Loko: Predictable Latency in Small Networks

Amaury Van Bemten*, Nemanja Deric*, Johannes Zerwas*, Andreas Blenk*, Stefan Schmid° and Wolfgang Kellerer*
amaur.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
Loko: Predictable Latency in Small Networks
Loko: Predictable Latency in Small Networks
Loko: Predictable Latency in Small Networks

Low-capacity

~kbps, up to few Mbps, predictable traffic patterns

Industrial robot (eHealth 5G Research Hub Munich)
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
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

Industrial robot (eHealth 5G Research Hub Munich)
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

- 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
Loko: Predictable Latency in Small Networks

Industrial robot (eHealth 5G Research Hub Munich)
Loko: Predictable Latency in Small Networks

Hard latency requirements

Per-packet **100% guaranteed** max. latency (~µs, ms)
Loko: Predictable Latency in Small Networks

State-of-the-Art?
Loko: Predictable Latency in Small Networks

State-of-the-Art?
Loko: Predictable Latency in Small Networks

State-of-the-Art?

proprietary or not interoperable: expensive, specialized hardware, vendor lock-in, inflexible
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?
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
- ~$90
- 5x1G
- 83 gr.
- 148 mm × 100mm

Banana Pi R2
- ~$125
- 5x1G
- 100 gr.
- 148 mm × 100.5mm

Zodiac FX
- ~$70
- 4x100M
- 115 gr.
- 100mm × 80mm

Zodiac GX
- ~$120
- 5x1G
- 765 gr.
- 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

---

**predictable latency SOLUTION for progr. networks**

### Hardware

<table>
<thead>
<tr>
<th>Device</th>
<th>Price</th>
<th>Ports</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana Pi R1</td>
<td>~$90</td>
<td>5x1G</td>
<td>148 mm × 100 mm</td>
</tr>
<tr>
<td>Banana Pi R2</td>
<td>~$125</td>
<td>5x1G</td>
<td>148 mm × 100.5 mm</td>
</tr>
<tr>
<td>Zodiac FX</td>
<td>~$70</td>
<td>4x100M</td>
<td>100 mm × 80 mm</td>
</tr>
<tr>
<td>Zodiac GX</td>
<td>~$120</td>
<td>5x1G</td>
<td>232 mm × 142 mm × 45 mm</td>
</tr>
</tbody>
</table>

### Solutions

- **Silo** [SIGCOMM15]
- **QJump** [NSDI15]
Loko: Predictable Latency in Small Networks

Predictable latency SOLUtion for progr. networks

State-of-the-Art?

small programmable HARDWARE

Banana Pi R1
- ~$90
- 5x1G
- 83 gr.
- 148 mm × 100 mm

Banana Pi R2
- ~$125
- 5x1G
- 100 gr.
- 148 mm × 100.5 mm

Zodiac FX
- ~$70
- 4x100M
- 115 gr.
- 100 mm × 80 mm

Zodiac GX
- ~$120
- 5x1G
- 765 gr.
- 232 mm × 142 mm × 45 mm

Silo [SIGCOMM15]

QJump [NSDI15]
Silo [SIGCOMM15]

Silo: Predictable Message Latency in the Cloud

Keen Jang
Intel Labs
Santa Clara, CA

Justine Sherry
UC Berkeley
Berkeley, CA

Hitesh Babani
Microsoft Research
Cambridge, UK

Toby Monpaster
University of Cambridge
Cambridge, UK

ABSTRACT
Message latency in the cloud is highly variable and difficult to predict. Silo provides predictable message latency by isolating messages into isolated silos. Silo is especially useful in serverless computing environments, where services are provisioned dynamically. Silo guarantees a response within a bounded amount of time, making it ideal for services that require fast response times.
**Silo** [SIGCOMM15]

