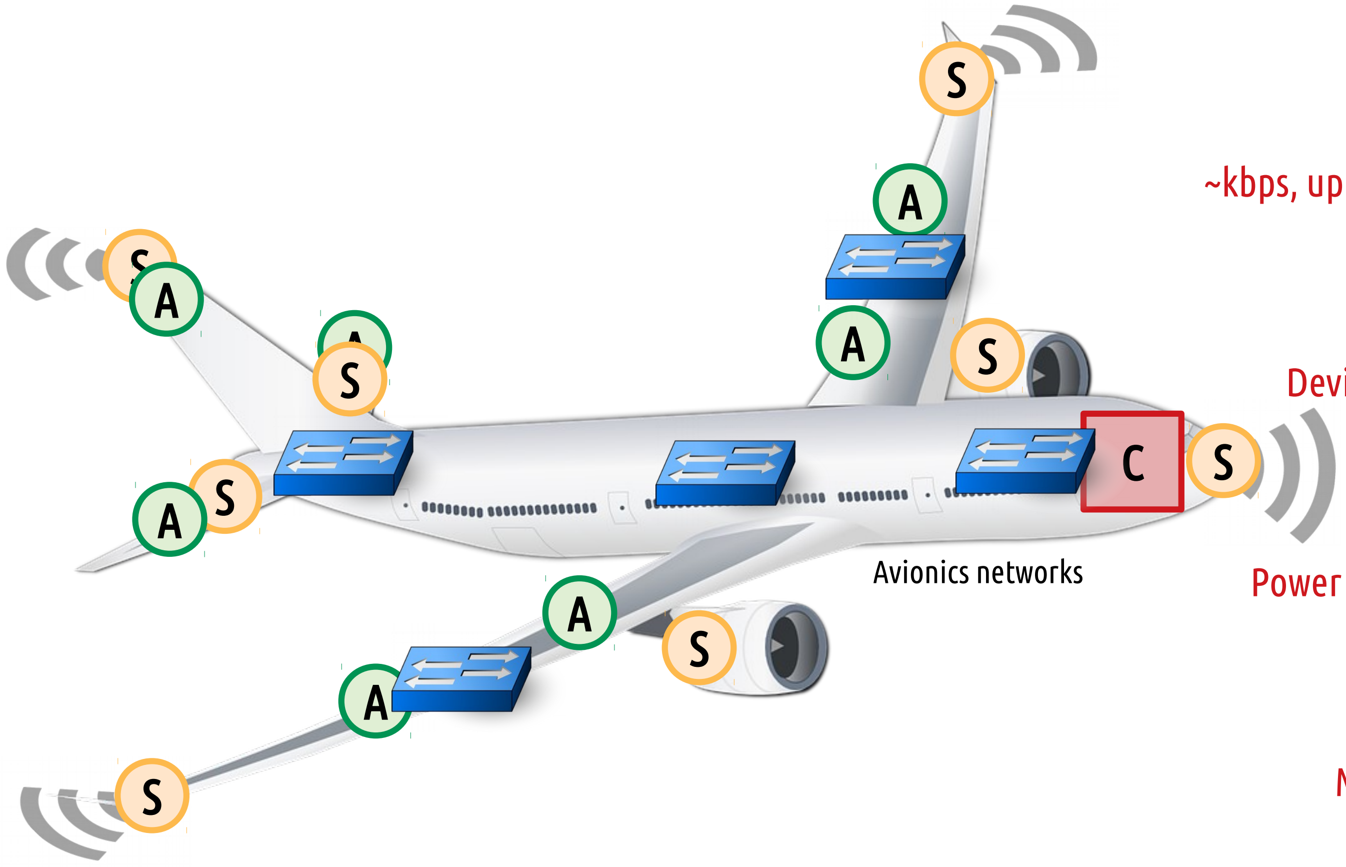
Lightweight

Power consumption and physical constraints

Low-cost

Many instances of such networks

Loko: Predictable Latency in Small Networks



Low-capacity

~kbps, up to few Mbps, predictable traffic patterns

Small devices

Devices have to fit in small (~cm²) areas

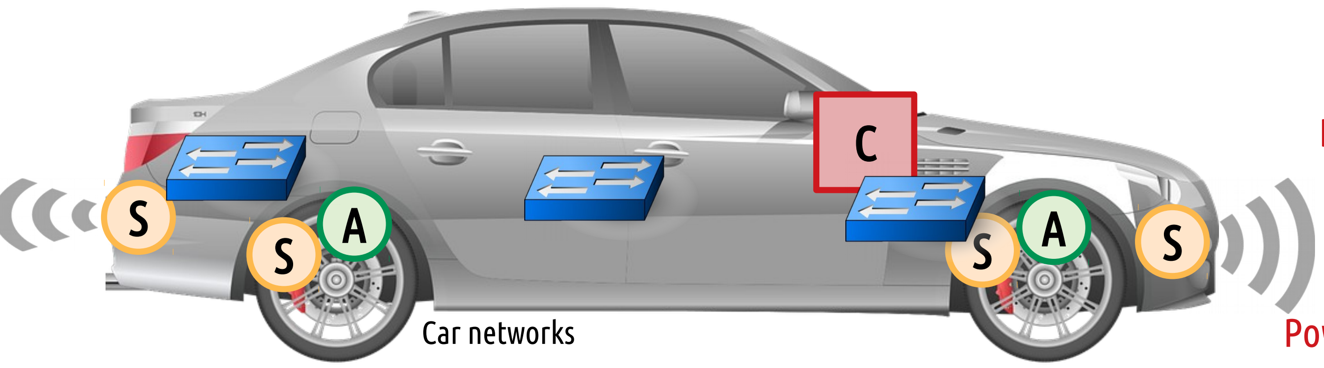
Lightweight

Power consumption and physical constraints

Low-cost

Many instances of such networks

Loko: Predictable Latency in Small Networks



Car networks

Low-capacity

~kbps, up to few Mbps, predictable traffic patterns

Small devices

Devices have to fit in small (~cm²) areas

Lightweight

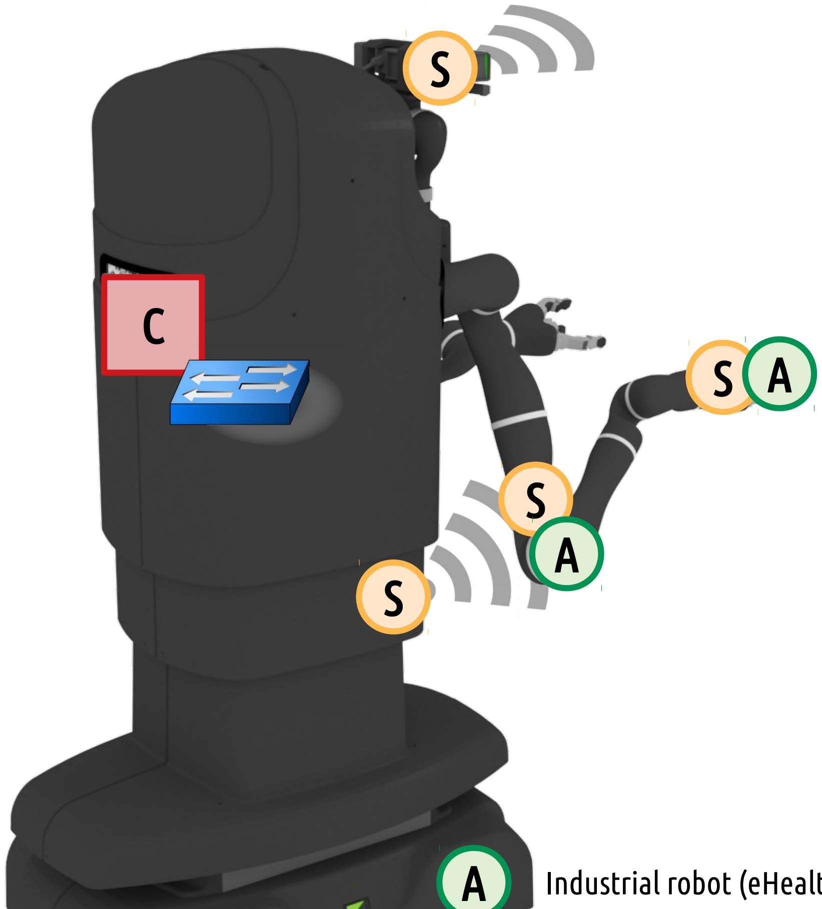
Power consumption and physical constraints

Low-cost

Many instances of such networks

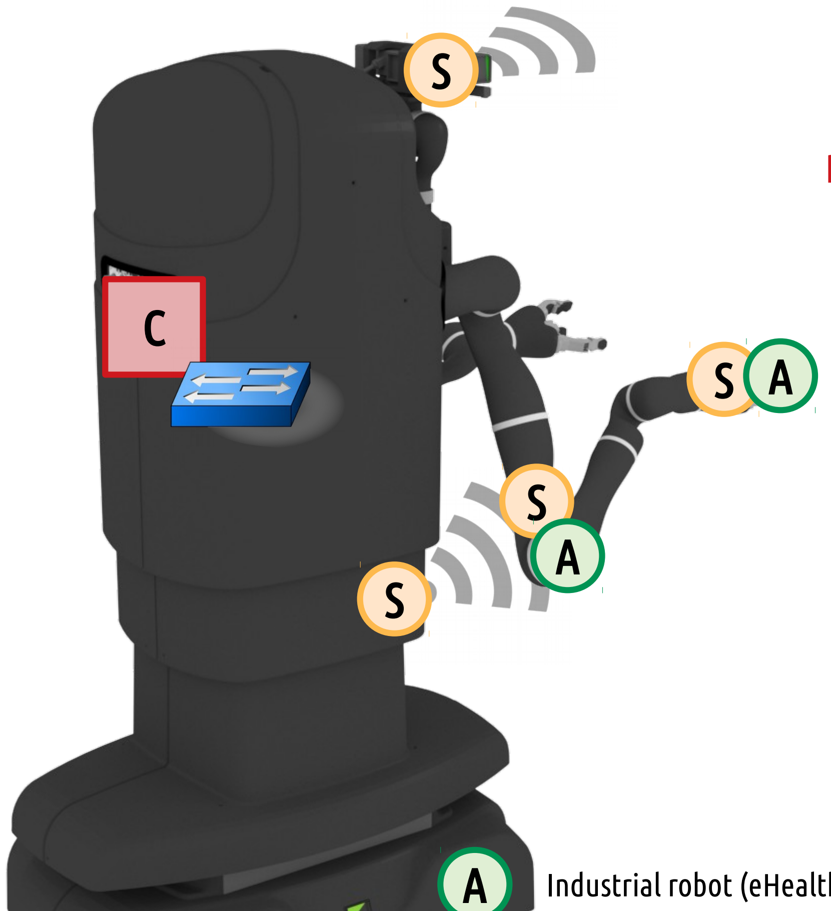
Loko: **Predictable Latency** in Small Networks

Loko: **Predictable Latency** in Small Networks



Industrial robot (eHealth 5G Research Hub Munich)

Loko: Predictable Latency in Small Networks



Industrial robot (eHealth 5G Research Hub Munich)

Hard latency requirements

Per-packet **100% guaranteed** max. latency (~µs, ms)

THEY SAY THAT TIME IS MONEY
THIS IS PARTICULARLY TRUE WHEN YOU HAVE A GOLD WATCH

Loko: Predictable Latency in Small Networks

State-of-the-Art?

Loko: Predictable Latency in Small Networks

<u>State-of-the-Art?</u>

PROFI NET®

EtherCAT®

EtherNet/IP™

ETHERNET POWERLINK

lin
LOCAL INTERCONNECT NETWORK

PROFI BUS®

FlexRay™

CAN

Modbus

Loko: Predictable Latency in Small Networks

<u>State-of-the-Art?</u>

PROFI NET®

EtherCAT®

EtherNet/IP™

ETHERNET POWERLINK

lin
LOCAL INTERCONNECT NETWORK

PROFI BUS®

FlexRay™

CAN

Modbus

?

proprietary or not interoperable: expensive, specialized hardware, vendor lock-in, **inflexible**

Loko: Predictable Latency in Small Networks

Loko: Predictable Latency in ~~Small Networks~~

Loko: Predictable Latency in Small Programmable Networks

Loko: Predictable Latency in ~~Small Networks~~

Loko: Predictable Latency in Small Programmable Networks

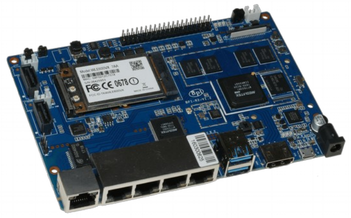
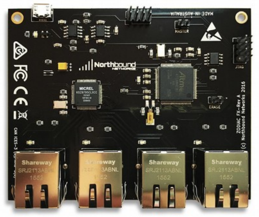
State-of-the-Art?

Loko: Predictable Latency in ~~Small Networks~~
Loko: Predictable Latency in Small Programmable Networks

State-of-the-Art?

small programmable
HARDWARE

Low-capacity
~kbps, up to few Mbps, predictable traffic patterns
Small devices
Devices have to fit in small (~cm²) areas
Lightweight
Power consumption and physical constraints
Low-cost
Many instances of such networks

Banana Pi R1	Banana Pi R2	Zodiac FX	Zodiac GX
~$90	~$125	~$70	~$120
5x1G	5x1G	4x100M	5x1G
83 gr.	100 gr.	115 gr.	765 gr.
148 mm × 100mm	148 mm × 100.5mm	100mm × 80mm	232mm × 142mm × 45mm

Loko: Predictable Latency in ~~Small Networks~~
Loko: Predictable Latency in Small Programmable Networks

State-of-the-Art?

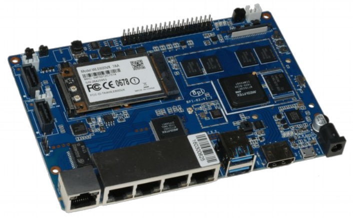
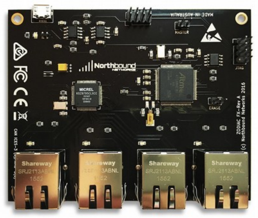
small programmable
HARDWARE

Low-capacity
~kbps, up to few Mbps, predictable traffic patterns

Small devices
Devices have to fit in small (~cm²) areas

Lightweight
Power consumption and physical constraints

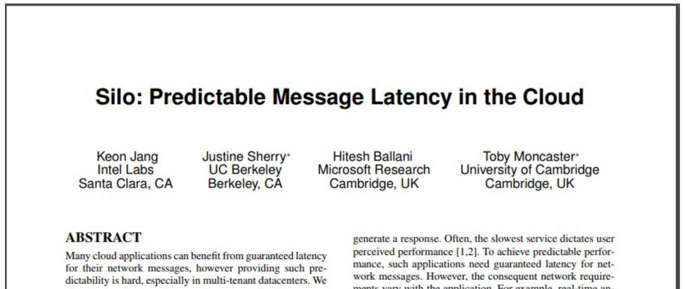
Low-cost
Many instances of such networks

Banana Pi R1	**Banana Pi R2**	**Zodiac FX**	**Zodiac GX**
~$90	~$125	~$70	~$120
5x1G	5x1G	4x100M	5x1G
83 gr.	100 gr.	115 gr.	765 gr.
148 mm × 100mm	148 mm × 100.5mm	100mm × 80mm	232mm × 142mm × 45mm

predictable latency
SOLUTION
for progr. networks

Silo [SIGCOMM15]



Silo: Predictable Message Latency in the Cloud

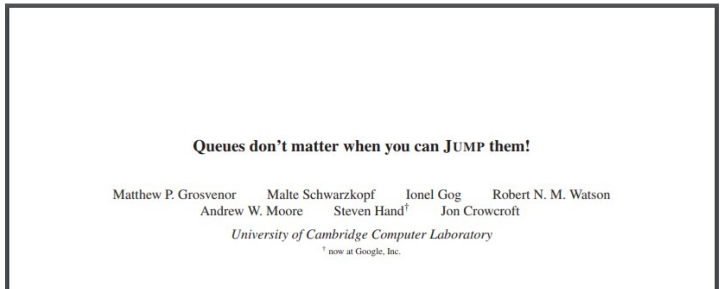
Keon Jang, Intel Labs, Santa Clara, CA
Justine Sherry*, UC Berkeley, Berkeley, CA
Hitesh Ballani, Microsoft Research, Cambridge, UK
Toby Moncaster*, University of Cambridge, Cambridge, UK

ABSTRACT

Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network require-

QJump [NSDI15]



Queues don't matter when you can JUMP them!

Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand[†], Jon Crowcroft

University of Cambridge Computer Laboratory

[†] now at Google, Inc.

Loko: Predictable Latency in ~~Small Networks~~

Loko: Predictable Latency in Small Programmable Networks

State-of-the-Art?

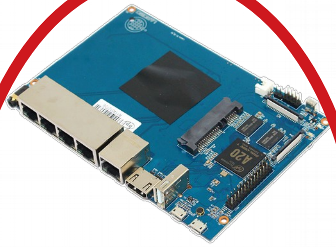
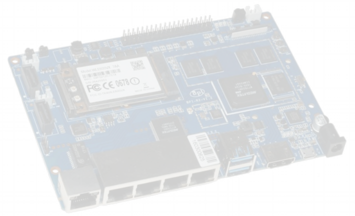
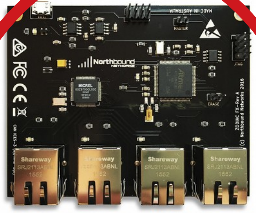
small programmable
HARDWARE

Low-capacity
~kbps, up to few Mbps, predictable traffic patterns

Small devices
Devices have to fit in small (~cm²) areas

Lightweight
Power consumption and physical constraints

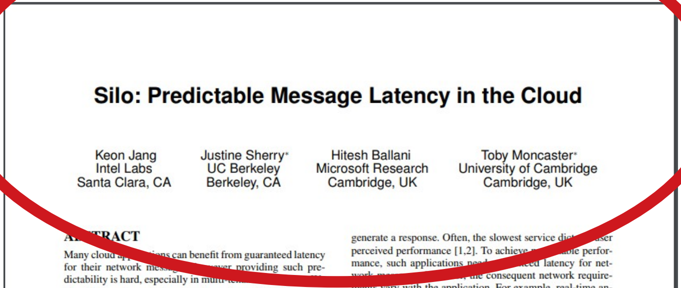
Low-cost
Many instances of such networks

Banana Pi R1	Banana Pi R2	Zodiac FX	Zodiac GX
~$90	~$125	~$70	~$120
5x1G	5x1G	4x100M	5x1G
83 gr.	100 gr.	115 gr.	765 gr.
148 mm × 100mm	148 mm × 100.5mm	100mm × 80mm	232mm × 142mm × 45mm

predictable latency
SOLUTION
for progr. networks

Silo [SIGCOMM15]



Silo: Predictable Message Latency in the Cloud

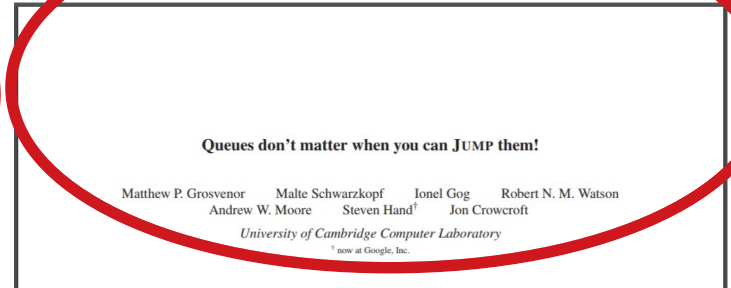
Keon Jang, Intel Labs, Santa Clara, CA
Justine Sherry*, UC Berkeley, Berkeley, CA
Hitesh Ballani, Microsoft Research, Cambridge, UK
Toby Moncaster*, University of Cambridge, Cambridge, UK

A[illegible]TRACT

Many cloud [illegible]ions can benefit from guaranteed latency for their network [illegible] providing such predictability is hard, especially in multi-te[illegible]

generate a response. Often, the slowest service [illegible] user perceived performance [1,2]. To achieve [illegible] performance, such applications [illegible] latency for network [illegible] the consequent network require-

QJump [NSDI15]

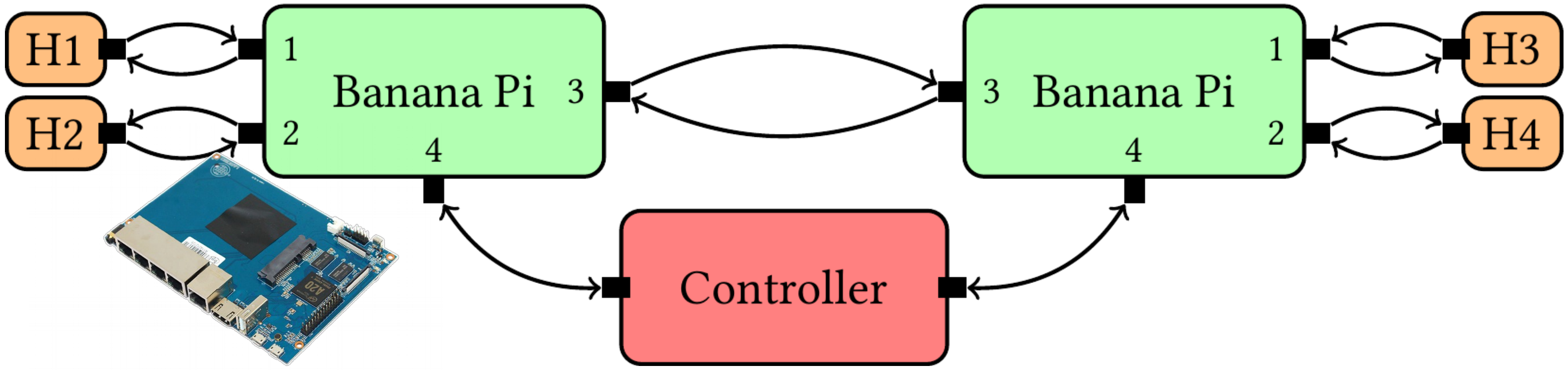


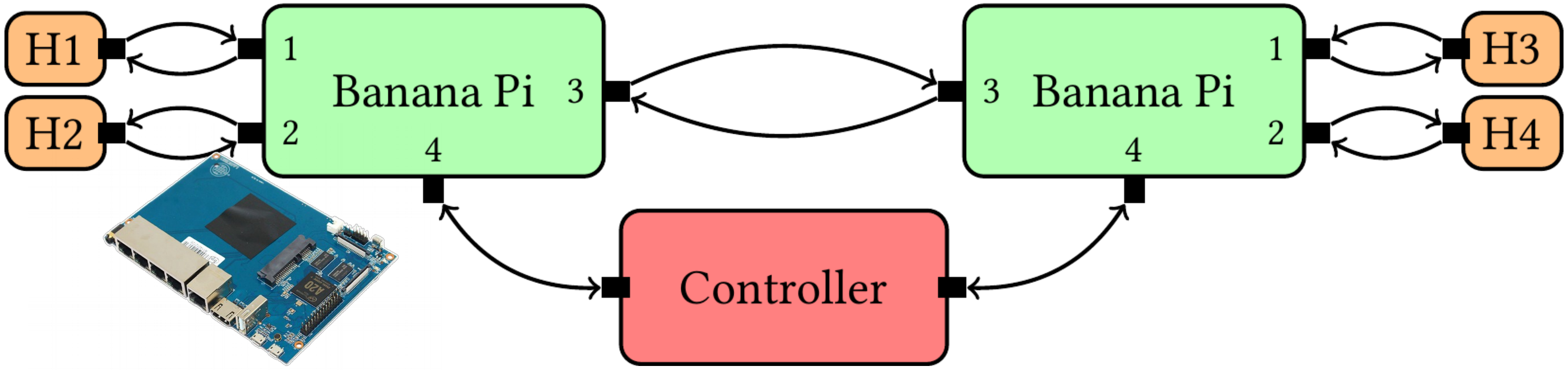
Queues don't matter when you can JUMP them!

Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand[†], Jon Crowcroft

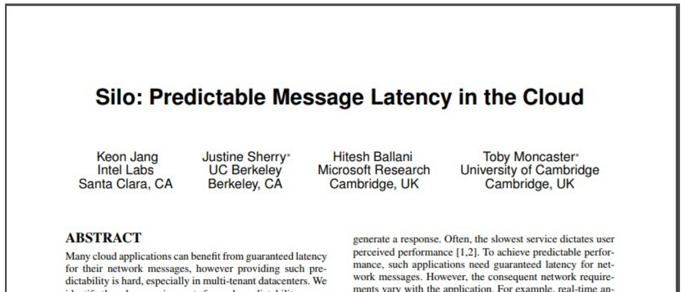
University of Cambridge Computer Laboratory

[†] now at Google, Inc.


H1
H2
H3
H4
1
2
3
4
Banana Pi
3
Banana Pi
1
2
4
Controller



Silo [SIGCOMM15]



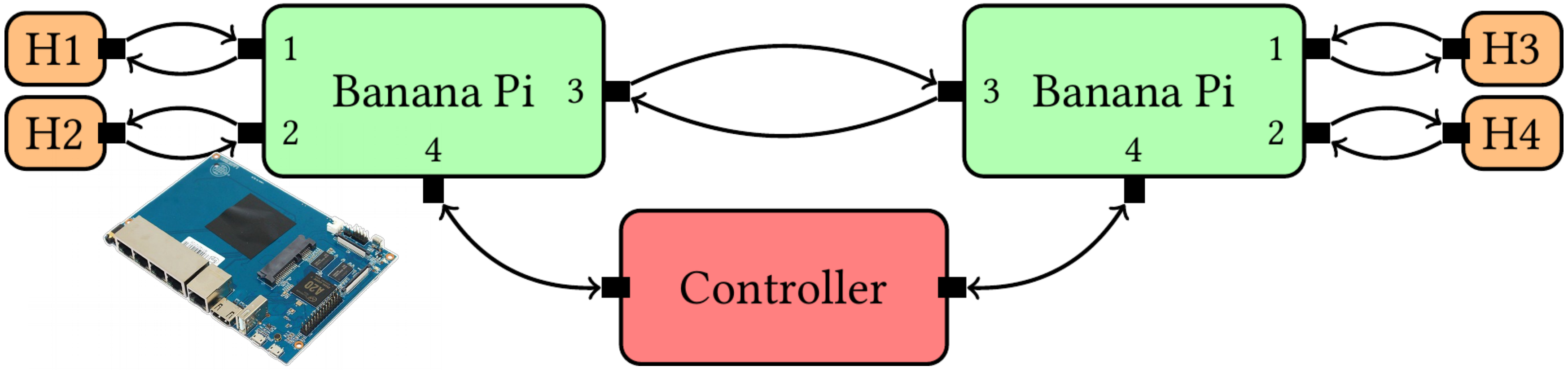
Silo: Predictable Message Latency in the Cloud

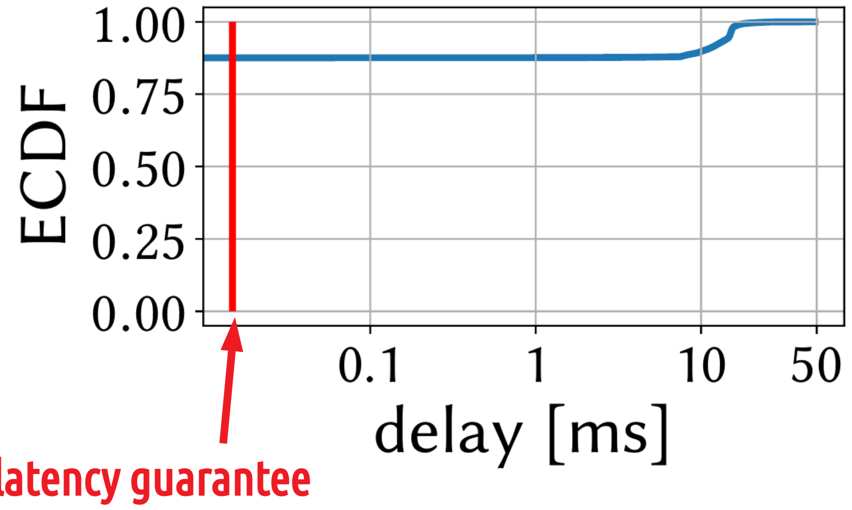
Keon Jang, Intel Labs, Santa Clara, CA
Justine Sherry*, UC Berkeley, Berkeley, CA
Hitesh Ballani, Microsoft Research, Cambridge, UK
Toby Moncaster*, University of Cambridge, Cambridge, UK

ABSTRACT

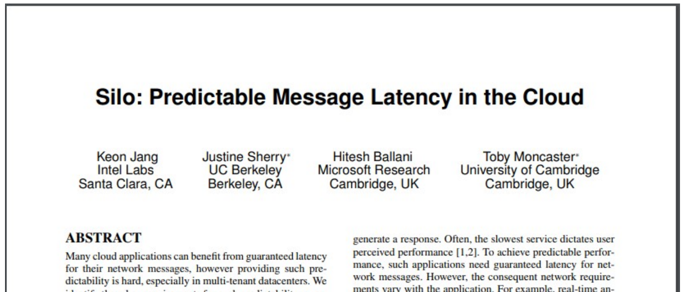
Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network requirements vary with the application. For example, real-time an-





Silo [SIGCOMM15]



Silo: Predictable Message Latency in the Cloud

Keon Jang
Intel Labs
Santa Clara, CA

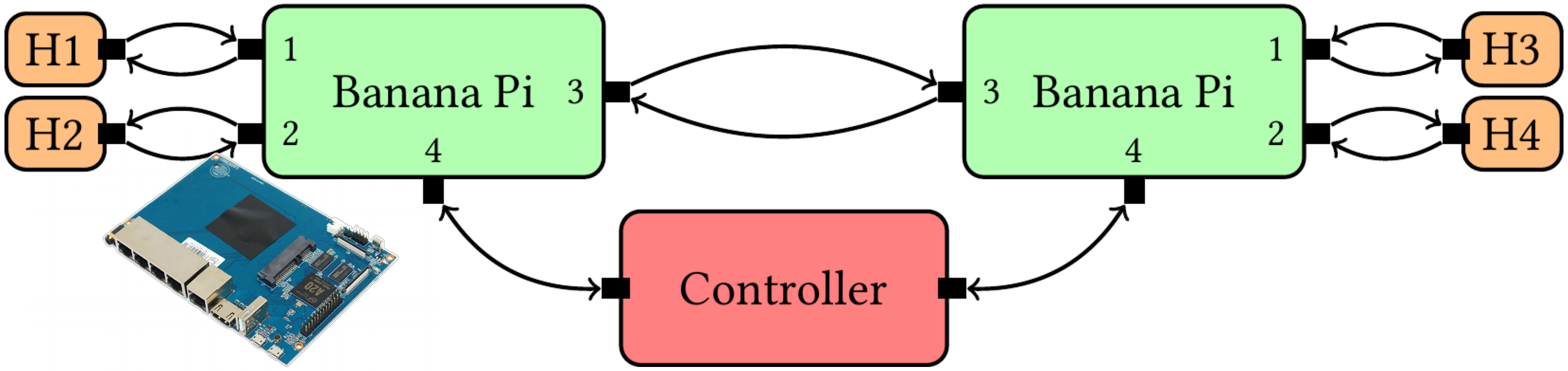
Justine Sherry*
UC Berkeley
Berkeley, CA

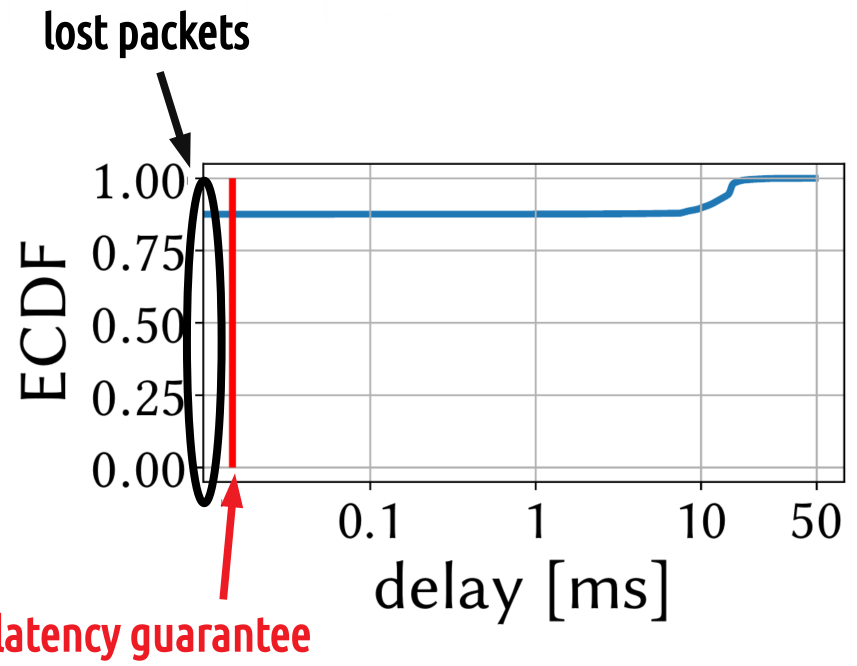
Hitesh Ballani
Microsoft Research
Cambridge, UK

Toby Moncaster*
University of Cambridge
Cambridge, UK

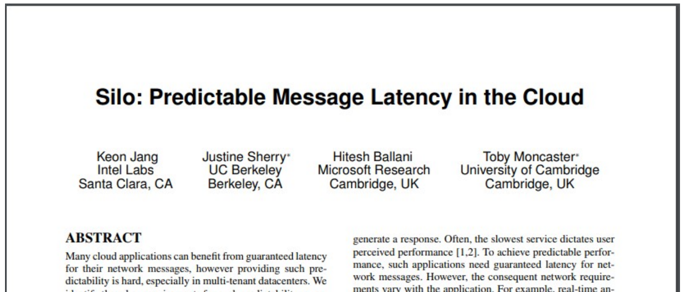
ABSTRACT
Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network requirements vary with the application. For example, real-time an-





Silo [SIGCOMM15]



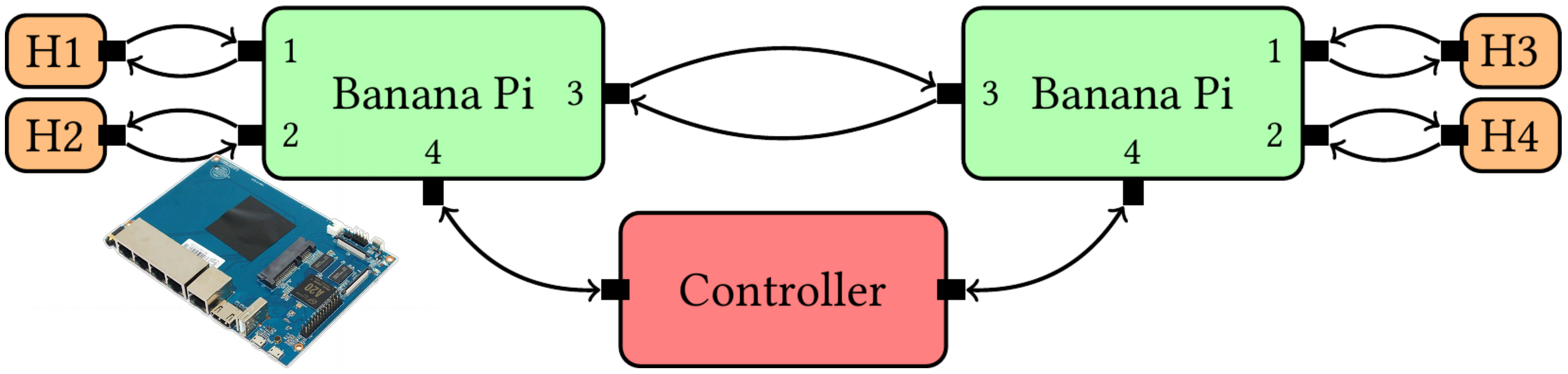
Silo: Predictable Message Latency in the Cloud

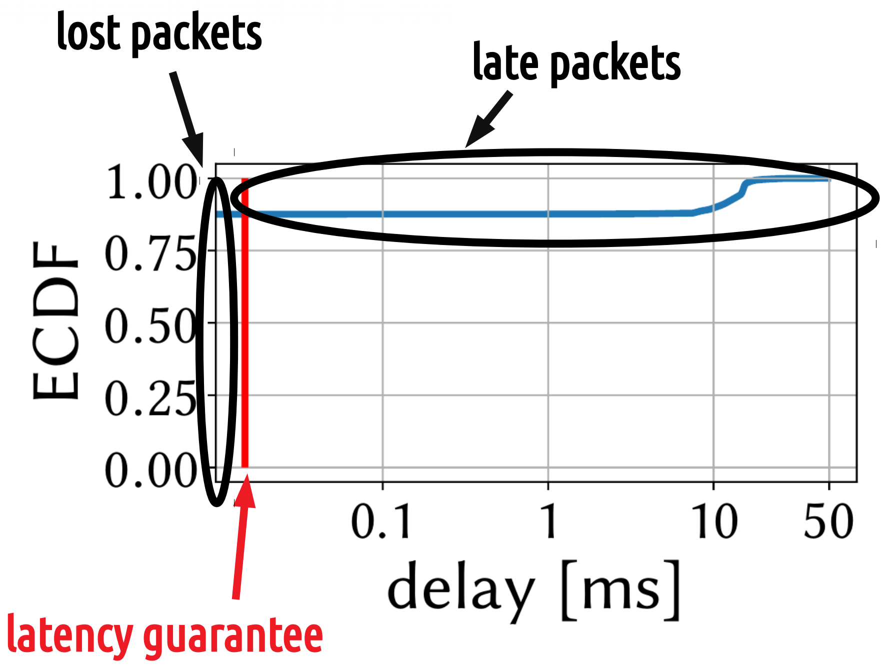
Keon Jang	Justine Sherry*	Hitesh Ballani	Toby Moncaster*
Intel Labs	UC Berkeley	Microsoft Research	University of Cambridge
Santa Clara, CA	Berkeley, CA	Cambridge, UK	Cambridge, UK

ABSTRACT

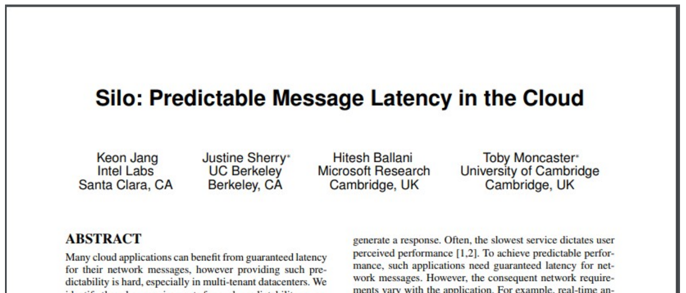
Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network requirements vary with the application. For example, real-time an-





Silo [SIGCOMM15]



Silo: Predictable Message Latency in the Cloud

Keon Jang
Intel Labs
Santa Clara, CA

Justine Sherry*
UC Berkeley
Berkeley, CA

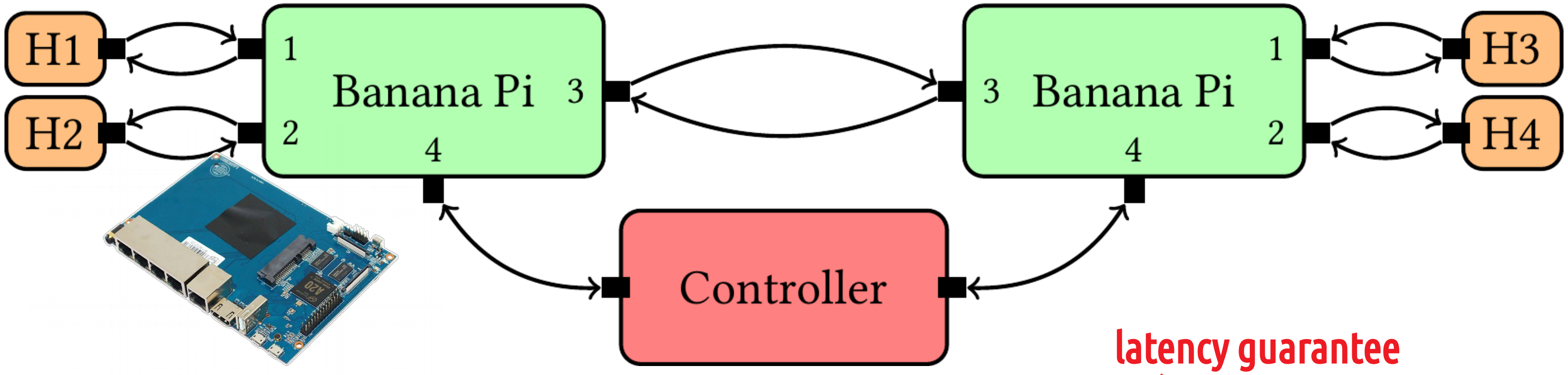
Hitesh Ballani
Microsoft Research
Cambridge, UK

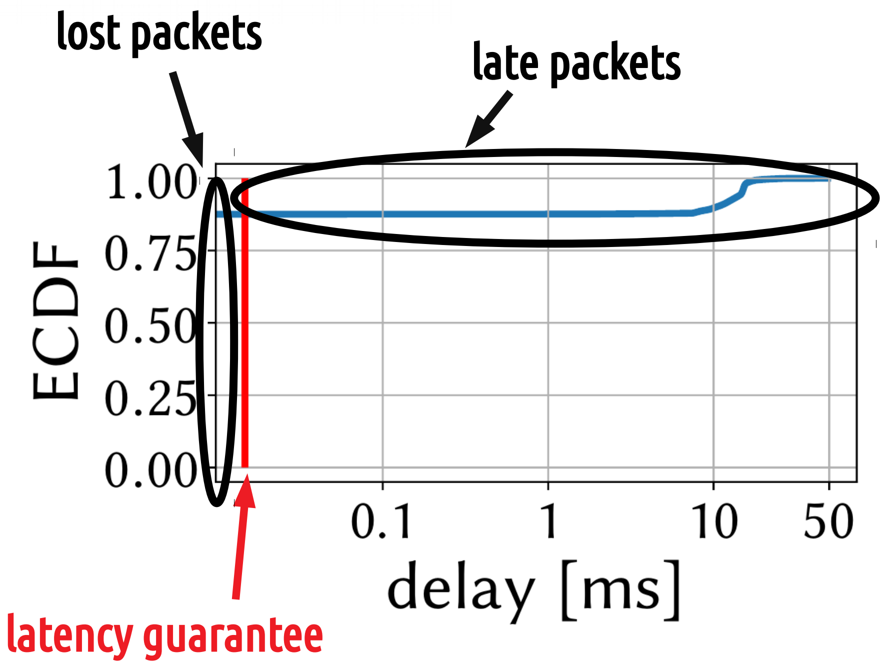
Toby Moncaster*
University of Cambridge
Cambridge, UK

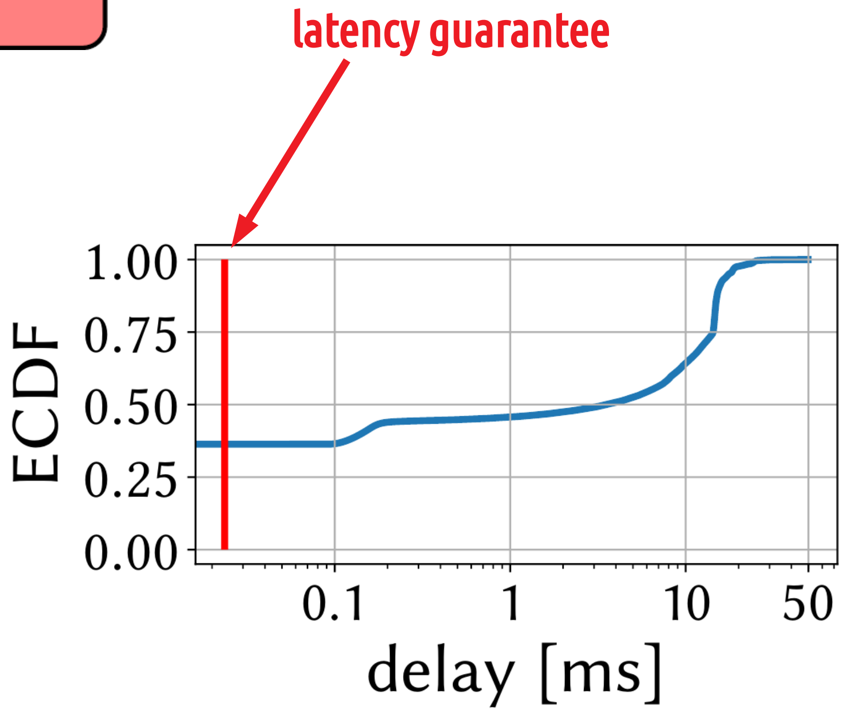
ABSTRACT

Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

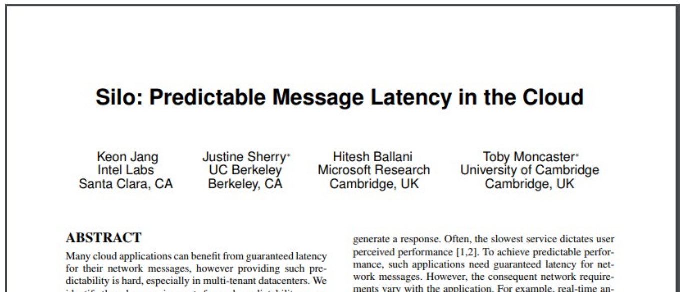
generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network requirements vary with the application. For example, real-time an-







