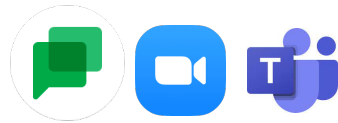


# Toward **Zero-waste** Terabit Networked Systems

Liangcheng (LC) Yu



# Ever-increasing user applications



Online Conferencing



Machine Learning



Video Streaming



Latency-critical Applications

Application



# Network systems, a packet forwarding engine



Networks serve to **forward user data**

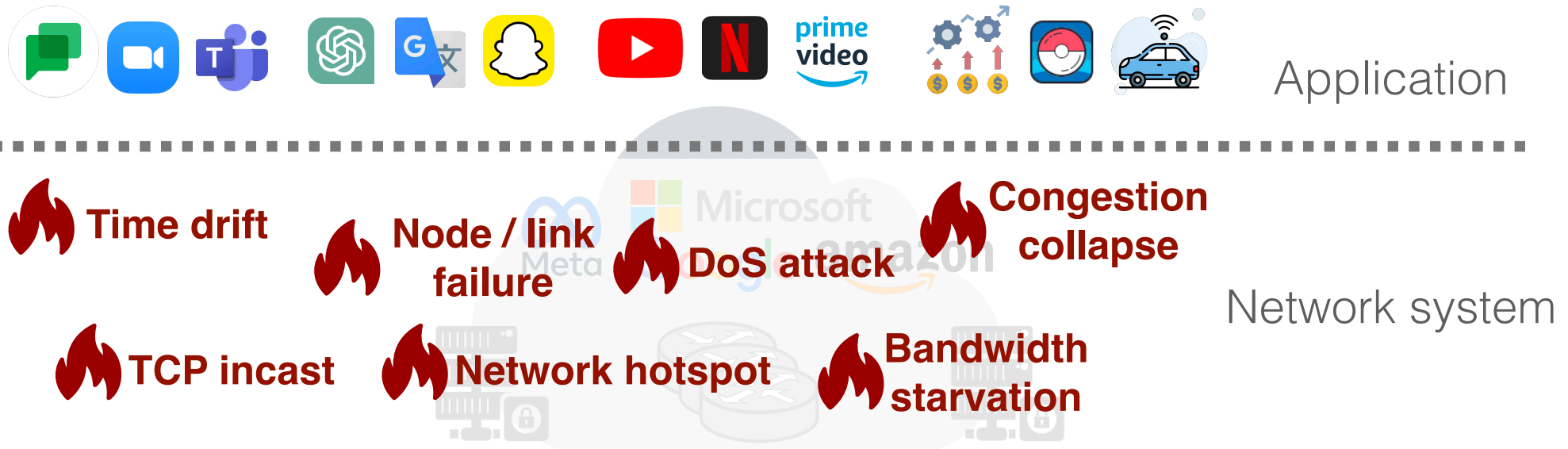
# Network systems, a packet forwarding engine



Networks serve to **forward user data**

*Today, networks are **far more complex!***

# Network systems: an operator's view



Networks serve to **forward user data**

*Today, networks are **far more complex!***

*...must handle **out-of-control events!***

# Network systems: an operator's view

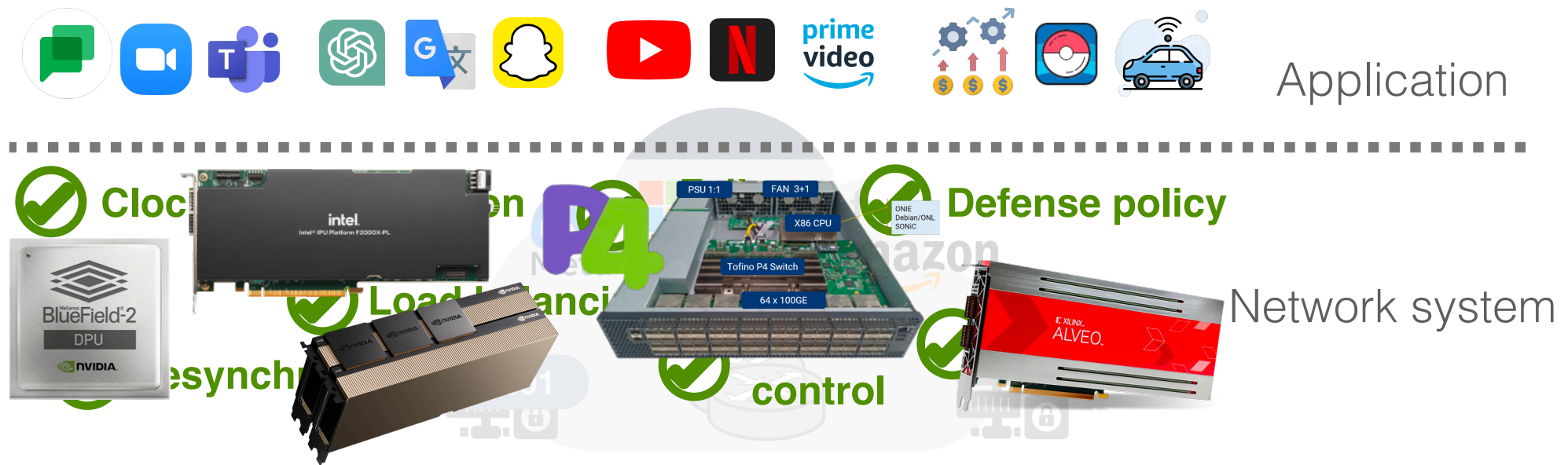


Networks serve to **forward user data**

*Today, networks are **far more complex!***

*...a vast array of control tasks*

# Network systems: an operator's view



Networks serve to **forward user data**

*Today, networks are **far more complex!***

*...a vast array of control tasks*

*...in-network computation w/ emerging HW accelerators*

*...and more!*

# Network systems: an operator's view



Today, network systems are  
**more than** just about **data forwarding!**

Networks serve to **forward user data**

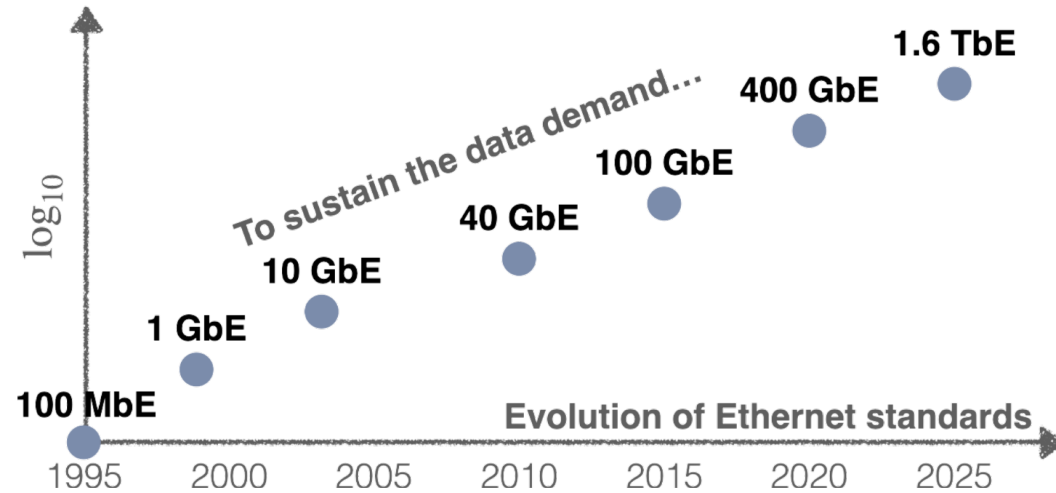
*Today, networks are **far more complex!***