*ABSTRACT*

Many cloud applications can benefit from guaranteed latency for their network messages. However, providing such guarantees is not easy to realize in a distributed environment. We present **Silo**, a protocol that enables predictable message latency in the cloud. **Silo** is designed to provide low-latency guarantees without sacrificing throughput. It uses a combination of latency-sensitive routing and a novel message scheduling algorithm to ensure that messages are delivered within a specified time. This allows applications to rely on **Silo** for critical latency-sensitive operations, thereby improving overall system performance and reliability.
Silo [SIGCOMM15]

Silo: Predictable Message Latency in the Cloud

Keun Jang Intel Labs Santa Clara, CA
Justin Czerney UC Berkeley Berkeley, CA
Hitesh Salbani 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 especially challenging in high-latency networks. We present Silo, a new approach that ensures predictable message latency in the cloud, using novel network management techniques and leveraging elastic network resources. Silo integrates a responsive, latency-guaranteed network stack with an efficient and scalable network control plane to achieve predictable latency guarantees even in high-latency environments. 

lost packets

latency guarantee
Silo [SIGCOMM15]

Silo: Predictable Message Latency in the Cloud

Keon Jang
Intel Labs
Santa Clara, CA

Justin Sherry
UC Berkeley
Berkeley, CA

Hitesh Balani
Microsoft Research
Cambridge, UK

Toby Moncrieff
University of Cambridge
Cambridge, UK

Abstract
Many cloud applications can benefit from guaranteed latency for their network messages, however providing such guarantees is challenging due to network variability. We present Silo, a system that provides guaranteed worst-case latency in the cloud. Silo achieves this by isolating critical traffic and using a combination of hardware and software techniques to ensure predictable latency. Silo is designed to be transparent to application developers and can be deployed in existing cloud environments.
Silo [SIGCOMM15]

ABSTRACT
Many cloud applications can benefit from guaranteed latency for their network messages, however providing such guarantees is challenging, especially in highly-varying link conditions. We introduce Silo, a system that provides low-latency communication guarantees for cloud applications while still supporting commodity hardware.

QJump [NSDI15]

Queues don’t matter when you can JUMP them!

Matthew F. Grovesneer, Malek Zaccaria, Joel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Haas, Jon Crowcroft
University of Cambridge Computer Laboratory
Silo [SIGCOMM15]

Silo: Predictable Message Latency in the Cloud

Keon Jang
Intel Labs
Santa Clara, CA

Justin J. Sherry
UC Berkeley
Berkeley, CA

Hitesh Balani
Microsoft Research
Cambridge, UK

Toby Menzies
University of Cambridge
Cambridge, UK

ABSTRACT

Many cloud applications can benefit from guaranteed latency for their network messages, however providing such predictability is challenging in modern datacenters. We present Silo, a platform that ensures predictable message latency by leveraging the hardware-based packet scheduling capabilities of modern network processors. Silo enables applications to tolerate unpredictable network behavior while maintaining predictable delays.

QJump [NSDI15]

Queues don’t matter when you can JUMP them!

Matthew P. Grimmer
Niko Schwerkopf
Joel Gog
Robert N. M. Watson
Andrew W. Moore
Steven Hand
Jon Crowcroft
University of Cambridge Computer Laboratory
Cambridge, UK
Silo [SIGCOMM15] and QJump [NSDI15] guarantee predictable message latency in the cloud. However, state-of-the-art guarantees are violated in practice, as shown in the graphs below.

The graphs illustrate the empirical cumulative distribution function (ECDF) of message delays and losses. For both Silo and QJump, the ECDFs show significant deviations from the expected latency guarantees, indicating that lost packets, late packets, and delayed packets are occurring more frequently than the state-of-the-art guarantees promise.
**Silo** [SIGCOMM15]

**QJump** [NSDI15]

---

**Silo: Predictable Message Latency in the Cloud**

**QJump: Queues don't matter when you can JUMP them!**
State-of-the-art guarantees are violated!

Silo [SIGCOMM15]

Silo: Predictable Message Latency in the Cloud

Kean Jang
Intel Labs
Santa Clara, CA
Justine Sherry
UC Berkeley
Berkeley, CA
Hitesh Balani
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 guarantees is challenging, especially in distributed systems. We introduce Silo, a novel framework that enables predictable message delivery over the network. Silo achieves this by implementing a distributed queueing and routing system that ensures messages are delivered within a specified time frame. We evaluate Silo in a large-scale cloud environment and demonstrate significant improvements in message latency compared to existing solutions.