Silo [SIGCOMM15]



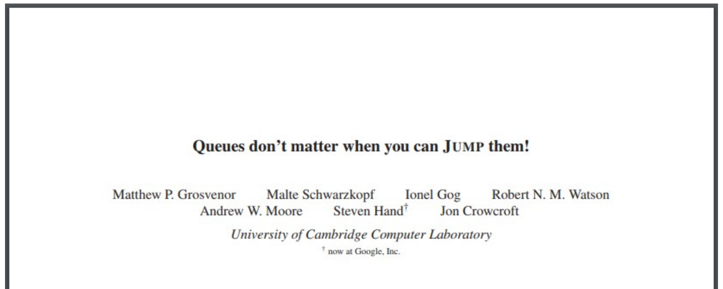
Silo: Predictable Message Latency in the Cloud

Keon Jang, Intel Labs, Santa Clara, CA
Justine Sherry*, UC Berkeley, Berkeley, CA
Hitesh Ballani, Microsoft Research, Cambridge, UK
Toby Moncaster*, University of Cambridge, Cambridge, UK

ABSTRACT
Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

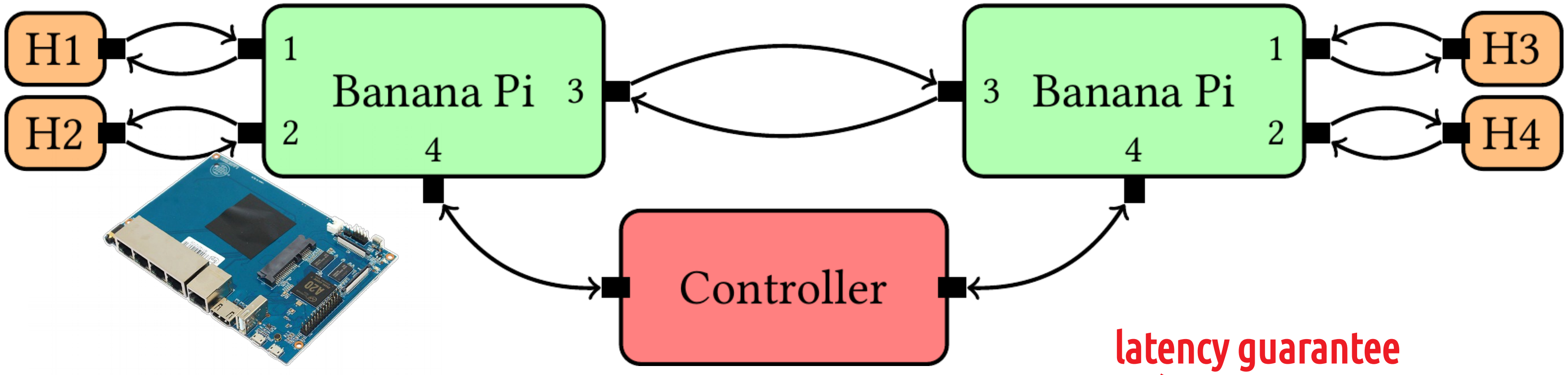
generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network require-

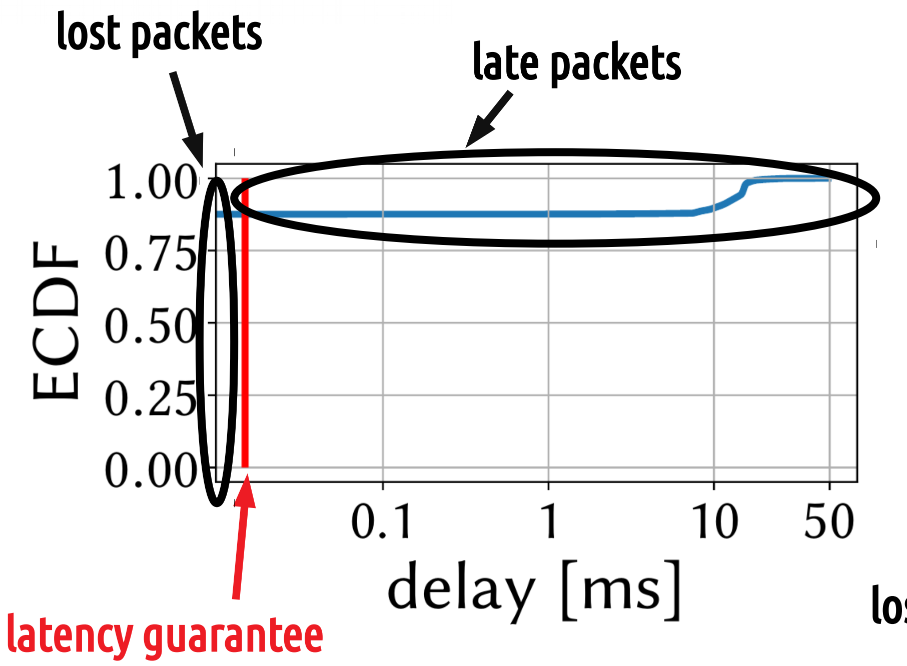
QJump [NSDI15]

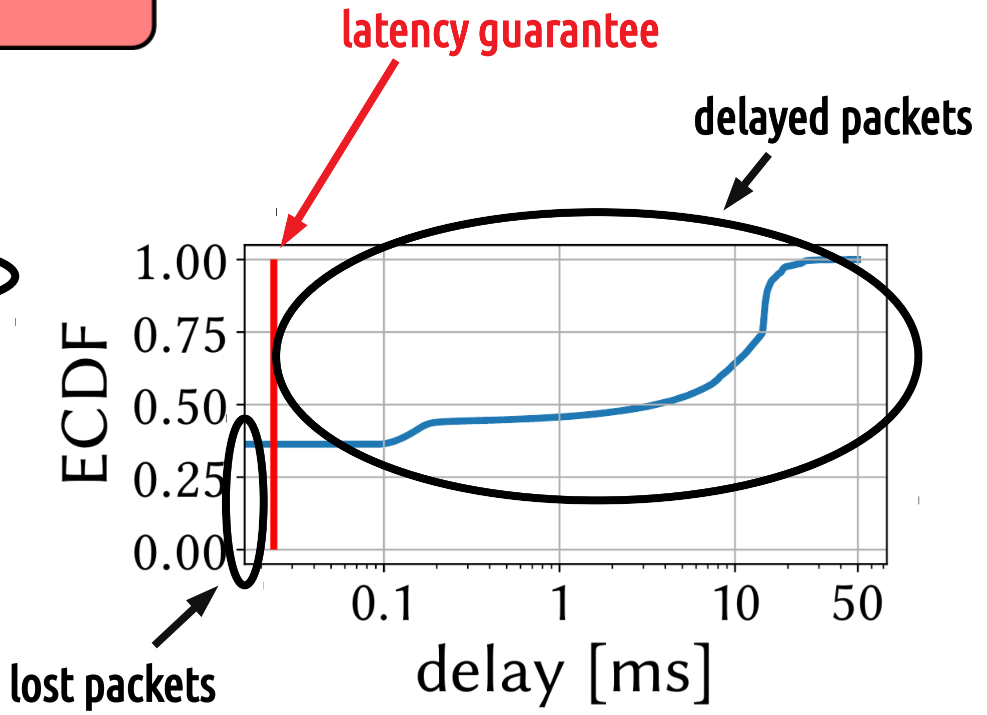


Queues don't matter when you can JUMP them!

Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson
Andrew W. Moore, Steven Hand†, Jon Crowcroft
University of Cambridge Computer Laboratory
† now at Google, Inc.

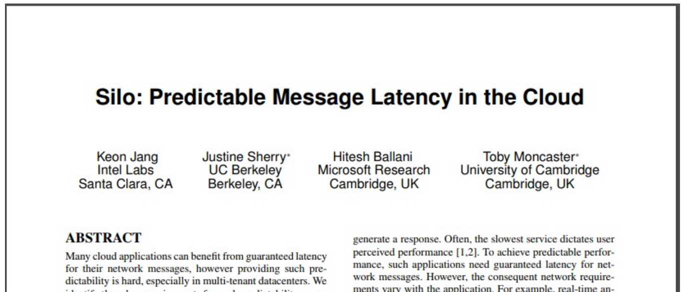






Silo [SIGCOMM15]

QJump [NSDI15]



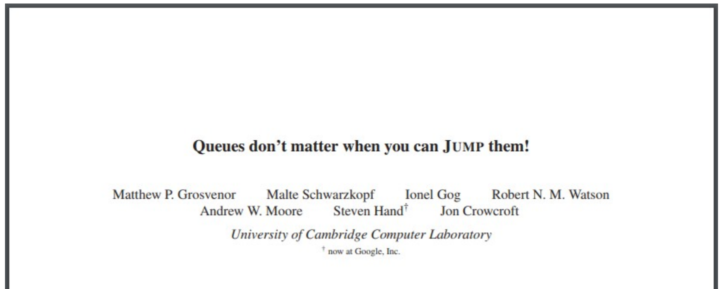
Silo: Predictable Message Latency in the Cloud

Keon Jang, Intel Labs, Santa Clara, CA
Justine Sherry*, UC Berkeley, Berkeley, CA
Hitesh Ballani, Microsoft Research, Cambridge, UK
Toby Moncaster*, University of Cambridge, Cambridge, UK

ABSTRACT

Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network require-

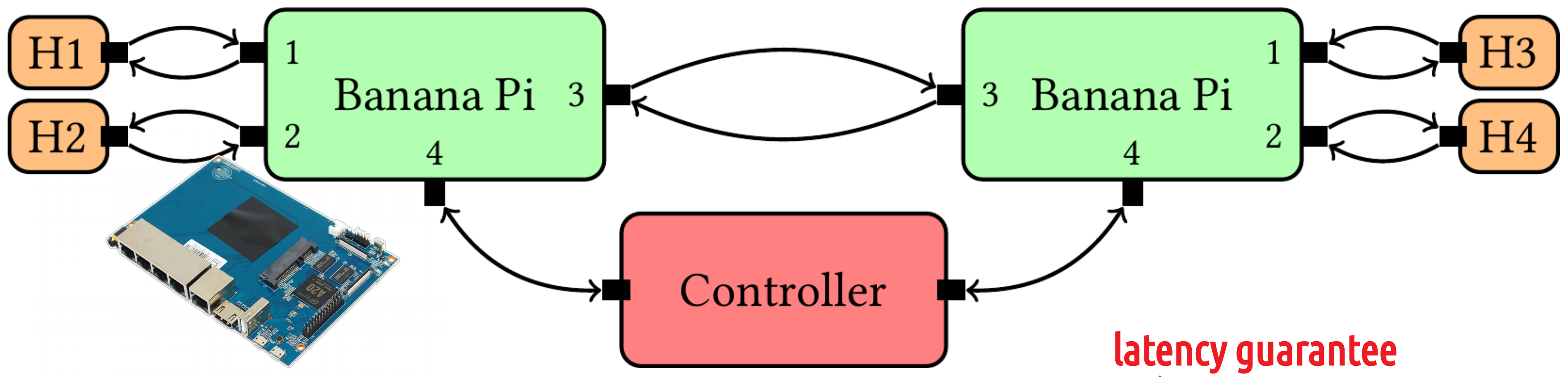


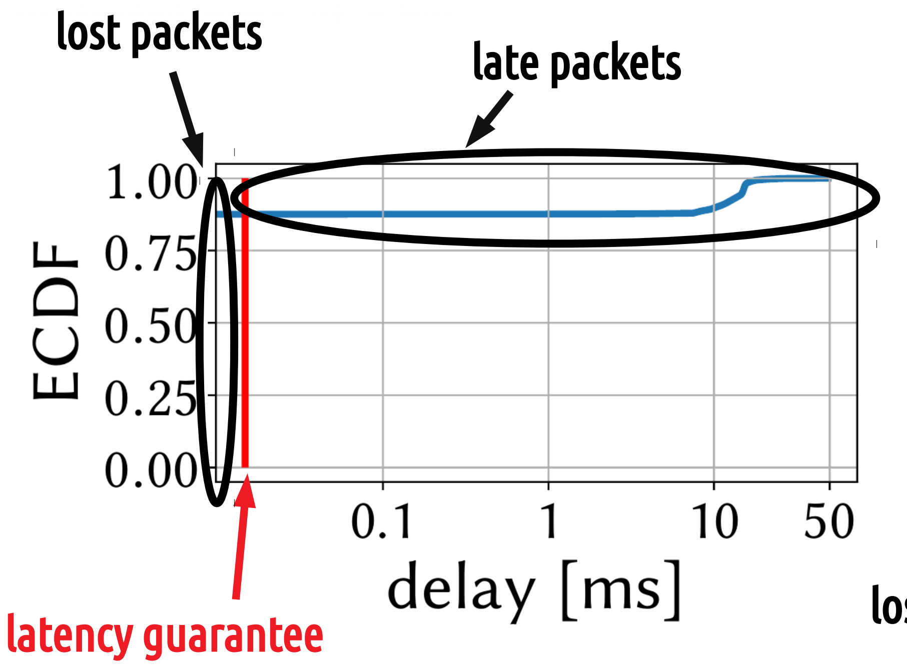
Queues don't matter when you can JUMP them!

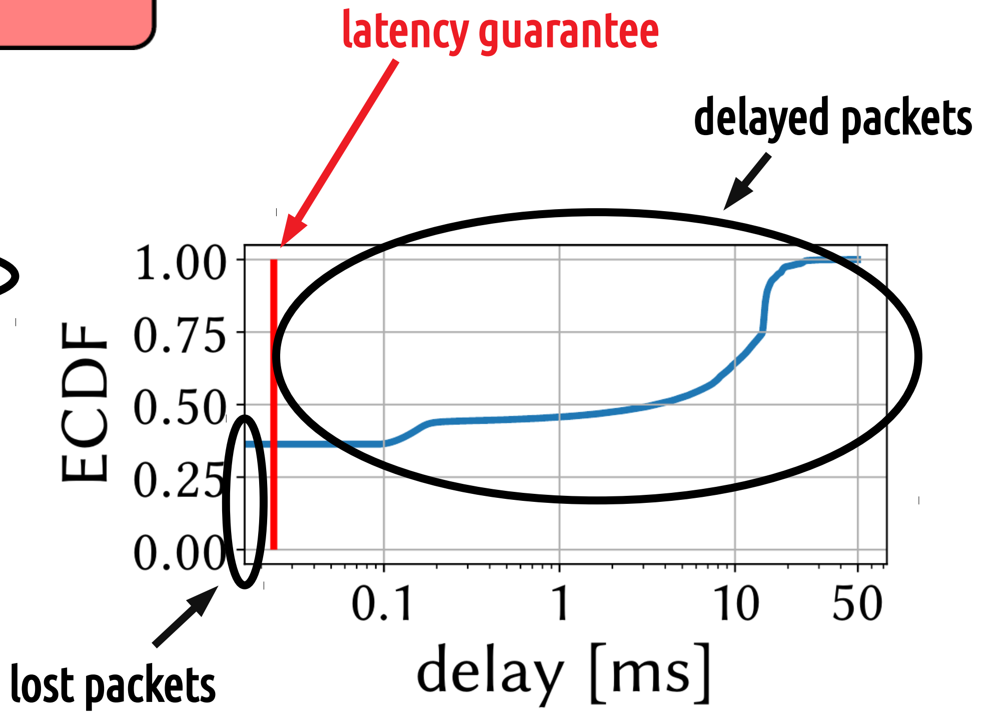
Matthew P. Grosvenor   Malte Schwarzkopf   Ionel Gog   Robert N. M. Watson
Andrew W. Moore   Steven Hand[†]   Jon Crowcroft

University of Cambridge Computer Laboratory

[†] now at Google, Inc.

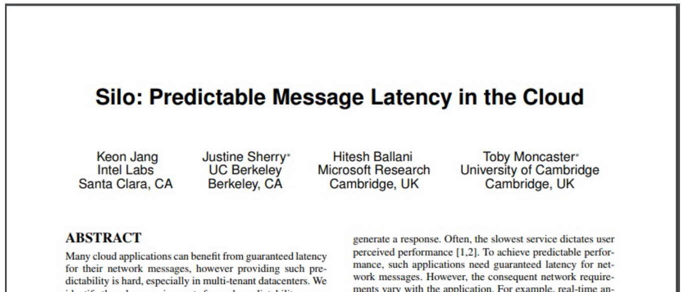






State-of-the-art guarantees are violated!

Silo [SIGCOMM15]



Silo: Predictable Message Latency in the Cloud

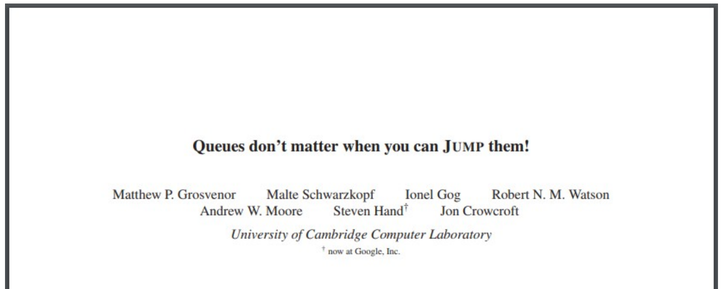
Keon Jang, Intel Labs, Santa Clara, CA
Justine Sherry*, UC Berkeley, Berkeley, CA
Hitesh Ballani, Microsoft Research, Cambridge, UK
Toby Moncaster*, University of Cambridge, Cambridge, UK

ABSTRACT

Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network requirements vary with the application. For example, real-time an-

QJump [NSDI15]

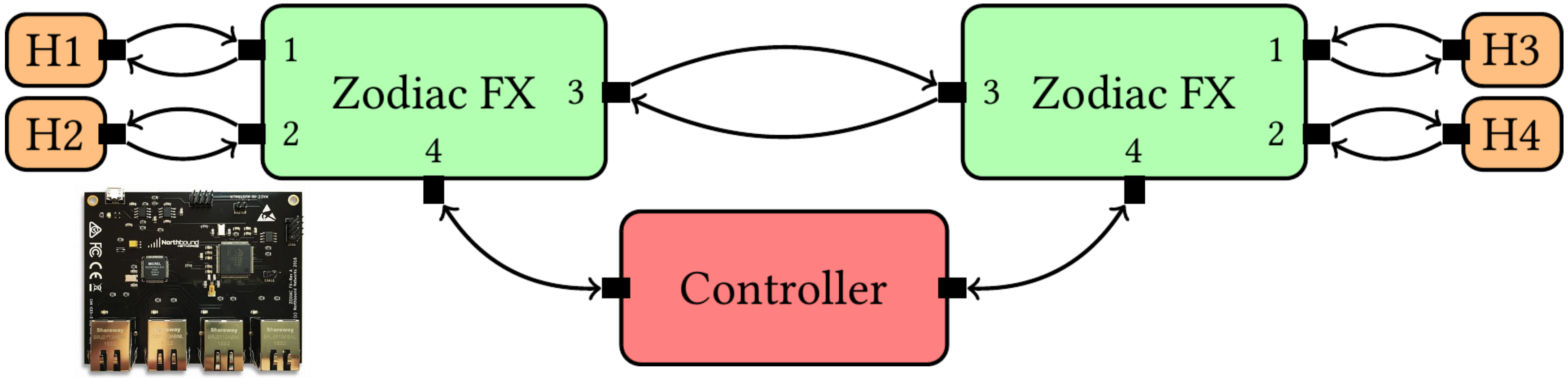


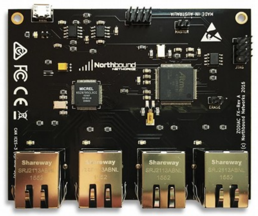
Queues don't matter when you can JUMP them!

Matthew P. Grosvenor   Malte Schwarzkopf   Ionel Gog   Robert N. M. Watson
Andrew W. Moore   Steven Hand†   Jon Crowcroft

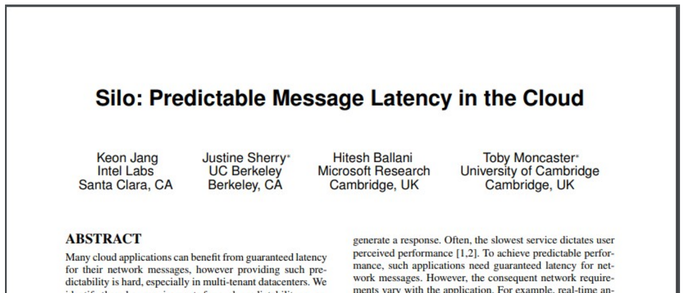
University of Cambridge Computer Laboratory

† now at Google, Inc.





Silo [SIGCOMM15]



Silo: Predictable Message Latency in the Cloud

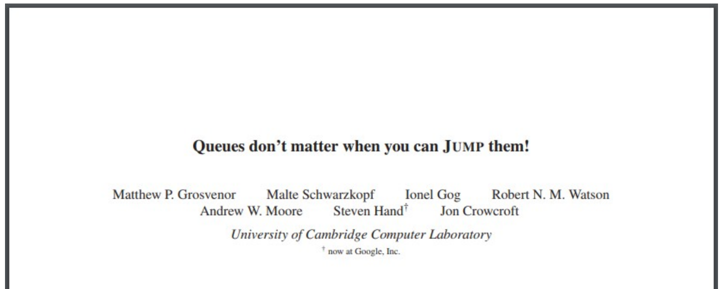
Keon Jang, Intel Labs, Santa Clara, CA
Justine Sherry, UC Berkeley, Berkeley, CA
Hitesh Ballani, Microsoft Research, Cambridge, UK
Toby Moncaster, University of Cambridge, Cambridge, UK

ABSTRACT

Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network requirements vary with the application. For example, real-time an

QJump [NSDI15]

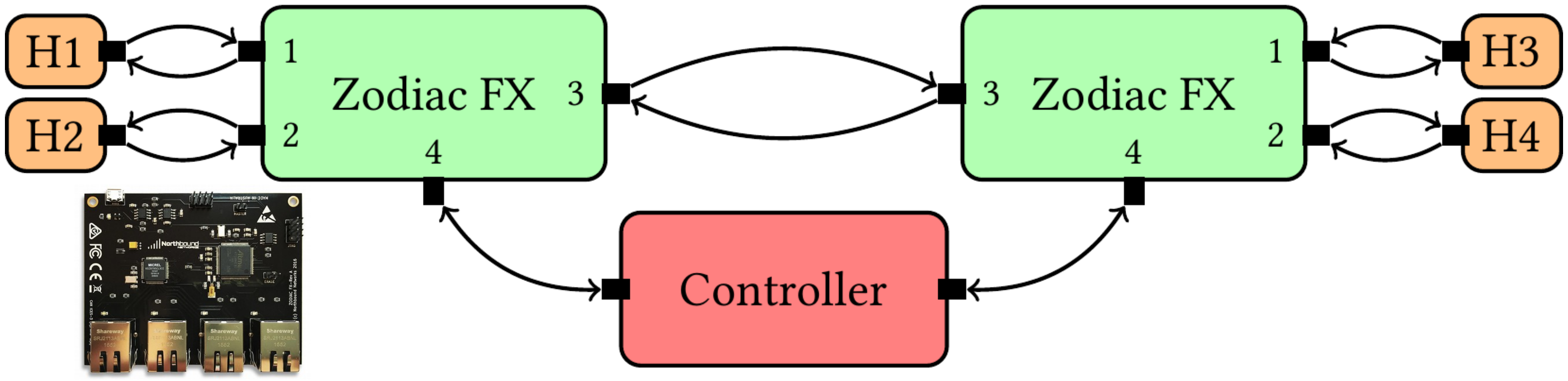


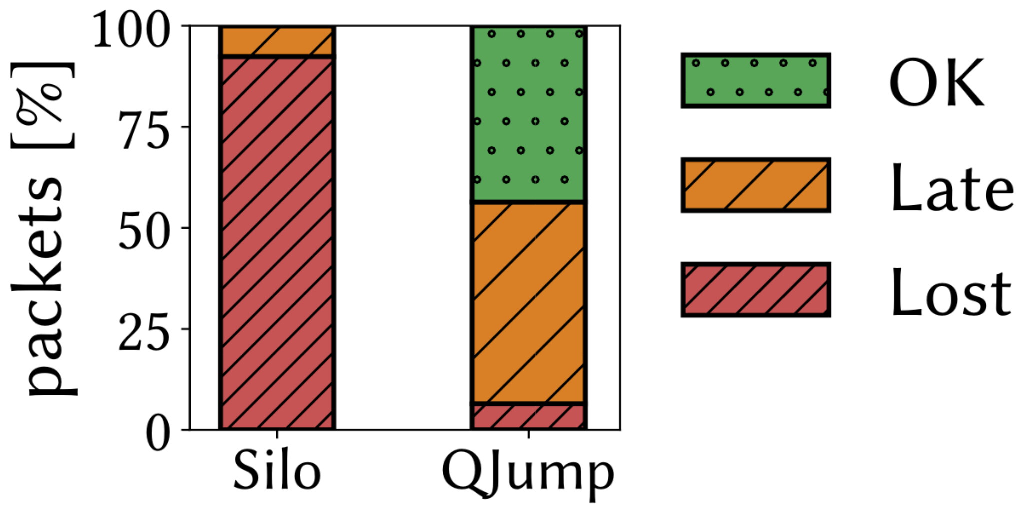
Queues don't matter when you can JUMP them!

Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand[†], Jon Crowcroft

University of Cambridge Computer Laboratory

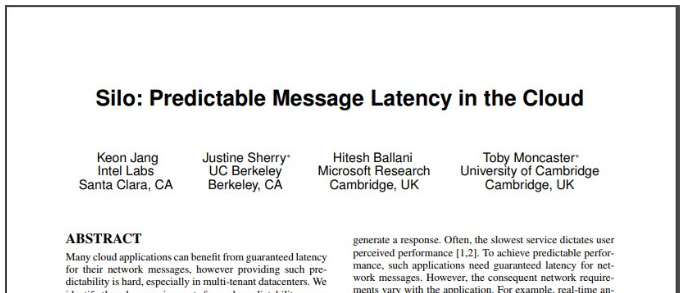
[†] now at Google, Inc.





State-of-the-art guarantees are violated!

Silo [SIGCOMM15]

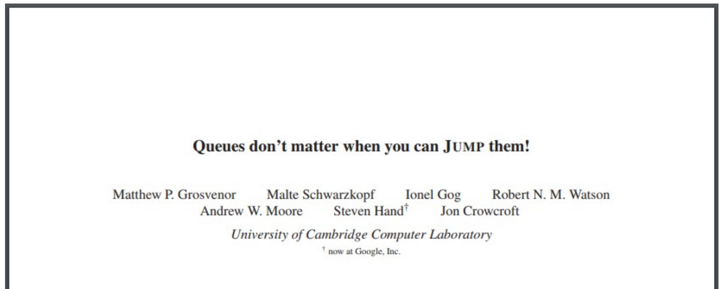

Silo: Predictable Message Latency in the Cloud

Keon Jang, Intel Labs, Santa Clara, CA
Justine Sherry*, UC Berkeley, Berkeley, CA
Hitesh Ballani, Microsoft Research, Cambridge, UK
Toby Moncaster*, University of Cambridge, Cambridge, UK

ABSTRACT
Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is hard, especially in multi-tenant datacenters. We

generate a response. Often, the slowest service dictates user perceived performance [1,2]. To achieve predictable performance, such applications need guaranteed latency for network messages. However, the consequent network requirements vary with the application. For example, real-time an-

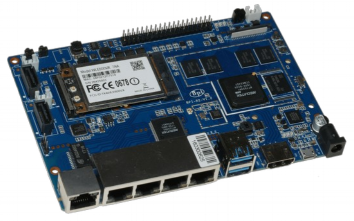
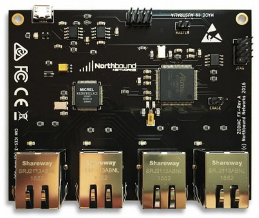
QJump [NSDI15]


Queues don't matter when you can JUMP them!

Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand†, Jon Crowcroft
University of Cambridge Computer Laboratory
† now at Google, Inc.

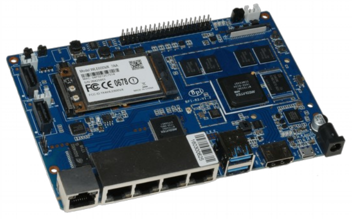
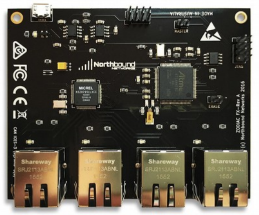
State-of-the-art guarantees are violated! Why?

State-of-the-art guarantees are violated! Why?

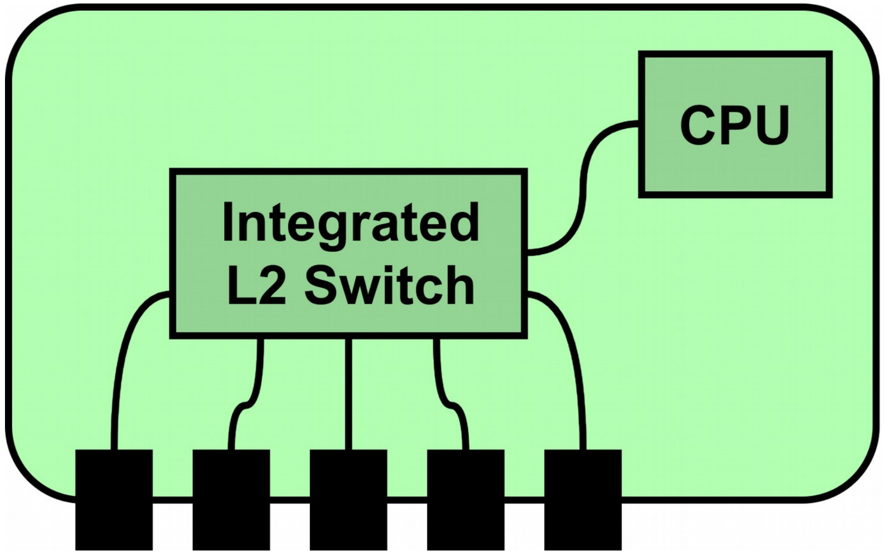



These low-cost switches share the same hardware architecture

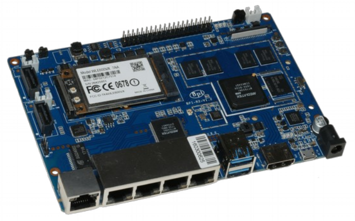
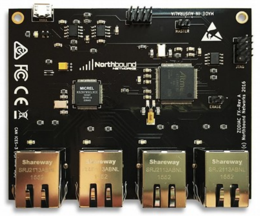
State-of-the-art guarantees are violated! Why?

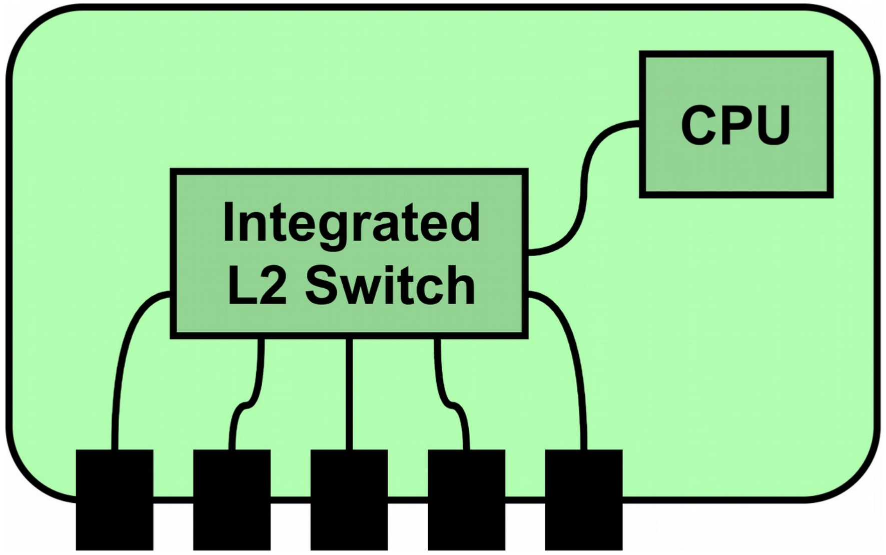
These low-cost switches share the same hardware architecture



State-of-the-art guarantees are violated! Why?

These low-cost switches share the same hardware architecture

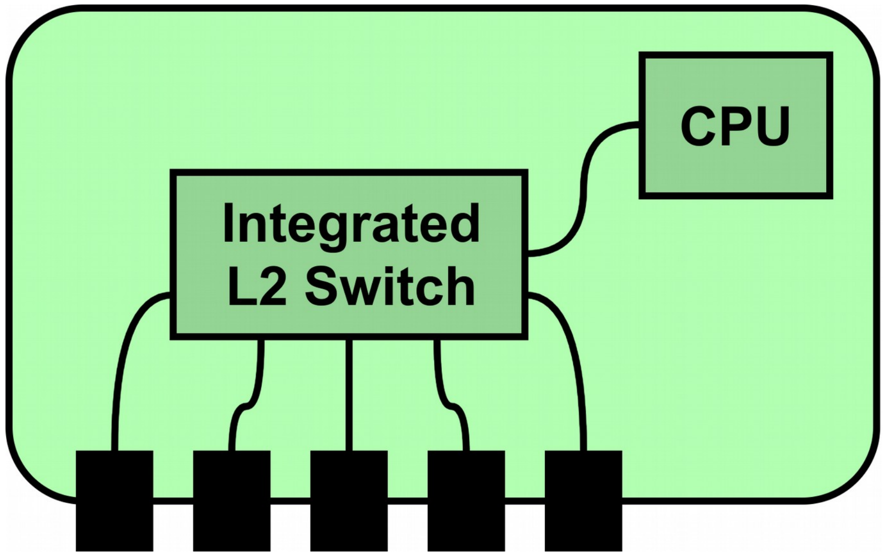


That's the only way to build a **cheap programmable chip**!

State-of-the-art guarantees are violated! Why?

Most SoA assumes

1. Switches can process packets at line rate
2. Ports do not interfere

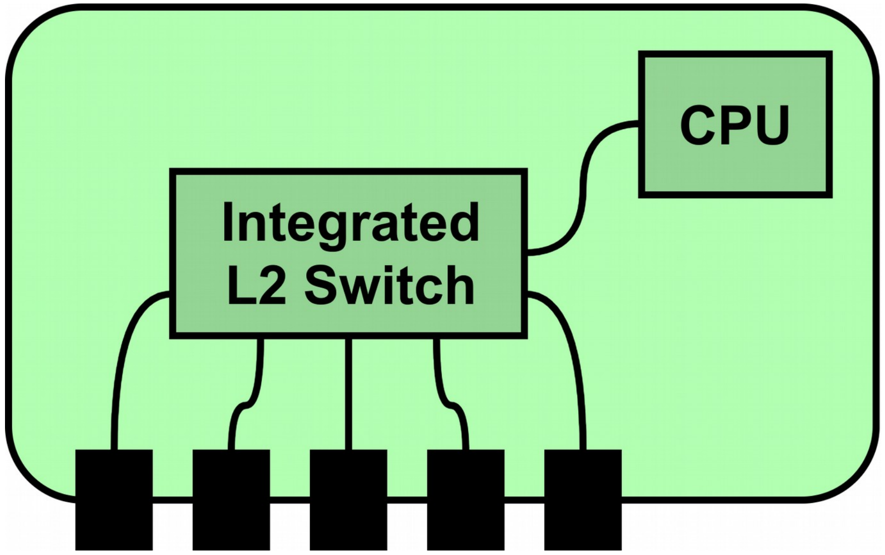


State-of-the-art guarantees are violated! Why?

Most SoA assumes

1. Switches can process packets at line rate
2. Ports do not interfere

Valid for traditional switches (e.g., data centers)



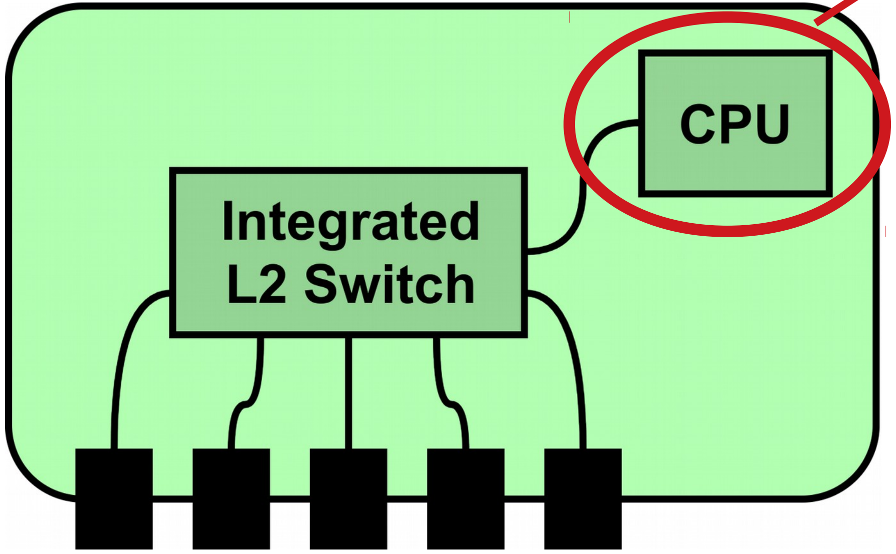
State-of-the-art guarantees are violated! Why?

Most SoA assumes

1. Switches can process packets at line rate
2. Ports do not interfere

Valid for traditional switches (e.g., data centers) **but not valid for such low-capacity switches**

1. CPU processing hardly at line rate
2. CPU shared by ports

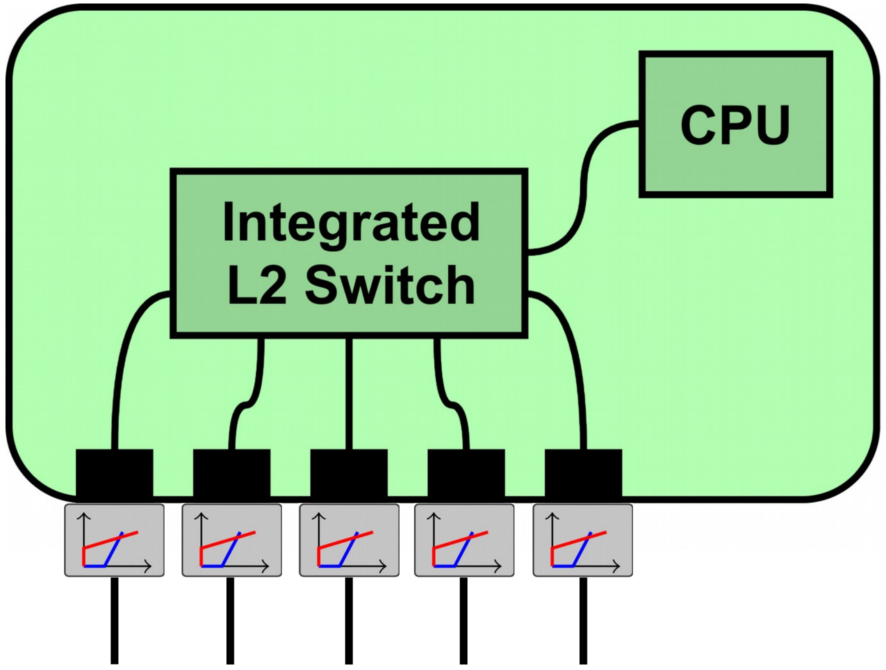


State-of-the-art guarantees are violated! Why?

Most SoA assumes

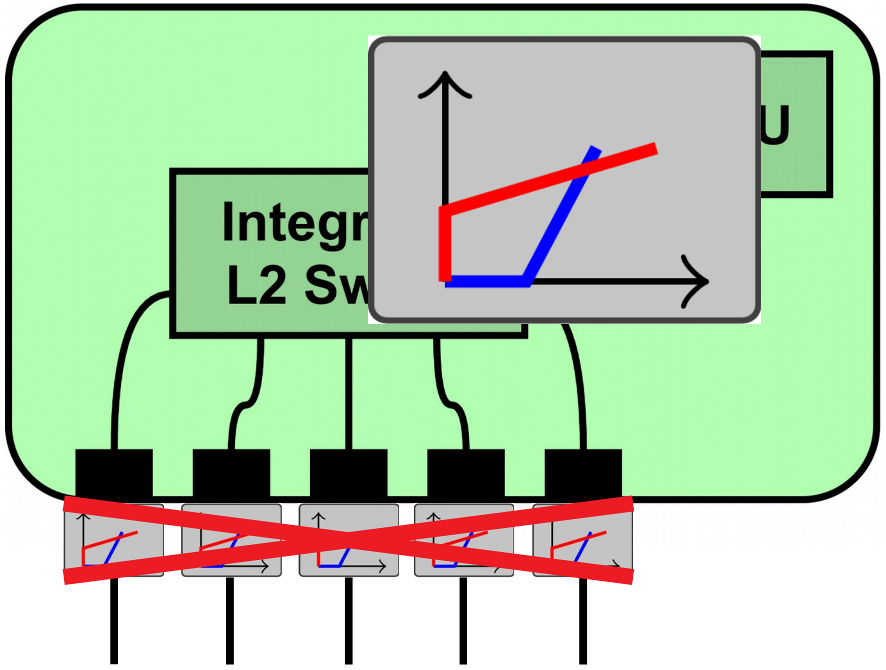
1. Switches can process packets at line rate
2. Ports do not interfere

For example Silo:



Defines one independent (network calculus) service per port

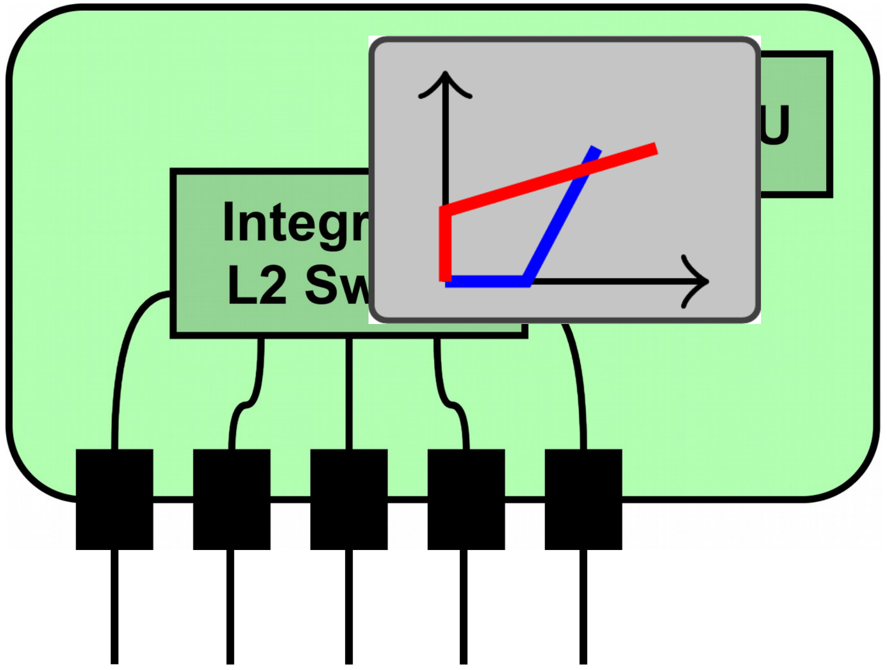
Instead, **such switches have to be modeled as a shared service**
which consists of the **Integrated Switch + CPU**



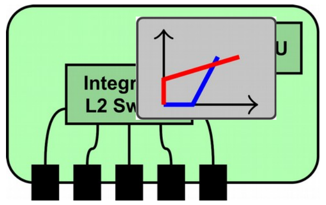
Instead, **such switches have to be modeled as a shared service**

which consists of the **Integrated Switch + CPU**

This forms the basis of Loko!