*...a vast array of control tasks*

*...in-network computation w/ emerging HW accelerators*

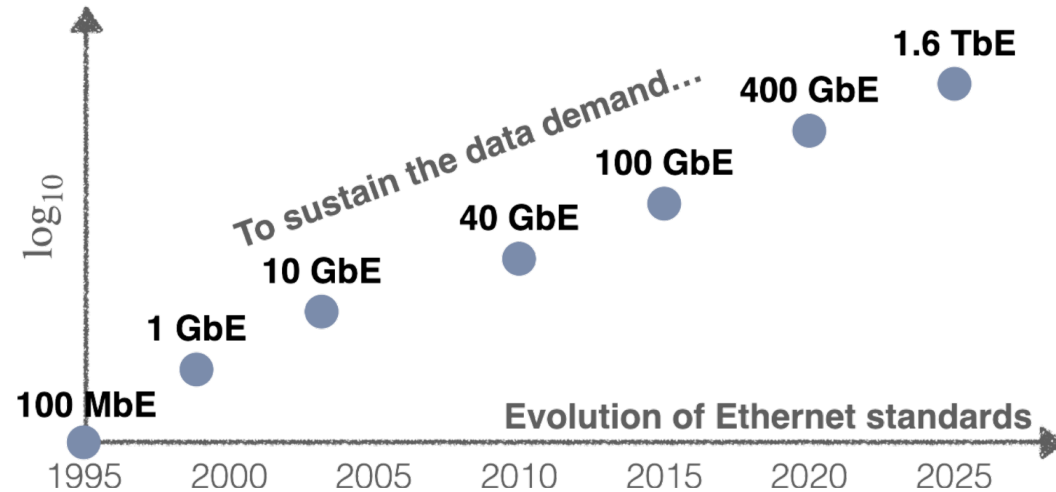
*...and more!*

# Trend toward terabit speed...



The speed of networking is **outpacing** many others

# Trend toward terabit speed...

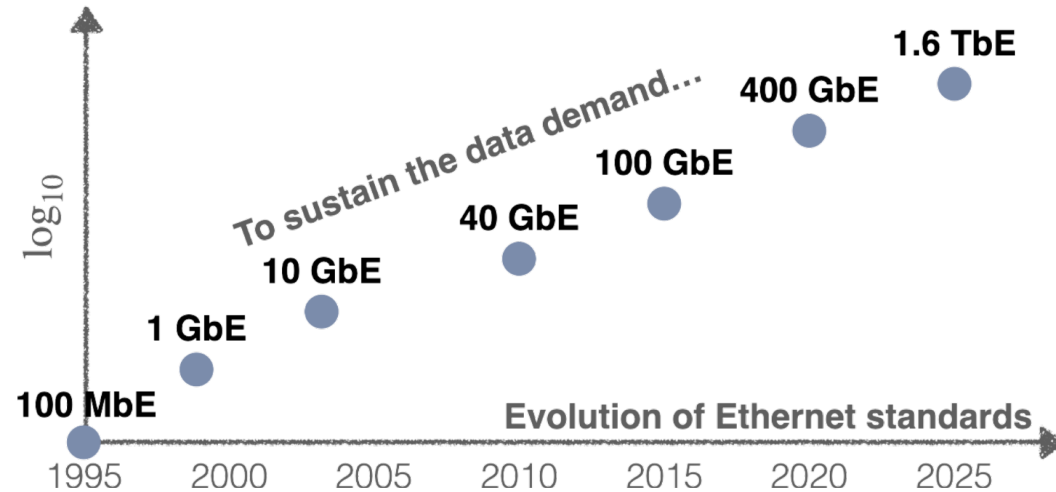


The speed of networking is **outpacing** many others

Great for *application data transfer* 😊



# Trend toward terabit speed...

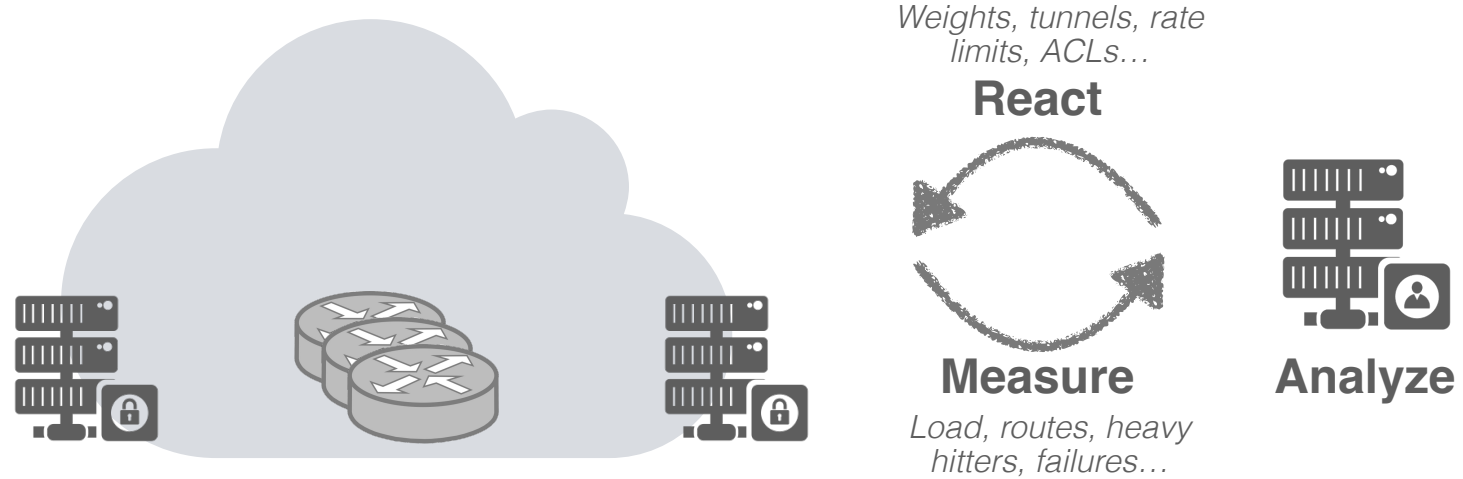


The speed of networking is **outpacing** many others

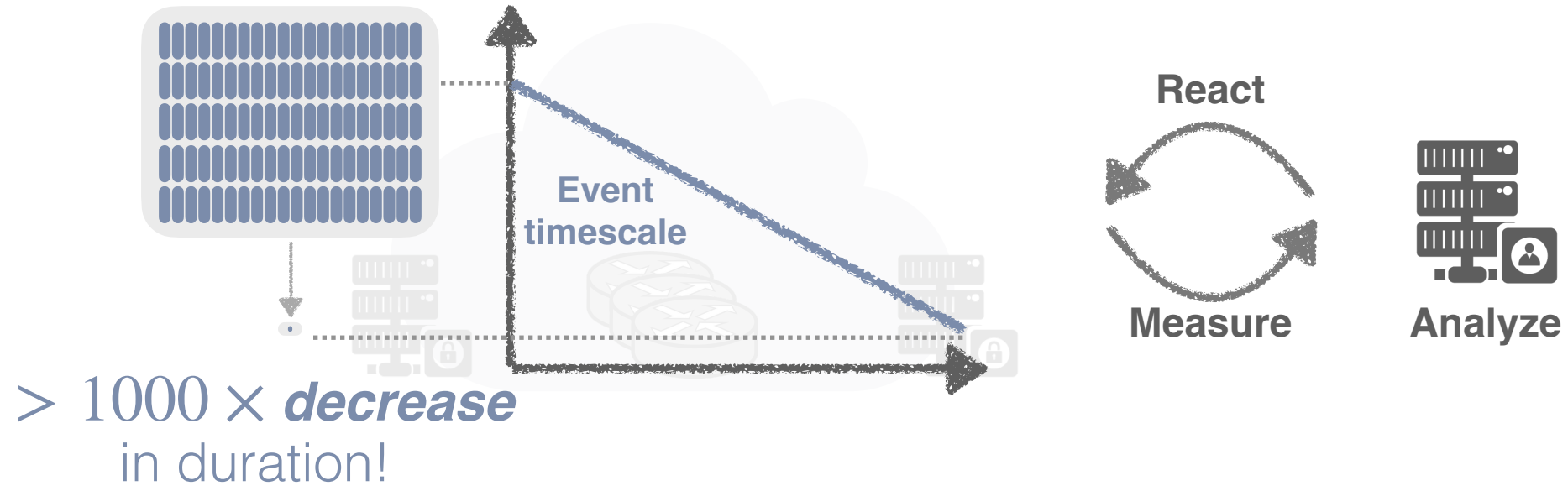
Great for *application data transfer* 😊

... **problematic for other tasks!** 😞

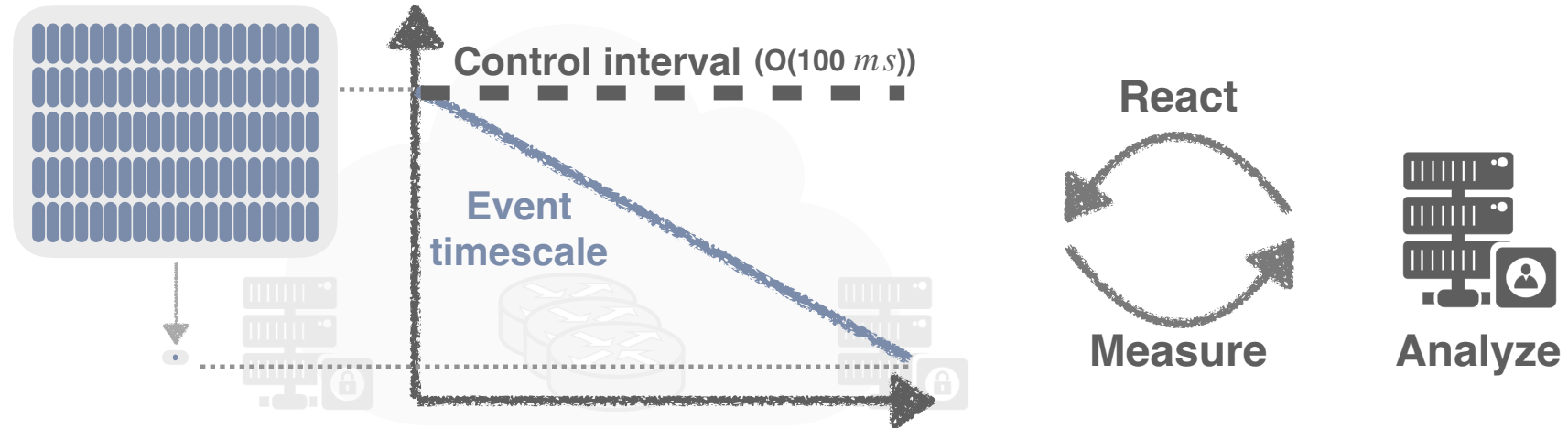
# Network control function as an example