QJump [NSDI15]

QJump: Don't let queues prevent your applications

Matthew P. Groves
Muhi Salmi
Jordi Gomila
Robert N. M. Watson
Andrew W. Moore
Steven Haskel
Jon Crowcroft
University of Cambridge Computer Laboratory
www.cs.cornell.edu

Queues don’t matter when you can JUMP them!
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!
Step 0: Identification of independent services
Step 0: Identification of independent services

Step 1: Benchmarking of the service(s)
Step 0: Identification of independent services

Step 1: Benchmarking of the service(s)

Step 2: Measurements → deterministic model for the service(s)
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 $\rightarrow$ deterministic model for the service(s)

**Step 3:** Switch model $\rightarrow$ 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:

```python
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*

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

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

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>64, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>

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

Measure (CP and DP) throughput, per-packet delay and buffer capacity 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

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>64, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>

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

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

Done in §3 of the paper The performance is indeed predictable.
Step 0: Identification of independent services

Step 1: Benchmarking of the service(s)

Step 2: Measurements $\rightarrow$ deterministic model for the service(s)

Step 3: Switch model $\rightarrow$ network model (admission control)
Step 2: Measurements $\rightarrow$ deterministic model for the service
Step 2: Measurements → deterministic model for the service

network calculus model
Step 2: Measurements → determinstic model for the service network calculus model

\[ R = \text{measured throughput} \]

\[ T = \text{measured processing time} \]
Step 2: Measurements $\rightarrow$ **deterministic** model for the service network calculus model

$T = $ measured processing time

$R = $ measured throughput

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

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>64, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>
Step 2: Measurements → deterministic model for the service

- \( R = \) measured throughput
- \( T = \) measured processing time

Take the worst-case for a given scenario

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, ( tp )-dst, dl-dst, masked-nw-dst, ( five )-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>64, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>
Step 2: Measurements $\rightarrow$ deterministic model for the service

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

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>64, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>
Step 2: Measurements $\rightarrow$ deterministic model for the service

network calculus model

$T = \text{measured processing time}$

$R = \text{measured throughput}$

Take the \textbf{worst-case} for a given \textit{scenario}

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>64, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>
Step 2: Measurements → **deterministic** model for the service

**network calculus model**

**T** = measured processing time

**R** = measured throughput

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

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>64, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>

For throughput: **for throughput**

For processing time: **for processing time**
Step 2: Measurements → deterministic model for the service
Step 2: Measurements $\rightarrow$ deterministic model for the service
Step 2: Measurements $\rightarrow$ 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

worst-case latency for all ports
Step 2: Measurements → deterministic model for the service

data

worst-case buffer usage

worst-case latency for all ports

time

token bucket flow from all ports fits CBR traffic pattern, typical for small networks
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 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.

\[ r: \text{max rate} \]
Step 3: Switch model → network model (admission control)

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

- \( r: \text{max rate} \)
- \( b: \text{max burst} \)
**Step 3:** Switch model → network model (admission control)

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

- $r$: max rate
- $b$: max burst
- $D$: max delay (ever)
**Step 3:** Switch model → network model (admission control)

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

- $r$: max rate
- $b$: max burst
- $D$: max delay (ever)
- $B$: max buffer usage (ever)
Step 3: Switch model → network model (admission control)

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

- **r**: max rate
- **b**: max burst
- **D**: max delay (ever)
- **B**: max buffer usage (ever)

Choose **r, b** such that **B ≤ buffer capacity**

to ensure no packet loss
Step 3: Switch model → network model (admission control)