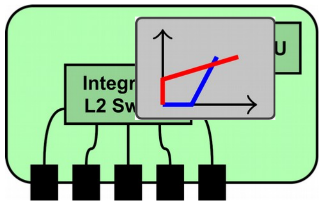


Loko

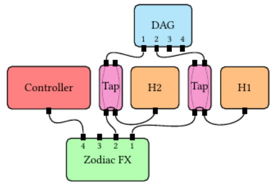


Step 0: Identification of independent services

Loko

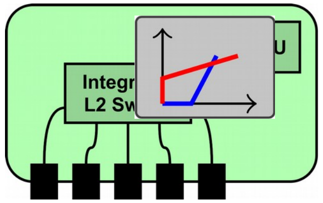


Step 0: Identification of independent services

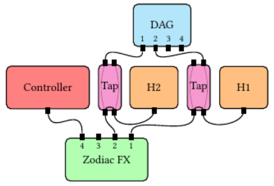


Step 1: Benchmarking of the service(s)

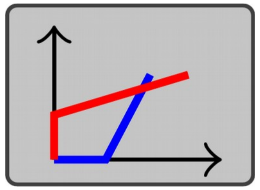
Loko



Step 0: Identification of independent services

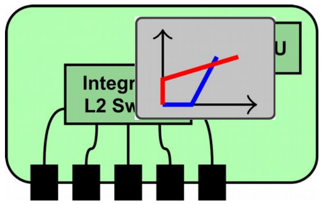


Step 1: Benchmarking of the service(s)

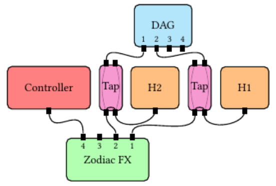


Step 2: Measurements → **deterministic** model for the service(s)

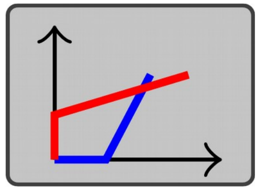
Loko



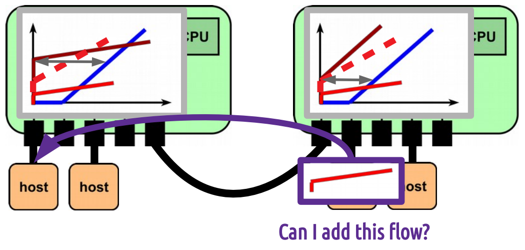
Step 0: Identification of independent services



Step 1: Benchmarking of the service(s)

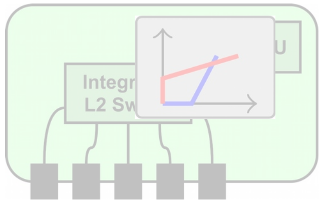


Step 2: Measurements → **deterministic** model for the service(s)

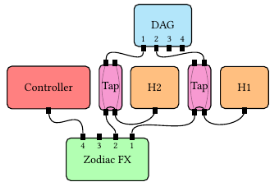


Step 3: Switch model → network model (admission control)

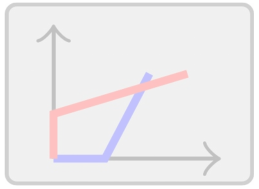
Loko



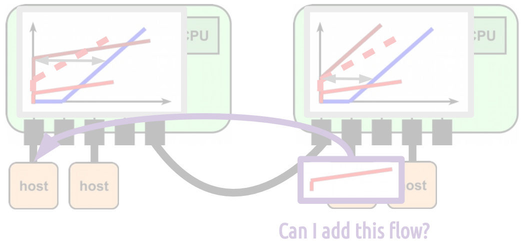
Step 0: Identification of independent services



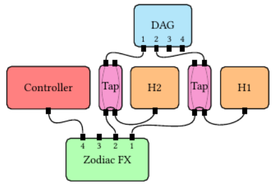
Step 1: Benchmarking of the service(s)



Step 2: Measurements → **deterministic** model for the service(s)

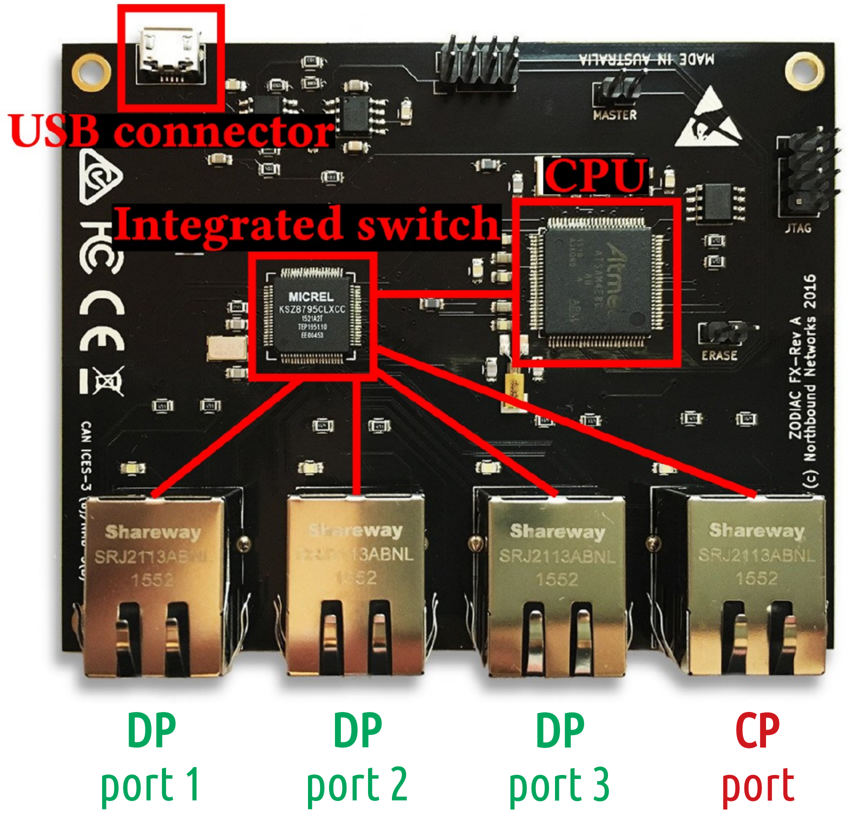


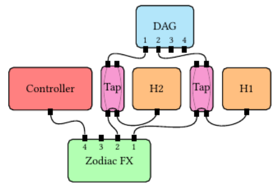
Step 3: Switch model → network model (admission control)



Step 1: Benchmarking of the service

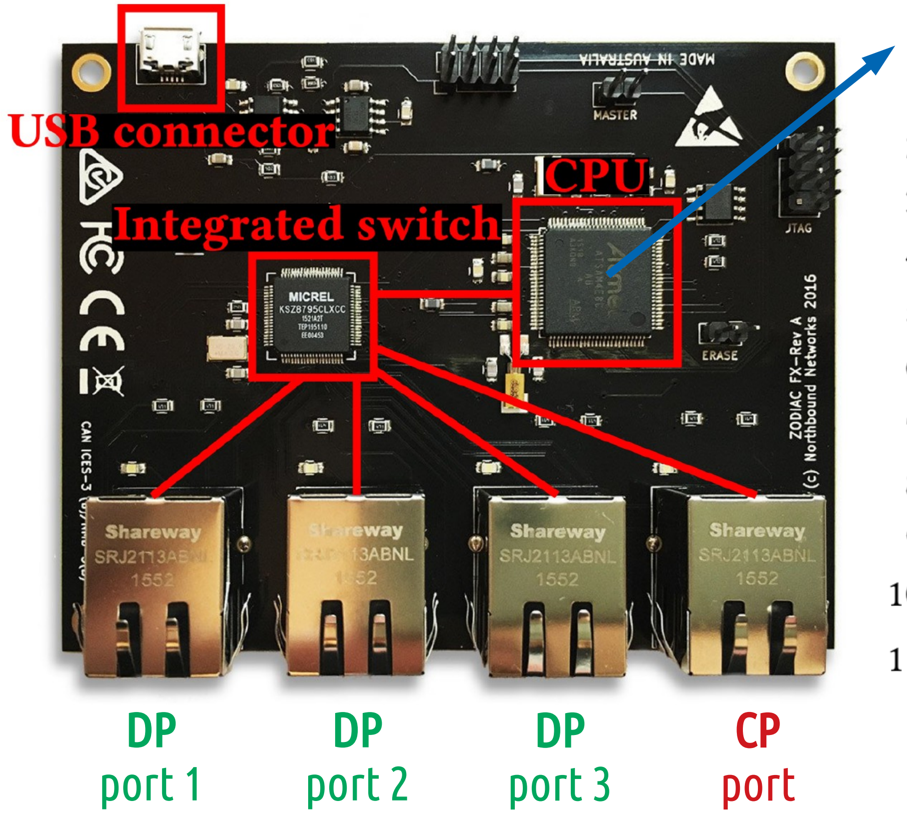
Let‘s see for the **Zodiac FX**





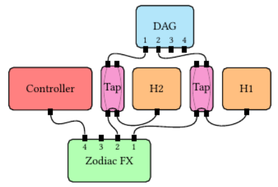
Step 1: Benchmarking of the service

Let's see for the **Zodiac FX**



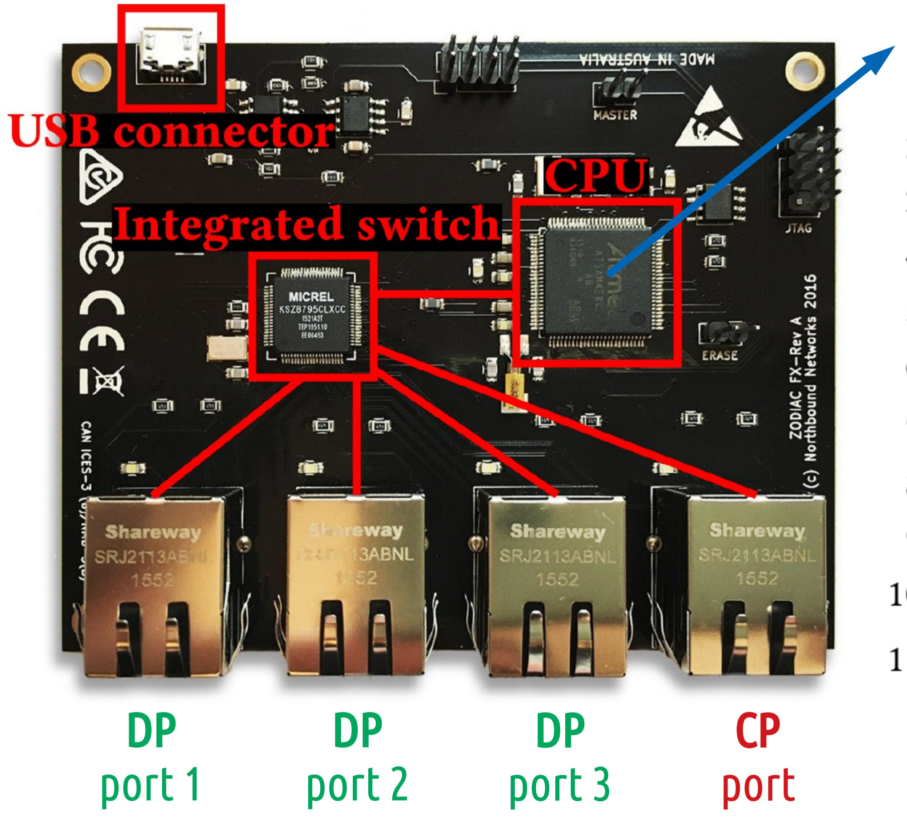
runs an embedded **OS-free** infinite loop:

```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```



Step 1: Benchmarking of the service

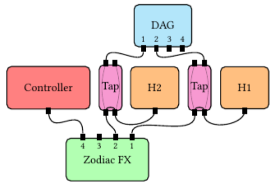
Let‘s see for the **Zodiac FX**



runs an embedded **OS-free** infinite loop:

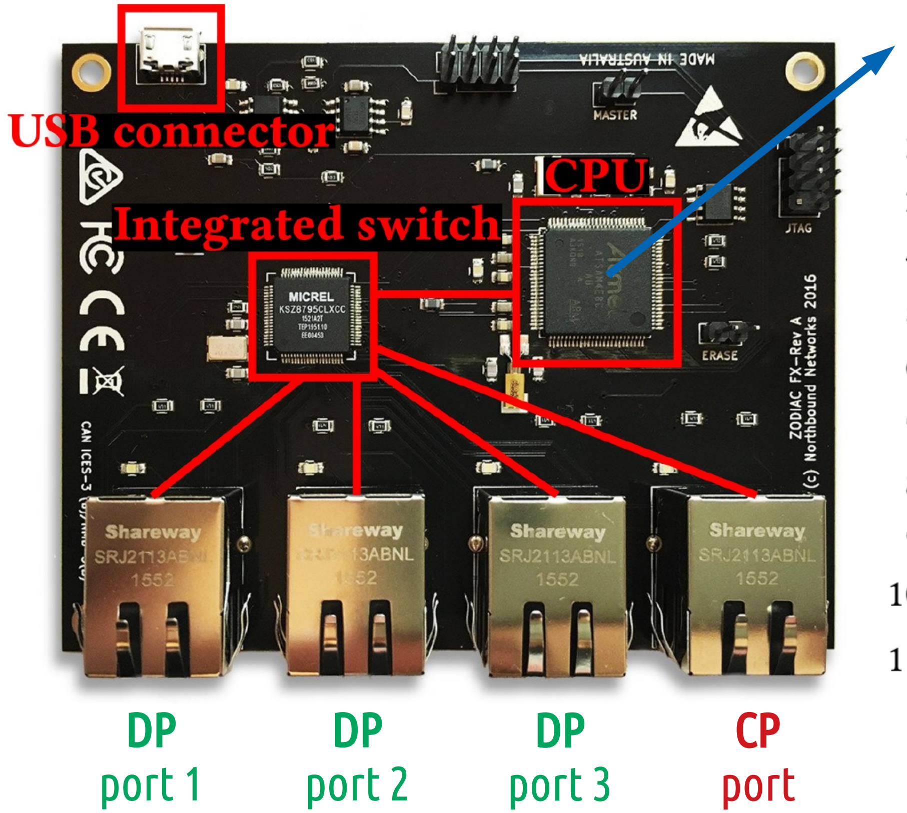
```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service

Let‘s see for the **Zodiac FX**

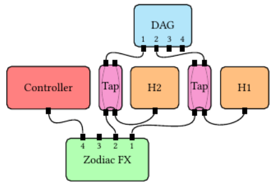


runs an embedded **OS-free** infinite loop:

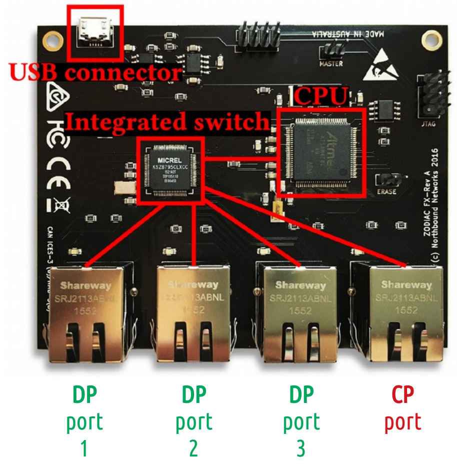
```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

and because open-source, we can!

For predictability, we have to **identify ANY source of delay**



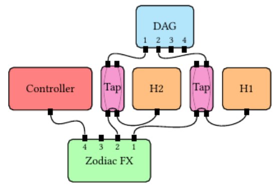
Step 1: Benchmarking of the service



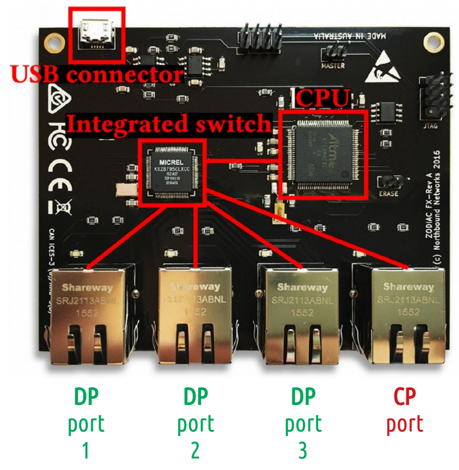
This is what we do in §2.1, §2.2, §3.1 of the paper, **we get**

Dimension	Values
nb. of entries	1, 17, 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all*
action	*output, set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, last*
priorities	*increasing, decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

This is the **exhaustive list** of dimensions that influence the switch processing!



Step 1: Benchmarking of the service



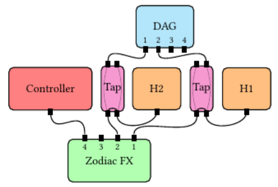
This is what we do in §2.1, §2.2, §3.1 of the paper, **we get**

Dimension	**Values**
nb. of entries	1, 17, 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all*
action	*output, set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, last*
priorities	*increasing, decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

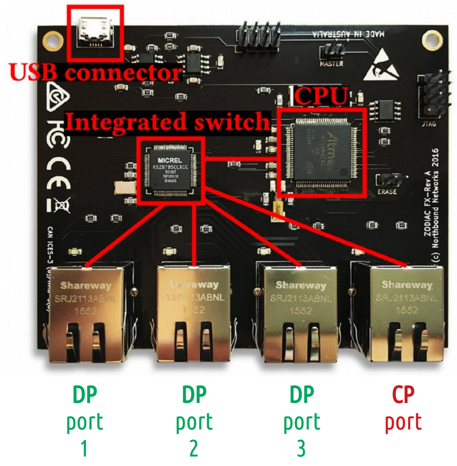
This is the **exhaustive list** of dimensions that influence the switch processing!

Measure (CP and DP) <u>throughput,</u> per-packet <u>delay</u> and <u>buffer capacity</u>
for each combination of the dimensions

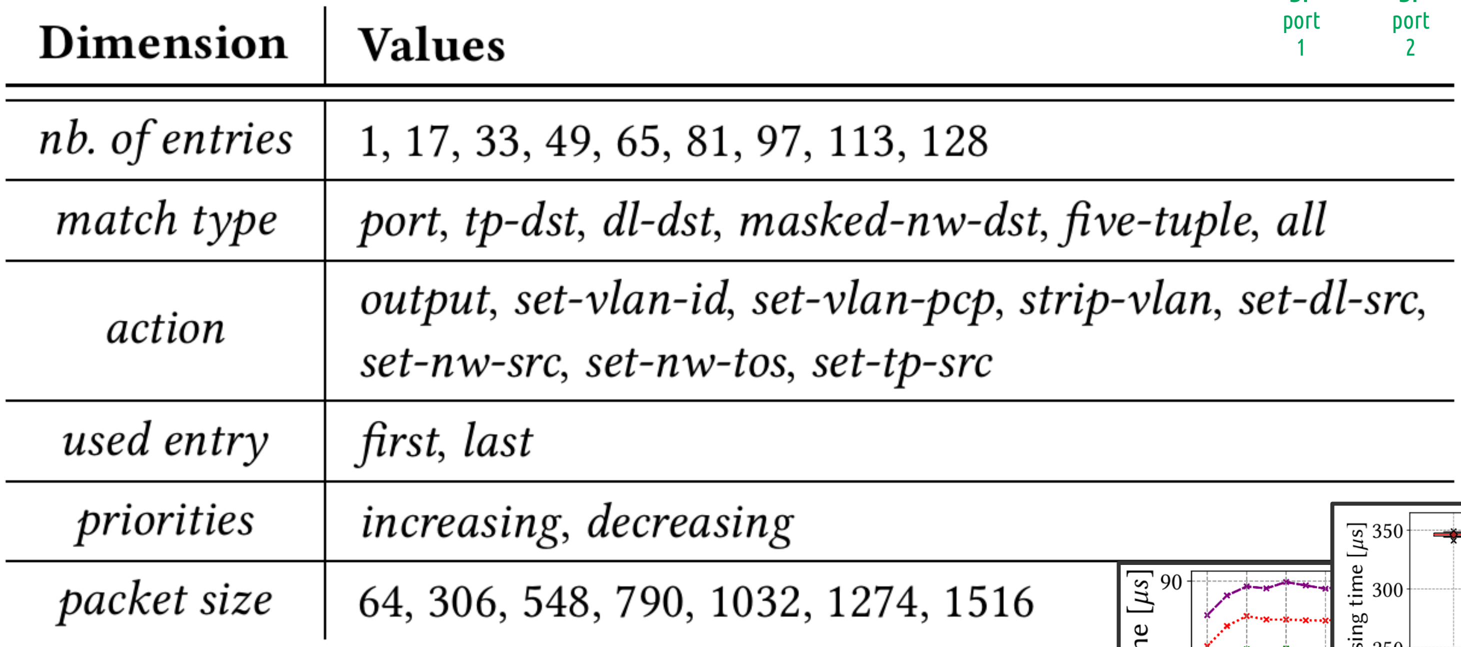
Done in §3 of the paper



Step 1: Benchmarking of the service



This is what we do in §2.1, §2.2, §3.1 of the paper, **we get**

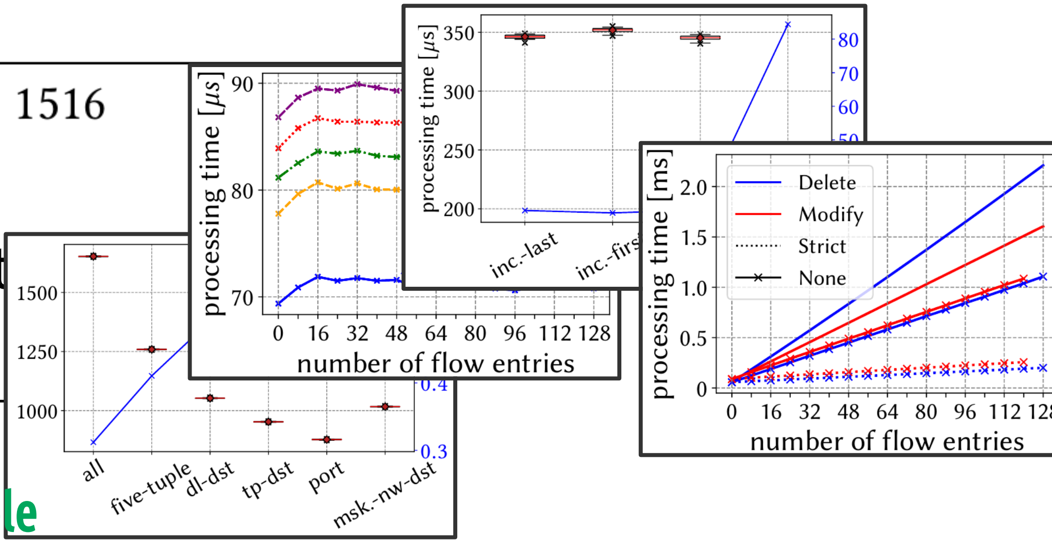


Dimension	Values
nb. of entries	1, 17, 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all*
action	*output, set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, last*
priorities	*increasing, decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

This is the **exhaustive list** of dimensions that

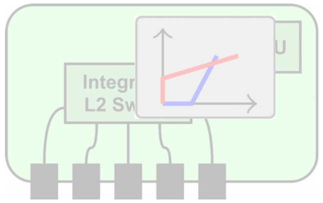
Measure (CP and DP) <u>throughput</u>, per-packet <u>delay</u> and b
for each combination of the dimensions

Done in §3 of the paper

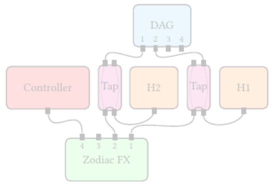


The performance is indeed predictable

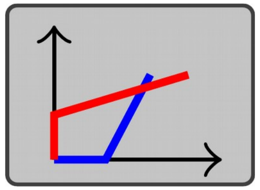
Loko



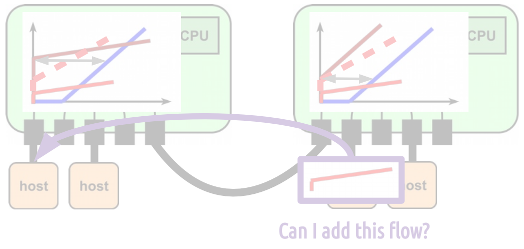
Step 0: Identification of independent services



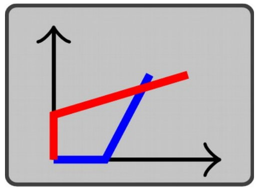
Step 1: Benchmarking of the service(s)



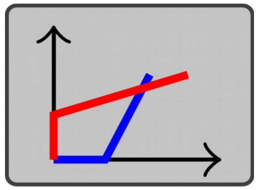
Step 2: Measurements → **deterministic** model for the service(s)



Step 3: Switch model → network model (admission control)

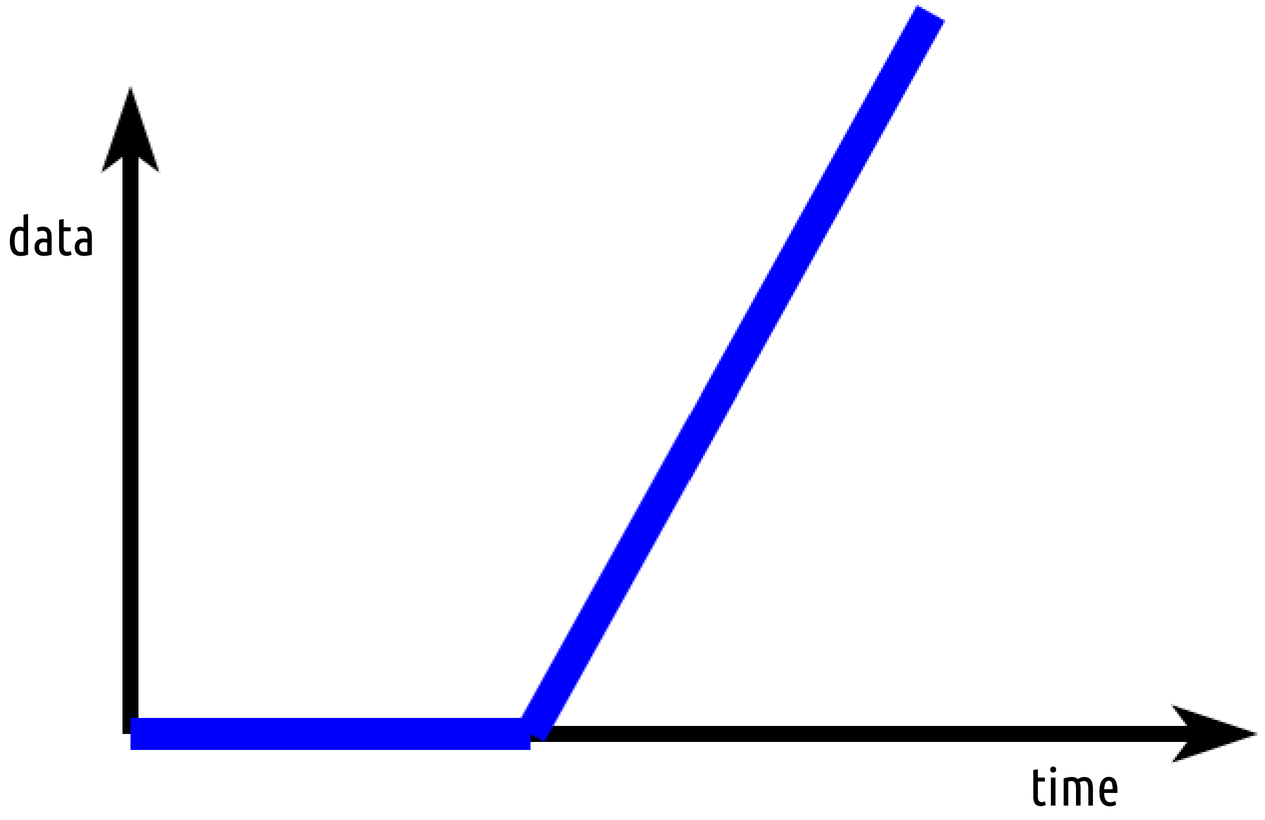


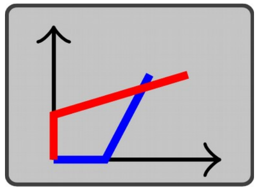
Step 2: Measurements → **deterministic** model for the service



Step 2: Measurements → **deterministic** model for the service

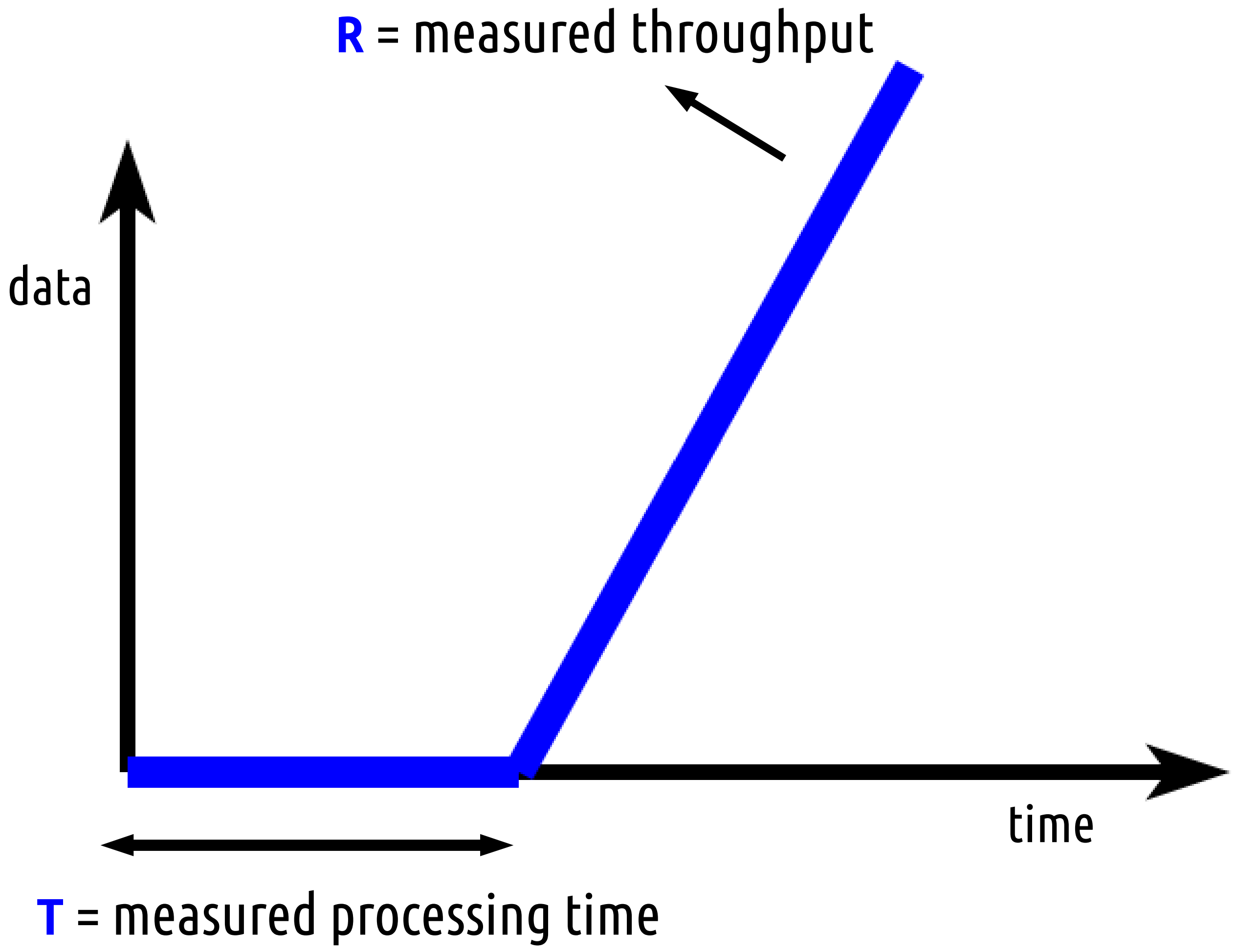
network calculus model





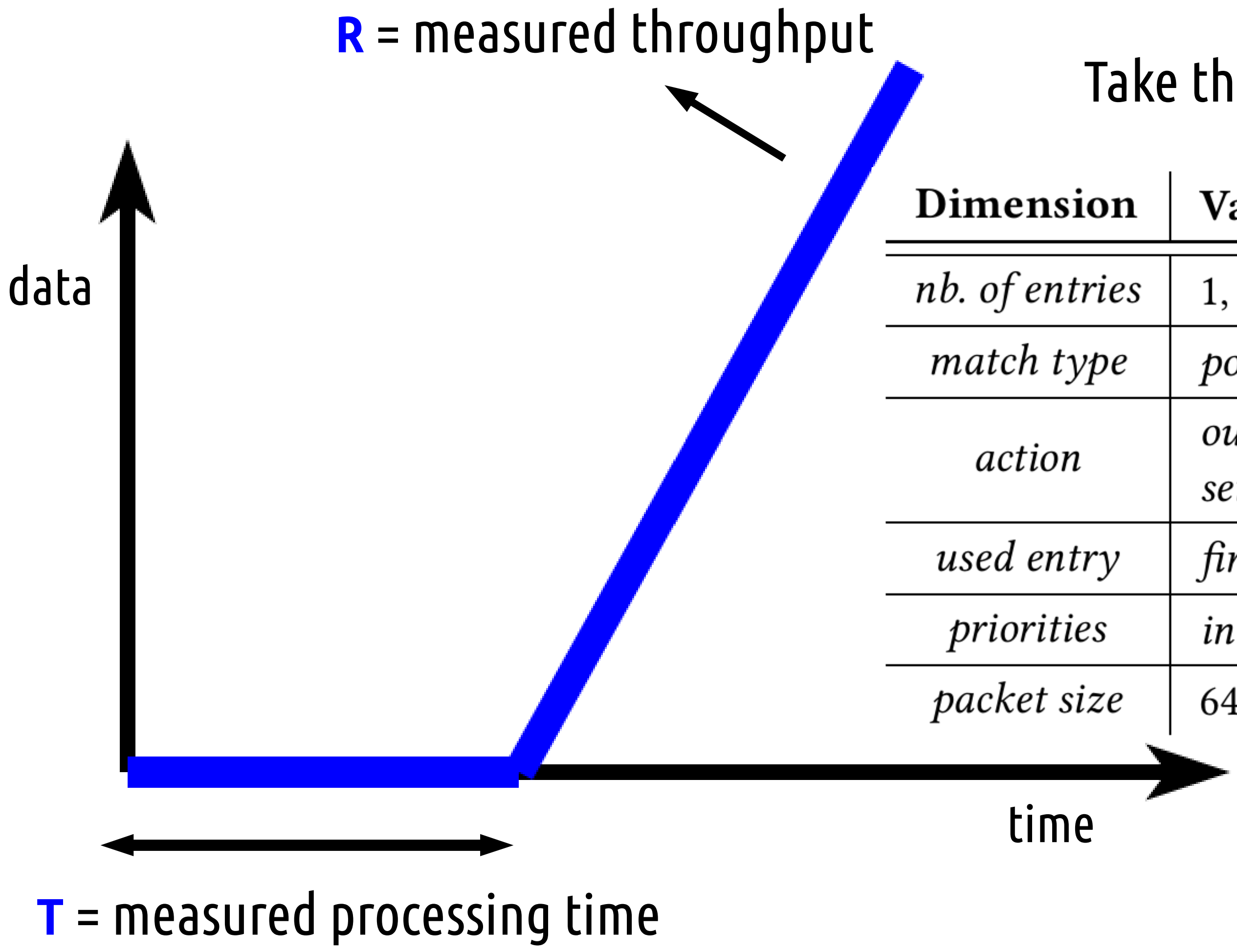
Step 2: Measurements → **deterministic** model for the service

network calculus model



Step 2: Measurements → **deterministic** model for the service

network calculus model

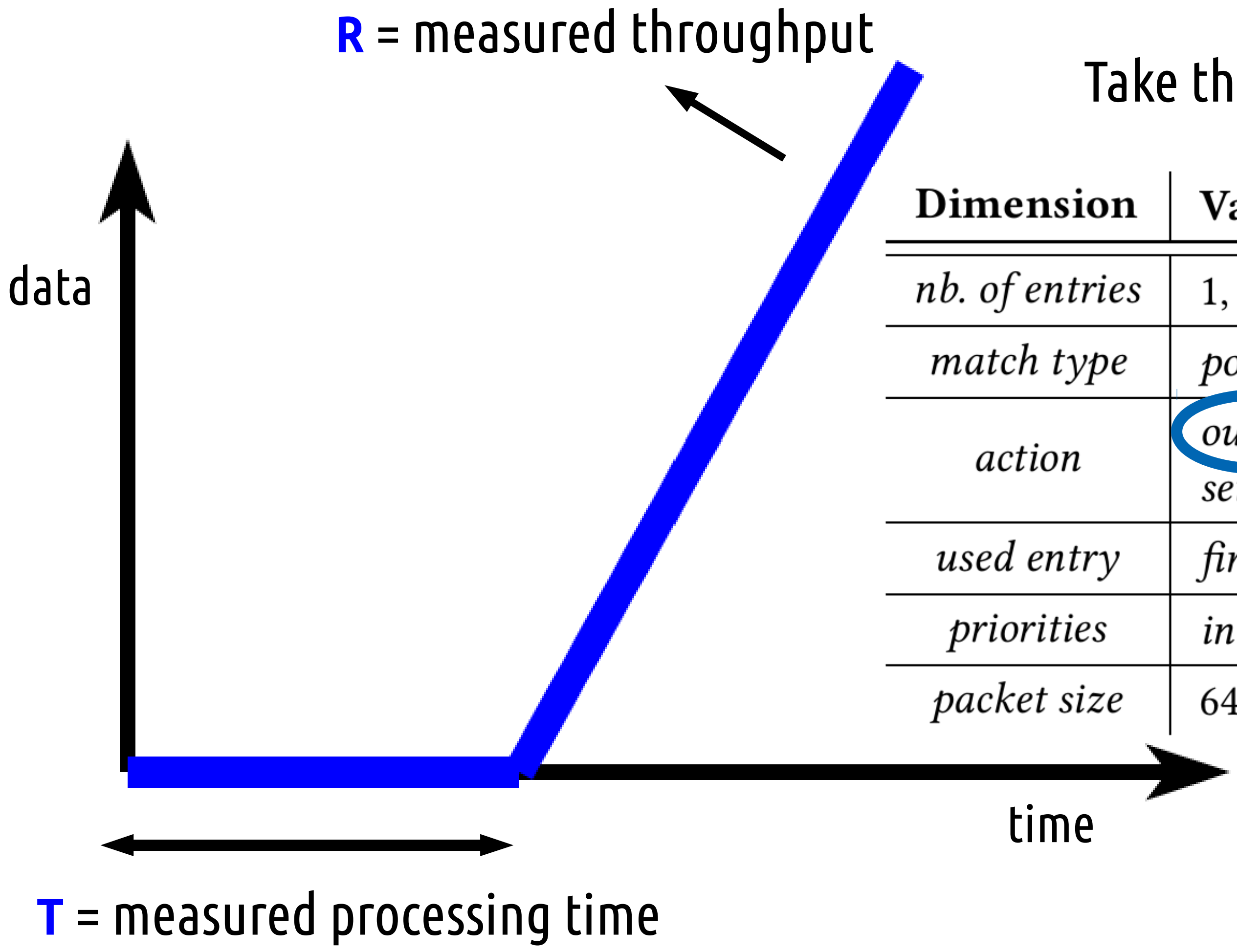


Take the worst-case for a given scenario

Dimension	Values
nb. of entries	1, 17, 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all*
action	*output, set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, last*
priorities	*increasing, decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

Step 2: Measurements → **deterministic** model for the service

network calculus model

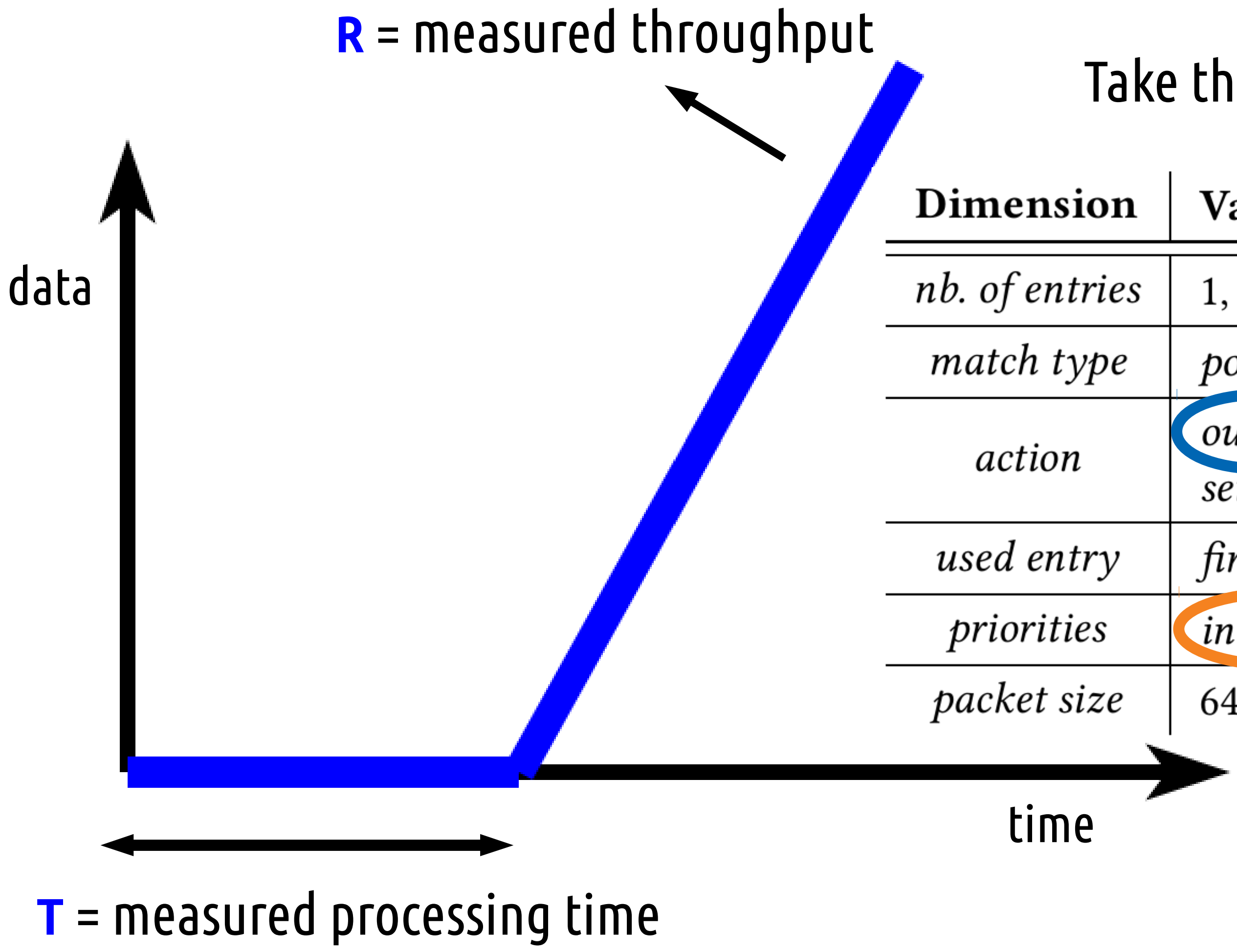


Take the worst-case for a given scenario

Dimension	Values
nb. of entries	1, 17, 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all*
action	*output, set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, last*
priorities	*increasing, decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

Step 2: Measurements → **deterministic** model for the service

network calculus model

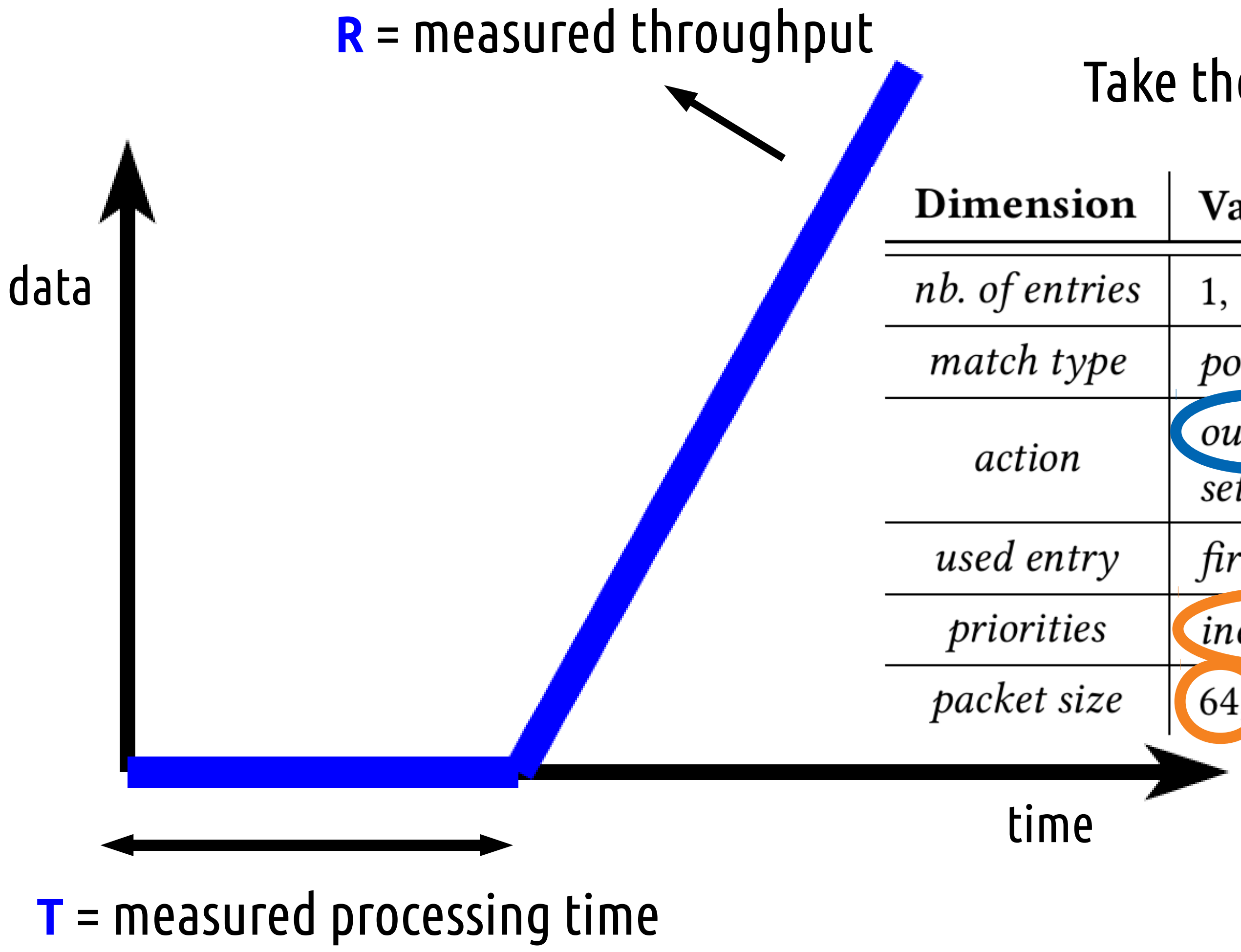


Take the worst-case for a given scenario

Dimension	Values
nb. of entries	1, 17, 33, [✓ 49], 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst,* [✓ *five-tuple*], *all*
action	[✓ *output*], *set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first,* [✓ *last*]
priorities	[✓ *increasing*], *decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

Step 2: Measurements → **deterministic** model for the service

network calculus model



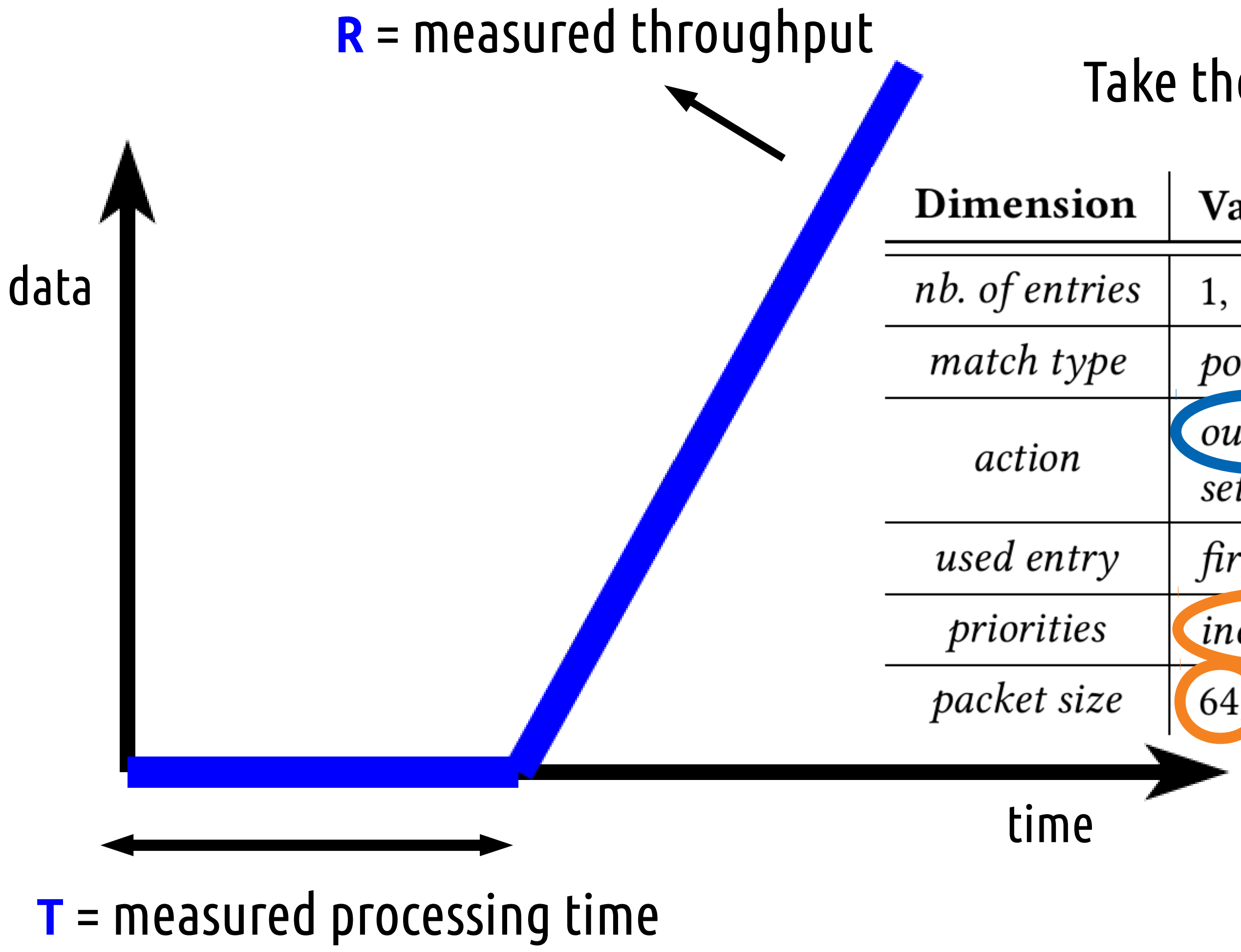
Take the worst-case for a given scenario

Dimension	Values
nb. of entries	1, 17, 33, [✓ 49], 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, [✓ five-tuple], all*
action	*[✓ output], set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, [✓ last]*
priorities	*[✓ increasing], decreasing*
packet size	[✓ 64], 306, 548, 790, 1032, 1274, 1516

for throughput

Step 2: Measurements → **deterministic** model for the service

network calculus model



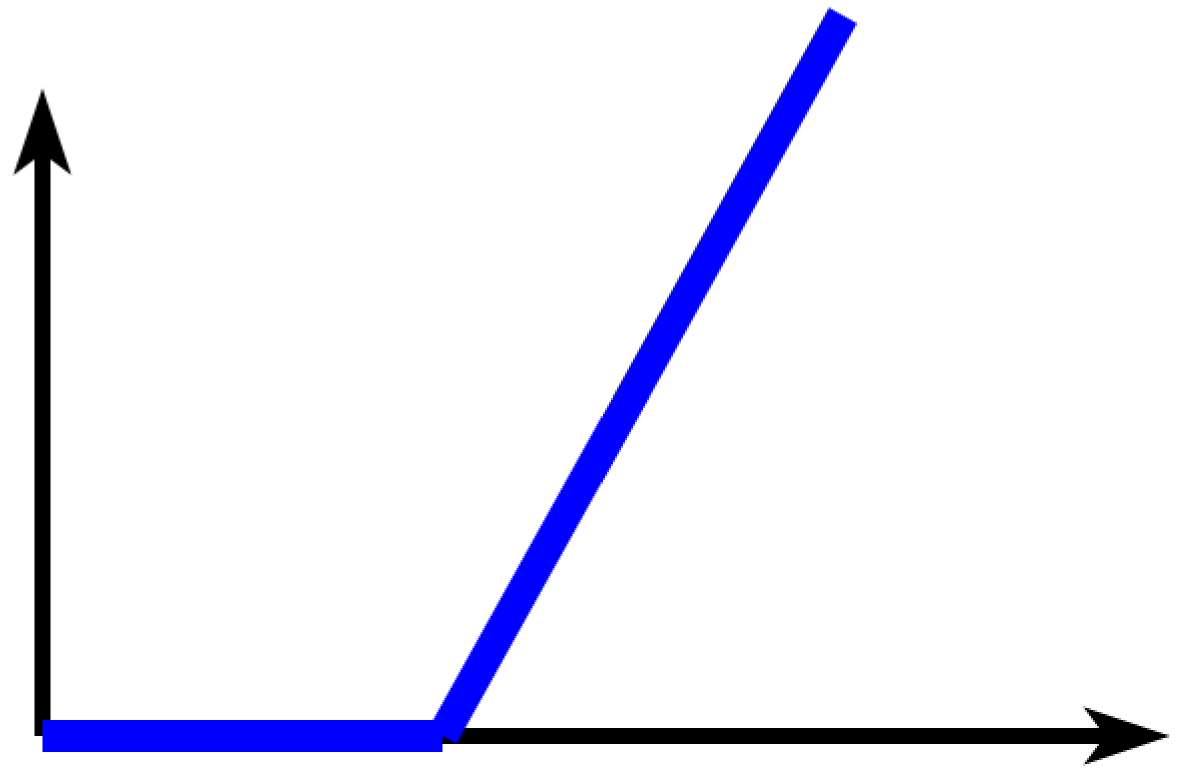
Take the worst-case for a given scenario

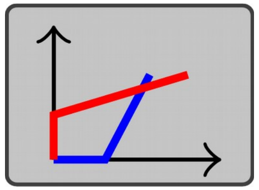
Dimension	Values
nb. of entries	1, 17, 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all*
action	*output, set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, last*
priorities	*increasing, decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

64: for throughput

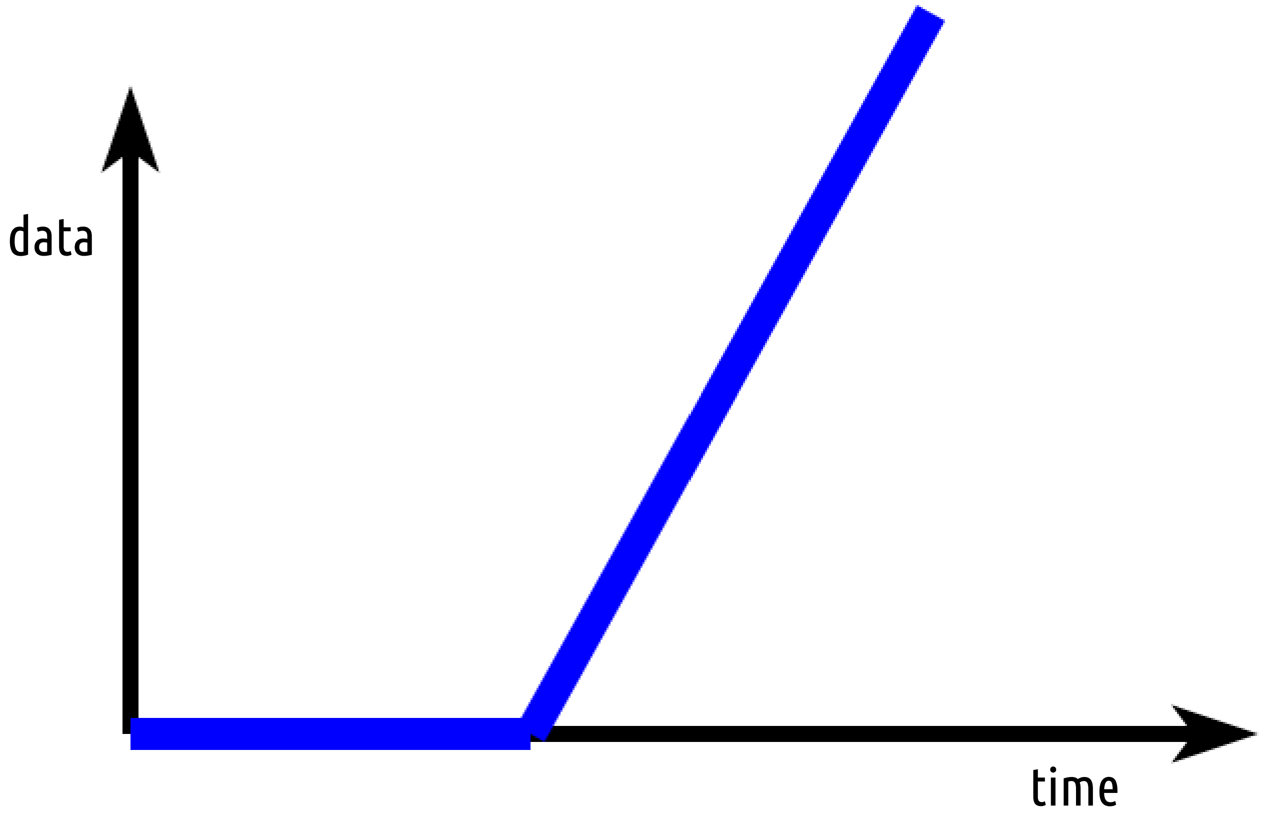
1516: for processing time

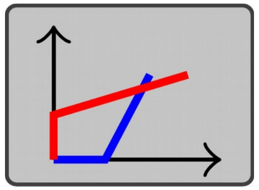
Step 2: Measurements → **deterministic** model for the service



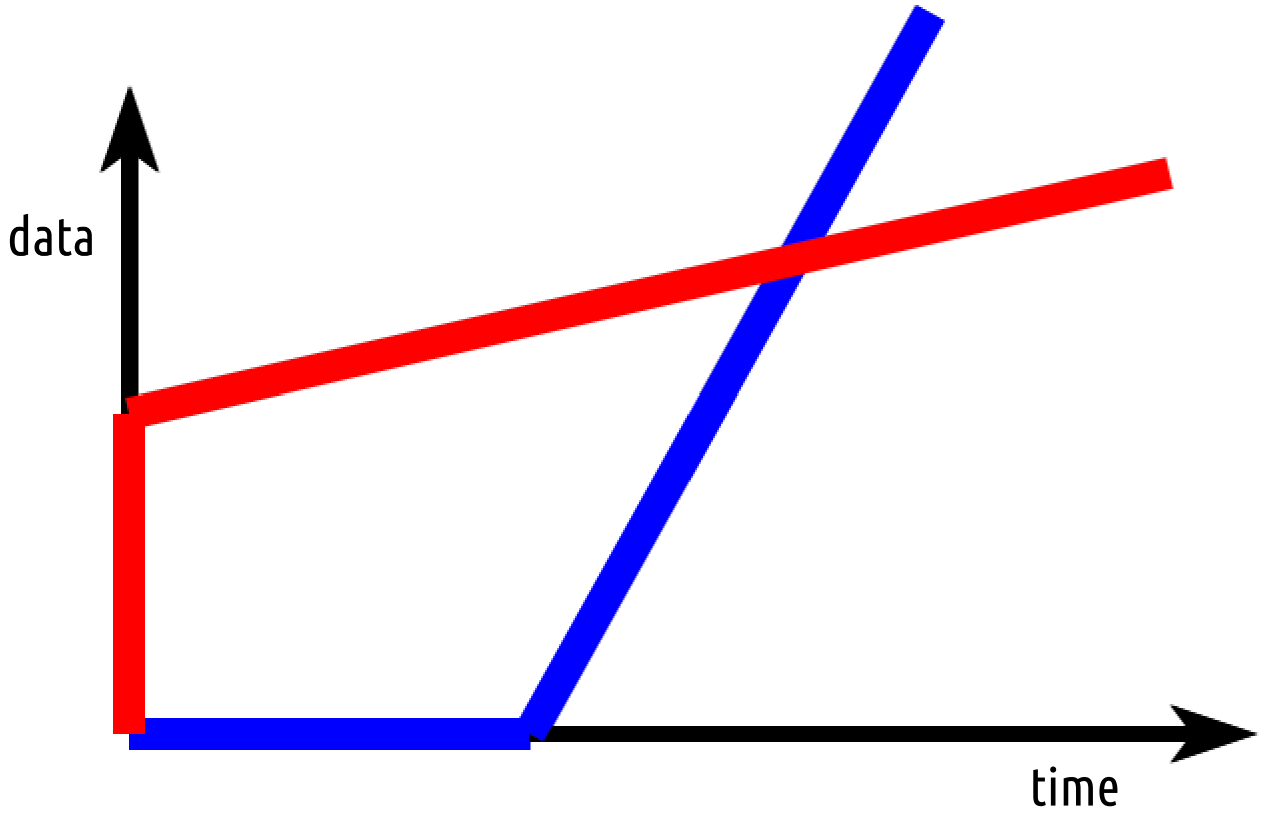


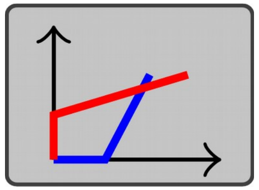
Step 2: Measurements → **deterministic** model for the service



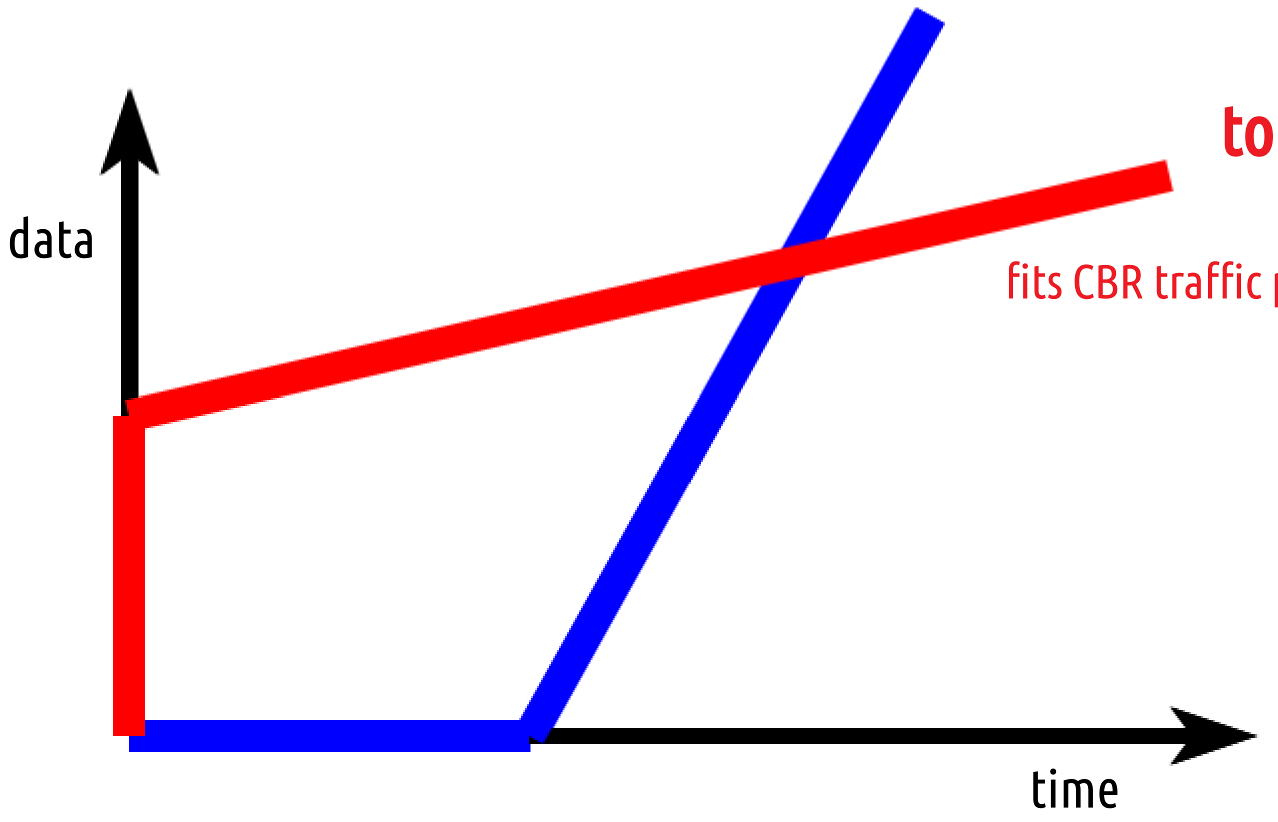


Step 2: Measurements → **deterministic** model for the service





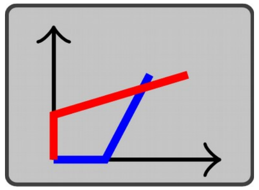
Step 2: Measurements → **deterministic** model for the service



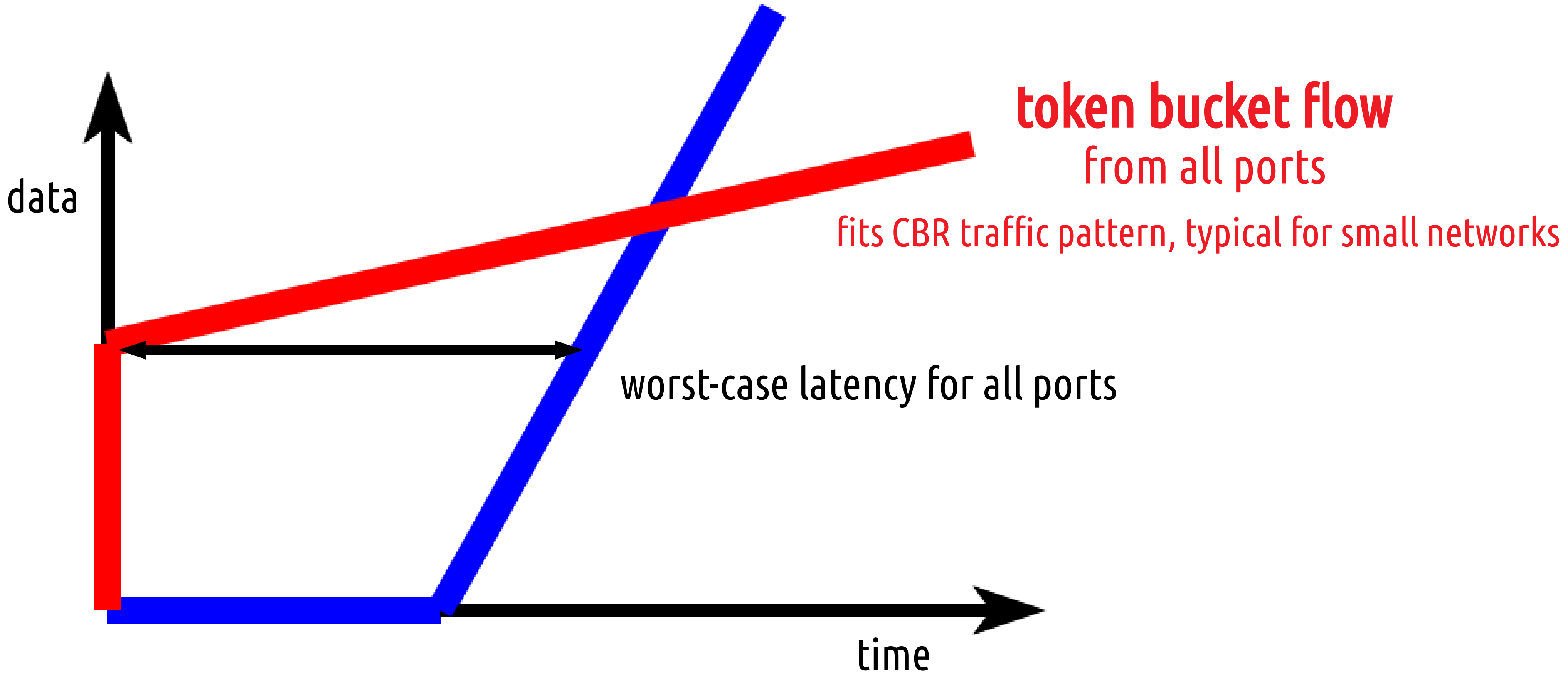
token bucket flow

from all ports

fits CBR traffic pattern, typical for small networks

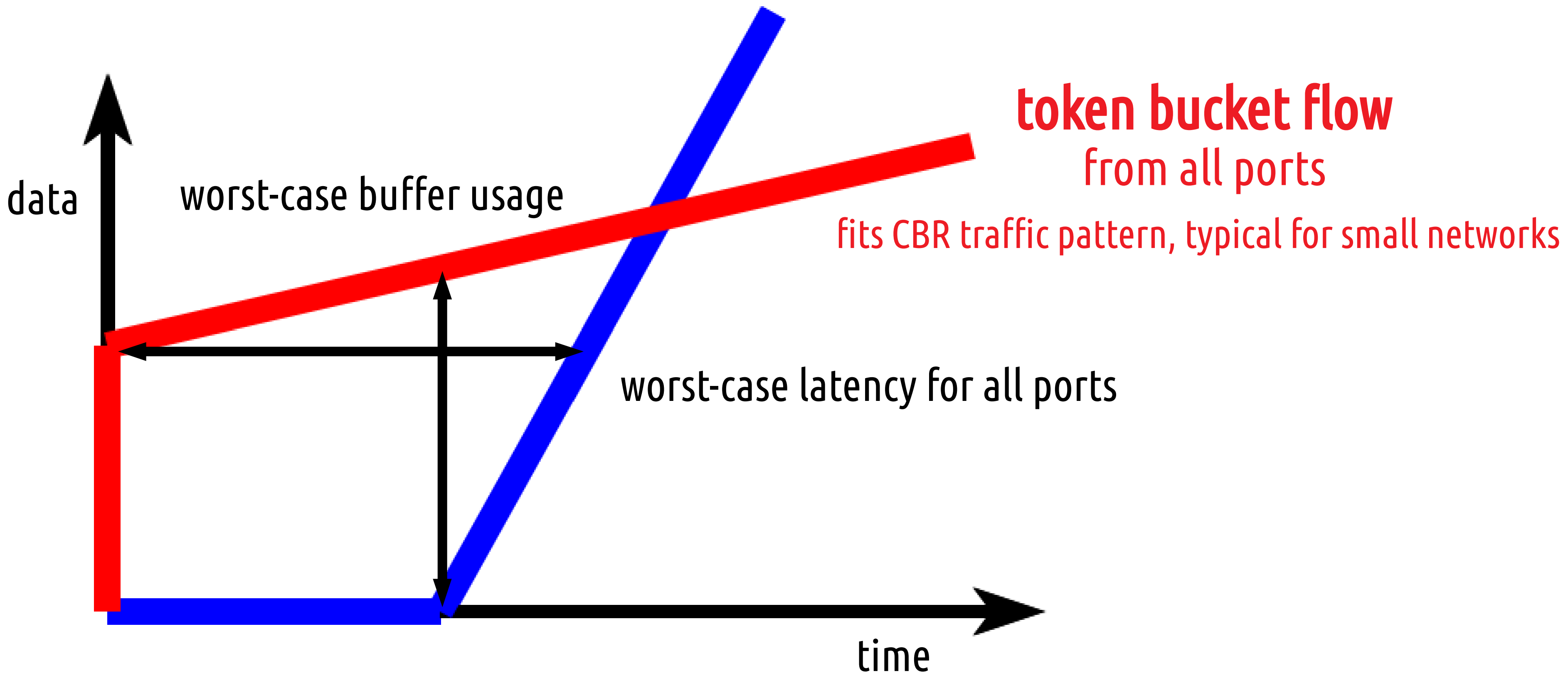


Step 2: Measurements → **deterministic** model for the service



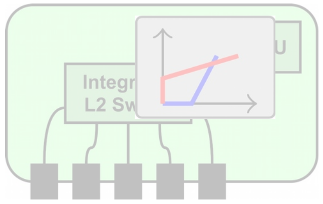
token bucket flow
from all ports
fits CBR traffic pattern, typical for small networks
data
worst-case latency for all ports
time

Step 2: Measurements → **deterministic** model for the service

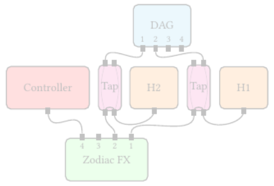


token bucket flow
from all ports
fits CBR traffic pattern, typical for small networks
data
worst-case buffer usage
worst-case latency for all ports
time