# Network control function as an example

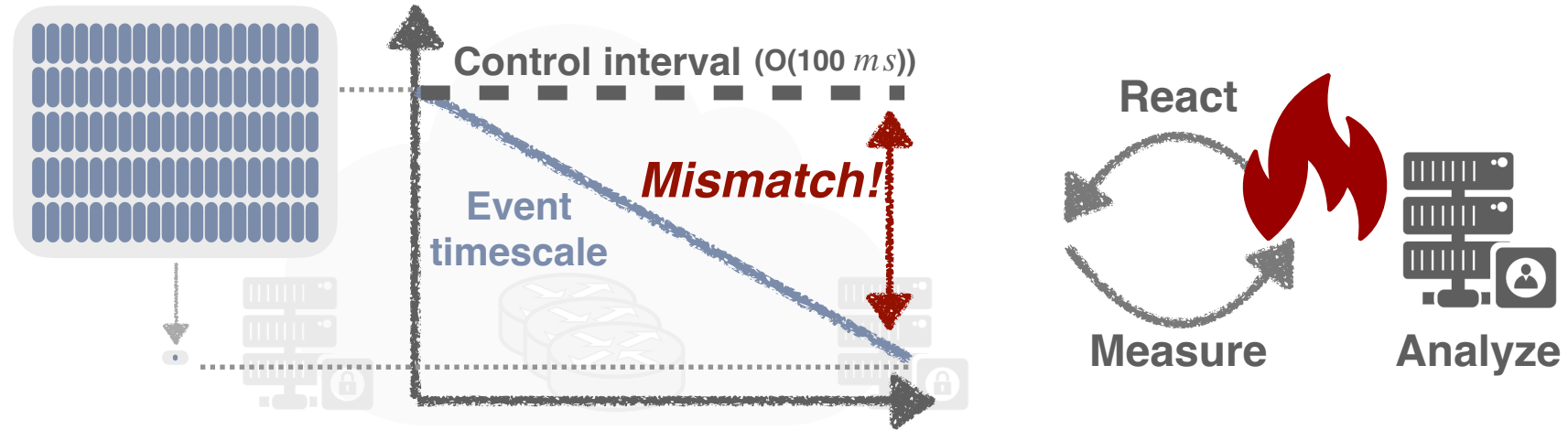


# Control, fast and slow



*If the control interval remains coarse-grained...*

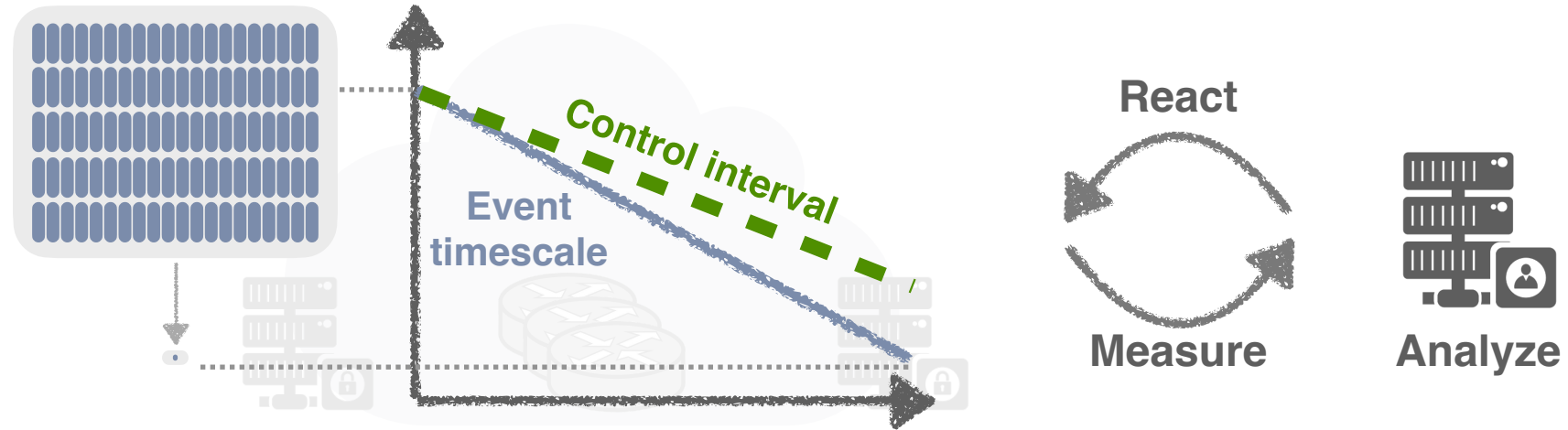
# Control, fast and slow



*If the control interval remains coarse-grained...*

☹️ **Hard to react to microscopic events**

# Control, fast and slow

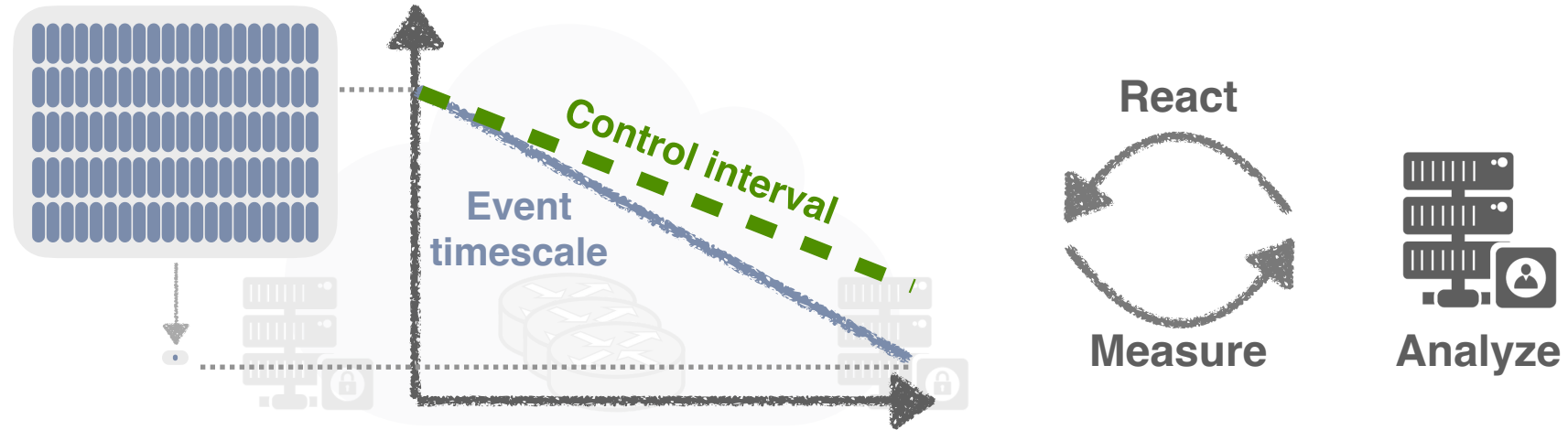


*If the control interval remains coarse-grained...*

☹️ **Hard to react to microscopic events**

*If we were to catch up with the link speeds...*

# Control, fast and slow



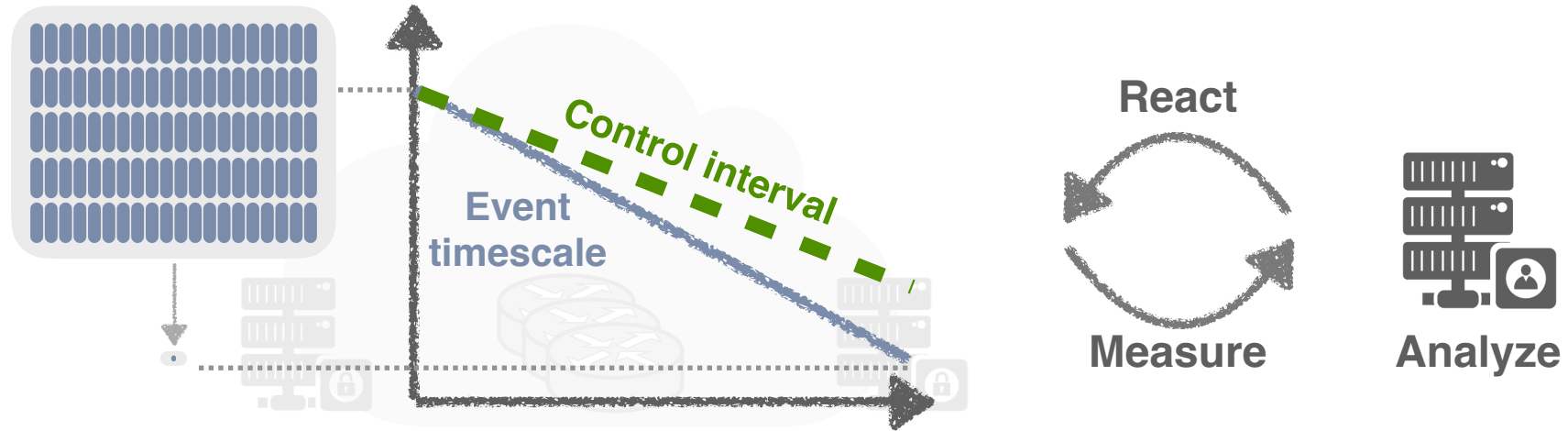
*If the control interval remains coarse-grained...*