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

\[ r: \text{max rate} \]
\[ b: \text{max burst} \]
\[ D: \text{max delay (ever)} \]
\[ B: \text{max buffer usage (ever)} \]

To ensure no packet loss:
Choose \( r, b \) such that \( B \leq \text{buffer capacity} \)

For example:
\[ r = R/5, \text{ max. } b \text{ such that } B \leq \text{buffer capacity} \]
Step 3: Switch model → network model (admission control)

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

- $r$: max rate
- $b$: max burst
- $D$: max delay (ever)
- $B$: max buffer usage (ever)

To ensure no packet loss, choose $r, b$ such that $B \leq$ buffer capacity

For example:

- $r = R/5$, max. $b$ such that $B \leq$ buffer capacity ........... or we can also do $r = R$, max. $b$ such that $B \leq$ buffer capacity
Step 3: Switch model → network model (admission control)

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

- **r**: max rate
- **b**: max burst
- **D**: max delay (ever)
- **B**: max buffer usage (ever)

To ensure no packet loss, choose \( r, b \) such that \( B \leq \text{buffer capacity} \)

For example:

\[
\begin{align*}
    r &= \frac{R}{5}, \max. \quad b \text{ such that } B \leq \text{buffer capacity} \\
\end{align*}
\]

or we can also do \( r = R, \max. \quad b \text{ such that } B \leq \text{buffer capacity} \)
Step 3: Switch model → network model (admission control)

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

- **b:** max burst
- **r:** max rate
- **D:** max delay (ever)
- **B:** max buffer usage (ever)

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

- **r:** max rate
- **b:** max burst
- **D:** max delay (ever)
- **B:** max buffer usage (ever)

**to ensure no packet loss**

choose **r, b** such that **B ≤ buffer capacity**

for example:

\[ r = \frac{R}{5}, \text{ max. } b \text{ such that } B \leq \text{buffer capacity} \ldots \ldots \text{ or we can also do } r = R, \text{ max. } b \text{ such that } B \leq \text{buffer capacity} \ldots \ldots \]

**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

1. Keep track of per-switch usage (burst and rate)
2. Accept as long as usage ≤ allocation
Step 3: Switch model $\rightarrow$ 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 $\leq$ 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**

Can I add this flow?
**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

Can I add this flow?
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

Can I add this flow? No!
**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
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**

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>64, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>
**Loko: Proof-of-Concept Implementation and Evaluation**

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

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb. of entries</td>
<td>1, 17, 33, 49, 65, 81, 97, 113, 128</td>
</tr>
<tr>
<td>match type</td>
<td>port, tp-dst, dl-dst, masked-nw-dst, five-tuple, all</td>
</tr>
<tr>
<td>used entry</td>
<td>first, last</td>
</tr>
<tr>
<td>priorities</td>
<td>increasing, decreasing</td>
</tr>
<tr>
<td>packet size</td>
<td>6, 306, 548, 790, 1032, 1274, 1516</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service curve</th>
<th>Res. all</th>
<th>max. rate</th>
<th>max. burst</th>
<th>max. delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R = 11.8 \text{ Mbps} )</td>
<td>full-rate</td>
<td>11.8 Mbps</td>
<td>2.02 kB</td>
<td>1.86 ms</td>
</tr>
<tr>
<td>( T = 0.46 \text{ ms} )</td>
<td>fifth-rate</td>
<td>2.37 Mbps</td>
<td>2.32 kB</td>
<td>2.07 ms</td>
</tr>
</tbody>
</table>
Loko: Proof-of-Concept Implementation and Evaluation

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

Remember! SoA was failing!
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 successfully provides latency guarantees!

(For more than 10M packets and 30 minutes)
Loko: Proof-of-Concept Implementation and Evaluation

$m_b$  burst multiplier: if $> 1$, flows send more than allowed

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!

burst multiplier: if $m_b > 1$, flows send more than allowed

packet loss

max delay observed

Packet loss

No packet loss

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

Network Calculus

\[ \alpha + \beta \rightarrow \alpha^* + \text{delay, backlog bounds} \]

\[ R(t) \rightarrow S \rightarrow R^*(t) \]

\[ \beta(t) = \beta_{R,T} \approx \mathbb{R} \]

\[ \alpha^*(t) = \gamma_{r,b+rT} \]

\[ \alpha(t) = \gamma_{r,b} \]

Data

\[ b + rT \]

Duration of any time interval

\[ T + b/R \]

\[ T \]
Scalability Analysis

Max. rate: 49% 41% 32% 25% 17% 8.4% 8.4%
Max. burst: 89% 100% 100% 100% 100% 85% 99%

Max. rate: 97% 100% 100% 100% 100% 100%
Max. burst: 15% 25% 34% 44% 53% 64% 76%

(a) Medium-sized flows.  (b) Artificially inc. buffer size.
**Step 1: Benchmarking of the service**

For predictability, we have to identify ANY source of delay.
Step 1: Benchmarking of the service

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

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

For predictability, we have to identify ANY source of delay.
Step 1: Benchmarking of the service

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