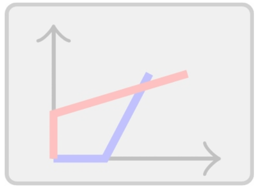
Loko



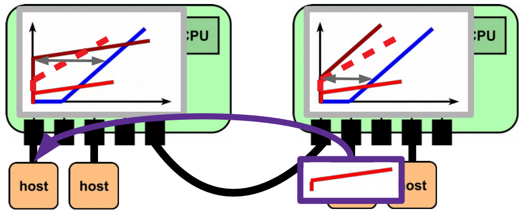
Step 0: Identification of independent services



Step 1: Benchmarking of the service(s)



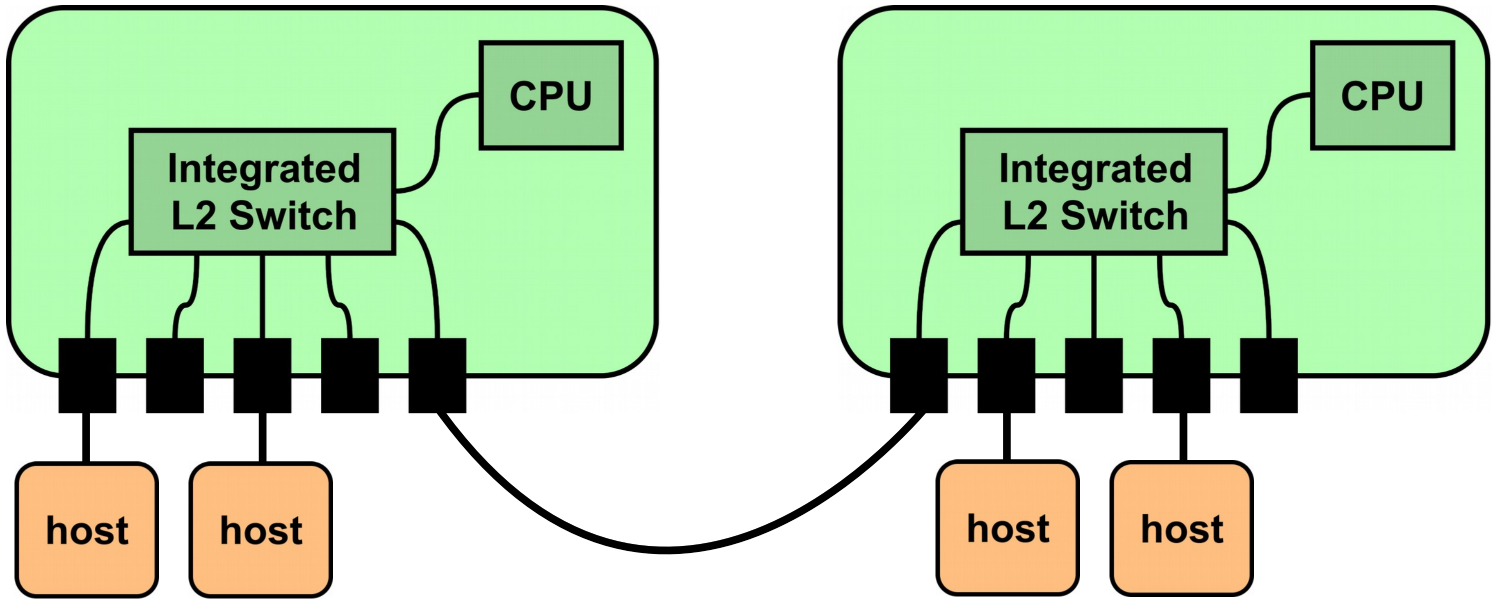
Step 2: Measurements → **deterministic** model for the service(s)



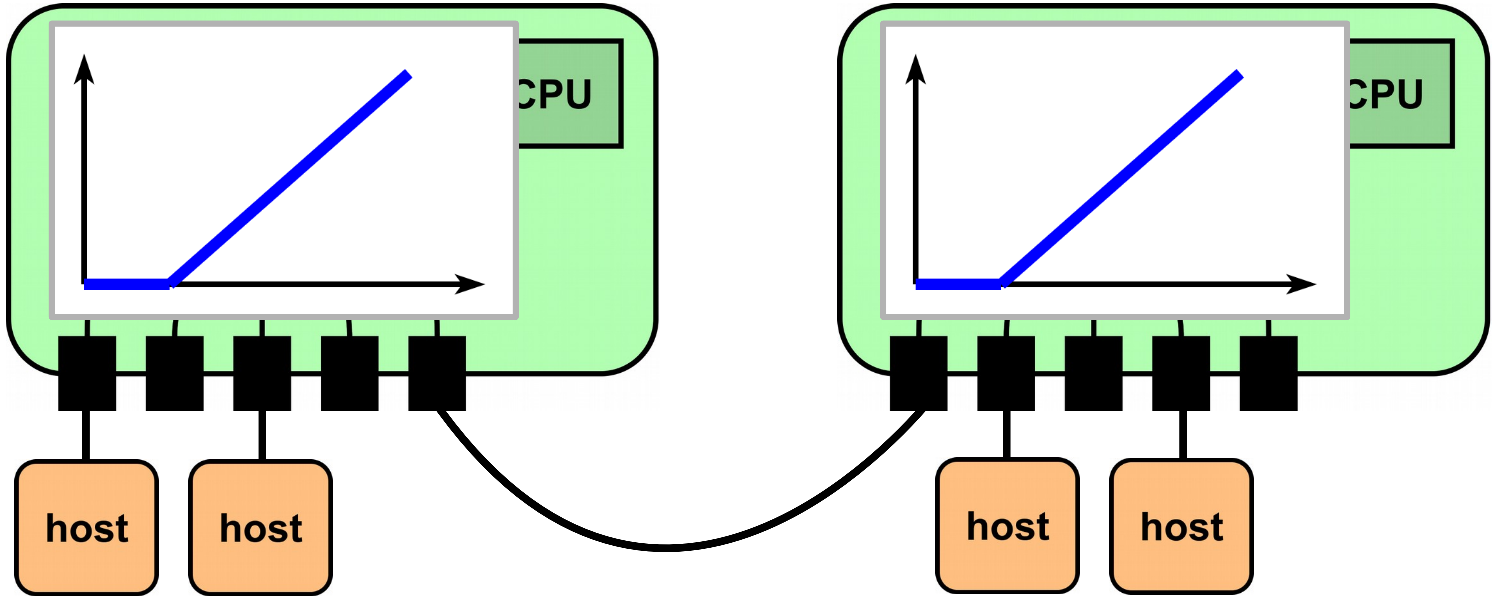
Can I add this flow?

Step 3: Switch model → network model (admission control)

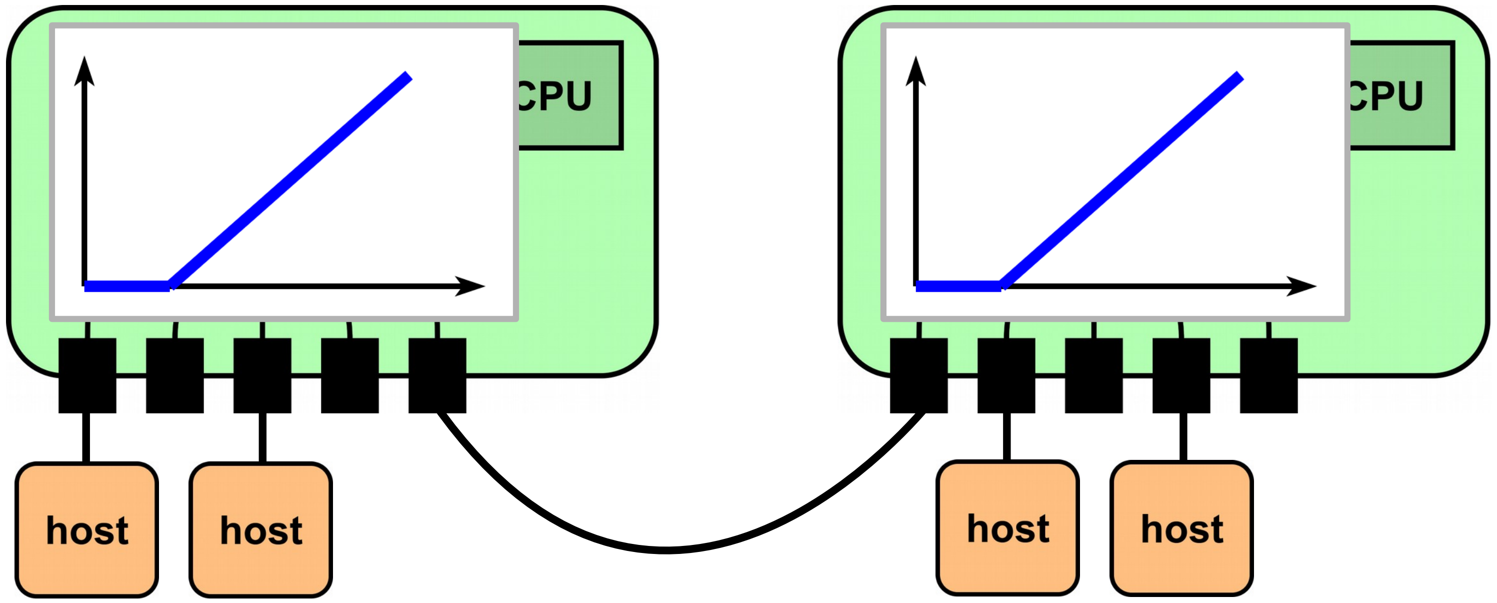
Step 3: Switch model → network model (admission control)



Step 3: Switch model → network model (admission control)

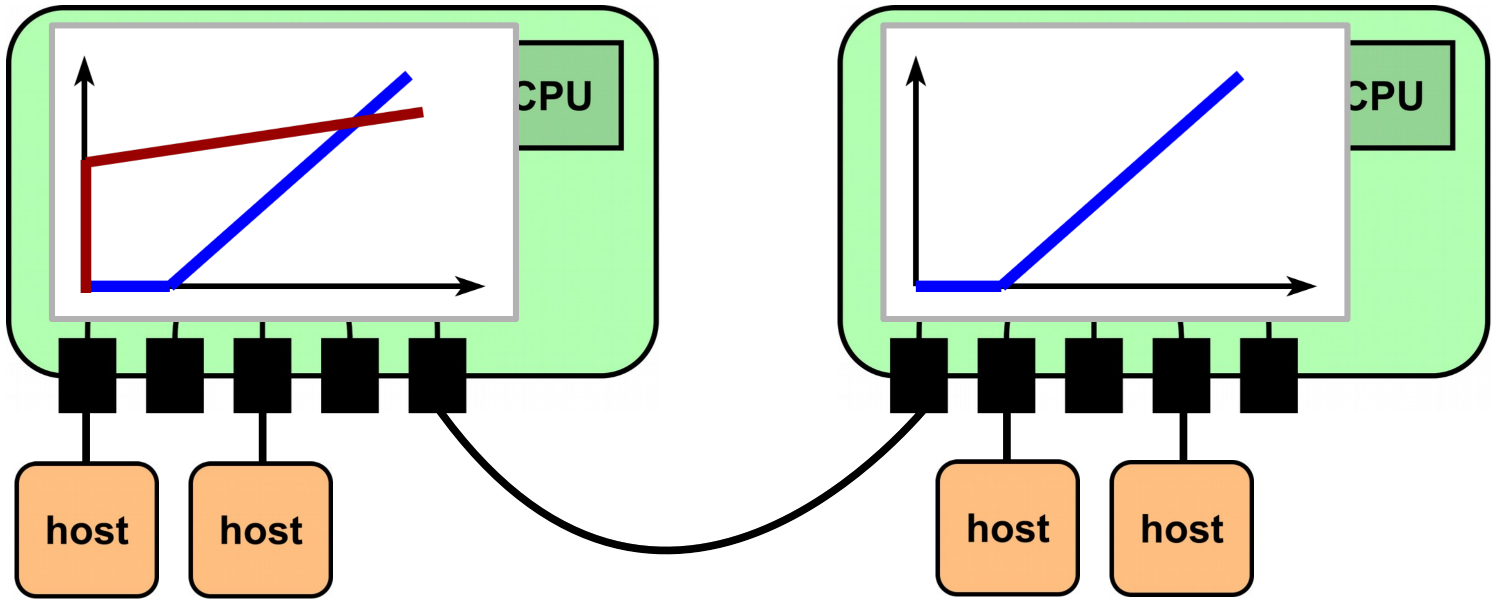


Step 3: Switch model → network model (admission control)



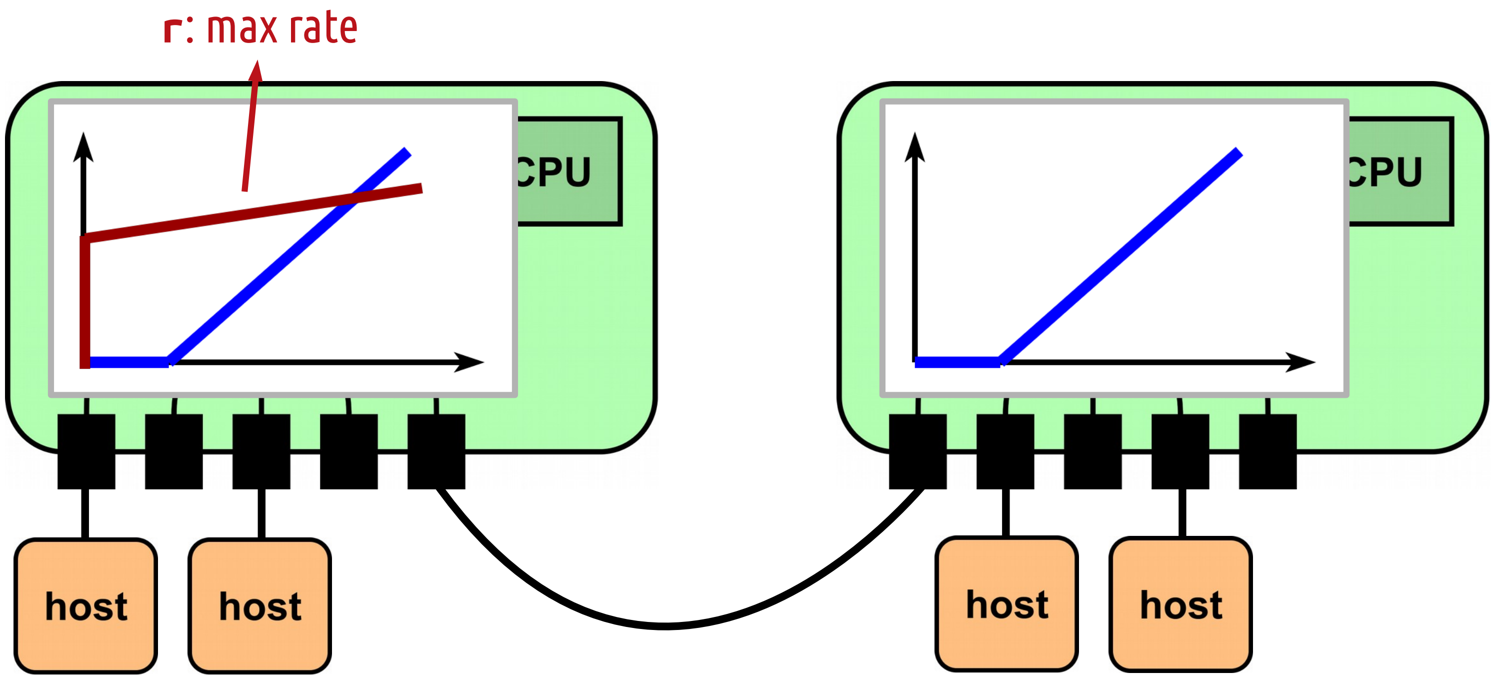
Step 3: Switch model → network model (admission control)

Resource allocation: logically allocate a **maximum rate and burst** to accept at each switch



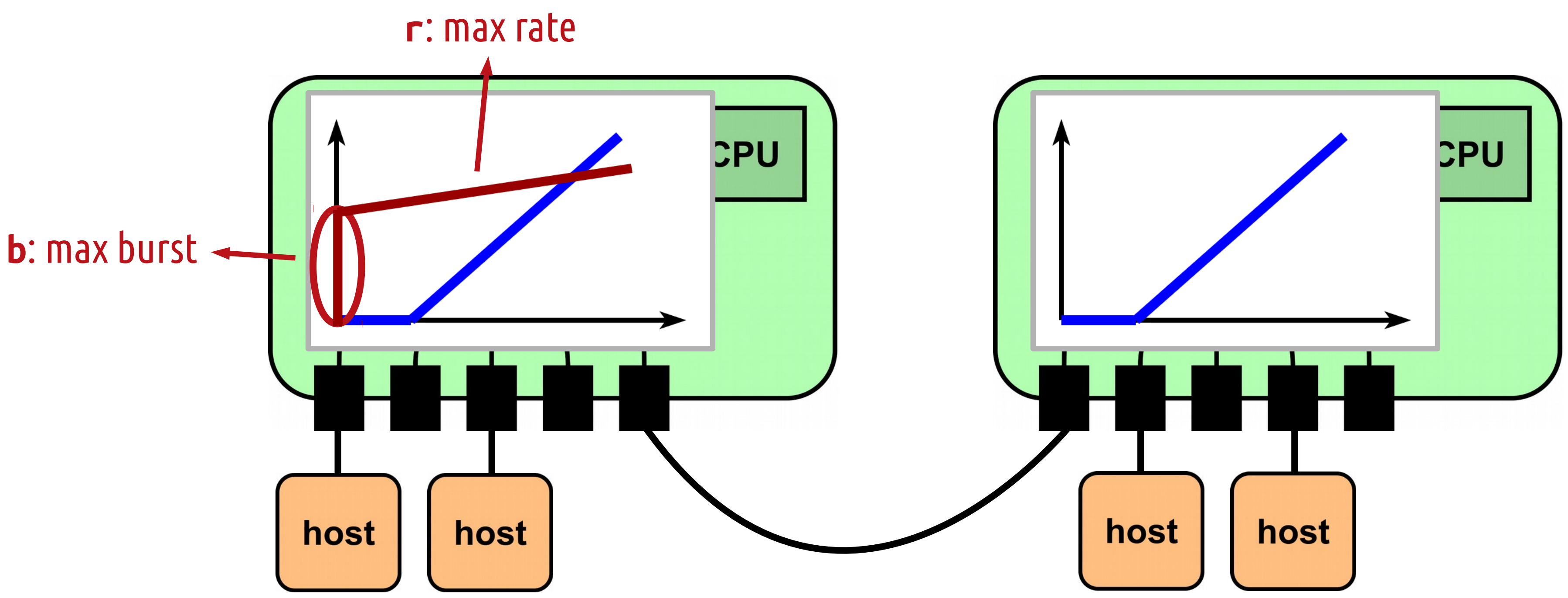
Step 3: Switch model → network model (admission control)

Resource allocation: **logically** allocate a **maximum rate and burst** to accept at each switch



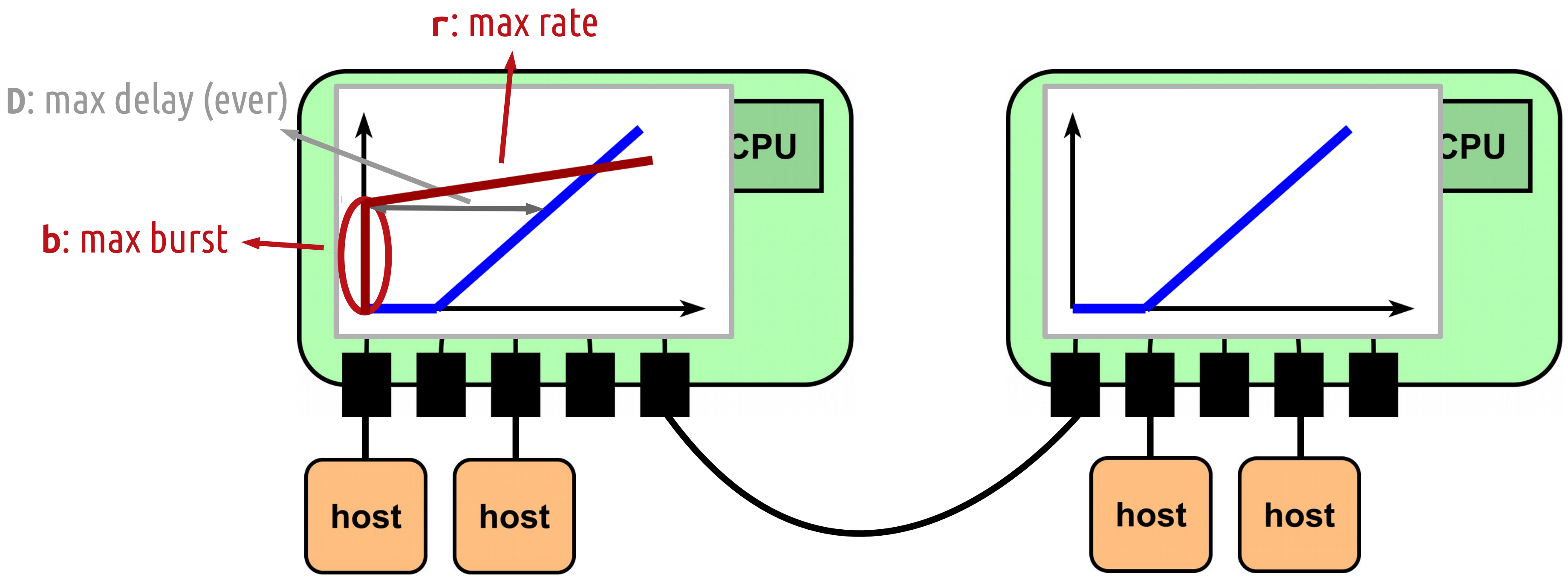
Step 3: Switch model → network model (admission control)

Resource allocation: logically allocate a **maximum rate and burst** to accept at each switch



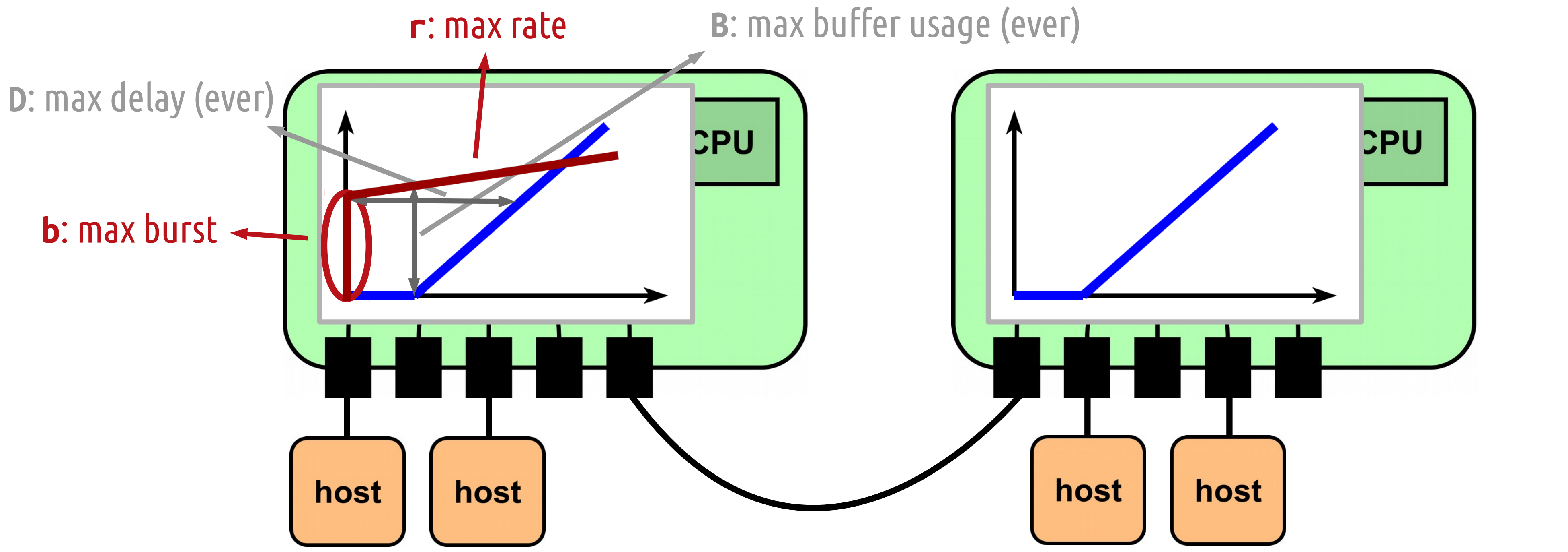
Step 3: Switch model → network model (admission control)

Resource allocation: logically allocate a **maximum rate and burst** to accept at each switch



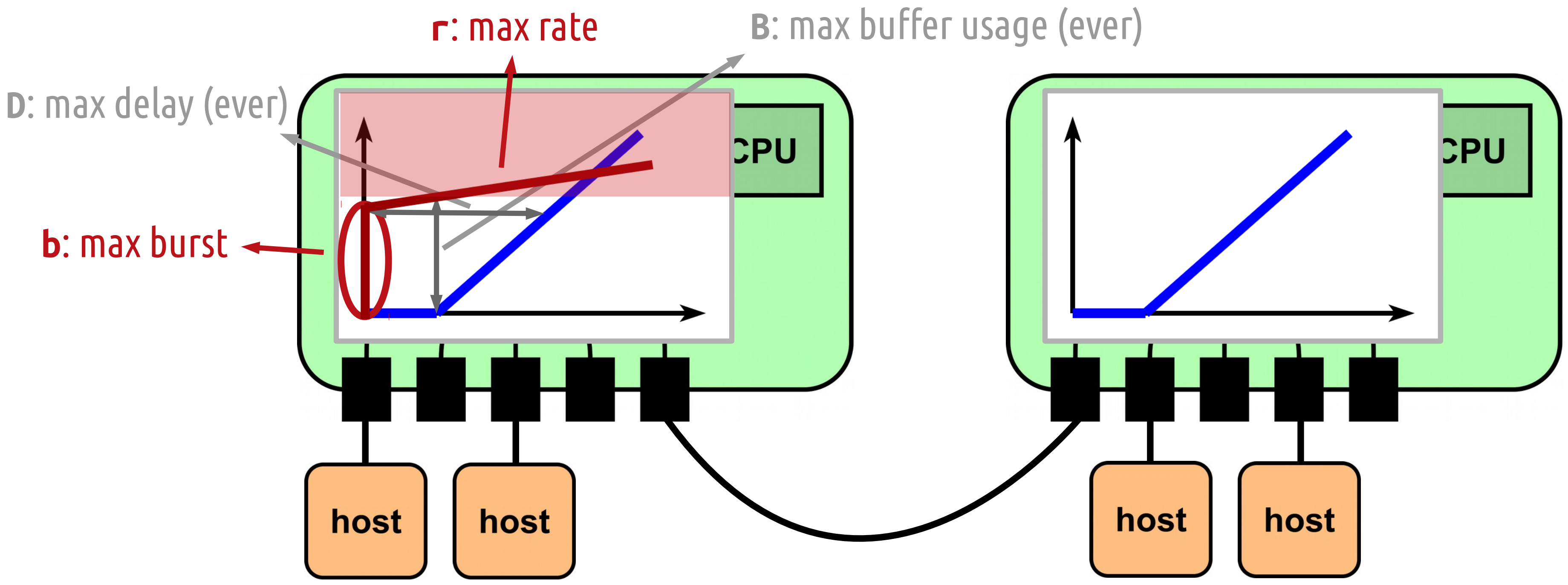
Step 3: Switch model → network model (admission control)

Resource allocation: logically allocate a **maximum rate and burst** to accept at each switch



Step 3: Switch model → network model (admission control)

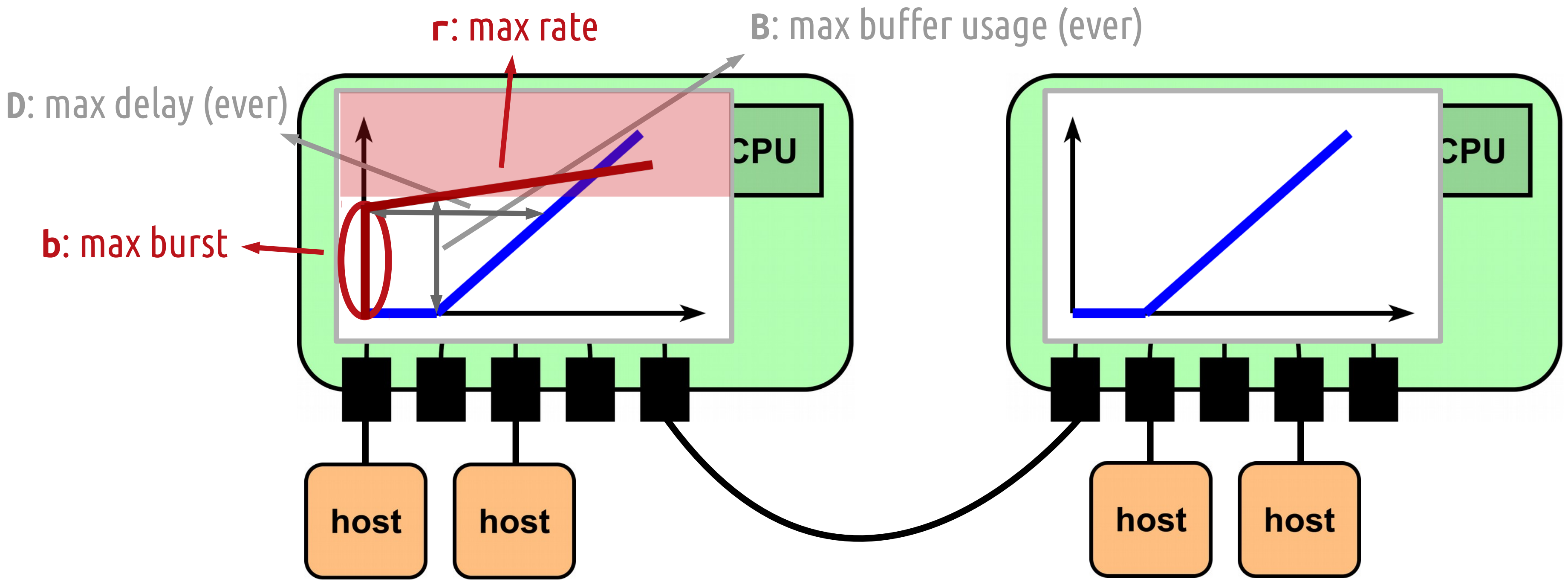
Resource allocation: logically allocate a **maximum rate and burst** to accept at each switch



to ensure no packet loss
choose **r**, **b** such that B ≤ buffer capacity

Step 3: Switch model → network model (admission control)

Resource allocation: logically allocate a **maximum rate and burst** to accept at each switch

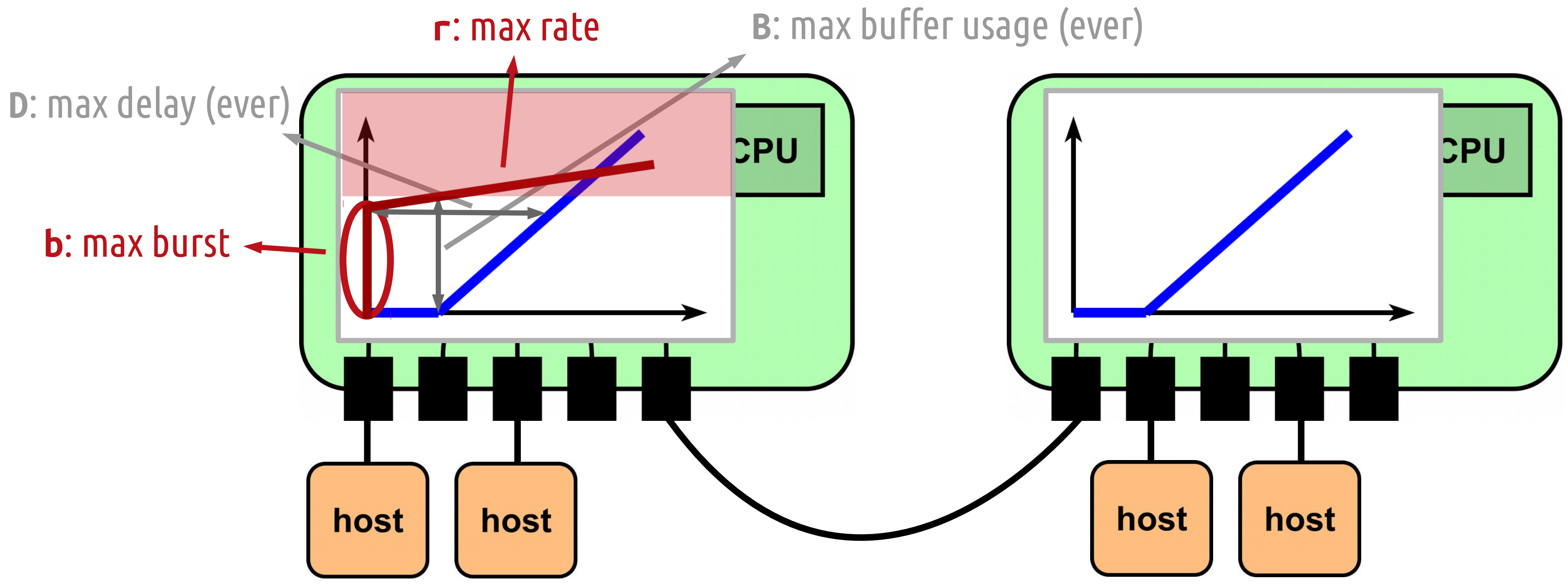


to ensure no packet loss
choose **r**, **b** such that **B** ≤ buffer capacity

for example:
r = **R**/5, max. **b** such that **B** ≤ buffer capacity

Step 3: Switch model → network model (admission control)

Resource allocation: **logically** allocate a **maximum rate and burst** to accept at each switch

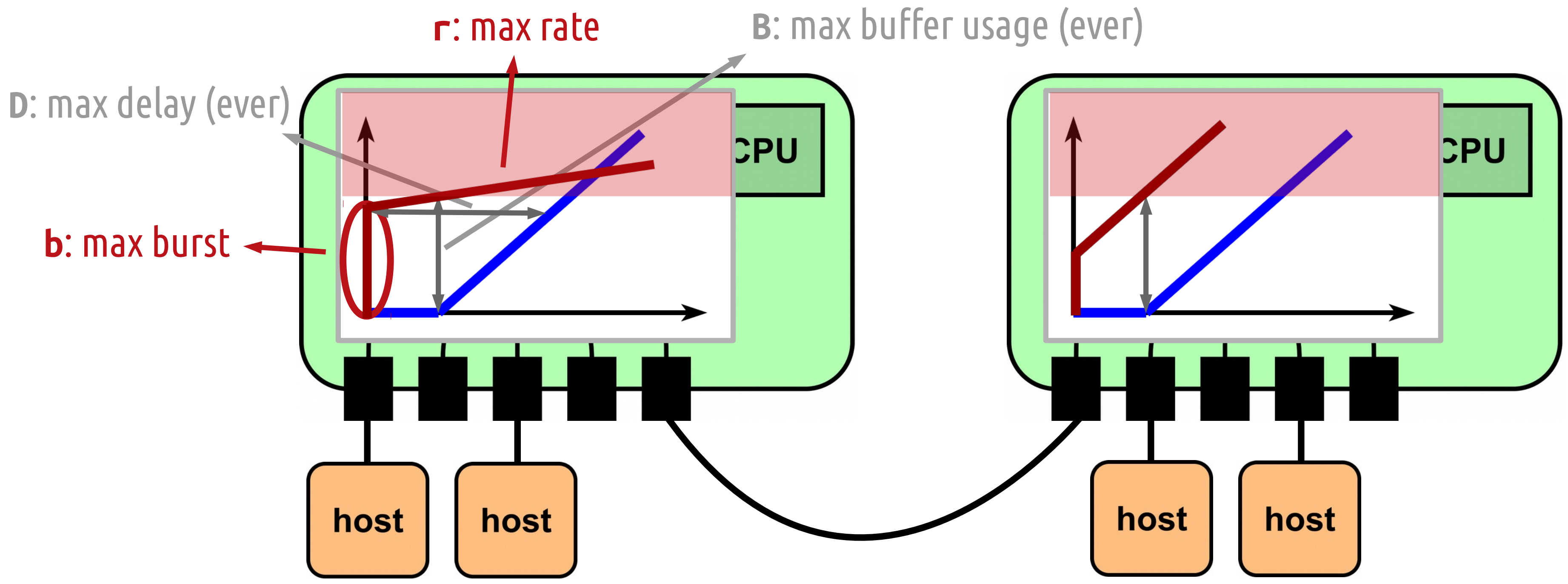


to ensure no packet loss
choose **r**, **b** such that B ≤ buffer capacity

for example:
r = **R**/5, max. **b** such that B ≤ buffer capacity ……… or we can also do **r** = **R**, max. **b** such that B ≤ buffer capacity

Step 3: Switch model → network model (admission control)

Resource allocation: logically allocate a **maximum rate and burst** to accept at each switch

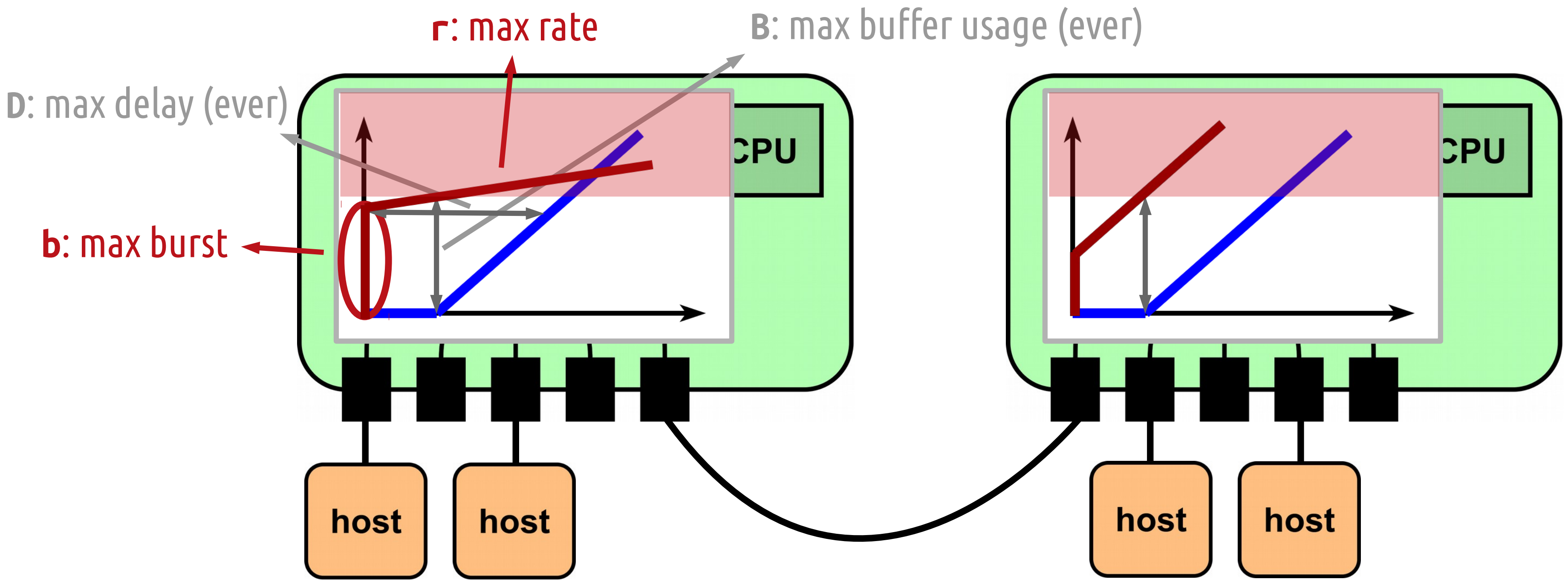


to ensure no packet loss
choose **r**, **b** such that B ≤ buffer capacity

for example:
r = **R**/5, max. **b** such that B ≤ buffer capacity ……… … or we can also do **r** = **R**, max. **b** such that B ≤ buffer capacity

Step 3: Switch model → network model (admission control)

Resource allocation: **logically** allocate a **maximum rate and burst** to accept at each switch

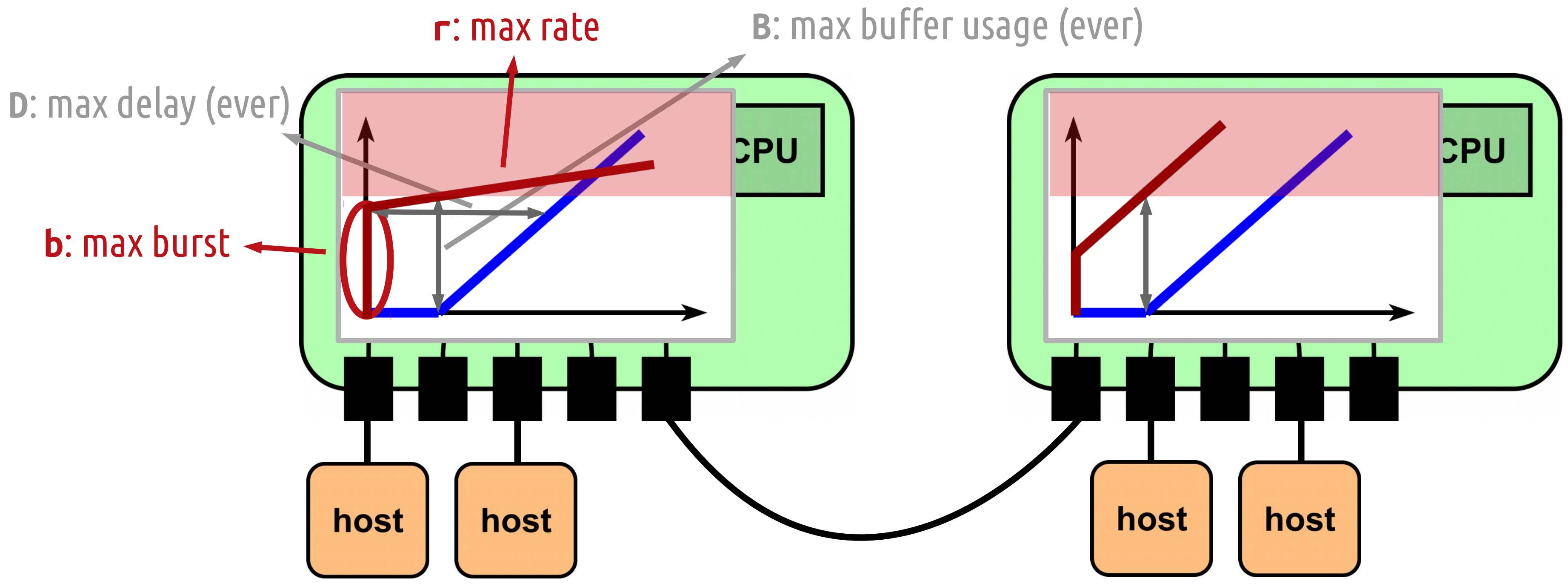


to ensure no packet loss
choose r, b such that B ≤ buffer capacity

for example:
r = R/5, max. b such that B ≤ buffer capacity ………… or we can also do r = R, max. b such that B ≤ buffer capacity … or, … …

Step 3: Switch model → network model (admission control)

Resource allocation: **logically** allocate a **maximum rate and burst** to accept at each switch



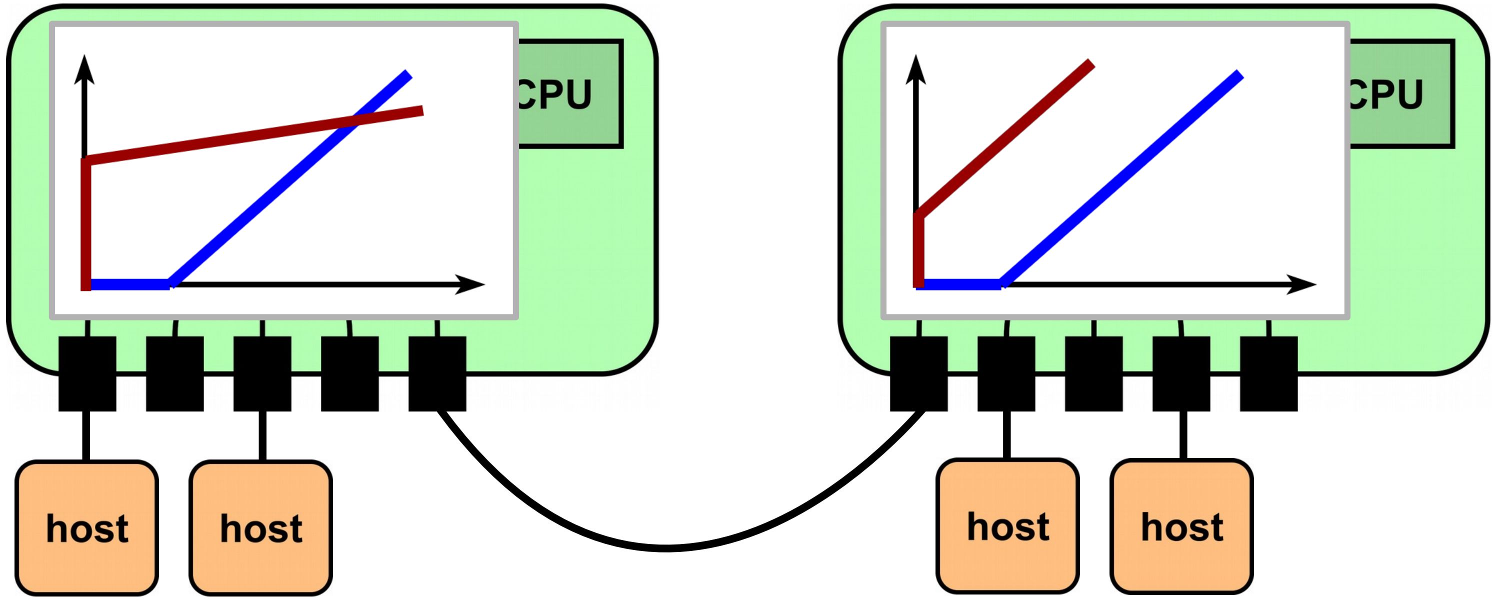
to ensure no packet loss
choose r, b such that B ≤ buffer capacity

for example:
r = R/5, max. b such that B ≤ buffer capacity ……… or we can also do r = R, max. b such that B ≤ buffer capacity … or, … …

Arbitrary decision, but should match traffic type!

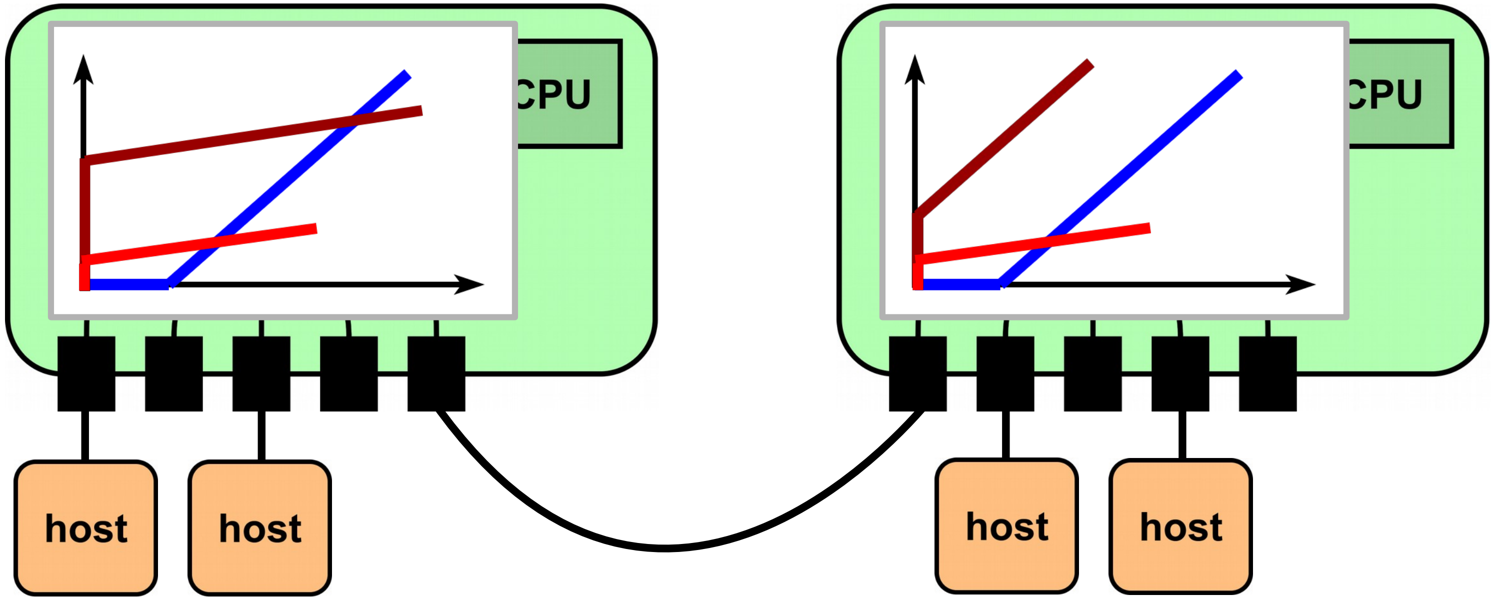
Step 3: Switch model → network model (admission control)

After per-switch **resource allocation**, **admission control** is easy



Step 3: Switch model → network model (admission control)

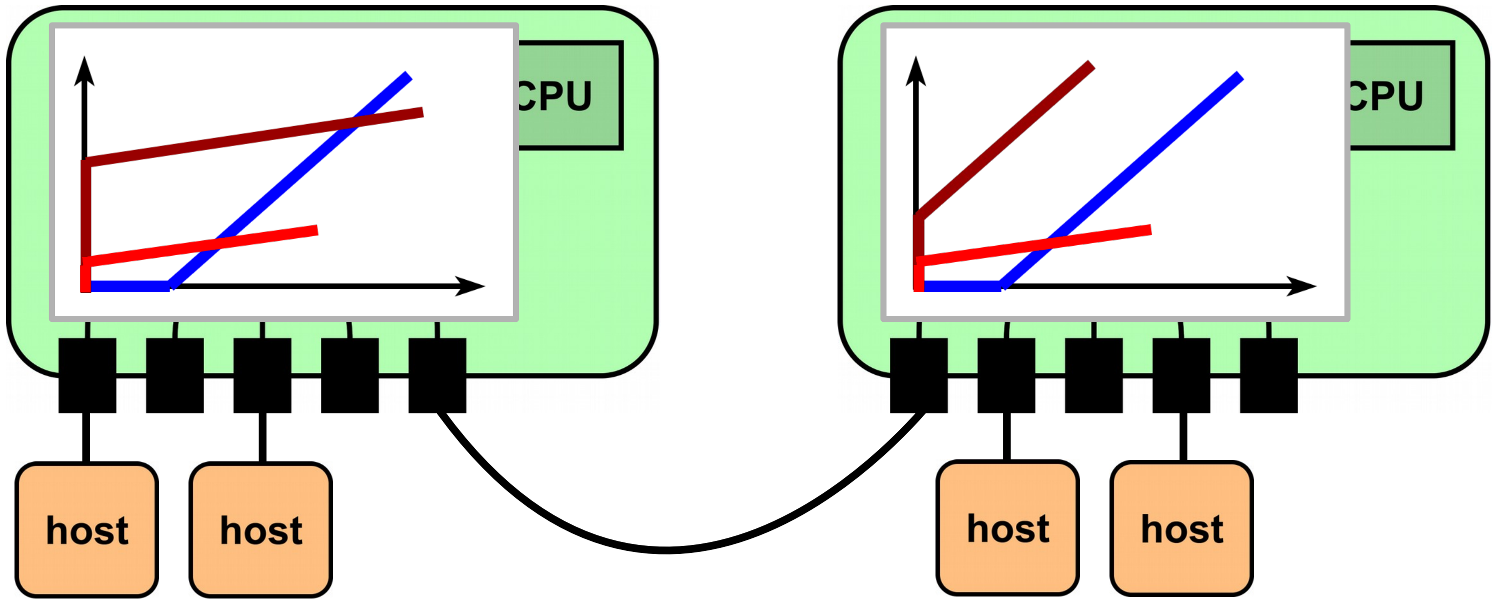
After per-switch **resource allocation**, **admission control** is easy



1. Keep track of **per-switch usage** (burst and rate)

Step 3: Switch model → network model (admission control)

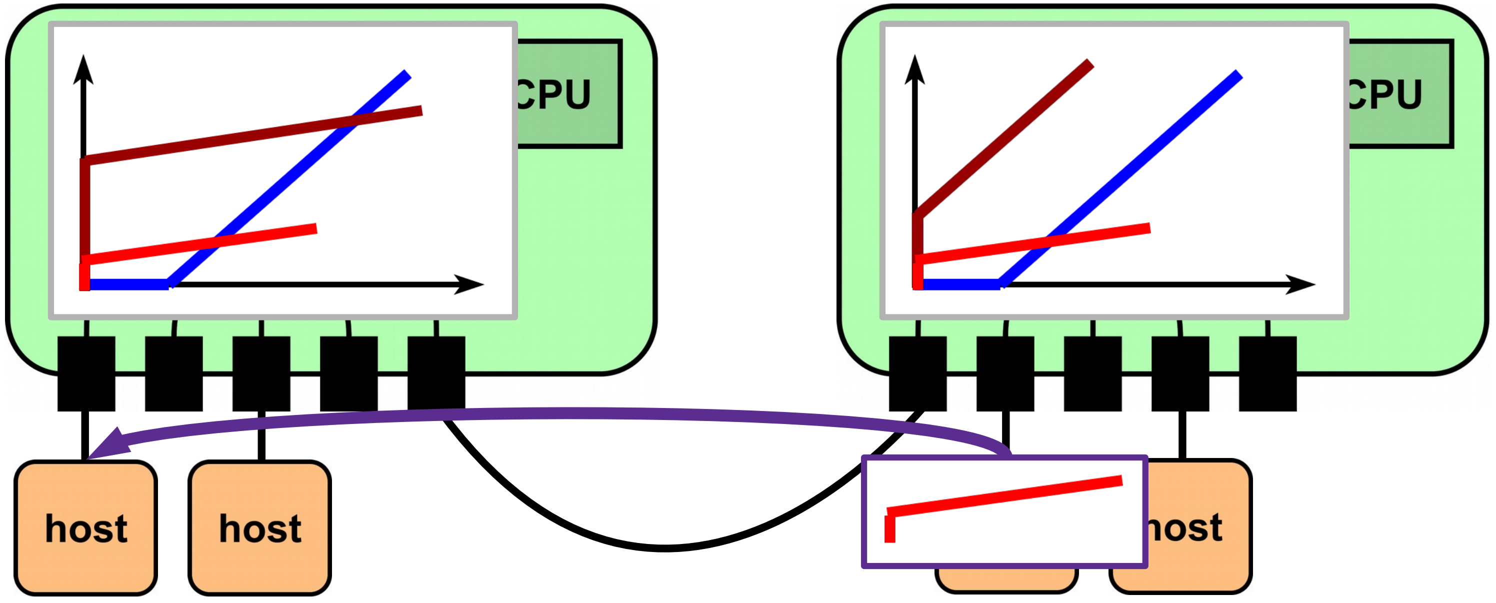
After per-switch **resource allocation**, **admission control** is easy



1. Keep track of **per-switch usage** (burst and rate)
2. Accept as long as **usage** ≤ **allocation**

Step 3: Switch model → network model (admission control)

After per-switch **resource allocation**, **admission control** is easy

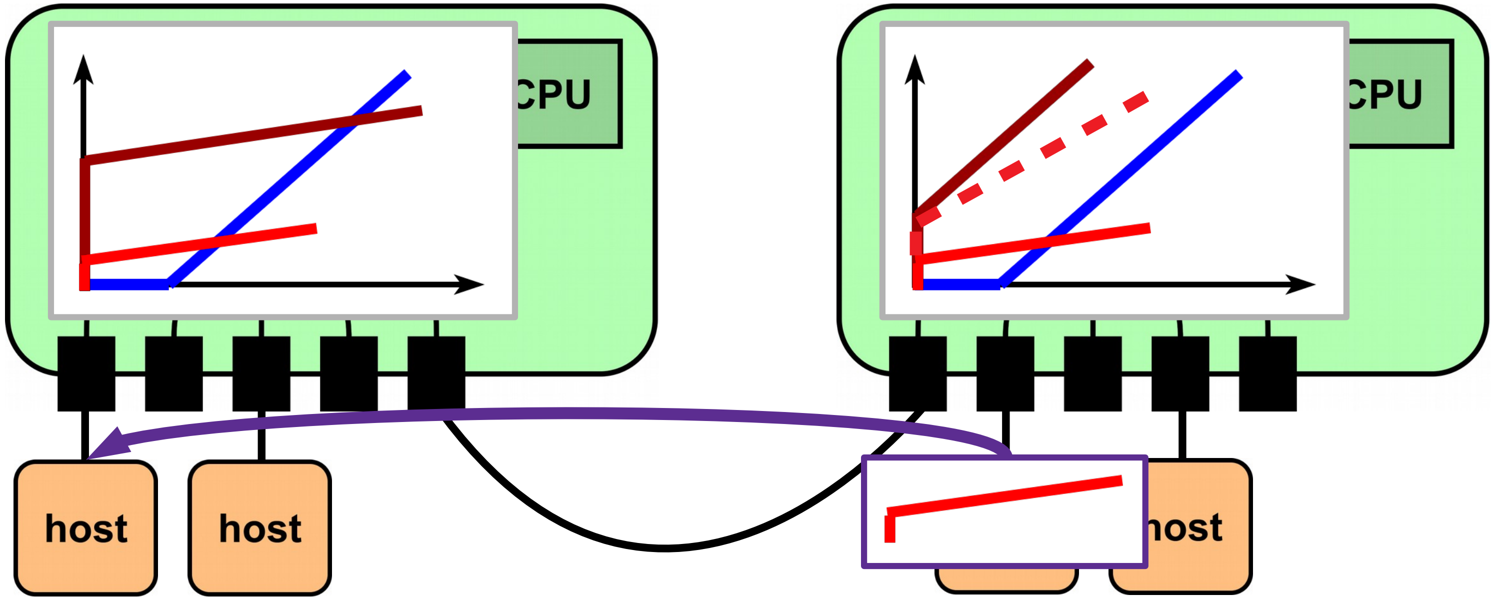


Can I add this flow?

1. Keep track of **per-switch usage** (burst and rate)
2. Accept as long as **usage** ≤ **allocation**

Step 3: Switch model → network model (admission control)

After per-switch **resource allocation**, **admission control** is easy

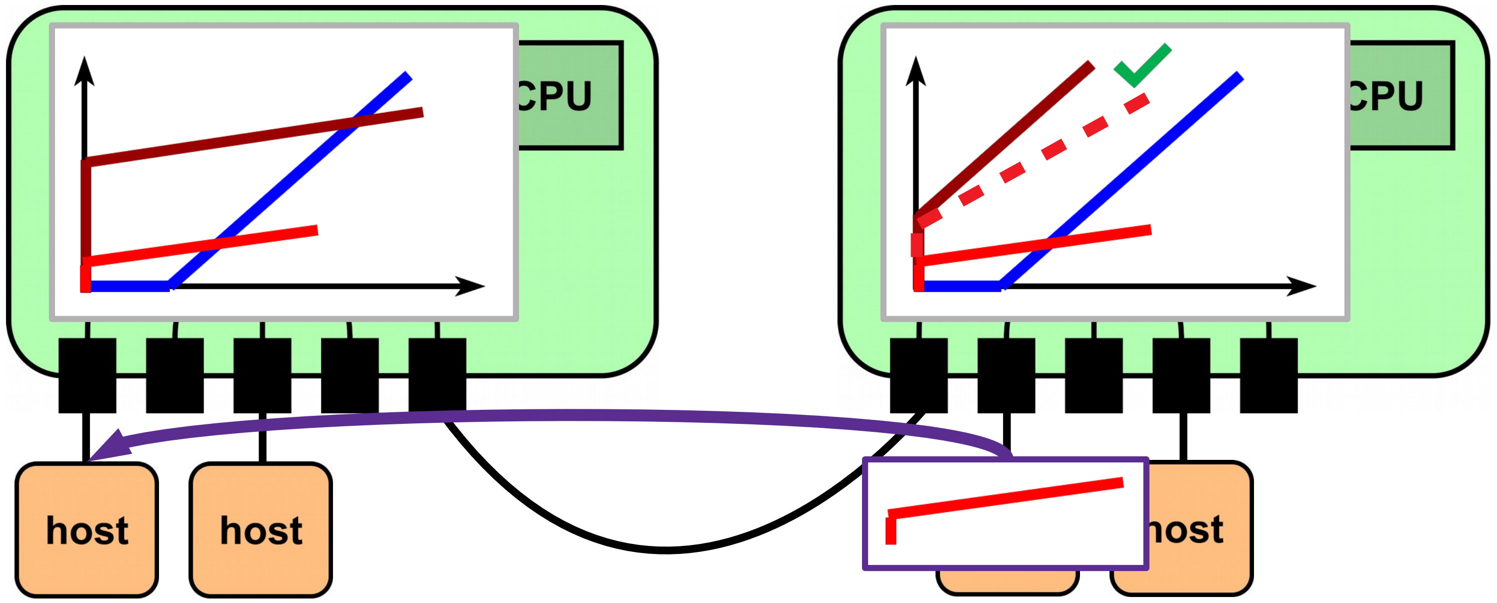


Can I add this flow?

1. Keep track of **per-switch usage** (burst and rate)
2. Accept as long as **usage** ≤ **allocation**

Step 3: Switch model → network model (admission control)

After per-switch **resource allocation**, **admission control** is easy

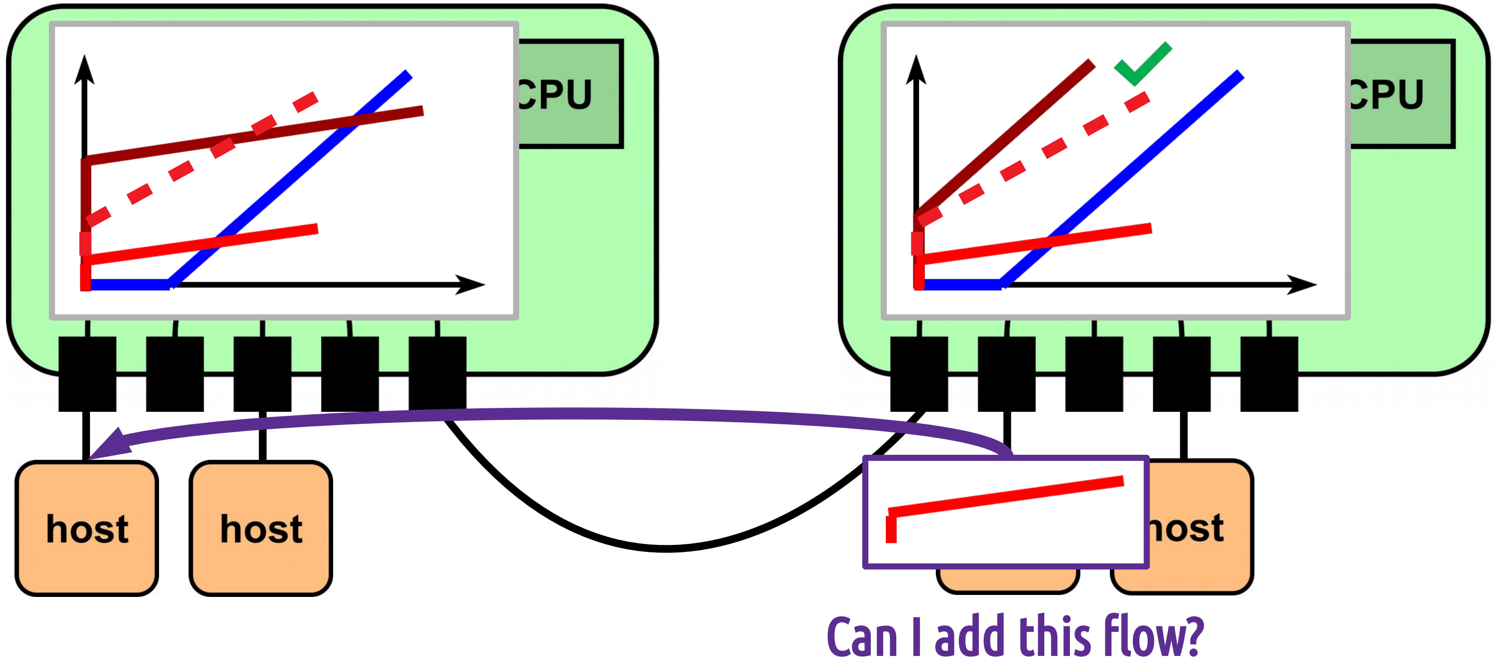


Can I add this flow?

1. Keep track of **per-switch usage** (burst and rate)
2. Accept as long as **usage** ≤ **allocation**

Step 3: Switch model → network model (admission control)

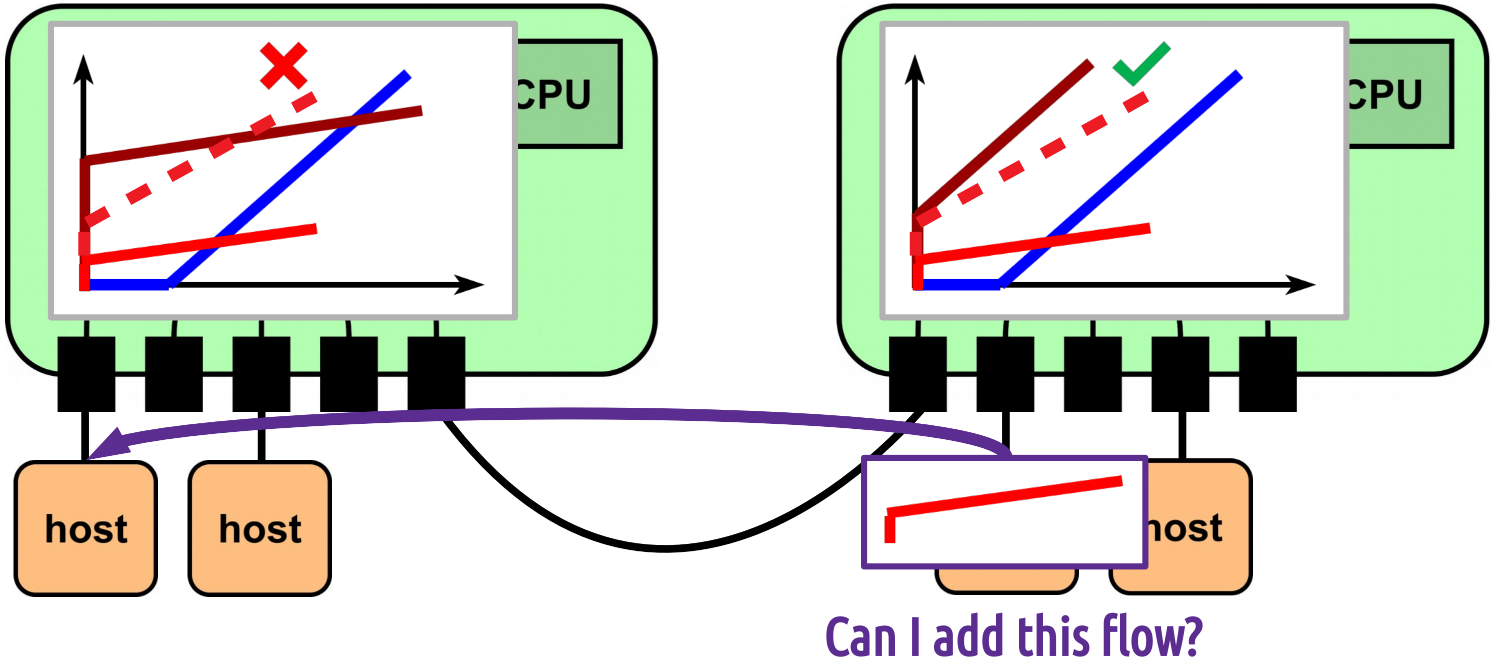
After per-switch **resource allocation**, **admission control** is easy



1. Keep track of **per-switch usage** (burst and rate)
2. Accept as long as **usage** ≤ **allocation**

Step 3: Switch model → network model (admission control)

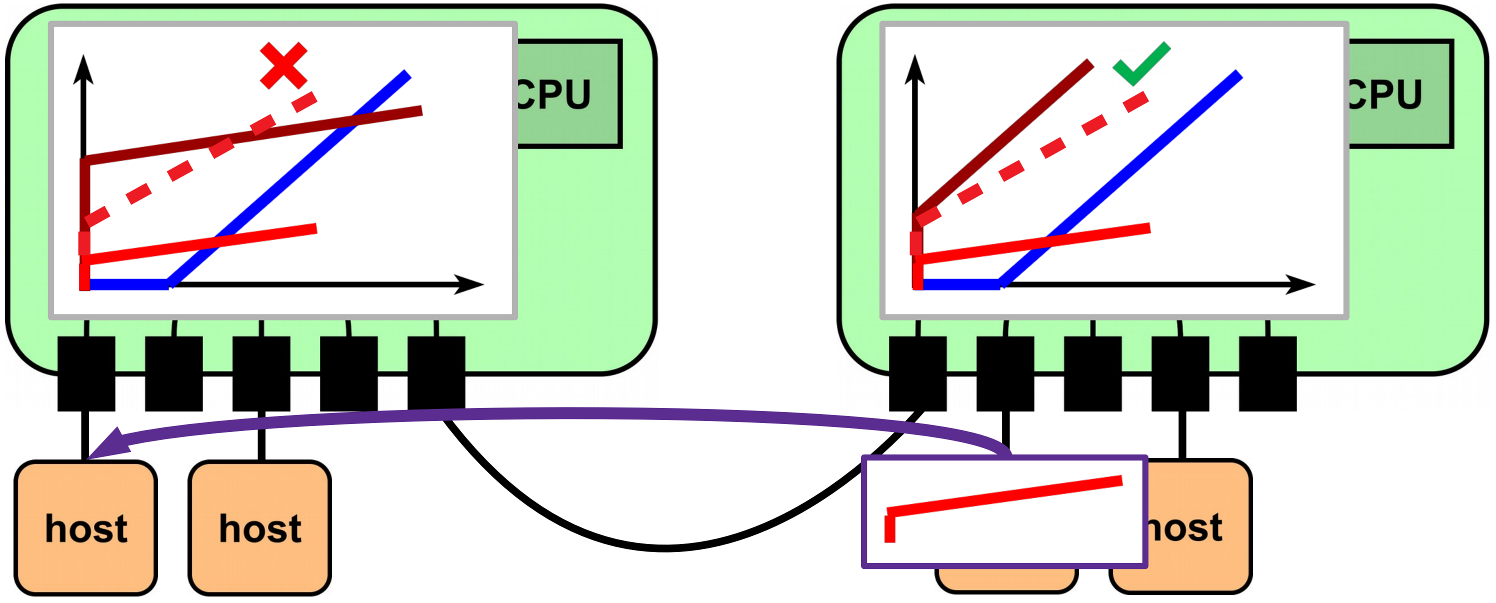
After per-switch **resource allocation**, **admission control** is easy



1. Keep track of **per-switch usage** (burst and rate)
2. Accept as long as **usage** ≤ **allocation**

Step 3: Switch model → network model (admission control)

After per-switch **resource allocation**, **admission control** is easy

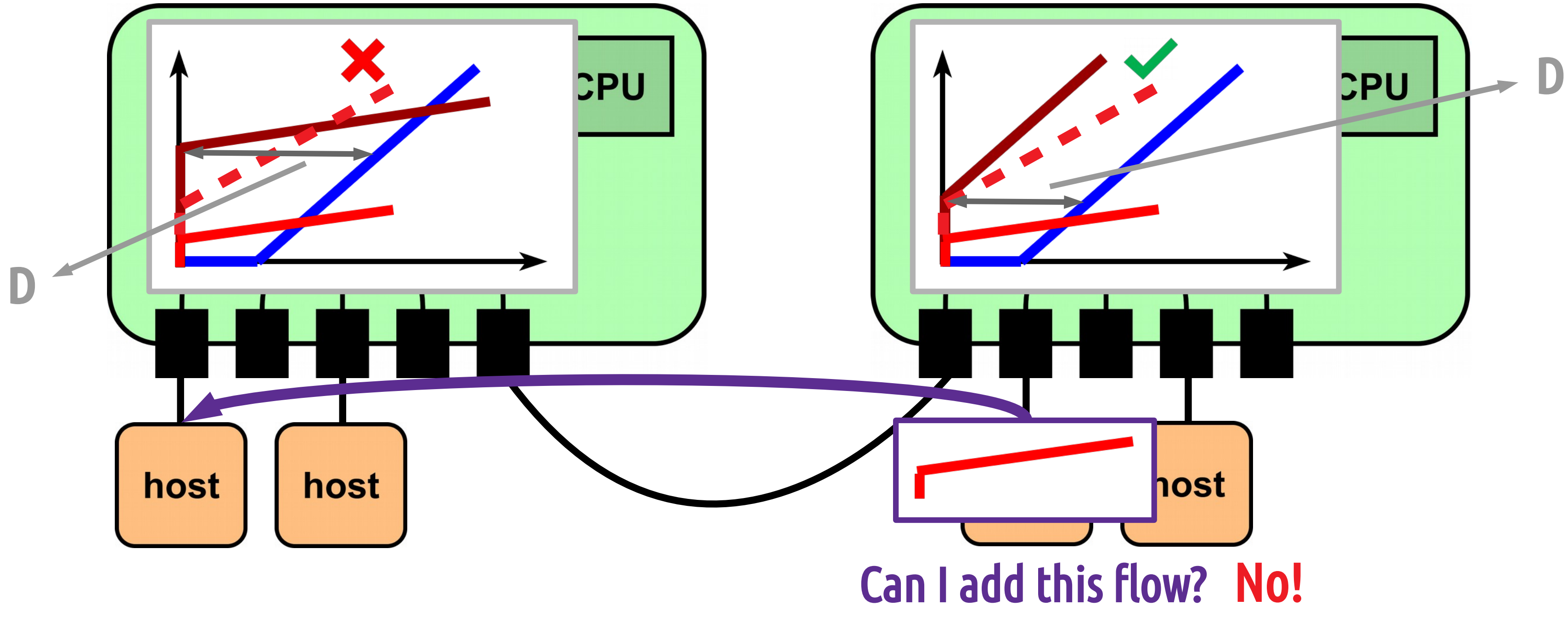


Can I add this flow? No!

1. Keep track of **per-switch usage** (burst and rate)
2. Accept as long as **usage** ≤ **allocation**

Step 3: Switch model → network model (admission control)

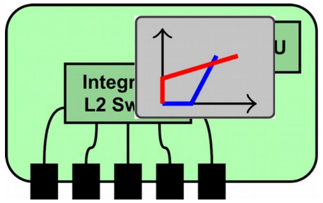
After per-switch resource allocation, admission control is easy