☹️ **Hard to react to microscopic events**

*If we were to catch up with the link speeds...*

***Allocate more cables, CPUs...?***

# Control, fast and slow



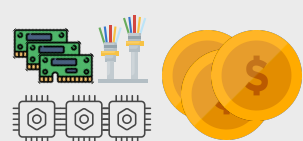
*If the control interval remains coarse-grained...*

☹️ **Hard to react to microscopic events**

*If we were to catch up with the link speeds...*

***Allocate more cables, CPUs...?***

☹️ **Costs!**



**Device purchasing**



**Embodied &  
operational carbon**



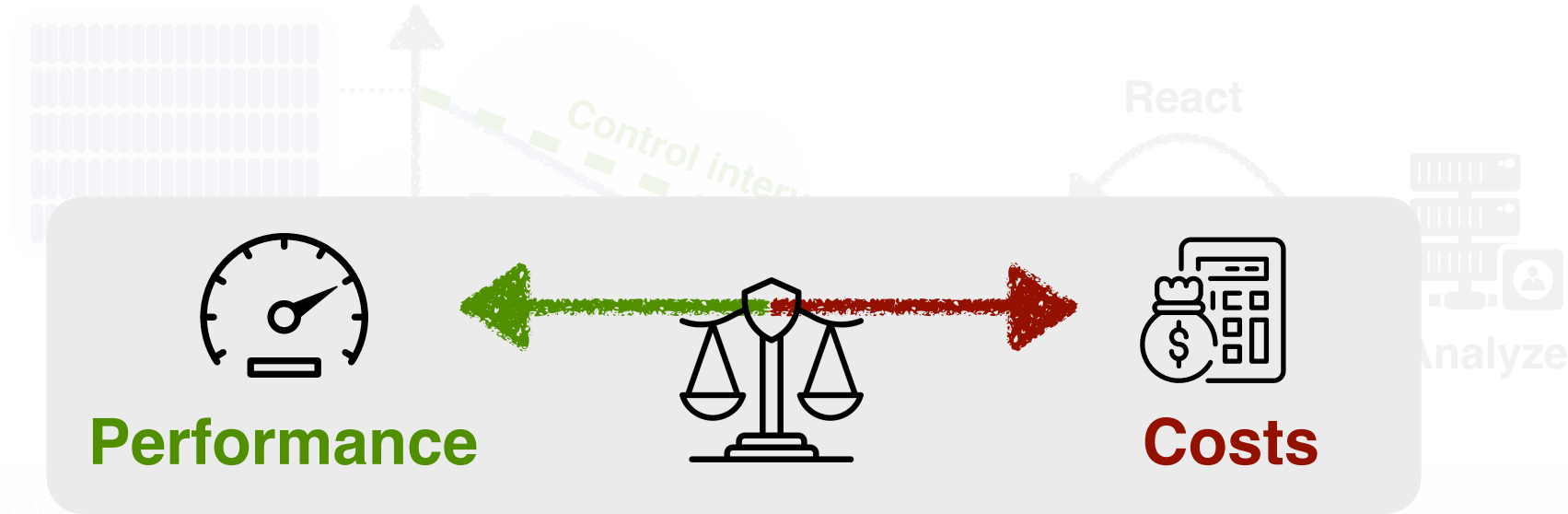
**Power & cooling**



**Impact to existing traffic**



# Control, fast and slow



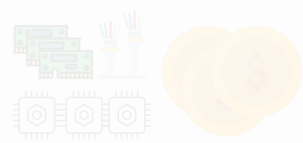
*If the control interval remains constant, growth is exponential*