```python
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

For predictability, we have to identify ANY source of delay
**Step 1: Benchmarking of the service**

For predictability, we have to **identify ANY source of delay**
Step 1: Benchmarking of the service

For predictability, we have to identify ANY source of delay

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

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()

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

7: function PROCESS_FRAME()
8: if packet from CP port then
9: if HTTP packet then SEND_TO_HTTP_SERVER()
10: if OpenFlow packet then SEND_TO_OF_AGENT()
11: if packet from DP port then SEND_TO_OF_PIPELINE()

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()

Rules one by one checks only higher priority

For predictability, we have to identify ANY source of delay
Step 1: Benchmarking of the service

For predictability, we have to identify ANY source of delay.
**Step 1: Benchmarking of the service**

```plaintext
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()

For predictability, we have to identify ANY source of delay

<table>
<thead>
<tr>
<th>id</th>
<th>matching</th>
<th>action</th>
<th>priority</th>
<th>counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dst_ip=10.0.X.X</td>
<td>output=1</td>
<td>150</td>
<td>counters</td>
</tr>
<tr>
<td>1</td>
<td>dst_ip=10.1.X.X</td>
<td>output=2</td>
<td>150000</td>
<td>counters</td>
</tr>
<tr>
<td>2</td>
<td>dst_ip=10.2.X.X</td>
<td>output=3</td>
<td>500</td>
<td>counters</td>
</tr>
<tr>
<td>3</td>
<td>dst_ip=10.2.5.5</td>
<td>output=1</td>
<td>200</td>
<td>counters</td>
</tr>
<tr>
<td>4</td>
<td>dst_ip=10.3.X.X</td>
<td>output=2</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>5</td>
<td>dst_ip=10.4.X.X</td>
<td>output=1</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>6</td>
<td>dst_ip=10.2.5.X</td>
<td>output=2</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>7</td>
<td>dst_ip=10.2.5.X</td>
<td>output=1</td>
<td>100</td>
<td>counters</td>
</tr>
<tr>
<td>8</td>
<td>dst_ip=10.2.5.X</td>
<td>output=3</td>
<td>300</td>
<td>counters</td>
</tr>
<tr>
<td>9</td>
<td>dst_ip=10.2.5.X</td>
<td>output=2</td>
<td>500</td>
<td>counters</td>
</tr>
<tr>
<td>10</td>
<td>dst_ip=10.2.5.X</td>
<td>output=1</td>
<td>500</td>
<td>counters</td>
</tr>
</tbody>
</table>

Rules one by one checks only higher priority

[packet|dst_ip=10.2.5.5]
Step 1: Benchmarking of the service

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

<table>
<thead>
<tr>
<th>id</th>
<th>matching</th>
<th>action</th>
<th>priority</th>
<th>counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dst_ip=10.0.X.X</td>
<td>output=1</td>
<td>150</td>
<td>counters</td>
</tr>
<tr>
<td>1</td>
<td>dst_ip=10.1.X.X</td>
<td>output=2</td>
<td>150000</td>
<td>counters</td>
</tr>
<tr>
<td>2</td>
<td>dst_ip=10.2.X.X</td>
<td>output=3</td>
<td>500</td>
<td>counters</td>
</tr>
<tr>
<td>3</td>
<td>dst_ip=10.2.5.5</td>
<td>output=1</td>
<td>200</td>
<td>counters</td>
</tr>
<tr>
<td>4</td>
<td>dst_ip=10.3.X.X</td>
<td>output=2</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>5</td>
<td>dst_ip=10.4.X.X</td>
<td>output=1</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>6</td>
<td>dst_ip=10.2.5.X</td>
<td>output=2</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>7</td>
<td>dst_ip=10.2.5.X</td>
<td>output=1</td>
<td>100</td>
<td>counters</td>
</tr>
<tr>
<td>8</td>
<td>dst_ip=10.2.5.X</td>
<td>output=3</td>
<td>300</td>
<td>counters</td>
</tr>
<tr>
<td>9</td>
<td>dst_ip=10.2.5.X</td>
<td>output=2</td>
<td>500</td>
<td>counters</td>
</tr>
<tr>
<td>10</td>
<td>dst_ip=10.2.X.X</td>
<td>output=1</td>
<td>500</td>
<td>counters</td>
</tr>
</tbody>
</table>

Rules one by one checks only higher priority.
**Step 1: Benchmarking of the service**