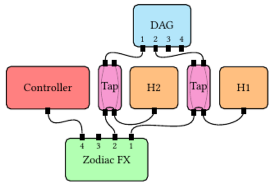
1. Keep track of per-switch usage (burst and rate)
2. Accept as long as usage ≤ allocation

Latency guarantee: sum of the D values at each hop

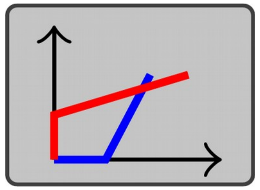
Loko



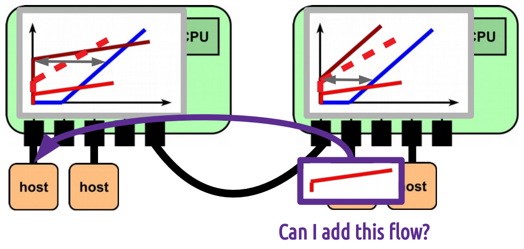
Step 0: Identification of independent services



Step 1: Benchmarking of the service(s)



Step 2: Measurements → **deterministic** model for the service(s)

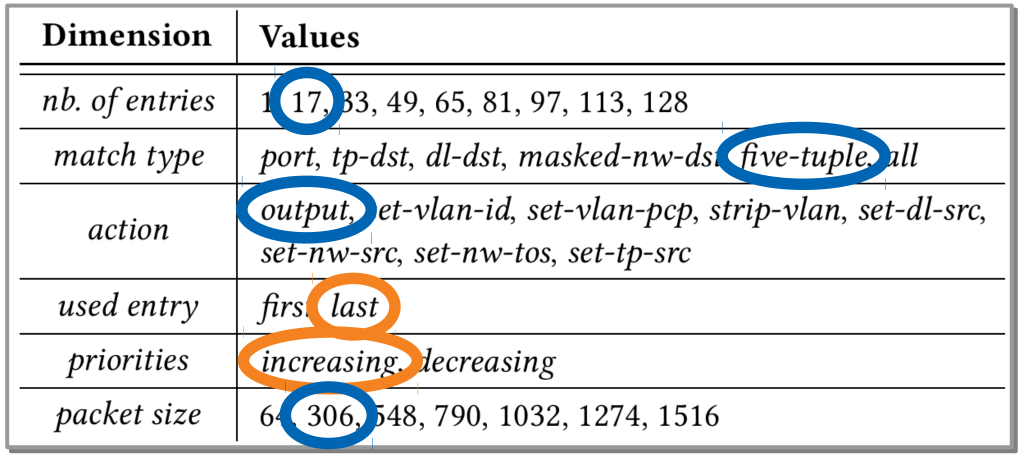


Step 3: Switch model → network model (admission control)

Loko: Proof-of-Concept Implementation and Evaluation

Loko: Proof-of-Concept Implementation and Evaluation

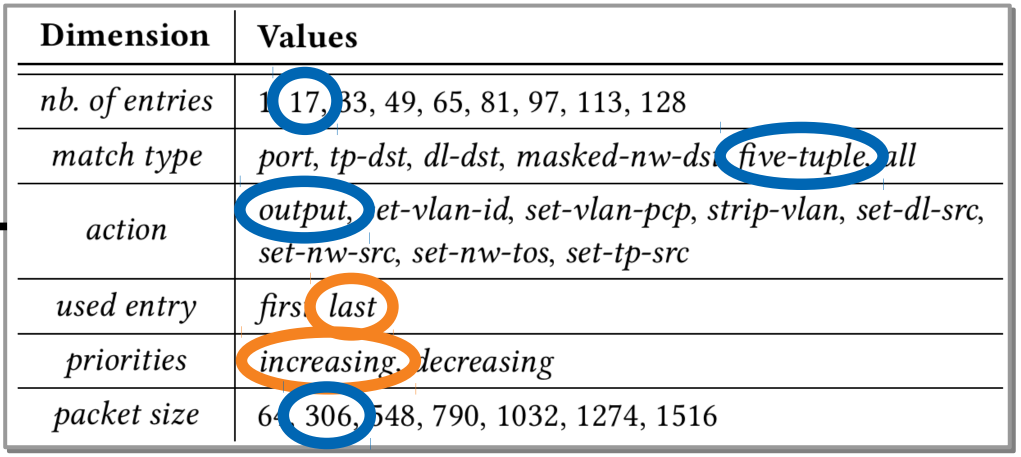
Take the worst-case
for a given scenario



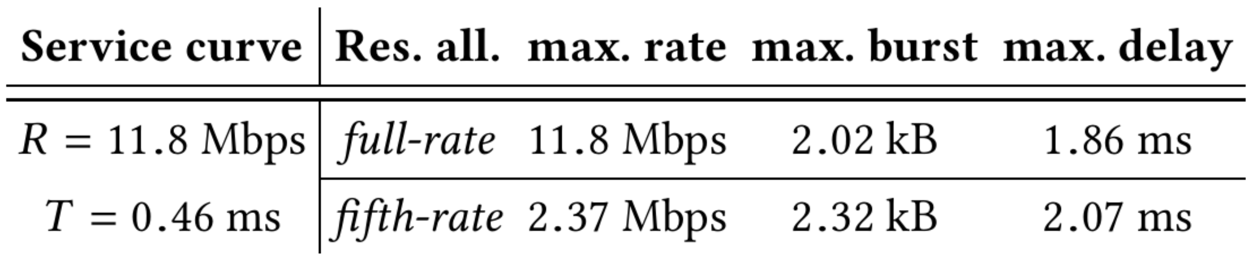
Dimension	Values
nb. of entries	1, [✓ 17], 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, [✓ five-tuple], all*
action	*[✓ output], set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, [✓ last]*
priorities	*[✓ increasing], decreasing*
packet size	64, [✓ 306], 548, 790, 1032, 1274, 1516

Loko: Proof-of-Concept Implementation and Evaluation

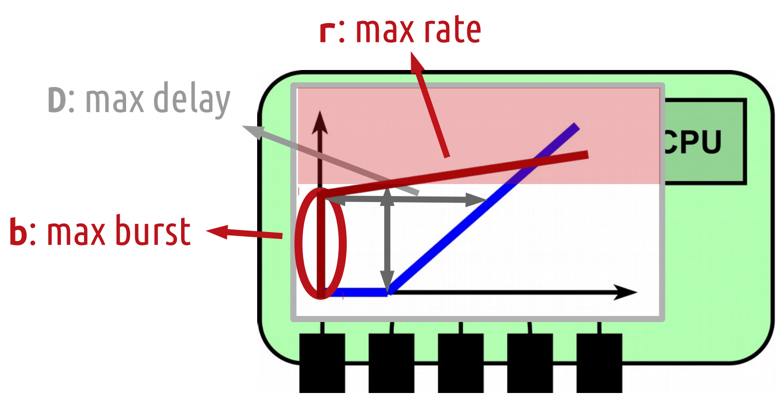
Take the **worst-case** for a given **scenario**



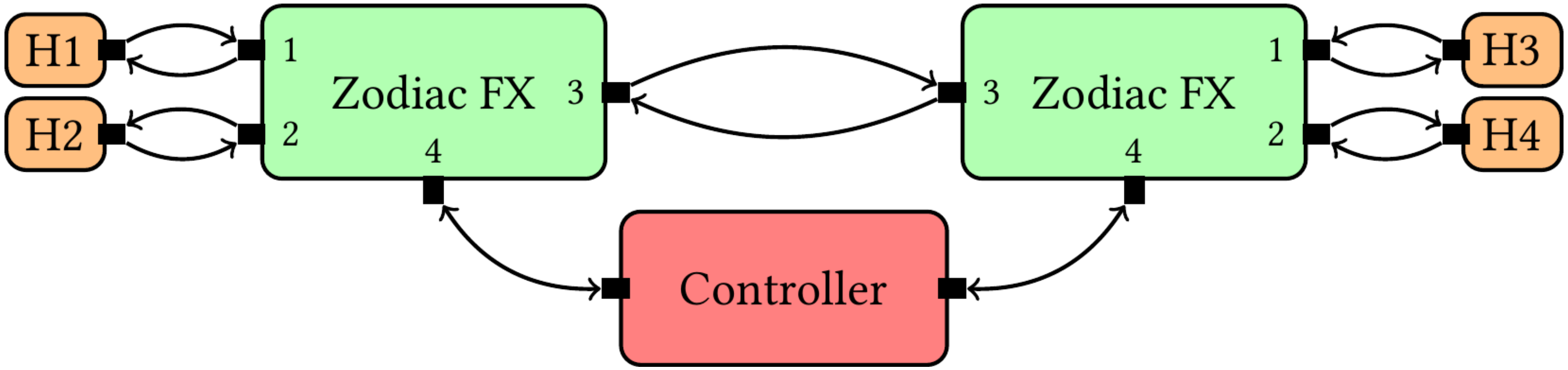
Dimension	Values
nb. of entries	1, [✓ 17], 33, 49, 65, 81, 97, 113, 128
match type	*port, tp-dst, dl-dst, masked-nw-dst, [✓ five-tuple], all*
action	*[✓ output], set-vlan-id, set-vlan-pcp, strip-vlan, set-dl-src, set-nw-src, set-nw-tos, set-tp-src*
used entry	*first, [✓ last]*
priorities	*[✓ increasing], decreasing*
packet size	64, [✓ 306], 548, 790, 1032, 1274, 1516



Service curve	Res. all.	max. rate	max. burst	max. delay
$R = 11.8$ Mbps	*full-rate*	11.8 Mbps	2.02 kB	1.86 ms
$T = 0.46$ ms	*fifth-rate*	2.37 Mbps	2.32 kB	2.07 ms

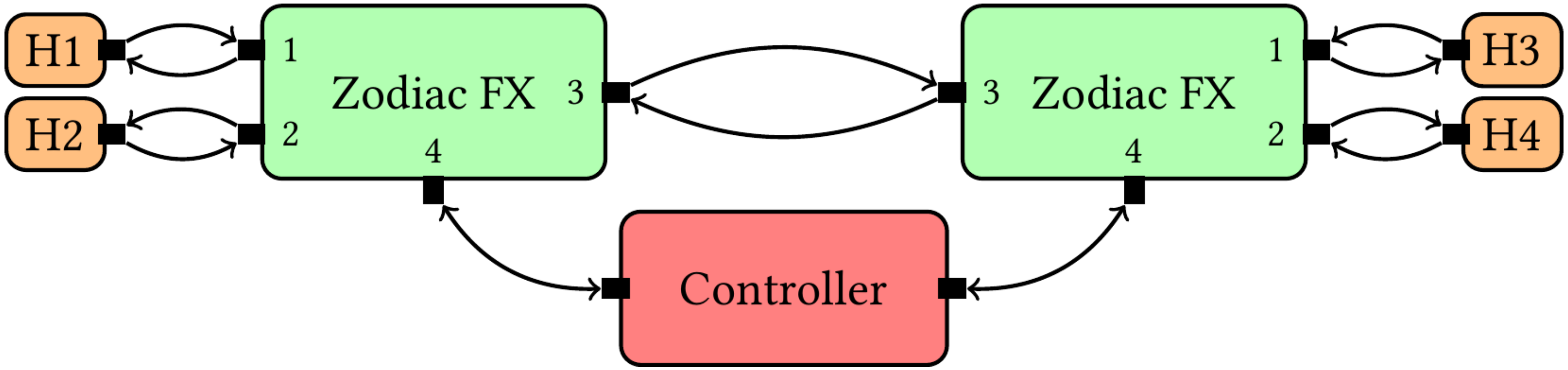


Loko: Proof-of-Concept Implementation and Evaluation



We add flows and observe delays/losses between H1–H3

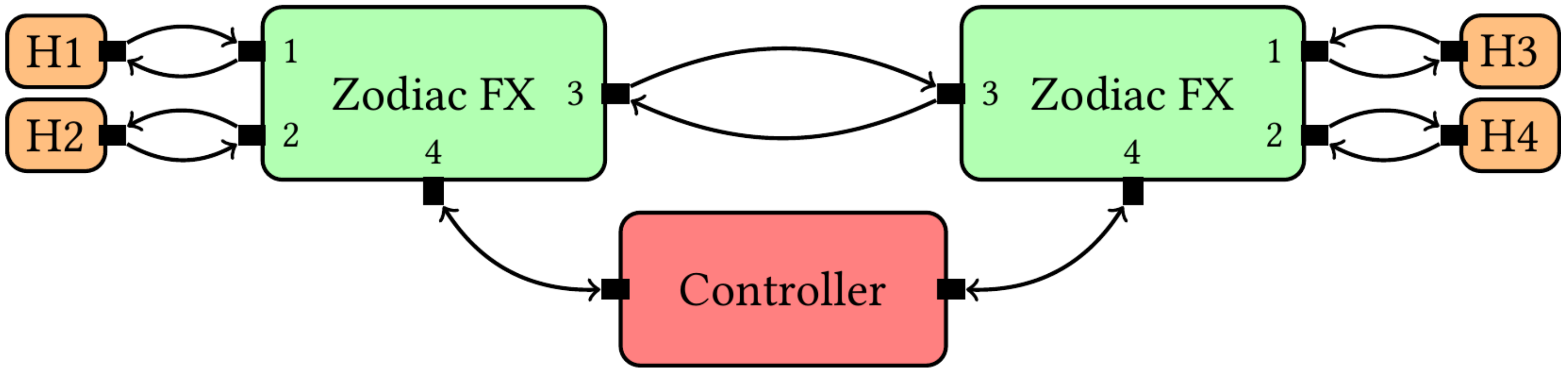
Loko: Proof-of-Concept Implementation and Evaluation



We add flows and observe delays/losses between **H1–H3**

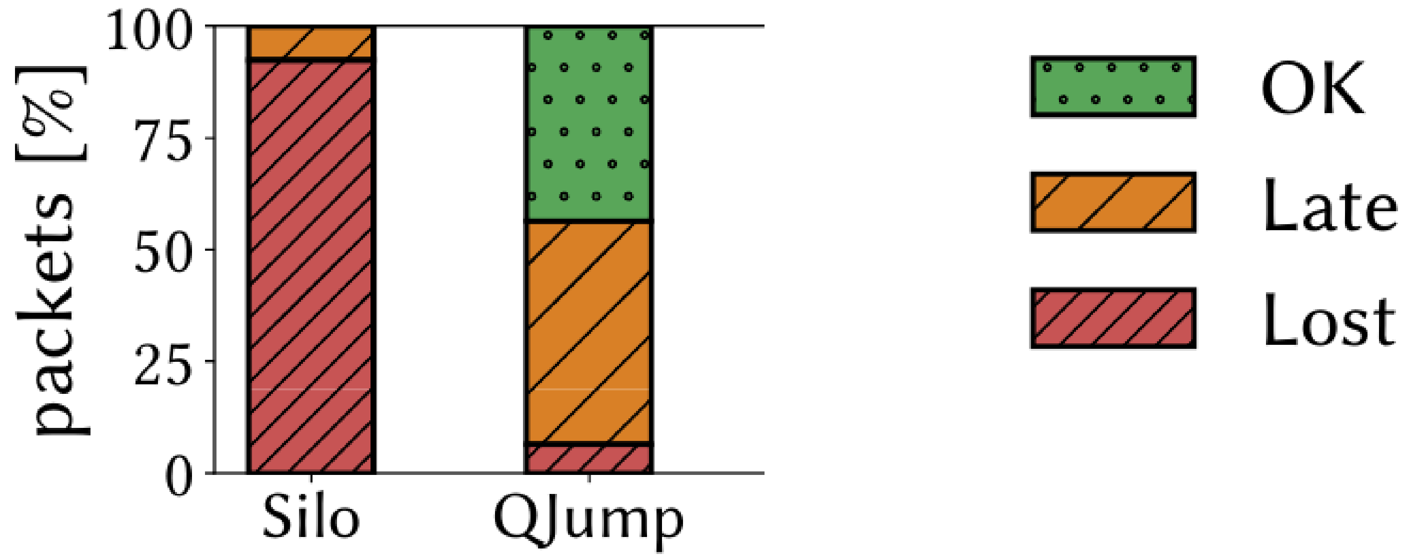
Remember! SoA was **failing**!

Loko: Proof-of-Concept Implementation and Evaluation

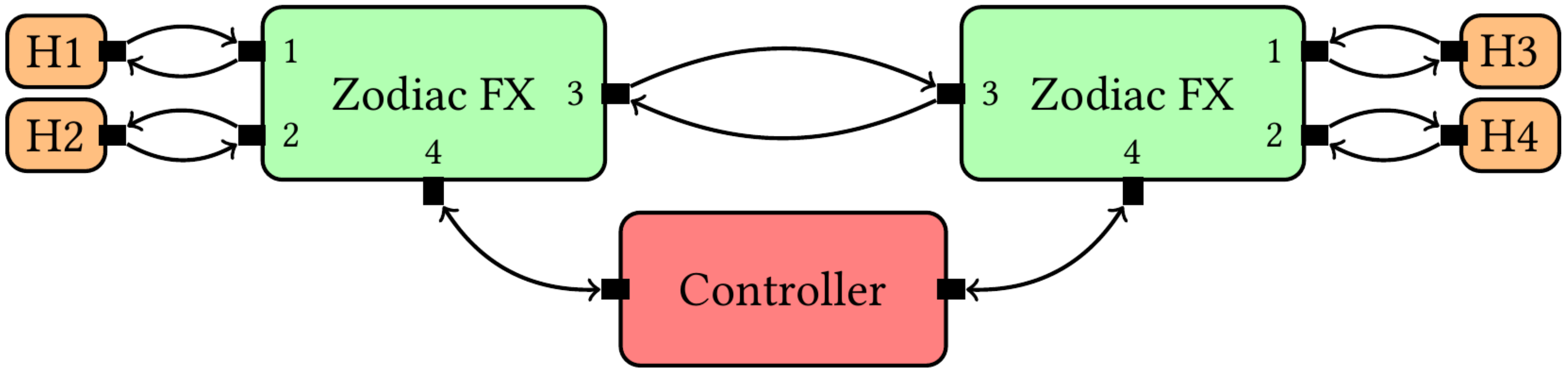


We add flows and observe delays/losses between **H1–H3**

Remember! SoA was **failing**!

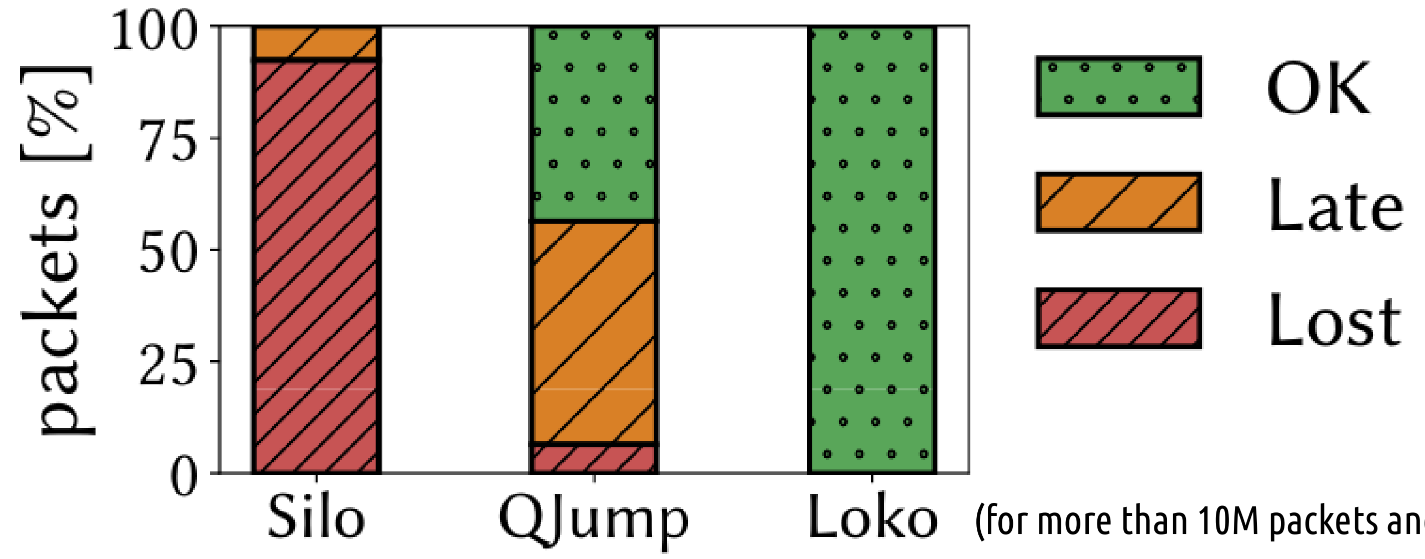


Loko: Proof-of-Concept Implementation and Evaluation



We add flows and observe delays/losses between **H1–H3**

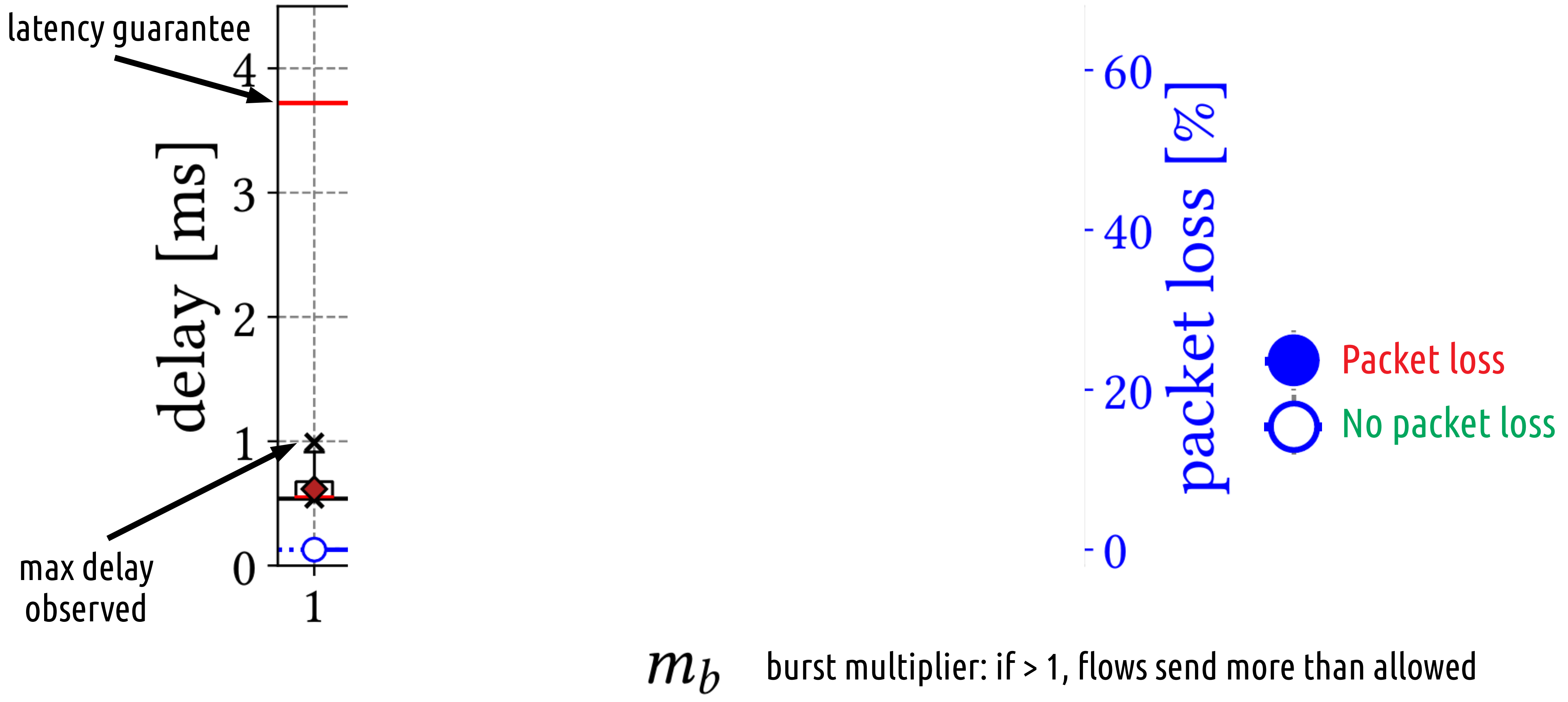
Remember! SoA was **failing**!



(for more than 10M packets and 30 minutes)

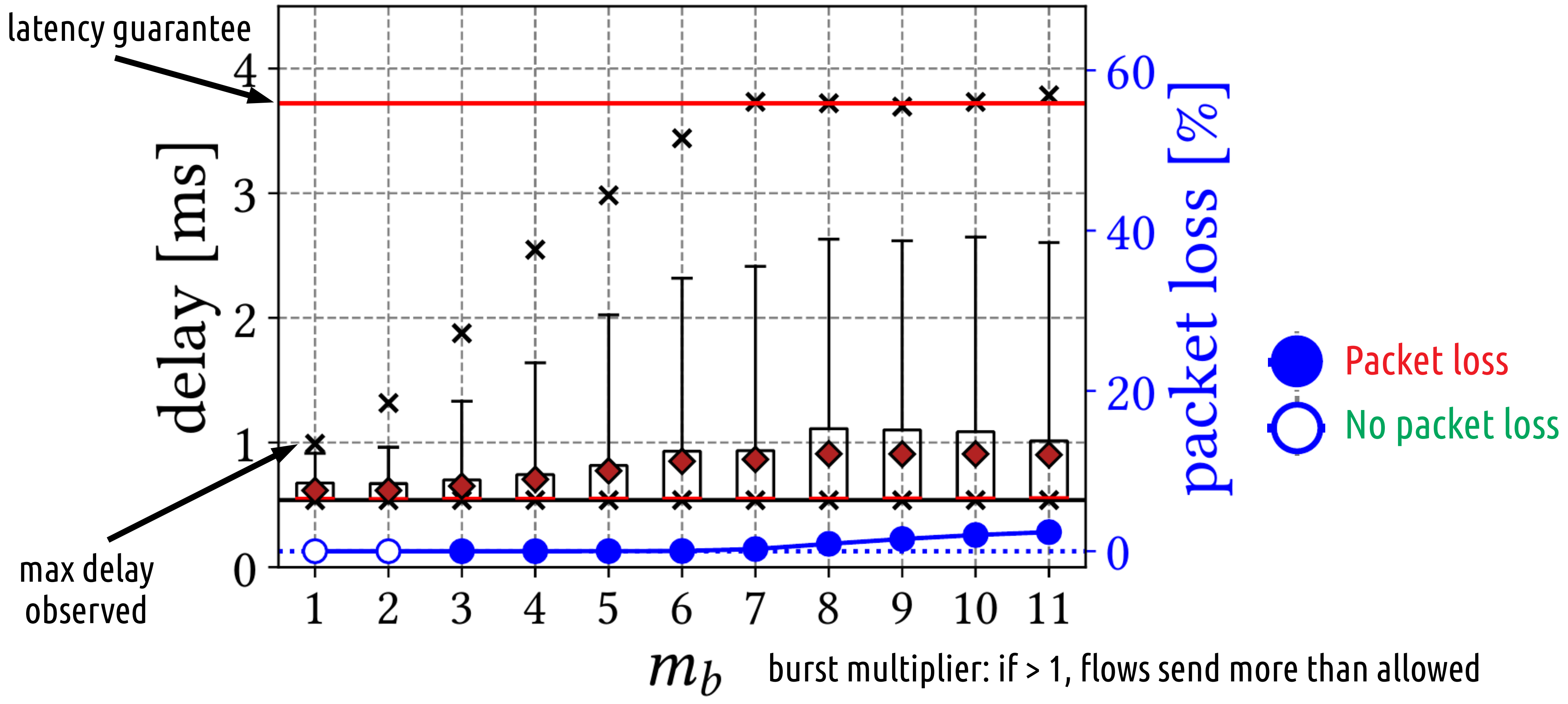
Loko successfully provides latency guarantees!

Loko: Proof-of-Concept Implementation and Evaluation



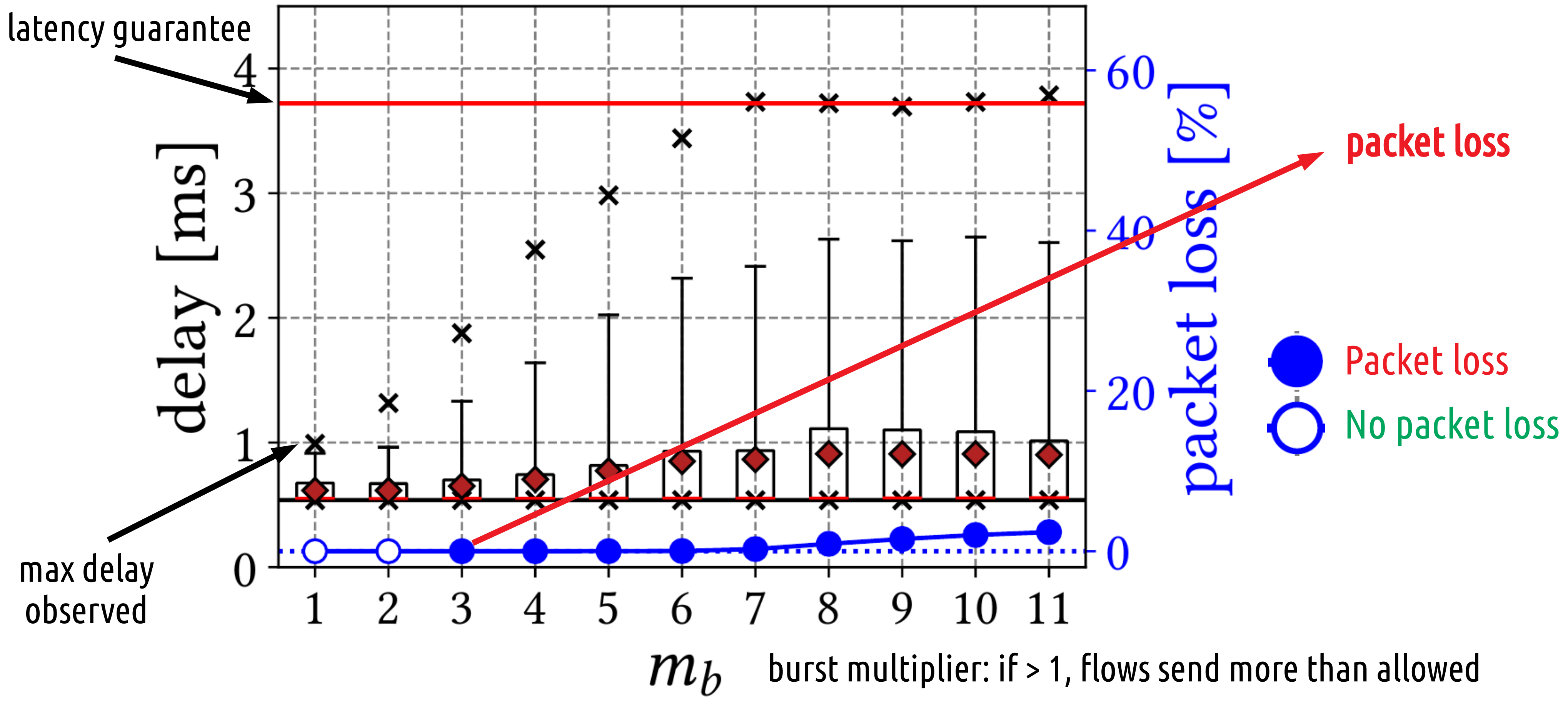
Loko successfully provides latency guarantees!

Loko: Proof-of-Concept Implementation and Evaluation



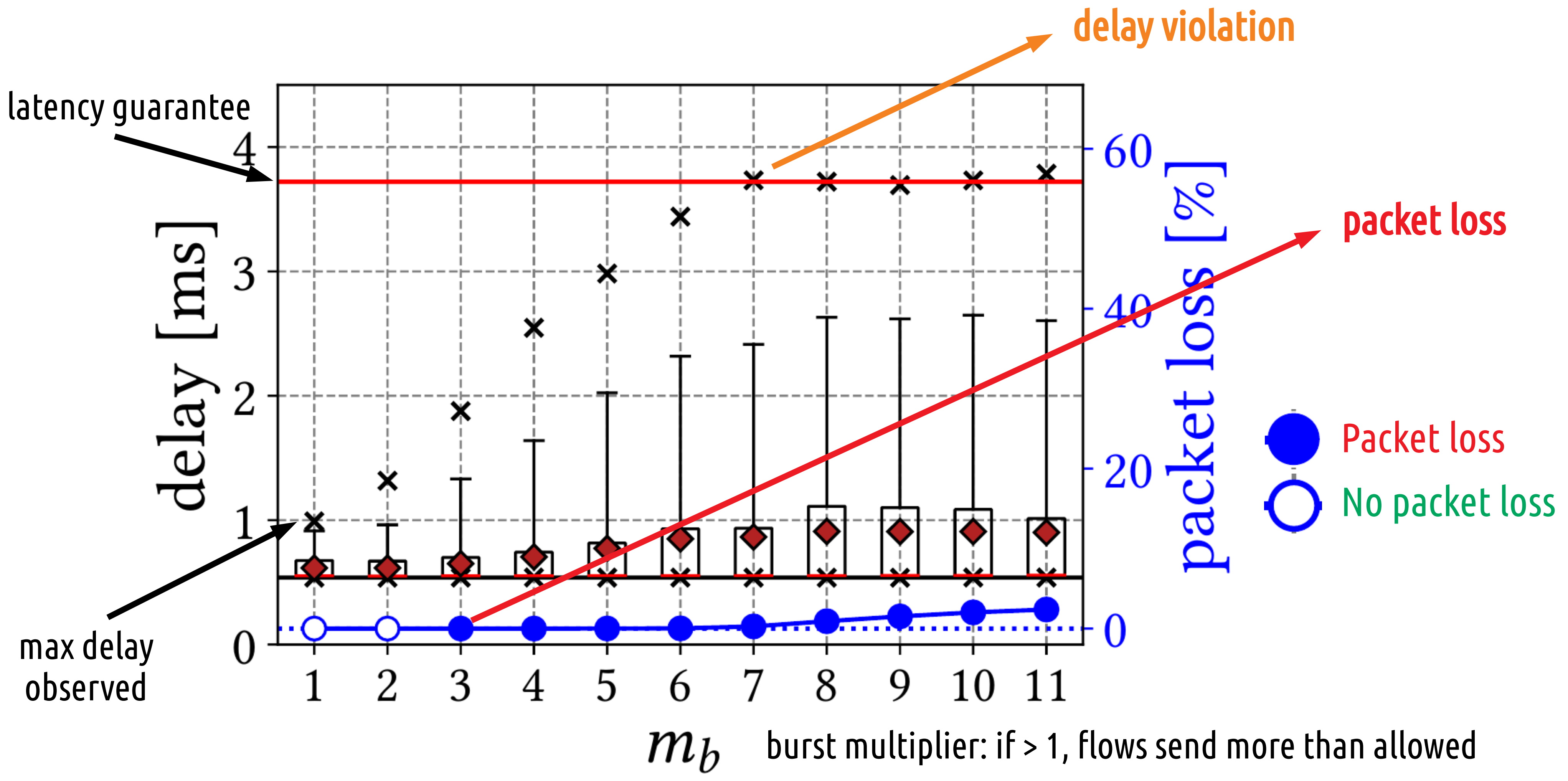
Loko successfully provides latency guarantees!

Loko: Proof-of-Concept Implementation and Evaluation



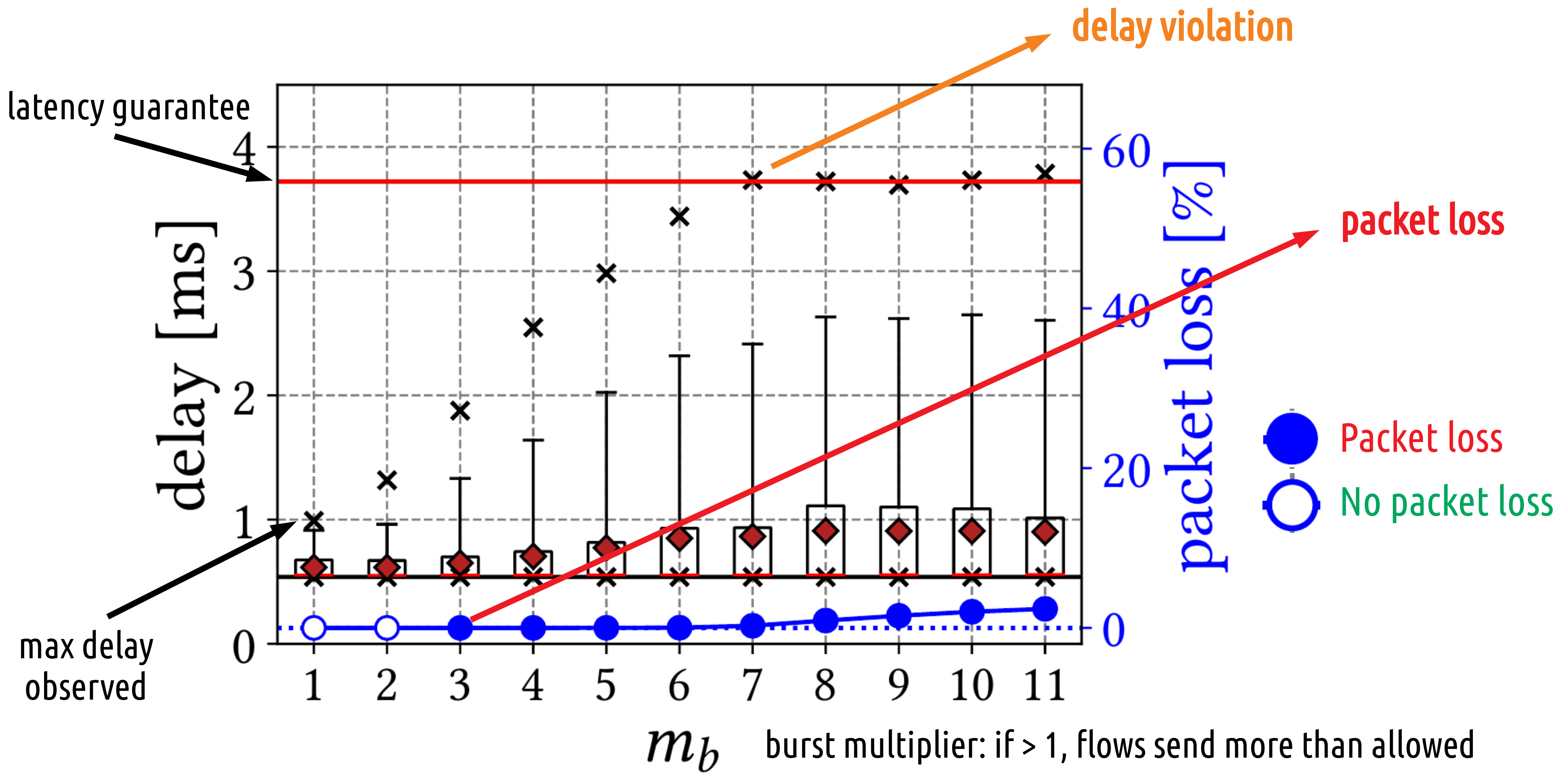
Loko successfully provides latency guarantees!

Loko: Proof-of-Concept Implementation and Evaluation



Loko successfully provides latency guarantees!

Loko: Proof-of-Concept Implementation and Evaluation



More evaluations, including control plane incorporation and scalability analysis in the paper (§6.1, §6.2)

Loko successfully provides latency guarantees!

Loko: Predictable Latency in Small Networks

What else can we say?

Loko: Predictable Latency in Small Networks

What else can we say?

Low-cost software implementations can be predictable and performant
provided there is no OS interference

Loko: Predictable Latency in Small Networks

What else can we say?

Low-cost software implementations can be predictable and performant

provided there is no OS interference

for small networks, but also maybe…

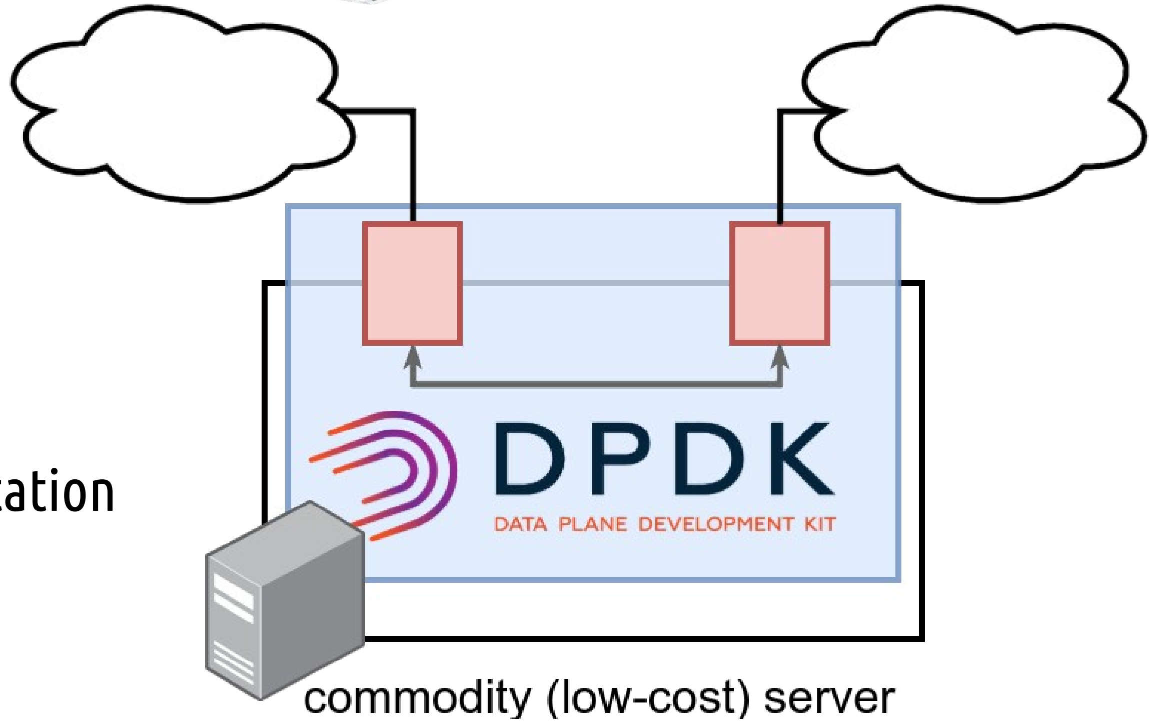
Loko: Predictable Latency in Small Networks

What else can we say?

Low-cost software implementations can be predictable and performant

provided there is no OS interference

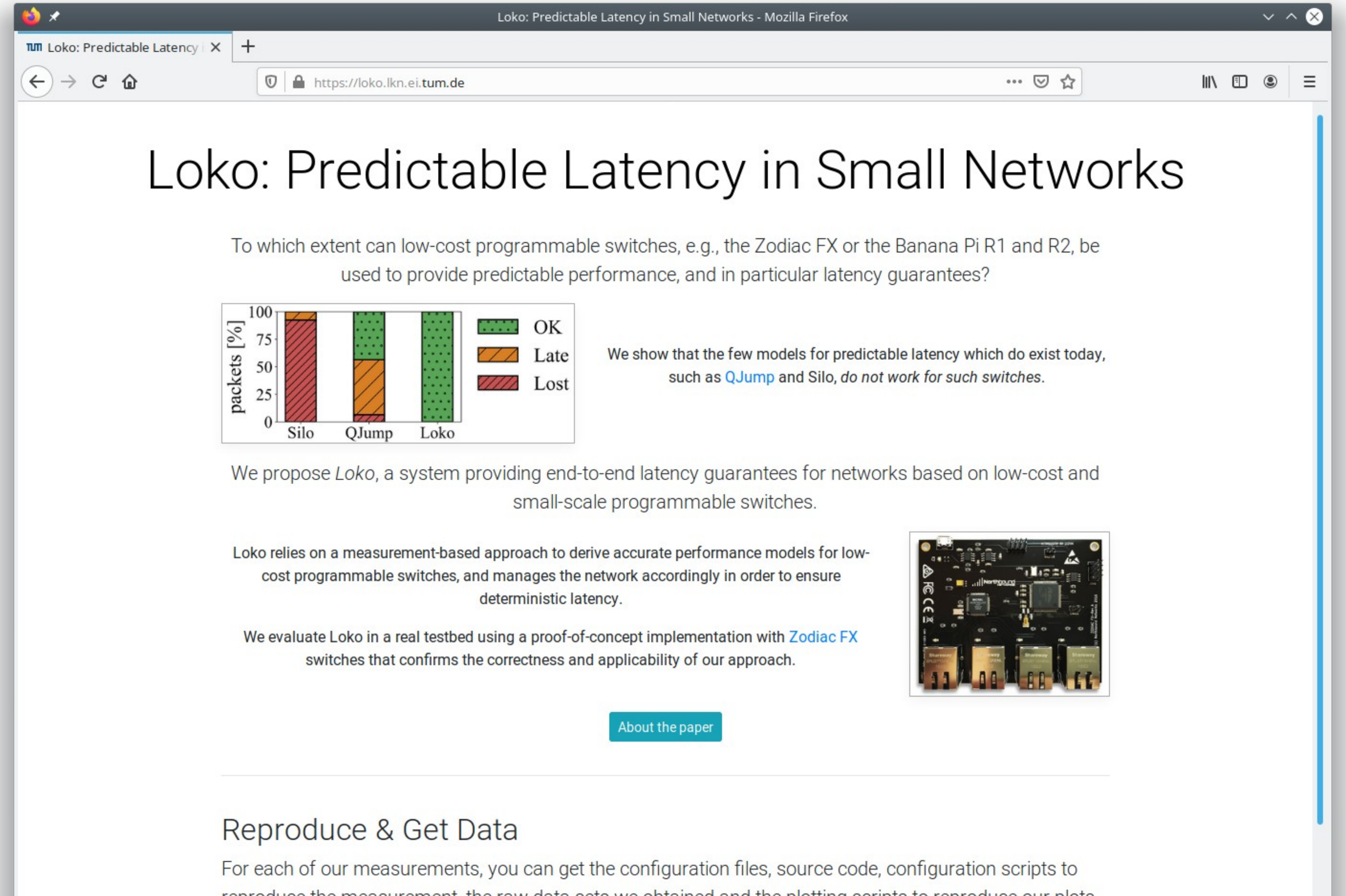
for small networks, but also maybe…



Loko-like approach for
proving the predictability of
software network functions implementation

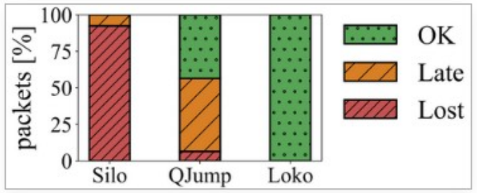
Thanks!

Data sets, traces, source code and configuration files available at
https://loko.lkn.ei.tum.de


Loko: Predictable Latency in Small Networks - Mozilla Firefox
Loko: Predictable Latency
https://loko.lkn.ei.tum.de

Loko: Predictable Latency in Small Networks

To which extent can low-cost programmable switches, e.g., the Zodiac FX or the Banana Pi R1 and R2, be used to provide predictable performance, and in particular latency guarantees?

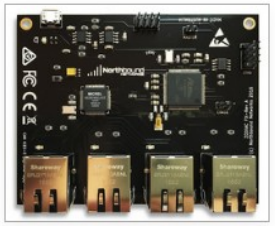


We show that the few models for predictable latency which do exist today, such as QJump and Silo, *do not work for such switches*.

We propose *Loko*, a system providing end-to-end latency guarantees for networks based on low-cost and small-scale programmable switches.

Loko relies on a measurement-based approach to derive accurate performance models for low-cost programmable switches, and manages the network accordingly in order to ensure deterministic latency.

We evaluate Loko in a real testbed using a proof-of-concept implementation with Zodiac FX switches that confirms the correctness and applicability of our approach.



About the paper

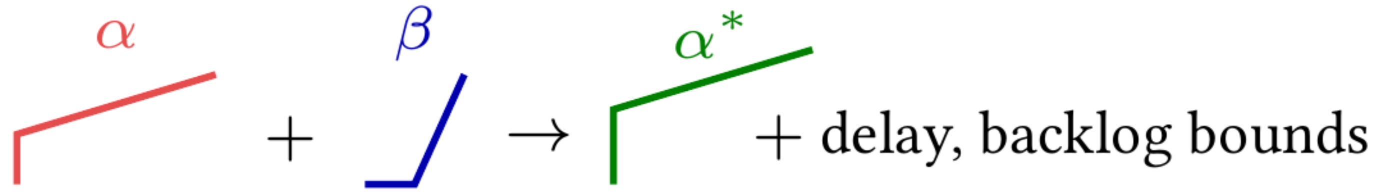
Reproduce & Get Data

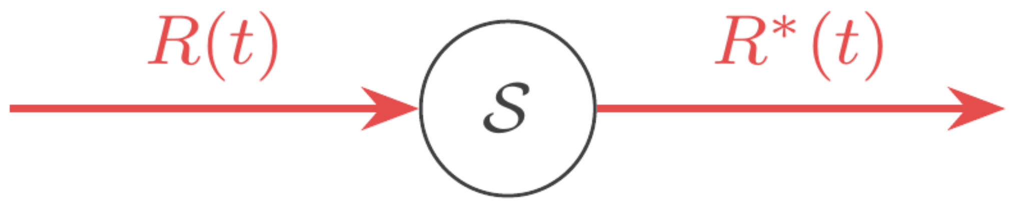
For each of our measurements, you can get the configuration files, source code, configuration scripts to

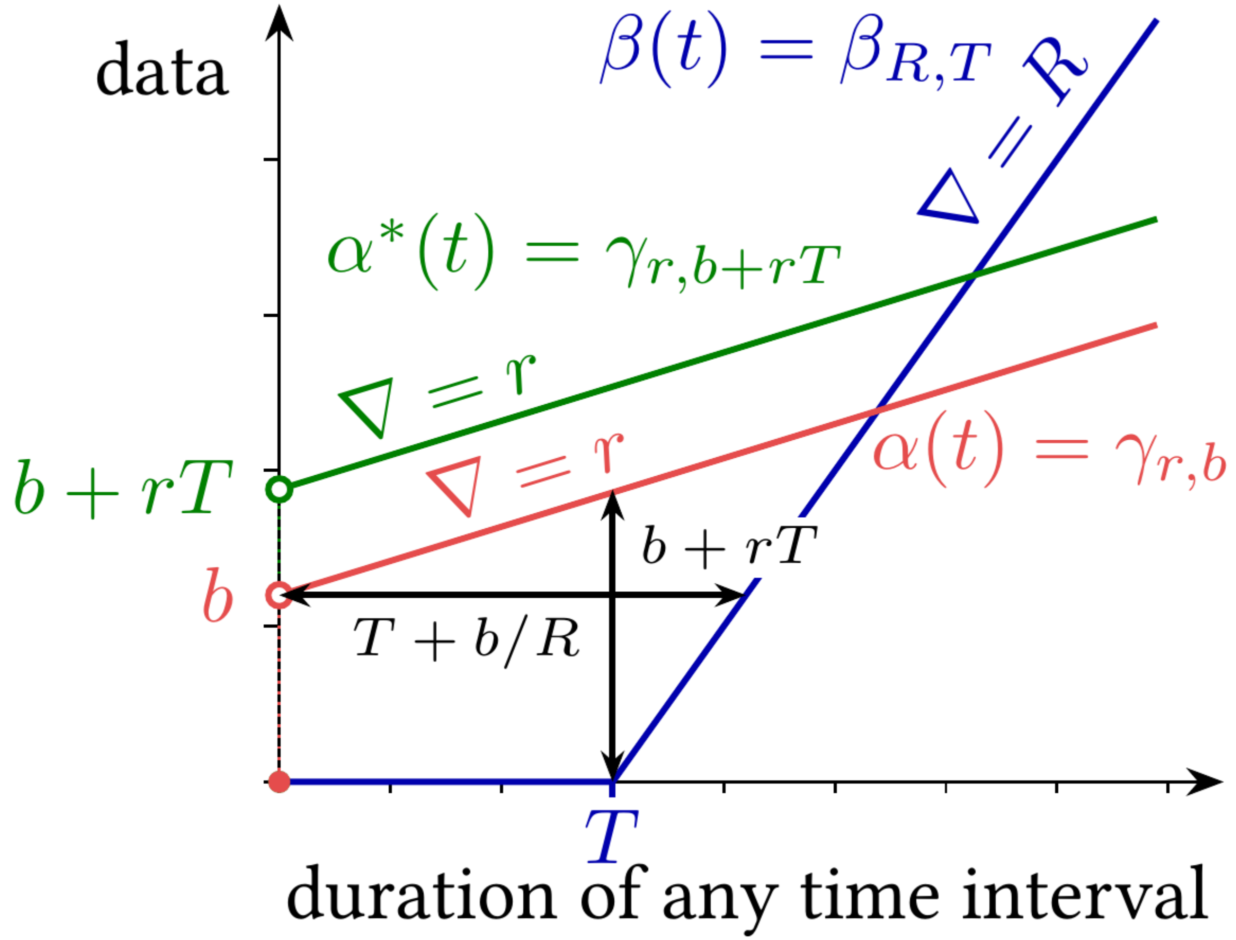
References

- **[NSDI15]** M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N.M. Watson, A. W. Moore, S. Hand, J. Crowcroft, „Queues Don't Matter When You Can JUMP Them!" – USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2015.
- **[SIGCOMM15]** K Jang, J Sherry, H Ballani, T Moncaster, „Silo: predictable message latency in the cloud" – ACM SIGCOMM, 2015.

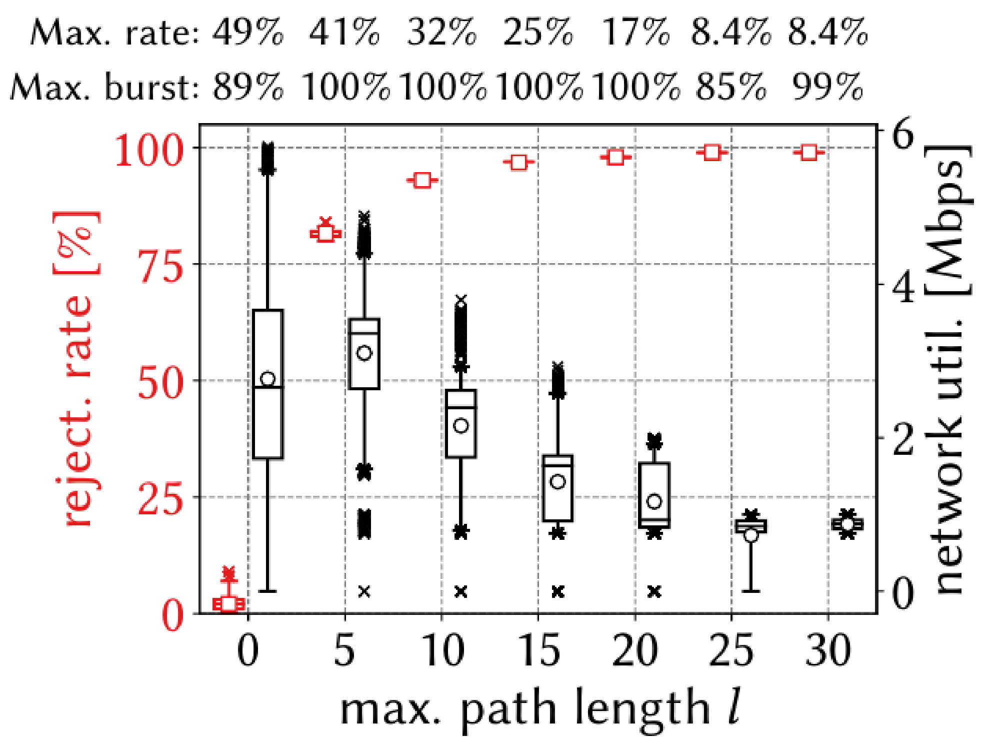
Network Calculus



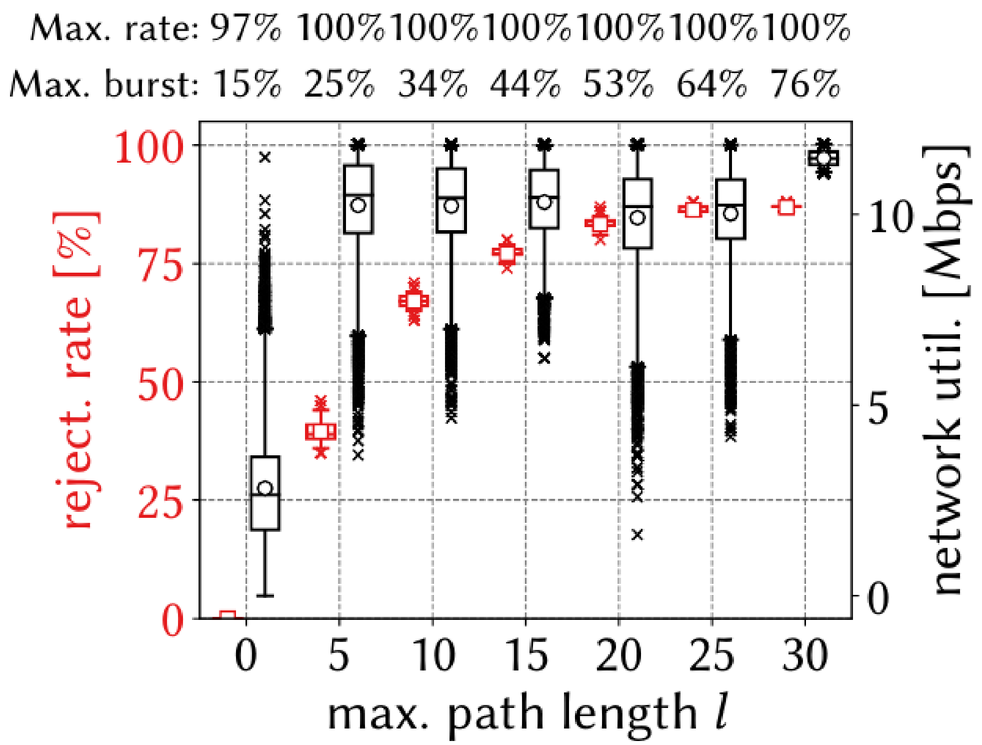




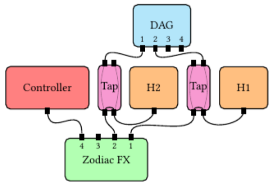
Scalability Analysis



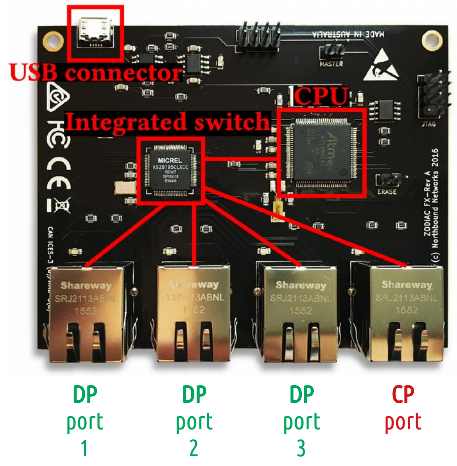
(a) Medium-sized flows.



(b) Artificially inc. buffer size.

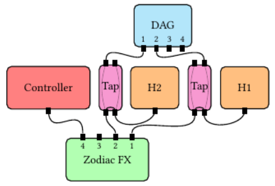


Step 1: Benchmarking of the service

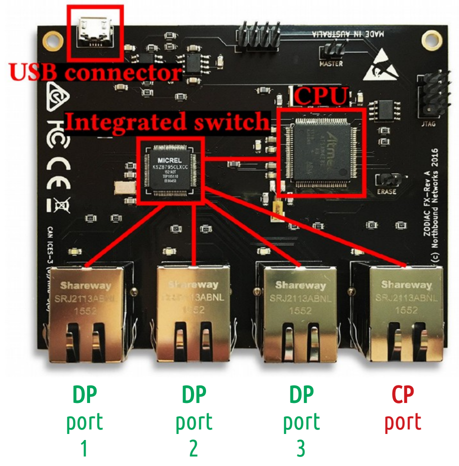