☹️ *Hard to react to microscopic events*

*If we were to catch up with the link speeds...*

*Allocate more resources (cables, CPUs...)?*

☹️ **Costs!**



Device purchasing



Embodied & operational carbon

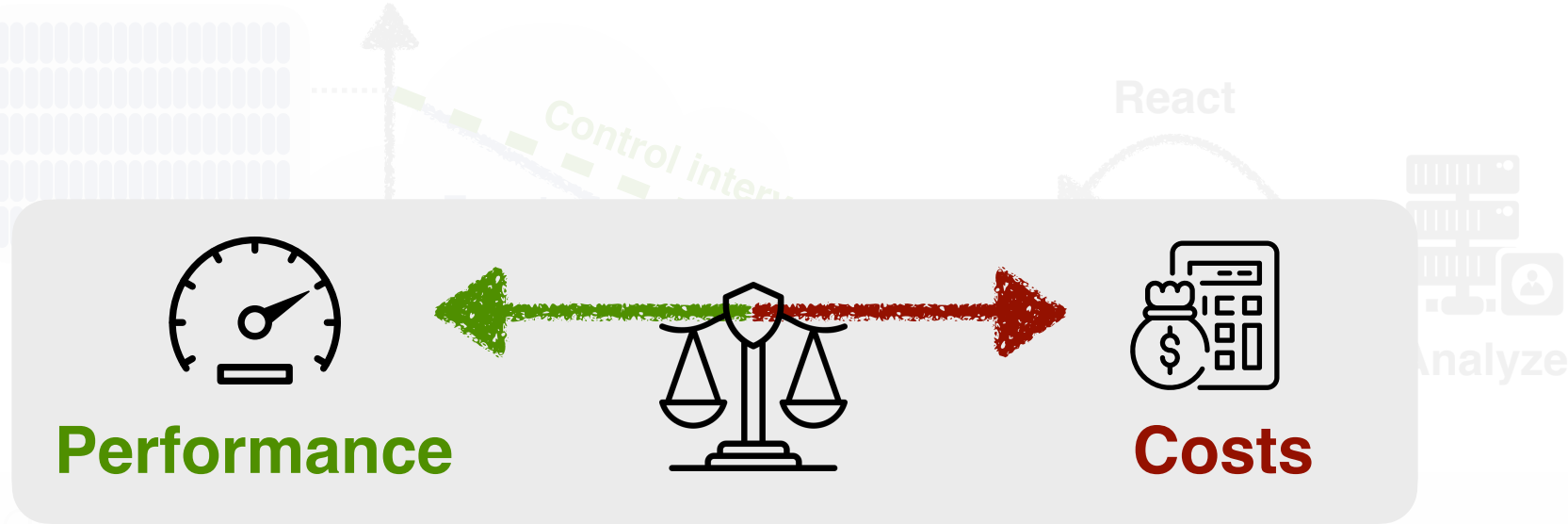


Power & cooling



Impact to existing traffic

# Control, fast and slow



*Can we break this tension?*

*If we were to catch up with the link speeds...*

*Allocate more resources (cables, CPUs...)?*



Device purchasing



Embodied & operational carbon

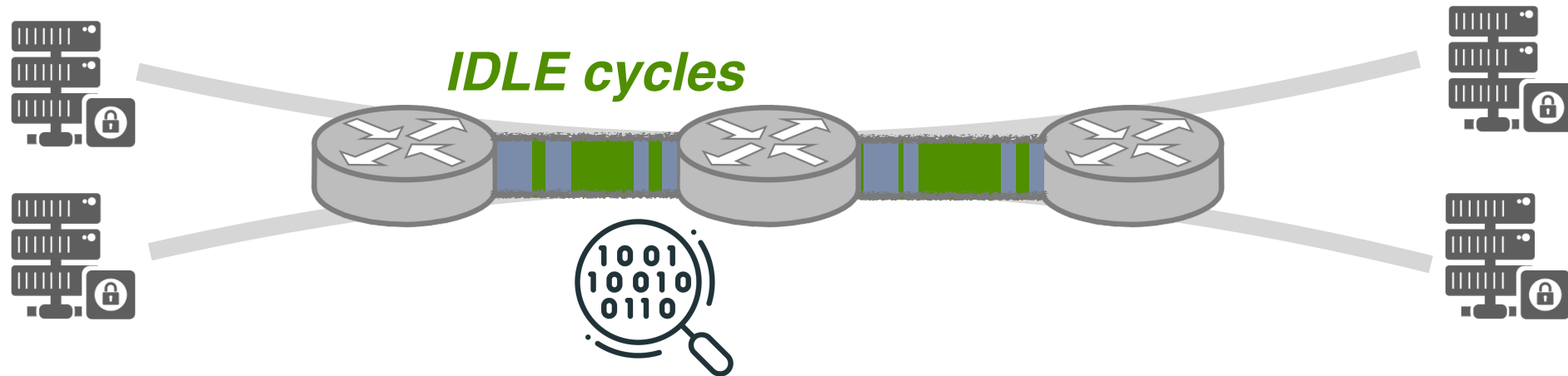


Power & cooling

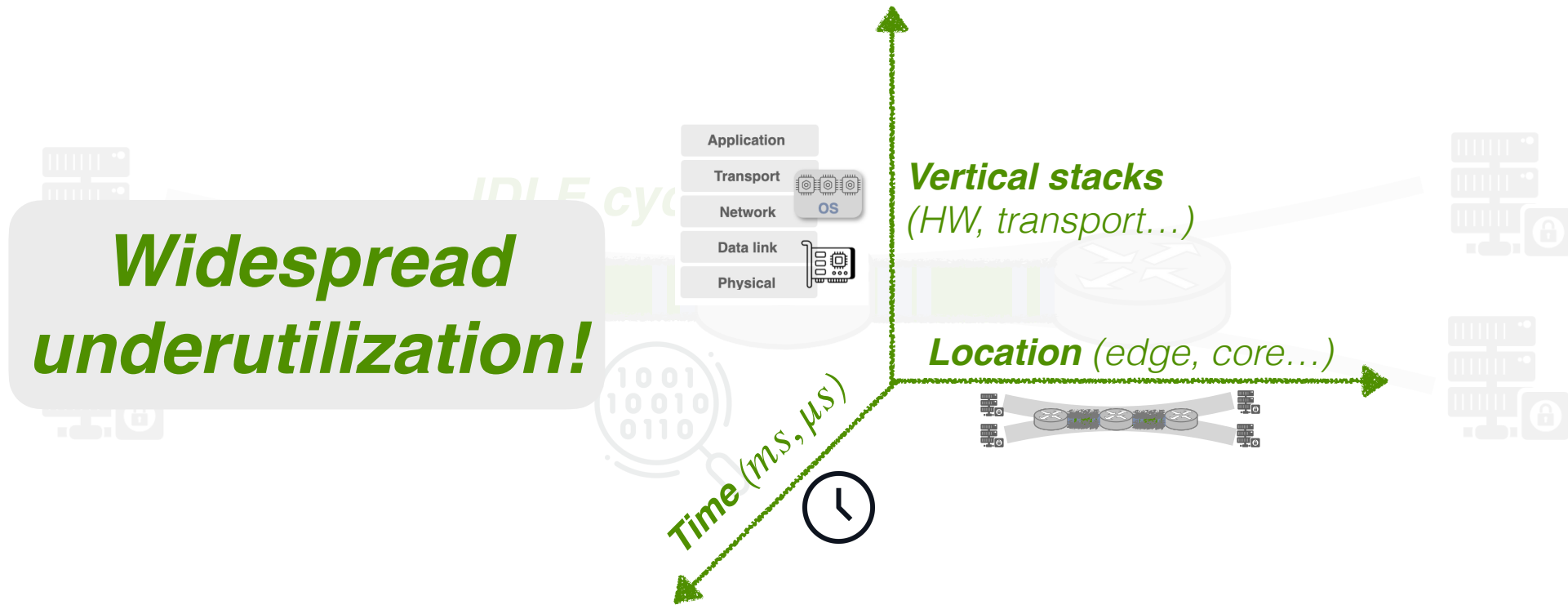


Impact to existing traffic

# Observation: in-network waste

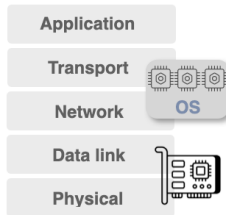


# Observation: in-network waste



# Observation: in-network waste

***Widespread underutilization!***



***Vertical stacks***  
(HW, transport...)

***Location*** (edge, core...)

***Time*** (ms,  $\mu$ s)



*Why not harness them to support background functions?*

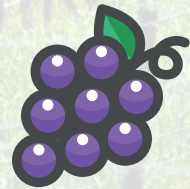


**A case of intercropping in farm systems...**





# A case of intercropping in farm systems...



**Cash plant  
(primary)**



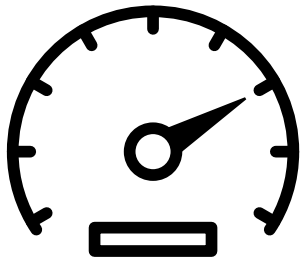
**Companion plant**

*Idle resources: sun light, soil, water, insects, 3D position...*



# This talk: a zero-waste design approach

## High-efficiency designs



Input: user workload

Goal: output a network that optimizes end-to-end performance metrics with minimal resource usage

## Zero-waste designs



Input: the workload **and the network**

Goal: maximize the utility of ***that network***, such as through uncovering the potential of the widespread in-network waste



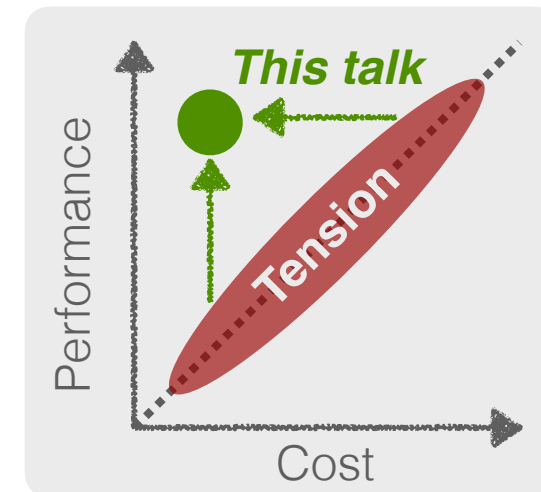
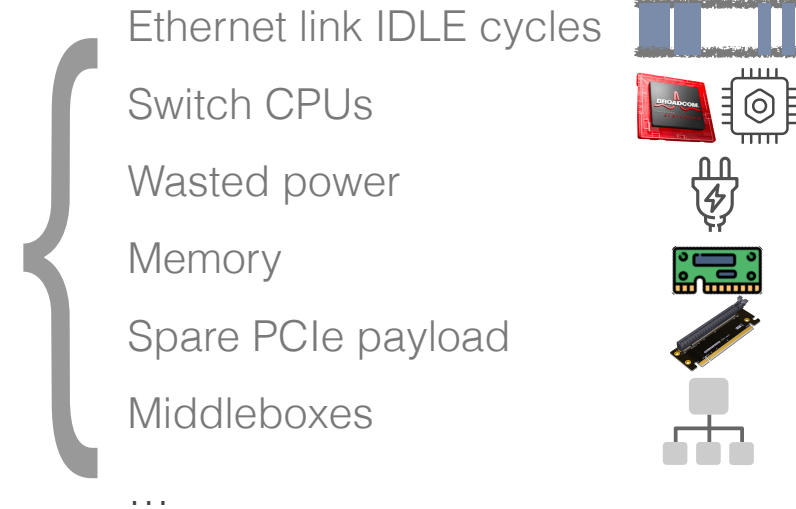
# This talk: takeaway