```
7: function PROCESS_FRAME()
8:   if packet from CP port then
9:     if HTTP packet then sendMessageToHttpServer()
10:    if OpenFlow packet then sendMessageToOFAgent()
11:   if packet from DP port then sendMessageToOFPipeline()
```

For predictability, we have to **identify ANY source of delay**.
Step 1: Benchmarking of the service

For predictability, we have to identify ANY source of delay

rules one by one checks only higher priority

For predictability, we have to **identify ANY** source of delay
Step 1: Benchmarking of the service

For predictability, we have to identify ANY source of delay.
Step 1: Benchmarking of the service

For predictability, we have to identify ANY source of delay

```python
7: function PROCESSFRAME()
8:     if packet from CP port then
9:         if HTTP packet then sendDataToHTTPServer()
10:        if OpenFlow packet then sendDataToOFAgent()
11:    if packet from DP port then sendDataToOFPipeline()
```

<table>
<thead>
<tr>
<th>id</th>
<th>matching</th>
<th>action</th>
<th>priority</th>
<th>counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dst_ip=10.0.X.X</td>
<td>output=1</td>
<td>150</td>
<td>counters</td>
</tr>
<tr>
<td>1</td>
<td>dst_ip=10.1.X.X</td>
<td>output=2</td>
<td>150000</td>
<td>counters</td>
</tr>
<tr>
<td>2</td>
<td>dst_ip=10.2.X.X</td>
<td>output=3</td>
<td>500</td>
<td>counters</td>
</tr>
<tr>
<td>3</td>
<td>dst_ip=10.2.5.X</td>
<td>output=1</td>
<td>200</td>
<td>counters</td>
</tr>
<tr>
<td>4</td>
<td>dst_ip=10.3.X.X</td>
<td>output=2</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>5</td>
<td>dst_ip=10.4.X.X</td>
<td>output=1</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>6</td>
<td>dst_ip=10.2.5.X</td>
<td>output=2</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>7</td>
<td>dst_ip=10.2.5.X</td>
<td>output=1</td>
<td>100</td>
<td>counters</td>
</tr>
<tr>
<td>8</td>
<td>dst_ip=10.2.5.X</td>
<td>output=3</td>
<td>300</td>
<td>counters</td>
</tr>
<tr>
<td>9</td>
<td>dst_ip=10.2.5.X</td>
<td>output=2</td>
<td>500</td>
<td>counters</td>
</tr>
<tr>
<td>10</td>
<td>dst_ip=10.2.X.X</td>
<td>output=1</td>
<td>500</td>
<td>counters</td>
</tr>
</tbody>
</table>
```