```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

For predictability, we have to **identify ANY source of delay**

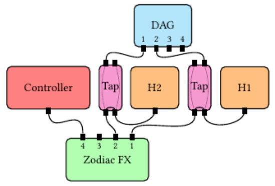


Step 1: Benchmarking of the service

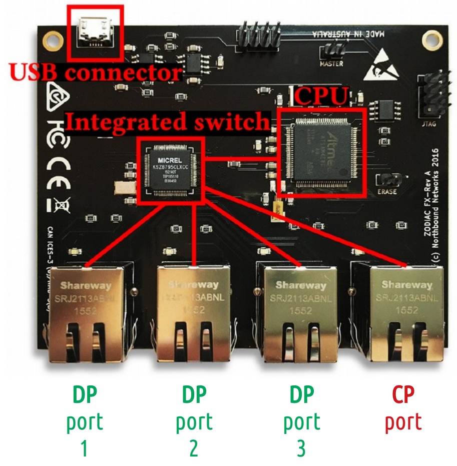


```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

For predictability, we have to **identify ANY source of delay**

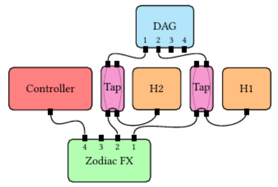


Step 1: Benchmarking of the service

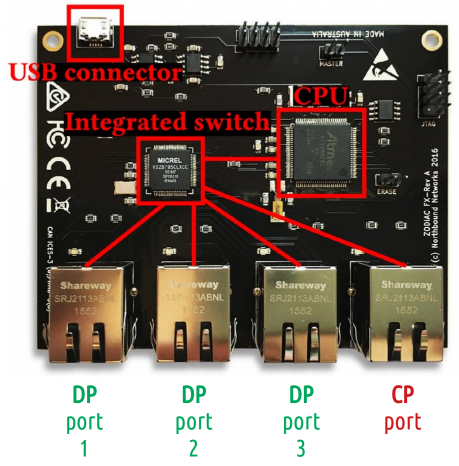


```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

For predictability, we have to **identify ANY source of delay**

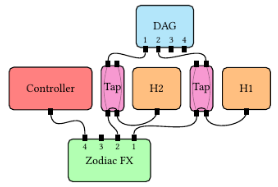


Step 1: Benchmarking of the service

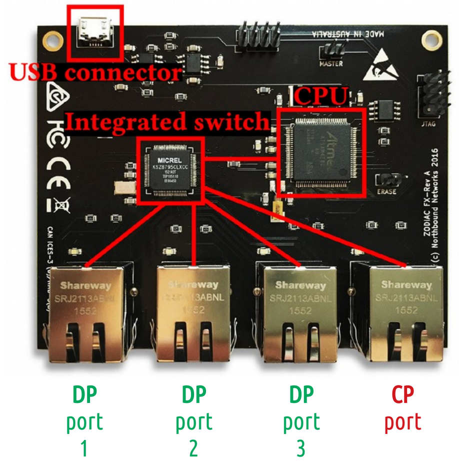


```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

For predictability, we have to **identify ANY source of delay**

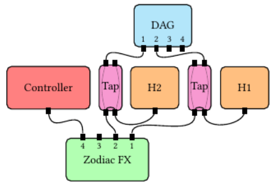


Step 1: Benchmarking of the service

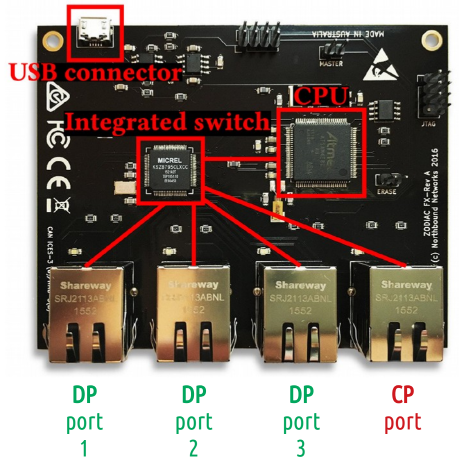


```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

For predictability, we have to **identify ANY source of delay**



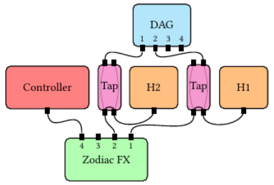
Step 1: Benchmarking of the service



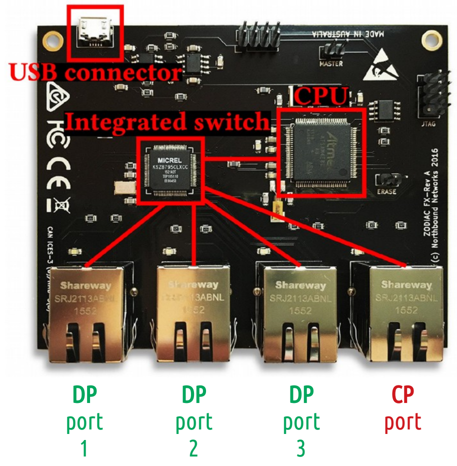
```
1:  while true do
2:      PROCESSFRAME()
3:      PROCESSCLI()
4:      PROTOCOLTIMERS()
5:      CHECKOFCONNECTION()
6:      if +500 ms since last OFCHECKS() then OFCHECKS()
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

Only interference with pure packet processing

For predictability, we have to **identify ANY source of delay**

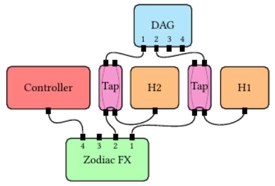


Step 1: Benchmarking of the service

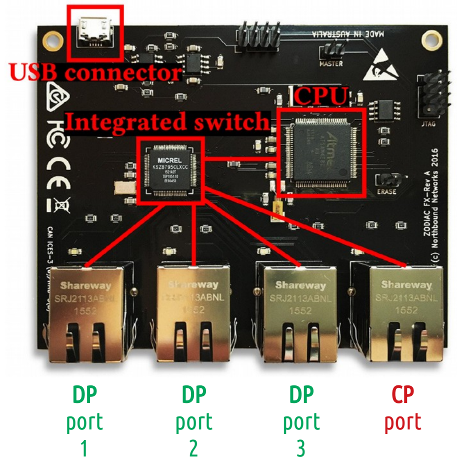


```
 7: function PROCESSFRAME()
 8:     if packet from CP port then
 9:         if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service

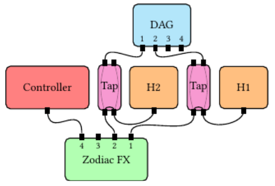


```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

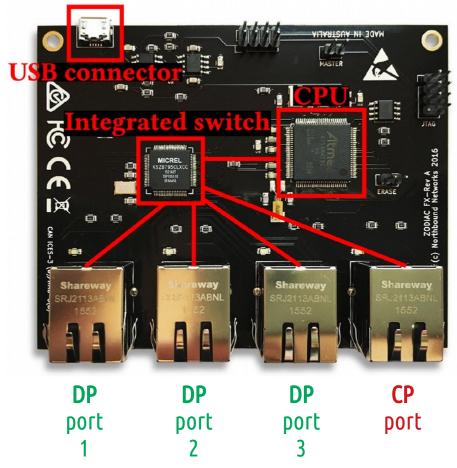
CP: §3.2 in paper

Let's analyze DP processing!

For predictability, we have to **identify ANY source of delay**

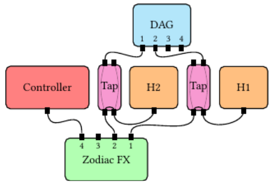


Step 1: Benchmarking of the service

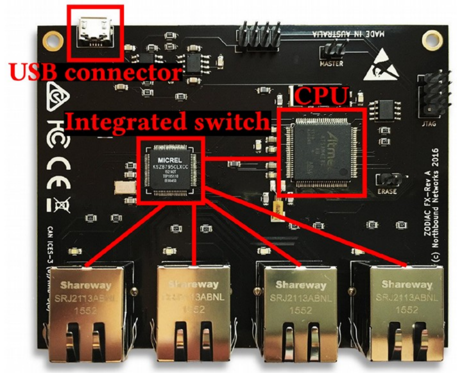


```
 7: function ProcessFrame()
 8:     if packet from CP port then
 9:         if HTTP packet then SendToHttpServer()
10:         if OpenFlow packet then SendToOFAgent()
11:     if packet from DP port then SendToOFPipeline()
```

For predictability, we have to **identify ANY source of delay**



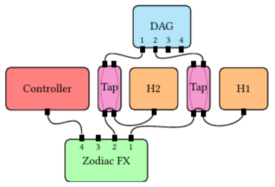
Step 1: Benchmarking of the service



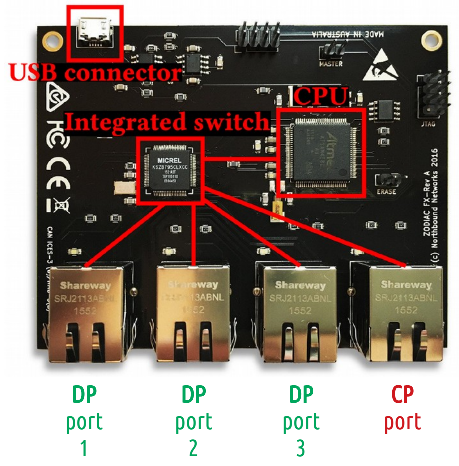
```
7: function ProcessFrame()
8:     if packet from CP port then
9:         if HTTP packet then SendToHttpServer()
10:        if OpenFlow packet then SendToOFAgent()
11:    if packet from DP port then SendToOFPipeline()
```

```
+------+---------------+
|packet|dst_ip=10.2.5.5|
+------+---------------+
```

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



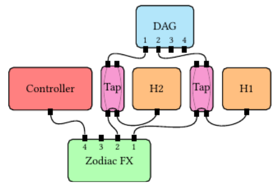
```
function PROCESSFRAME()
    if packet from CP port then
        if HTTP packet then SENDTOHTTPSERVER()
        if OpenFlow packet then SENDTOOFAGENT()
    if packet from DP port then SENDTOOFPIPELINE()
```

```
+------+---------------+
|packet|dst_ip=10.2.5.5|
+------+---------------+
```

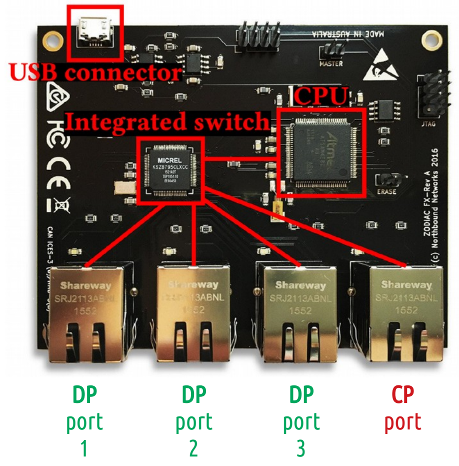
MATCHING TABLE

id	matching	action	priority	counters
0	dst_ip=10.0.X.X	output=1	150	counters
1	dst_ip=10.1.X.X	output=2	150000	counters
2	dst_ip=10.2.X.X	output=3	500	counters
3	dst_ip=10.2.5.5	output=1	200	counters
4	dst_ip=10.3.X.X	output=2	250000	counters
5	dst_ip=10.4.X.X	output=1	250000	counters
6	dst_ip=10.2.5.X	output=2	250000	counters
7	dst_ip=10.2.5.X	output=1	100	counters
8	dst_ip=10.2.5.X	output=3	300	counters
9	dst_ip=10.2.5.X	output=2	500	counters
10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
function PROCESSFRAME()
    if packet from CP port then
        if HTTP packet then SENDTOHTTPSERVER()
        if OpenFlow packet then SENDTOOFAGENT()
    if packet from DP port then SENDTOOFPIPELINE()
```

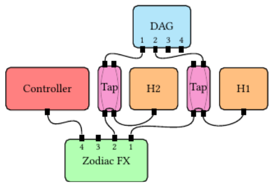
```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

rules one by one
checks only higher priority

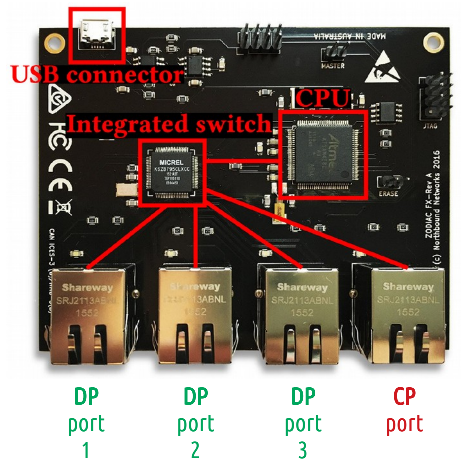
MATCHING TABLE

id	matching	action	priority	counters
0	dst_ip=10.0.X.X	output=1	150	counters
1	dst_ip=10.1.X.X	output=2	150000	counters
2	dst_ip=10.2.X.X	output=3	500	counters
3	dst_ip=10.2.5.5	output=1	200	counters
4	dst_ip=10.3.X.X	output=2	250000	counters
5	dst_ip=10.4.X.X	output=1	250000	counters
6	dst_ip=10.2.5.X	output=2	250000	counters
7	dst_ip=10.2.5.X	output=1	100	counters
8	dst_ip=10.2.5.X	output=3	300	counters
9	dst_ip=10.2.5.X	output=2	500	counters
10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

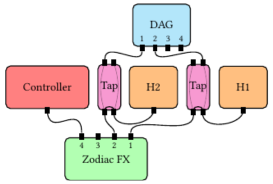
```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

rules one by one
checks only higher priority

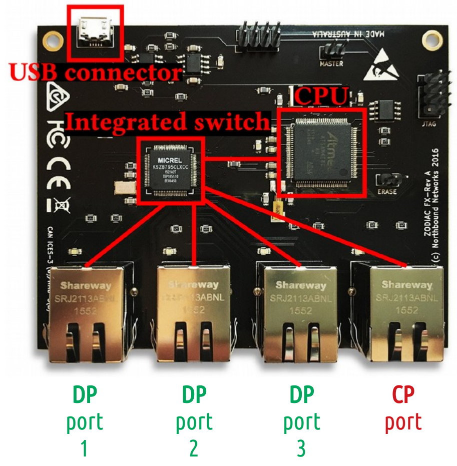
MATCHING TABLE

id	matching	action	priority	counters
0	dst_ip=10.0.X.X	output=1	150	counters
1	dst_ip=10.1.X.X	output=2	150000	counters
2	dst_ip=10.2.X.X	output=3	500	counters
3	dst_ip=10.2.5.5	output=1	200	counters
4	dst_ip=10.3.X.X	output=2	250000	counters
5	dst_ip=10.4.X.X	output=1	250000	counters
6	dst_ip=10.2.5.X	output=2	250000	counters
7	dst_ip=10.2.5.X	output=1	100	counters
8	dst_ip=10.2.5.X	output=3	300	counters
9	dst_ip=10.2.5.X	output=2	500	counters
10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
 7: function ProcessFrame()
 8:     if packet from CP port then
 9:         if HTTP packet then SendToHttpServer()
10:         if OpenFlow packet then SendToOFAgent()
11:     if packet from DP port then SendToOFPipeline()
```

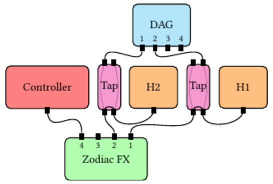
```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

rules one by one
checks only higher priority

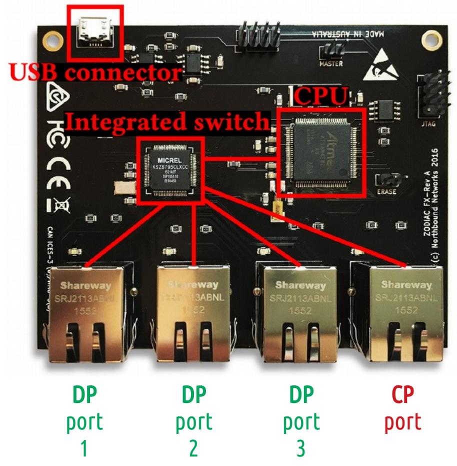
MATCHING TABLE

id	matching	action	priority	counters
0	dst_ip=10.0.X.X	output=1	150	counters
1	dst_ip=10.1.X.X	output=2	150000	counters
2	dst_ip=10.2.X.X	output=3	500	counters
3	dst_ip=10.2.5.5	output=1	200	counters
4	dst_ip=10.3.X.X	output=2	250000	counters
5	dst_ip=10.4.X.X	output=1	250000	counters
6	dst_ip=10.2.5.X	output=2	250000	counters
7	dst_ip=10.2.5.X	output=1	100	counters
8	dst_ip=10.2.5.X	output=3	300	counters
9	dst_ip=10.2.5.X	output=2	500	counters
10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

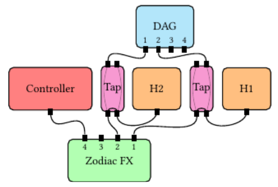
```
+------+---------------+
|packet|dst_ip=10.2.5.5|
+------+---------------+
```

rules one by one
checks only higher priority

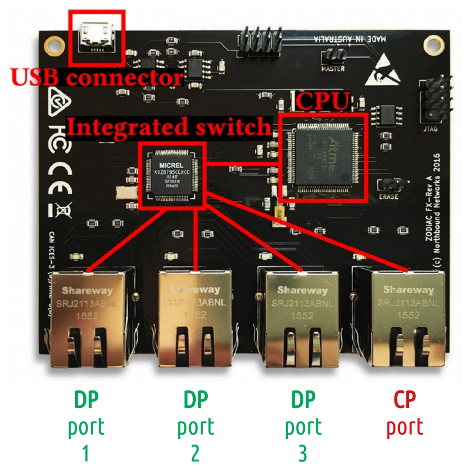
MATCHING TABLE

id	matching	action	priority	counters
✗ 0	dst_ip=10.0.X.X	output=1	150	counters
✗ 1	dst_ip=10.1.X.X	output=2	150000	counters
✓ 2	dst_ip=10.2.X.X	output=3	500	counters
3	dst_ip=10.2.5.5	output=1	200	counters
4	dst_ip=10.3.X.X	output=2	250000	counters
5	dst_ip=10.4.X.X	output=1	250000	counters
6	dst_ip=10.2.5.X	output=2	250000	counters
7	dst_ip=10.2.5.X	output=1	100	counters
8	dst_ip=10.2.5.X	output=3	300	counters
9	dst_ip=10.2.5.X	output=2	500	counters
10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