1

In-network waste is **widespread**, and in **numerous forms**

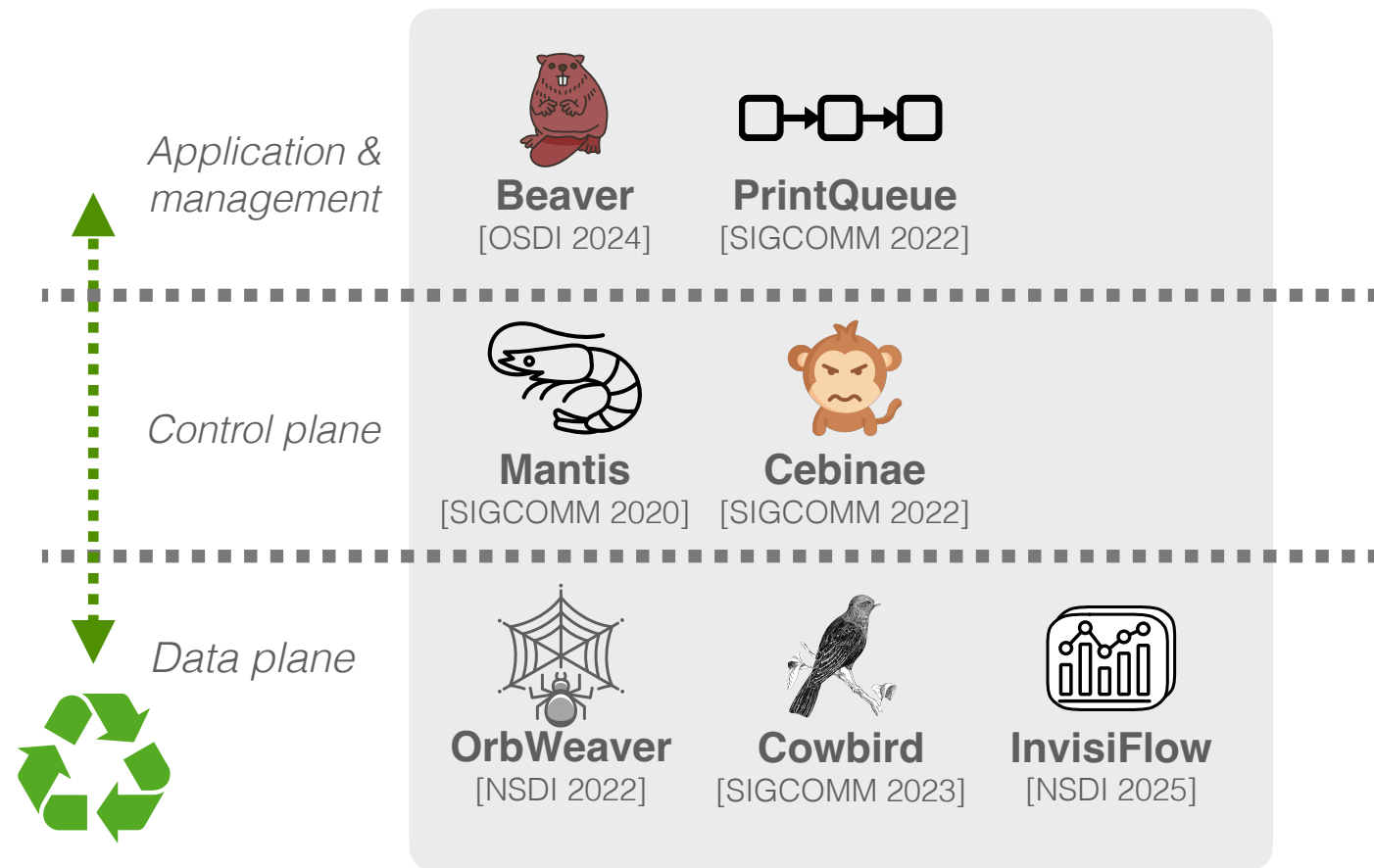
2

By exploiting domain-specific underutilization, it is **possible** to integrate performant functions with **near-zero costs**

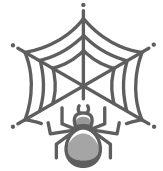


# Research overview

*Rethink the co-design of applications, software, and hardware to  
**minimize waste** in networked systems*



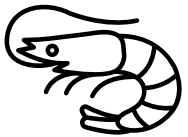
# Instantiations of zero-waste designs



**OrbWeaver** (*NSDI 2022*)

Reusing IDLE link cycles for in-band control communication

**Reuse**



**Mantis** (*SIGCOMM 2020*)

Recycling switch resources for flexible, sub-RTT reactions

**Recycle**

Zero-waste  
designs

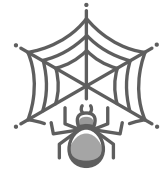


**Beaver** (*OSDI 2024*)

Reducing 'tax' of partial snapshots for distributed cloud services

**Reduce**

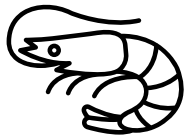
# Outline



**OrbWeaver** (*NSDI 2022*)

Reusing IDLE link cycles for in-band control communication

**Reuse**



**Mantis** (*SIGCOMM 2020*)

Recycling switch resources for flexible, sub-RTT reactions

**Recycle**

Zero-waste  
designs



**Beaver** (*OSDI 2024*)

Reducing 'tax' of partial snapshots for distributed cloud services

**Reduce**

# Networks are woven from packets

- A primary goal of computer networks: ***delivery packets***

# Networks are woven from packets

- A primary goal of computer networks: ***delivery packets***
  - ***User application***: video streaming, web browsing, file transfer...

# Networks are woven from packets

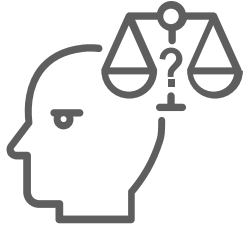
- A primary goal of computer networks: ***delivery packets***
  - ***User application***: video streaming, web browsing, file transfer...
  - ***Non-user application***: control messages, probes about network state, keep alive heartbeats...

# Networks are woven from packets

- A primary goal of computer networks: ***delivery packets***
  - ***User application***: video streaming, web browsing, file transfer...
  - ***Non-user application***: control messages, probes about network state, keep alive heartbeats...
- Often, two classes of traffic ***multiplex*** the same network



# When introducing a distributed coordination function...



To cost **extra bandwidth** for **efficacy**, or not?

*Time synchronization*

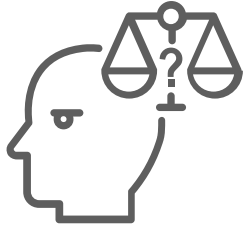
*Failure detector*

*Congestion notification*

*In-band telemetry*

*...*

# When introducing a distributed coordination function...



To cost **extra bandwidth** for **efficacy**, or not?

*Time synchronization*

**clock-sync rate**  $\leftrightarrow$  **clock precision**

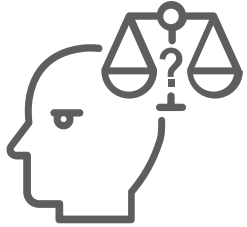
*Failure detector*

*Congestion notification*

*In-band telemetry*

...

# When introducing a distributed coordination function...



To cost **extra bandwidth** for **efficacy**, or not?

*Time synchronization*

**clock-sync rate** ↔ **clock precision**

*Failure detector*

**keep alive message frequency** ↔ **detection speed**

*Congestion notification*

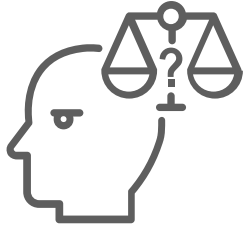
**probe data/rate** ↔ **measurement accuracy**

*In-band telemetry*

**INT postcard volume** ↔ **post-mortem analysis**

...

# When introducing a distributed coordination function...



To cost **extra bandwidth** for **efficacy**, or not?

*Time synchronization*

**clock-sync rate** ↔ **clock precision**

*Failure detector*

**keep alive message frequency** ↔ **detection speed**

*Congestion notification*

**probe data/rate** ↔ **measurement accuracy**

*In-band telemetry*

**INT postcard volume** ↔ **post-mortem analysis**

...

Is this trade-off between overhead and fidelity necessary?

When introducing an in-band control function...

To consume **extra bandwidth** for **efficacy**, or not to?