Rules one by one checks only higher priority.
Step 1: Benchmarking of the service

7: function PROCESS_FRAME()
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

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

#### Function `PROCESSFRAME()`

```python
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()
```

<table>
<thead>
<tr>
<th>packet</th>
<th>dst_ip=10.2.5.5</th>
</tr>
</thead>
</table>

#### Matching Table

<table>
<thead>
<tr>
<th>id</th>
<th>matching</th>
<th>action</th>
<th>priority</th>
<th>counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dst_ip=10.0.X.X</td>
<td>output=1</td>
<td>150</td>
<td>counters</td>
</tr>
<tr>
<td>1</td>
<td>dst_ip=10.1.X.X</td>
<td>output=2</td>
<td>150000</td>
<td>counters</td>
</tr>
<tr>
<td>2</td>
<td>dst_ip=10.2.X.X</td>
<td>output=3</td>
<td>500</td>
<td>counters</td>
</tr>
<tr>
<td>3</td>
<td>dst_ip=10.2.5.5</td>
<td>output=1</td>
<td>200</td>
<td>counters</td>
</tr>
<tr>
<td>4</td>
<td>dst_ip=10.3.X.X</td>
<td>output=2</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>5</td>
<td>dst_ip=10.4.X.X</td>
<td>output=1</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>6</td>
<td>dst_ip=10.2.5.5</td>
<td>output=2</td>
<td>250000</td>
<td>counters</td>
</tr>
<tr>
<td>7</td>
<td>dst_ip=10.2.5.X</td>
<td>output=1</td>
<td>100</td>
<td>counters</td>
</tr>
<tr>
<td>8</td>
<td>dst_ip=10.2.5.X</td>
<td>output=3</td>
<td>300</td>
<td>counters</td>
</tr>
<tr>
<td>9</td>
<td>dst_ip=10.2.5.X</td>
<td>output=2</td>
<td>500</td>
<td>counters</td>
</tr>
<tr>
<td>10</td>
<td>dst_ip=10.2.X.X</td>
<td>output=1</td>
<td>500</td>
<td>counters</td>
</tr>
</tbody>
</table>

Rules one by one checks only higher priority.

**Dimension**

- nb. of entries
- match type

For predictability, we have to **identify ANY source of delay**
Step 1: Benchmarking of the service

7: function PROCESS_FRAME()
8: if packet from CP port then
9:    if HTTP packet then SEND_TO_HTTP_SERVER()
10:   if OpenFlow packet then SEND_TO_OF_AGENT()
11:  if packet from DP port then SEND_TO_OF_PIPELINE()

For predictability, we have to identify ANY source of delay

rules one by one
checks only higher priority

Dimension

<table>
<thead>
<tr>
<th>nb. of entries</th>
<th>match type</th>
<th>action</th>
</tr>
</thead>
</table>

For predictability, we have to identify ANY source of delay
Step 1: Benchmarking of the service

7: function PROCESS_FRAME()
8: if packet from CP port then
9:     if HTTP packet then SEND_TO_HTTPSERVER()
10:    if OpenFlow packet then SEND_TO_OFAGENT()
11:   if packet from DP port then SEND_TO_OF_PIPELINE()

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

rules one by one checks only higher priority

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()

Rules one by one checks only higher priority

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**

Measure **throughput and per-packet delay** for each combination of the dimensions.
Step 1: Benchmarking of the service

Measure throughput and per-packet delay for each combination of the dimensions

17 entries five-tuple matching 790-byte packets output action
Step 1: Benchmarking of the service

Measure throughput and per-packet delay for each combination of the dimensions.
Step 1: Benchmarking of the service

Measure throughput and per-packet delay for each combination of the dimensions.

Capture the data from 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!