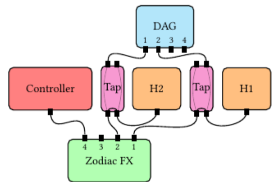
```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

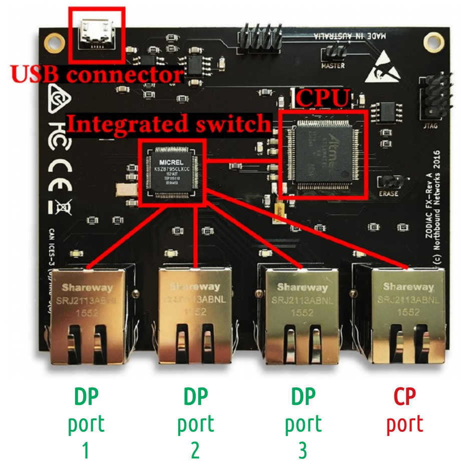
rules one by one
checks only higher priority

```
+------------------------------------------------------+
|                    MATCHING TABLE                    |
+----+-----------------+----------+----------+----------+
| id | matching        | action   | priority | counters |
+----+-----------------+----------+----------+----------|
✗ 0  | dst_ip=10.0.X.X | output=1 | 150      | counters |
✗ 1  | dst_ip=10.1.X.X | output=2 | 150000   | counters |
✓ 2  | dst_ip=10.2.X.X | output=3 | 500      | counters |
– 3  | dst_ip=10.2.5.5 | output=1 | 200      | counters |
| 4  | dst_ip=10.3.X.X | output=2 | 250000   | counters |
| 5  | dst_ip=10.4.X.X | output=1 | 250000   | counters |
| 6  | dst_ip=10.2.5.X | output=2 | 250000   | counters |
| 7  | dst_ip=10.2.5.X | output=1 | 100      | counters |
| 8  | dst_ip=10.2.5.X | output=3 | 300      | counters |
| 9  | dst_ip=10.2.5.X | output=2 | 500      | counters |
| 10 | dst_ip=10.2.X.X | output=1 | 500      | counters |
+----+-----------------+----------+----------+----------+
```

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
 7: function PROCESSFRAME()
 8:     if packet from CP port then
 9:         if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

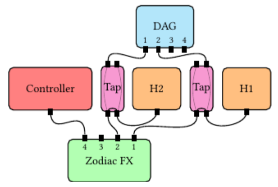
```
+------+---------------+
|packet|dst_ip=10.2.5.5|
+------+---------------+
```

rules one by one
checks only higher priority

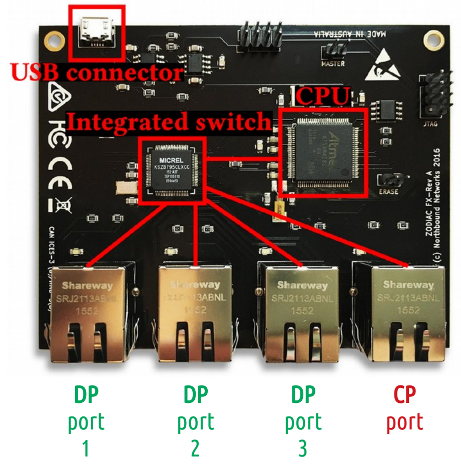
MATCHING TABLE

	id	matching	action	priority	counters
✗	0	dst_ip=10.0.X.X	output=1	150	counters
✗	1	dst_ip=10.1.X.X	output=2	150000	counters
✓	2	dst_ip=10.2.X.X	output=3	500	counters
–	3	dst_ip=10.2.5.5	output=1	200	counters
✗	4	dst_ip=10.3.X.X	output=2	250000	counters
	5	dst_ip=10.4.X.X	output=1	250000	counters
	6	dst_ip=10.2.5.X	output=2	250000	counters
	7	dst_ip=10.2.5.X	output=1	100	counters
	8	dst_ip=10.2.5.X	output=3	300	counters
	9	dst_ip=10.2.5.X	output=2	500	counters
	10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

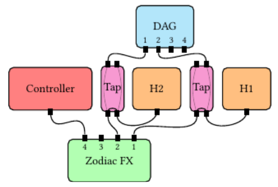
```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

rules one by one
checks only higher priority

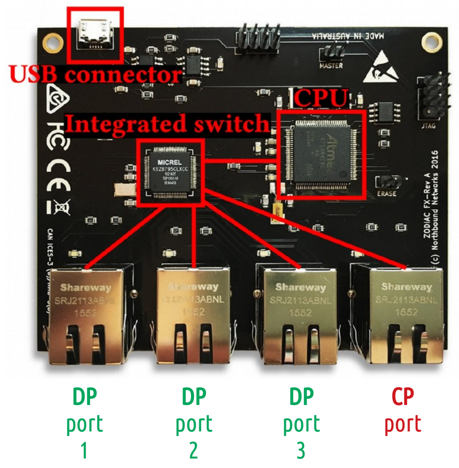
MATCHING TABLE

	id	matching	action	priority	counters
✗	0	dst_ip=10.0.X.X	output=1	150	counters
✗	1	dst_ip=10.1.X.X	output=2	150000	counters
✓	2	dst_ip=10.2.X.X	output=3	500	counters
−	3	dst_ip=10.2.5.5	output=1	200	counters
✗	4	dst_ip=10.3.X.X	output=2	250000	counters
✗	5	dst_ip=10.4.X.X	output=1	250000	counters
	6	dst_ip=10.2.5.X	output=2	250000	counters
	7	dst_ip=10.2.5.X	output=1	100	counters
	8	dst_ip=10.2.5.X	output=3	300	counters
	9	dst_ip=10.2.5.X	output=2	500	counters
	10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

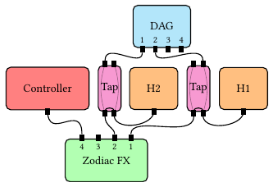
```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

rules one by one
checks only higher priority

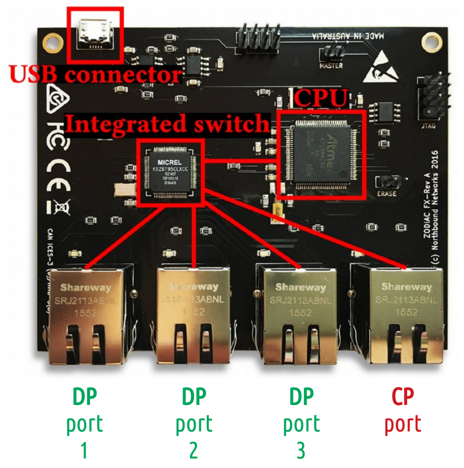
MATCHING TABLE

	id	matching	action	priority	counters
✗	0	dst_ip=10.0.X.X	output=1	150	counters
✗	1	dst_ip=10.1.X.X	output=2	150000	counters
✓	2	dst_ip=10.2.X.X	output=3	500	counters
–	3	dst_ip=10.2.5.5	output=1	200	counters
✗	4	dst_ip=10.3.X.X	output=2	250000	counters
✗	5	dst_ip=10.4.X.X	output=1	250000	counters
✓	6	dst_ip=10.2.5.X	output=2	250000	counters
	7	dst_ip=10.2.5.X	output=1	100	counters
	8	dst_ip=10.2.5.X	output=3	300	counters
	9	dst_ip=10.2.5.X	output=2	500	counters
	10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
 7: function PROCESSFRAME()
 8:     if packet from CP port then
 9:         if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

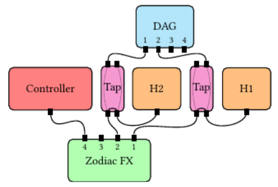
```
+------+---------------+
|packet|dst_ip=10.2.5.5|
+------+---------------+
```

rules one by one
checks only higher priority

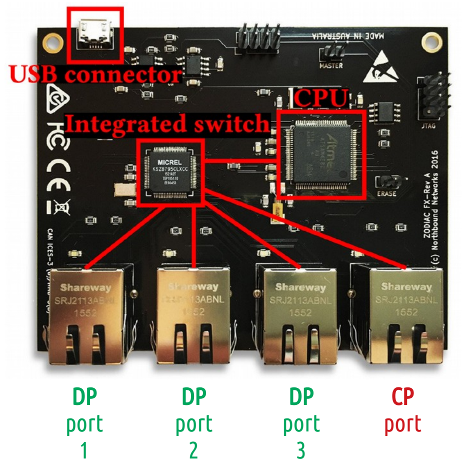
MATCHING TABLE

	id	matching	action	priority	counters
✗	0	dst_ip=10.0.X.X	output=1	150	counters
✗	1	dst_ip=10.1.X.X	output=2	150000	counters
✓	2	dst_ip=10.2.X.X	output=3	500	counters
–	3	dst_ip=10.2.5.5	output=1	200	counters
✗	4	dst_ip=10.3.X.X	output=2	250000	counters
✗	5	dst_ip=10.4.X.X	output=1	250000	counters
✓	6	dst_ip=10.2.5.X	output=2	250000	counters
–	7	dst_ip=10.2.5.X	output=1	100	counters
–	8	dst_ip=10.2.5.X	output=3	300	counters
–	9	dst_ip=10.2.5.X	output=2	500	counters
–	10	dst_ip=10.2.X.X	output=1	500	counters

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

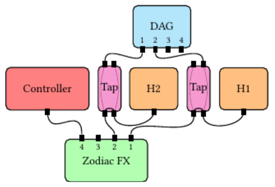
```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

rules one by one
checks only higher priority

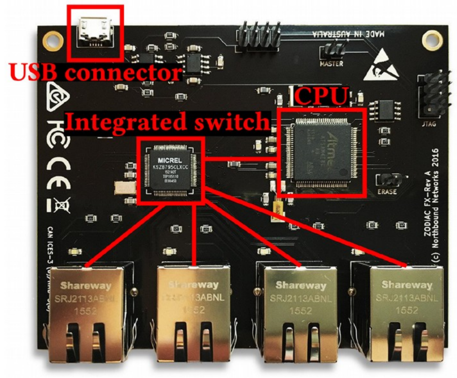
```
  +----------------------------------------------------+
  |                   MATCHING TABLE                   |
  +----+-----------------+----------+----------+----------+
  | id | matching        | action   | priority | counters |
  +----+-----------------+----------+----------+----------+
✗ | 0  | dst_ip=10.0.X.X | output=1 | 150      | counters |
✗ | 1  | dst_ip=10.1.X.X | output=2 | 150000   | counters |
✓ | 2  | dst_ip=10.2.X.X | output=3 | 500      | counters |
– | 3  | dst_ip=10.2.5.5 | output=1 | 200      | counters |
✗ | 4  | dst_ip=10.3.X.X | output=2 | 250000   | counters |
✗ | 5  | dst_ip=10.4.X.X | output=1 | 250000   | counters |
✓ | 6  | dst_ip=10.2.5.X | output=2 | 250000   | counters |
– | 7  | dst_ip=10.2.5.X | output=1 | 100      | counters |
– | 8  | dst_ip=10.2.5.X | output=3 | 300      | counters |
– | 9  | dst_ip=10.2.5.X | output=2 | 500      | counters |
– | 10 | dst_ip=10.2.X.X | output=1 | 500      | counters |
  +----+-----------------+----------+----------+----------+
```

Dimension

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

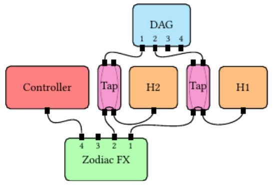
rules one by one
checks only higher priority

MATCHING TABLE

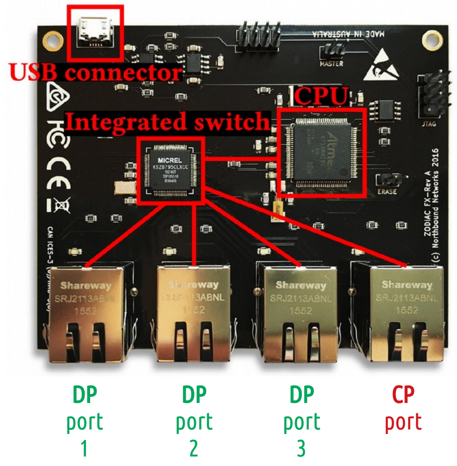
	id	matching	action	priority	counters
✗	0	dst_ip=10.0.X.X	output=1	150	counters
✗	1	dst_ip=10.1.X.X	output=2	150000	counters
✓	2	dst_ip=10.2.X.X	output=3	500	counters
–	3	dst_ip=10.2.5.5	output=1	200	counters
✗	4	dst_ip=10.3.X.X	output=2	250000	counters
✗	5	dst_ip=10.4.X.X	output=1	250000	counters
✓	6	dst_ip=10.2.5.X	output=2	250000	counters
–	7	dst_ip=10.2.5.X	output=1	100	counters
–	8	dst_ip=10.2.5.X	output=3	300	counters
–	9	dst_ip=10.2.5.X	output=2	500	counters
–	10	dst_ip=10.2.X.X	output=1	500	counters

Dimension
nb. of entries

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

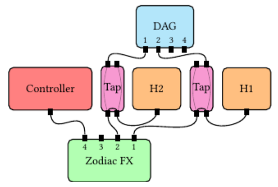
rules one by one
checks only higher priority

MATCHING TABLE

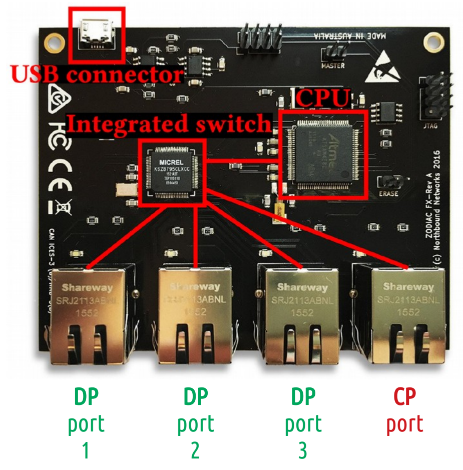
	id	matching	action	priority	counters
✗	0	dst_ip=10.0.X.X	output=1	150	counters
✗	1	dst_ip=10.1.X.X	output=2	150000	counters
✓	2	dst_ip=10.2.X.X	output=3	500	counters
–	3	dst_ip=10.2.5.5	output=1	200	counters
✗	4	dst_ip=10.3.X.X	output=2	250000	counters
✗	5	dst_ip=10.4.X.X	output=1	250000	counters
✓	6	dst_ip=10.2.5.X	output=2	250000	counters
–	7	dst_ip=10.2.5.X	output=1	100	counters
–	8	dst_ip=10.2.5.X	output=3	300	counters
–	9	dst_ip=10.2.5.X	output=2	500	counters
–	10	dst_ip=10.2.X.X	output=1	500	counters

Dimension
nb. of entries
match type

For predictability, we have to identify ANY source of delay



Step 1: Benchmarking of the service



```
 7: function PROCESSFRAME()
 8:     if packet from CP port then
 9:         if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

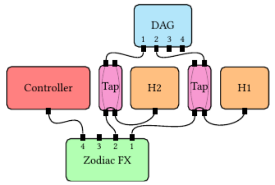
```
+------+---------------+
|packet|dst_ip=10.2.5.5|
+------+---------------+
```

rules one by one
checks only higher priority

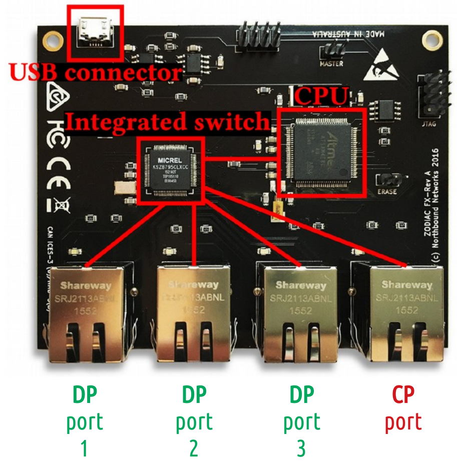
```
+-------------------------------------------------------------+
|                      MATCHING TABLE                         |
+----+------------------+-----------+-----------+-----------+
| id | matching         | action    | priority  | counters  |
+----+------------------+-----------+-----------+-----------+
✗  0 | dst_ip=10.0.X.X  | output=1  | 150       | counters  |
✗  1 | dst_ip=10.1.X.X  | output=2  | 150000    | counters  |
✓  2 | dst_ip=10.2.X.X  | output=3  | 500       | counters  |
–  3 | dst_ip=10.2.5.5  | output=1  | 200       | counters  |
✗  4 | dst_ip=10.3.X.X  | output=2  | 250000    | counters  |
✗  5 | dst_ip=10.4.X.X  | output=1  | 250000    | counters  |
✓  6 | dst_ip=10.2.5.X  | output=2  | 250000    | counters  |
–  7 | dst_ip=10.2.5.X  | output=1  | 100       | counters  |
–  8 | dst_ip=10.2.5.X  | output=3  | 300       | counters  |
–  9 | dst_ip=10.2.5.X  | output=2  | 500       | counters  |
– 10 | dst_ip=10.2.X.X  | output=1  | 500       | counters  |
+----+------------------+-----------+-----------+-----------+
```

Dimension
nb. of entries
match type
action

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
 7: function PROCESSFRAME()
 8:     if packet from CP port then
 9:         if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

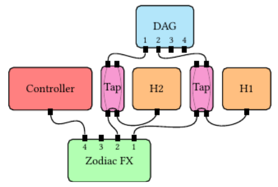
```
+------+---------------+
|packet|dst_ip=10.2.5.5|
+------+---------------+
```

rules one by one
checks only higher priority

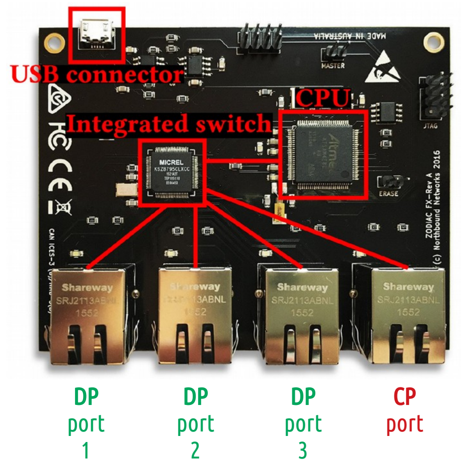
```
+----------------------------------------------------------+
|                     MATCHING TABLE                       |
+----+-----------------+----------+----------+----------+
| id | matching        | action   | priority | counters |
+----+-----------------+----------+----------+----------+
| 0  | dst_ip=10.0.X.X | output=1 | 150      | counters |
| 1  | dst_ip=10.1.X.X | output=2 | 150000   | counters |
| 2  | dst_ip=10.2.X.X | output=3 | 500      | counters |
| 3  | dst_ip=10.2.5.5 | output=1 | 200      | counters |
| 4  | dst_ip=10.3.X.X | output=2 | 250000   | counters |
| 5  | dst_ip=10.4.X.X | output=1 | 250000   | counters |
| 6  | dst_ip=10.2.5.X | output=2 | 250000   | counters |
| 7  | dst_ip=10.2.5.X | output=1 | 100      | counters |
| 8  | dst_ip=10.2.5.X | output=3 | 300      | counters |
| 9  | dst_ip=10.2.5.X | output=2 | 500      | counters |
| 10 | dst_ip=10.2.X.X | output=1 | 500      | counters |
+----+-----------------+----------+----------+----------+
```

Dimension
nb. of entries
match type
action
used entry

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

```
+------+---------------+
|packet|dst_ip=10.2.5.5|
+------+---------------+
```

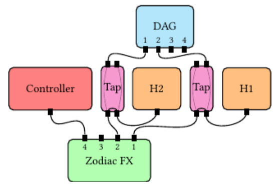
rules one by one
checks only higher priority

MATCHING TABLE

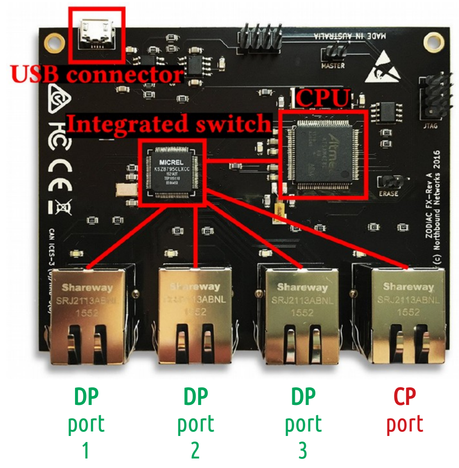
	id	matching	action	priority	counters
✗	0	dst_ip=10.0.X.X	output=1	150	counters
✗	1	dst_ip=10.1.X.X	output=2	150000	counters
✓	2	dst_ip=10.2.X.X	output=3	500	counters
–	3	dst_ip=10.2.5.5	output=1	200	counters
✗	4	dst_ip=10.3.X.X	output=2	250000	counters
✗	5	dst_ip=10.4.X.X	output=1	250000	counters
✓	6	dst_ip=10.2.5.X	output=2	250000	counters
–	7	dst_ip=10.2.5.X	output=1	100	counters
–	8	dst_ip=10.2.5.X	output=3	300	counters
–	9	dst_ip=10.2.5.X	output=2	500	counters
–	10	dst_ip=10.2.X.X	output=1	500	counters

Dimension
nb. of entries
match type
action
used entry
priorities

For predictability, we have to **identify ANY source of delay**



Step 1: Benchmarking of the service



```
7:  function PROCESSFRAME()
8:      if packet from CP port then
9:          if HTTP packet then SENDTOHTTPSERVER()
10:         if OpenFlow packet then SENDTOOFAGENT()
11:     if packet from DP port then SENDTOOFPIPELINE()
```

```
+------+----------------+
|packet|dst_ip=10.2.5.5|
+------+----------------+
```

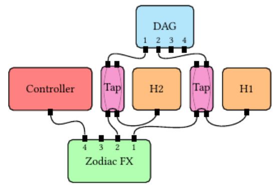
rules one by one
checks only higher priority

MATCHING TABLE

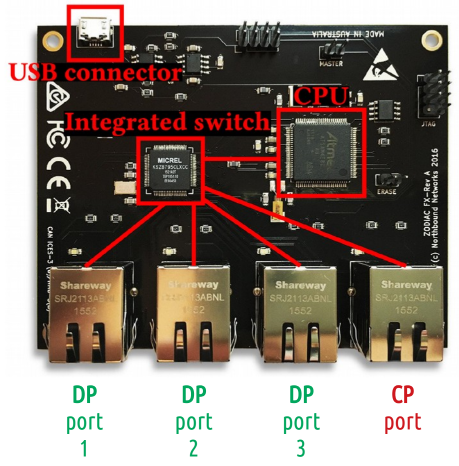
	id	matching	action	priority	counters
✗	0	dst_ip=10.0.X.X	output=1	150	counters
✗	1	dst_ip=10.1.X.X	output=2	150000	counters
✓	2	dst_ip=10.2.X.X	output=3	500	counters
—	3	dst_ip=10.2.5.5	output=1	200	counters
✗	4	dst_ip=10.3.X.X	output=2	250000	counters
✗	5	dst_ip=10.4.X.X	output=1	250000	counters
✓	6	dst_ip=10.2.5.X	output=2	250000	counters
—	7	dst_ip=10.2.5.X	output=1	100	counters
—	8	dst_ip=10.2.5.X	output=3	300	counters
—	9	dst_ip=10.2.5.X	output=2	500	counters
—	10	dst_ip=10.2.X.X	output=1	500	counters

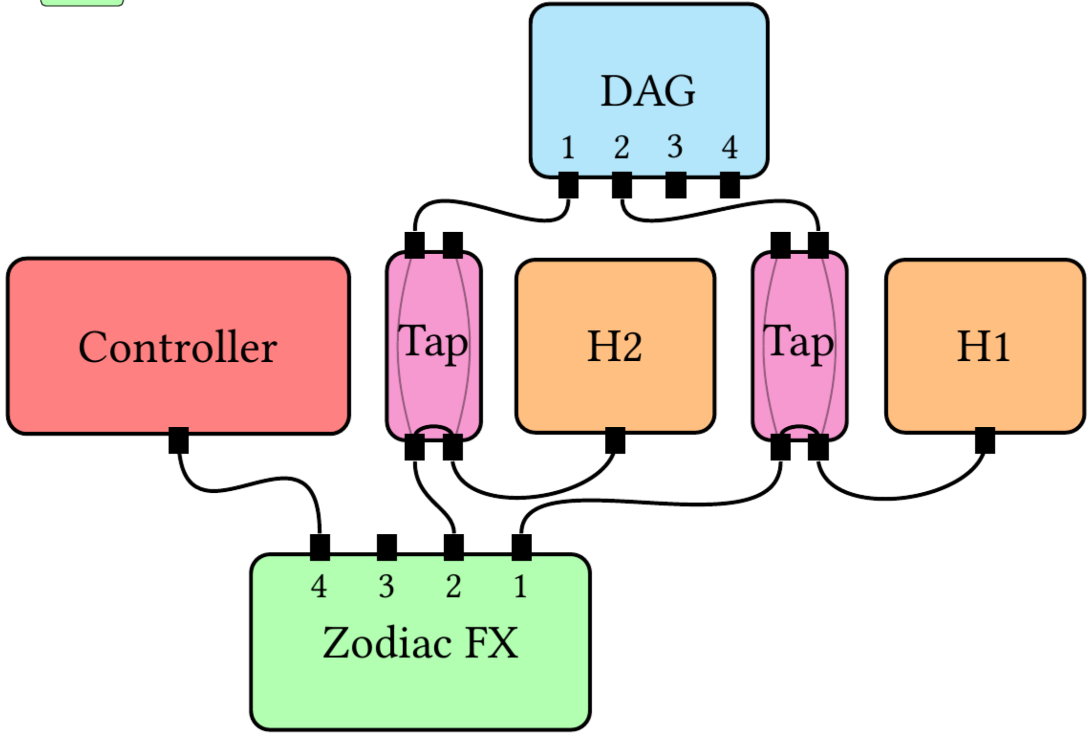
Dimension
nb. of entries
match type
action
used entry
priorities
packet size

For predictability, we have to **identify ANY source of delay**

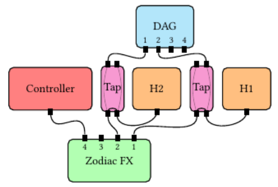


Step 1: Benchmarking of the service

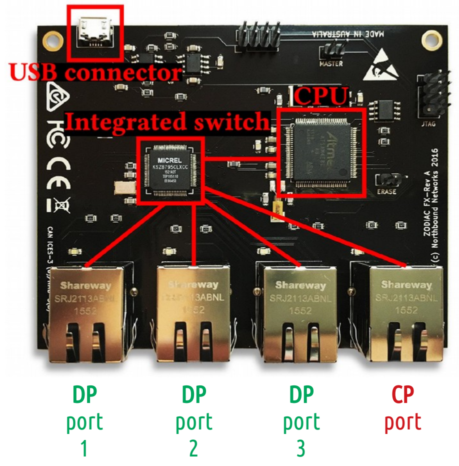


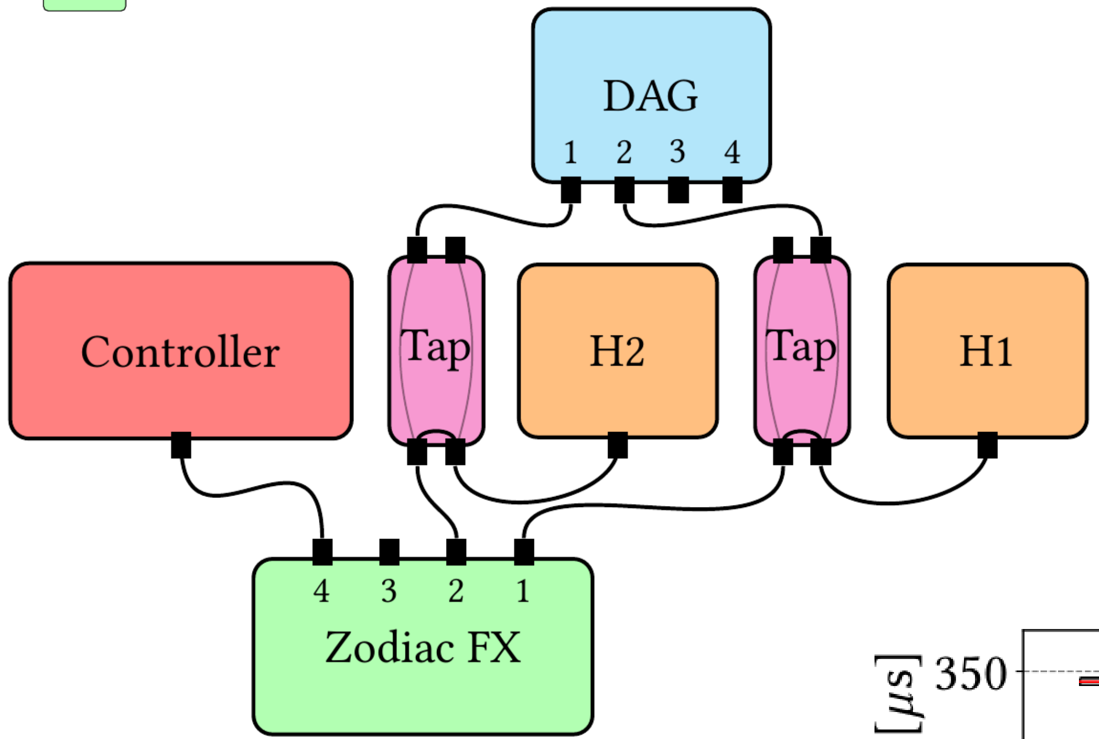


Measure <u>throughput</u> and per-packet <u>delay</u>
for each combination of the dimensions



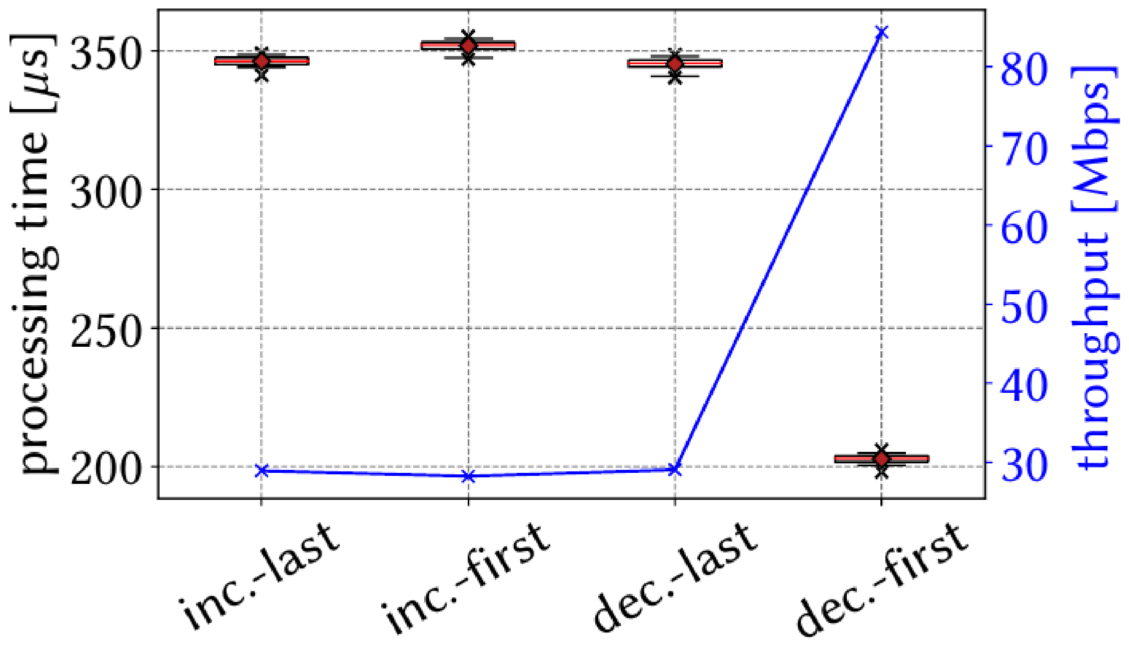
Step 1: Benchmarking of the service



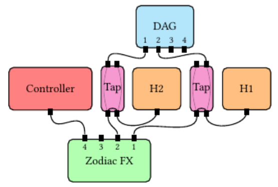


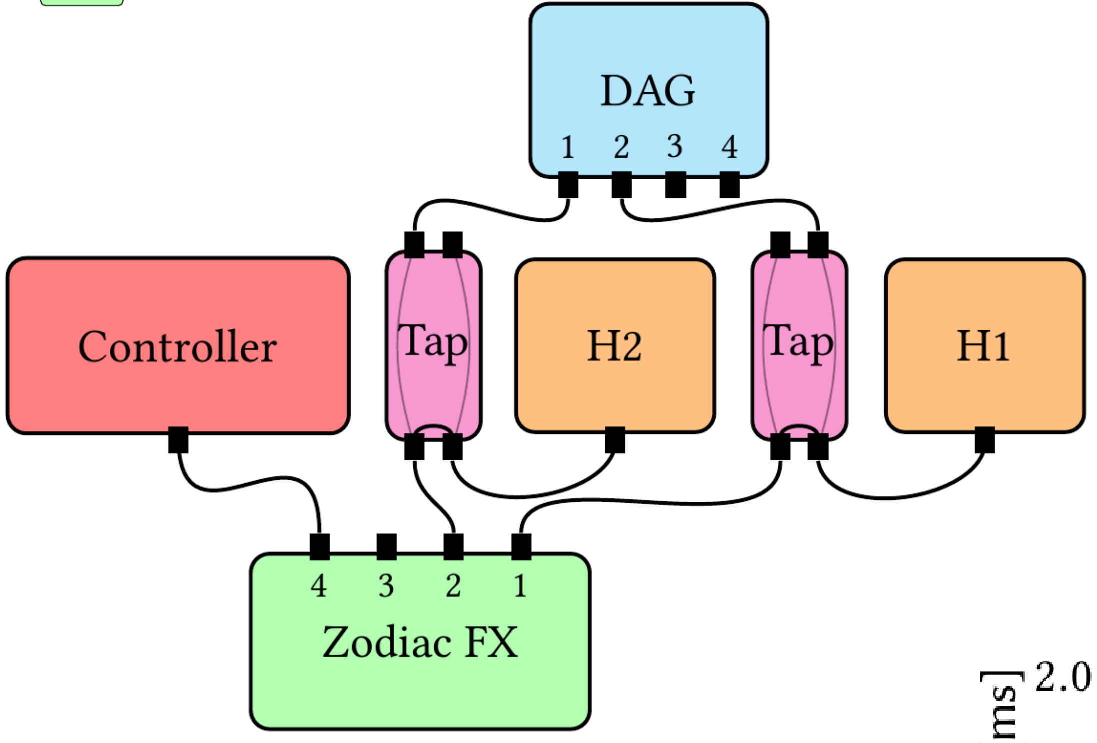
Measure <u>throughput</u> and per-packet <u>delay</u>
for each combination of the dimensions

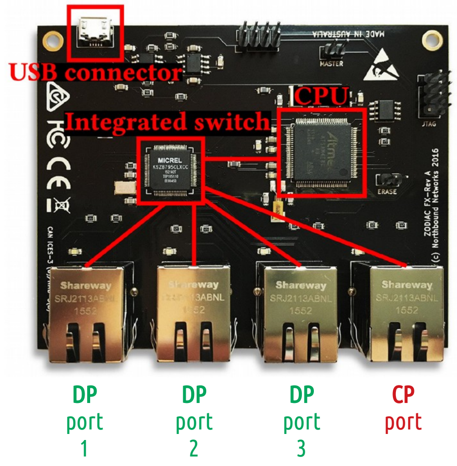
17 entries
five-tuple matching
790-byte packets
output action



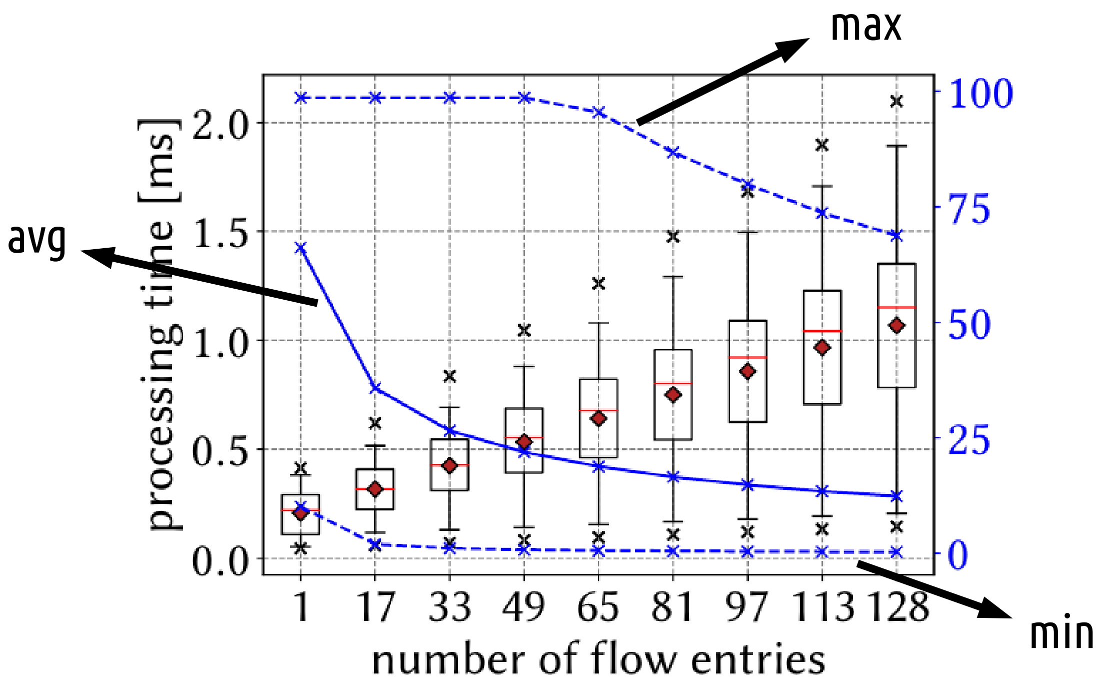
Step 1: Benchmarking of the service

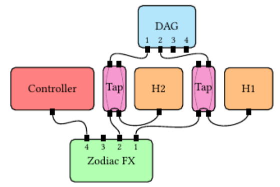




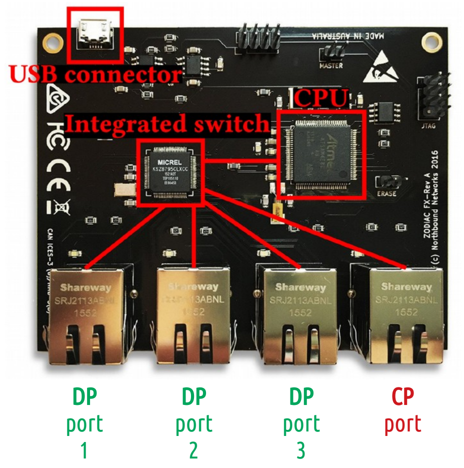


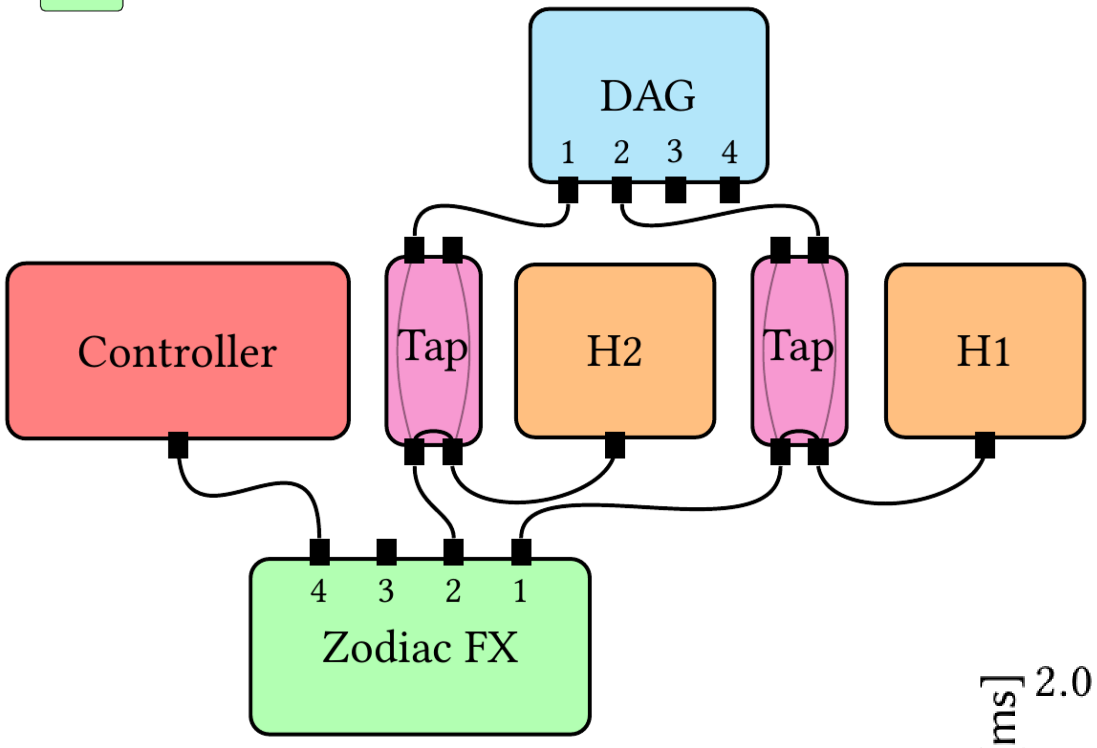
Measure <u>throughput</u> and per-packet <u>delay</u>
for each combination of the dimensions





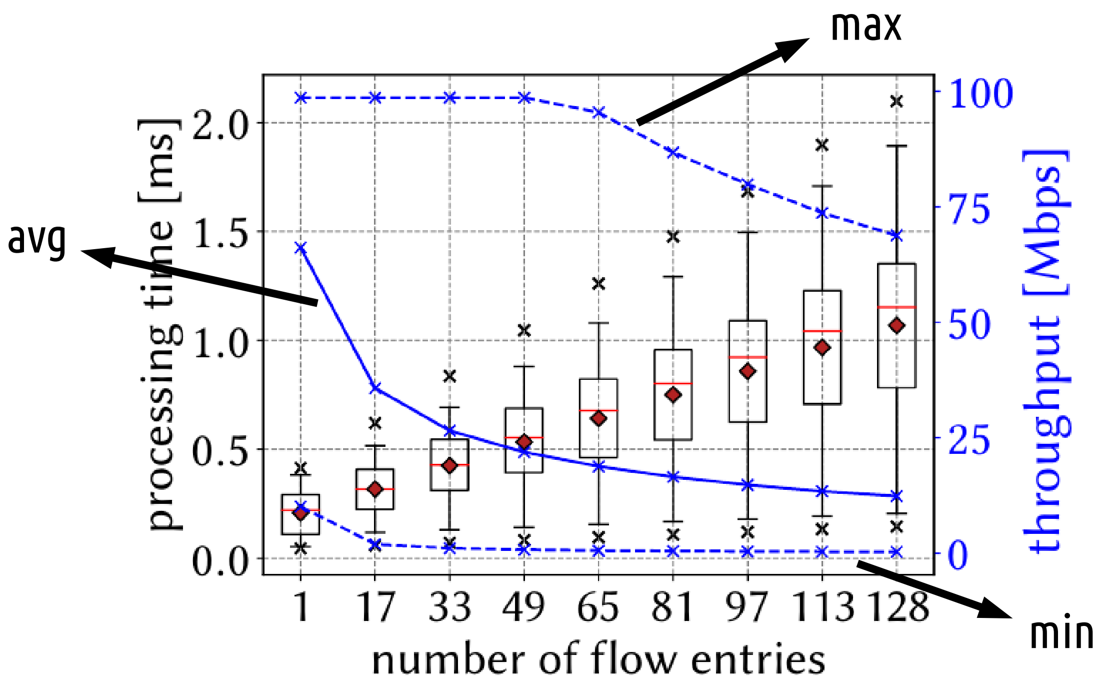
Step 1: Benchmarking of the service

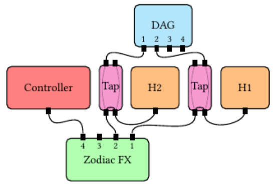




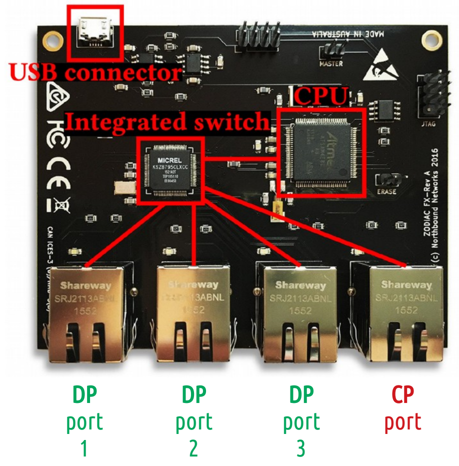
Measure <u>throughput</u> and per-packet <u>delay</u>
for each combination of the dimensions

all cases aggregated





Step 1: Benchmarking of the service



Buffer capacity: §3.5 in paper

Depends only on packet size

from 3 packets (1516 bytes) to 25 packets (64 bytes)

Very scarce resource!