Time synchronization    **clock-sync rate**  $\leftrightarrow$  **clock precision**

Can we coordinate at **high-fidelity** with a **near-zero cost** (to usable bandwidth, latency...)?

Is this trade-off between fidelity and overhead necessary?

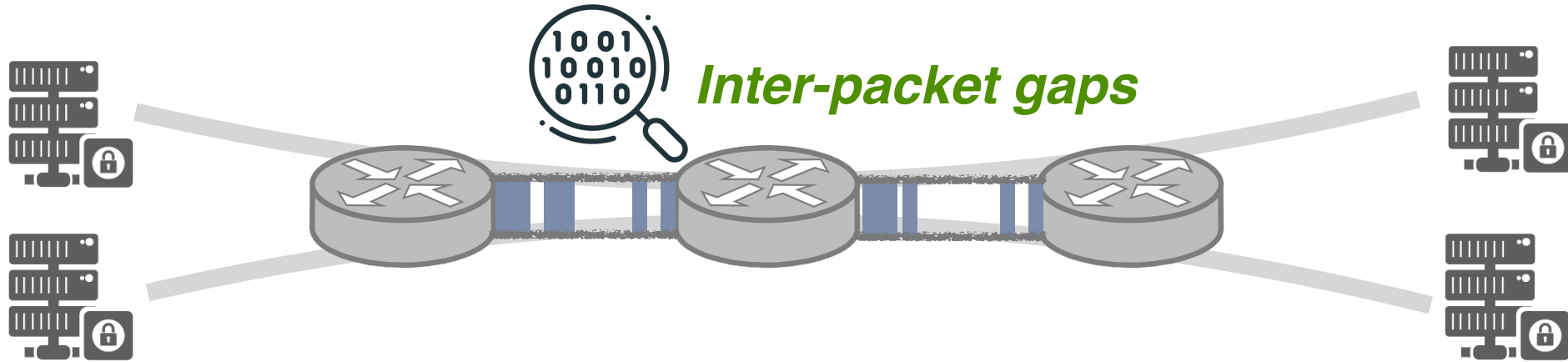
Can we coordinate at **high-fidelity** with a **near-zero cost** (to usable bandwidth, latency...)?



**Idea: Weaved Stream**

- Exploit **every gap** ( $O(100\text{ns})$ ) between user packets opportunistically
- Inject customizable **IDLE packets** carrying information across devices

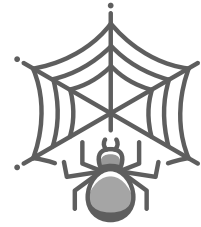
# Opportunity: $< \mu s$ gaps are prevalent



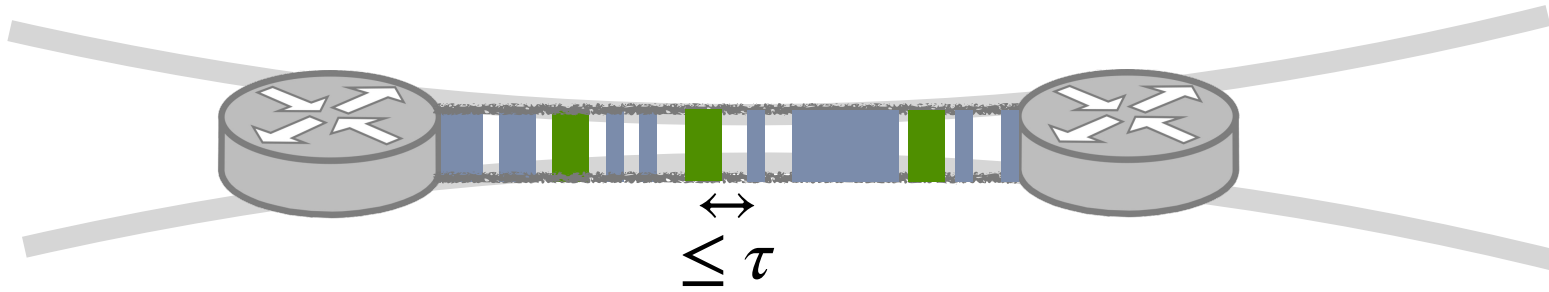
## *Root causes?*

- Uncertainties in application load patterns (e.g., burstiness)
- Conservative resource provisioning for peak usages
- Bottlenecks at CPU processing vs network BW
- TCP effects
- Structural asymmetry
- ...

# Abstraction: weaved stream



Union of **user** AND **IDLE** (injected) packets



**[R1 Predictability]** Interval between **any** two consecutive packets  $\leq \tau$

$$\tau = B_{100Gbps} / MTU_{1500B} = 120ns$$

**[R2 Little-to-zero overhead]** Not impact user packets or power draw



# Abstraction: weaved stream

Union of **user** and **IDLE** (injected) packets

Implement many ***in-network applications***  
(failure detection, clock sync, congestion notification...)  
***for free!***

[R1]

$$\tau = B_{100Gbps} / MTU_{1500B} = 120ns$$

[R2 ***Little-to-zero overhead***] Not impact user packets or power draw

# Abstraction: weaved stream

Union of **user** and **IDLE** (injected) packets

## Crazy idea?

### Extending IDLE characters to higher layers

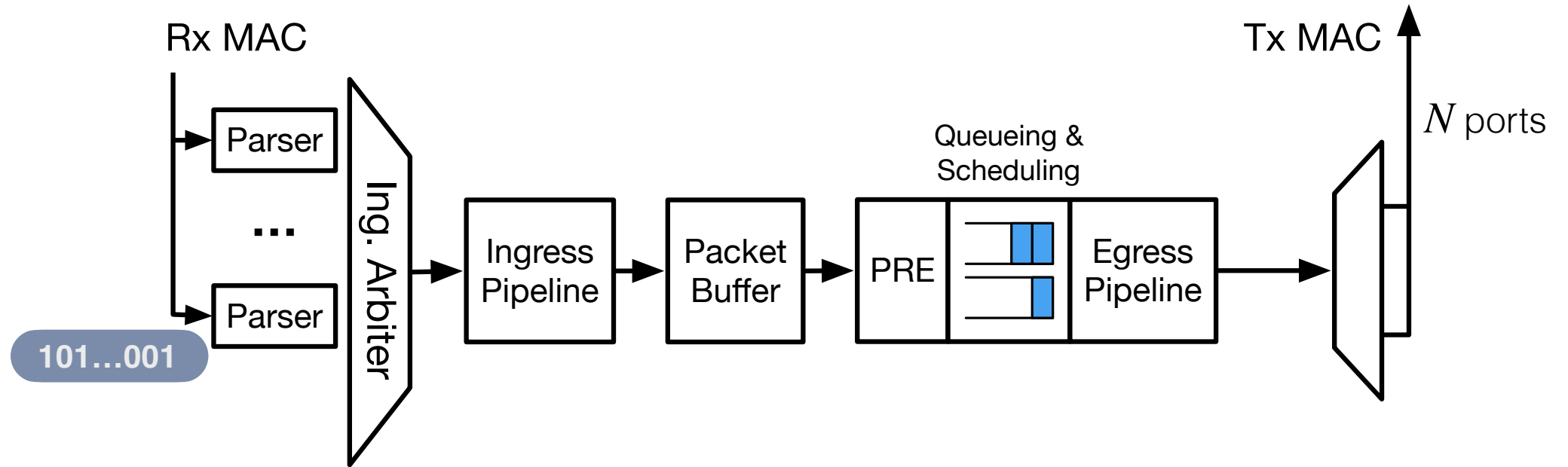
- Data plane packet generator
- Replication engine
- Data plane programmability
- Flexible switch configuration (priorities, buffers...)

**[R2 Little-to-zero overhead]** Not impact user packets or power draw

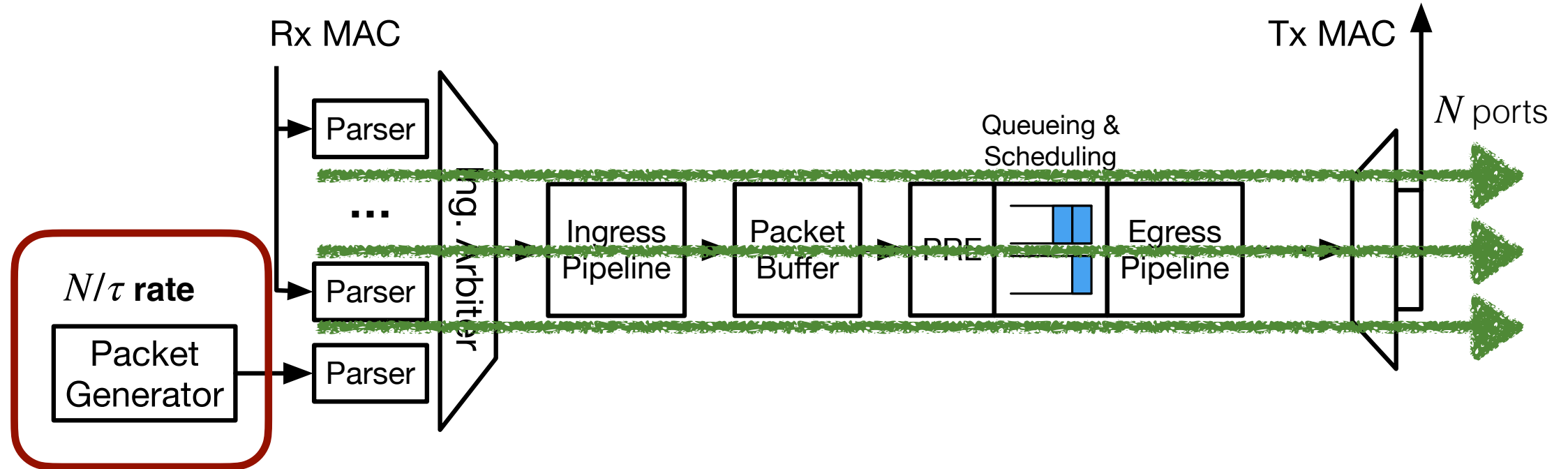
# OrbWeaver: outline

1. Switch data plane architecture
2. Implementing weaved stream abstraction
3. OrbWeaver applications

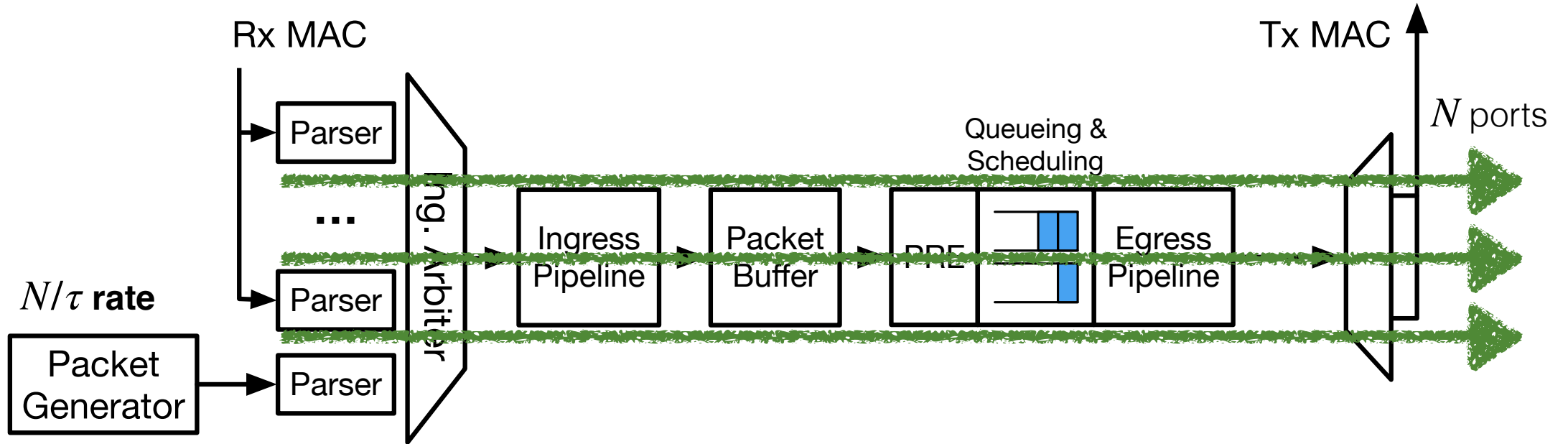
# RMT switch architecture




# Strawman: blind packet generation

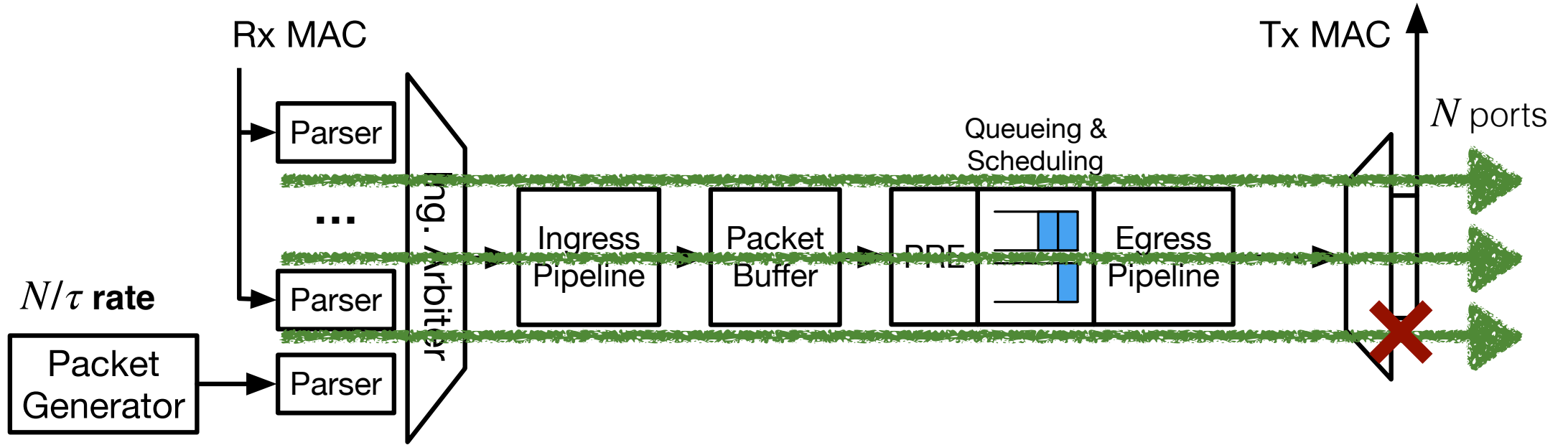


# Strawman: blind packet generation



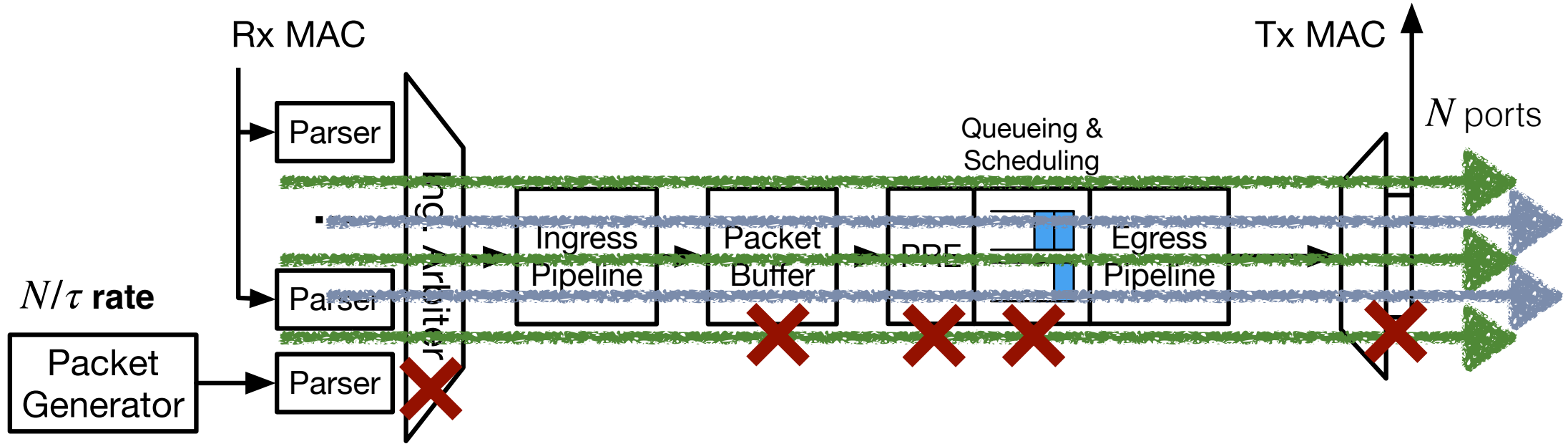
Predictability even there is no user traffic 

# Problems with blind packet generation



**#1 Scalability:** overwhelm generator capacity to satisfy target rate for all ports

# Problems with blind packet generation



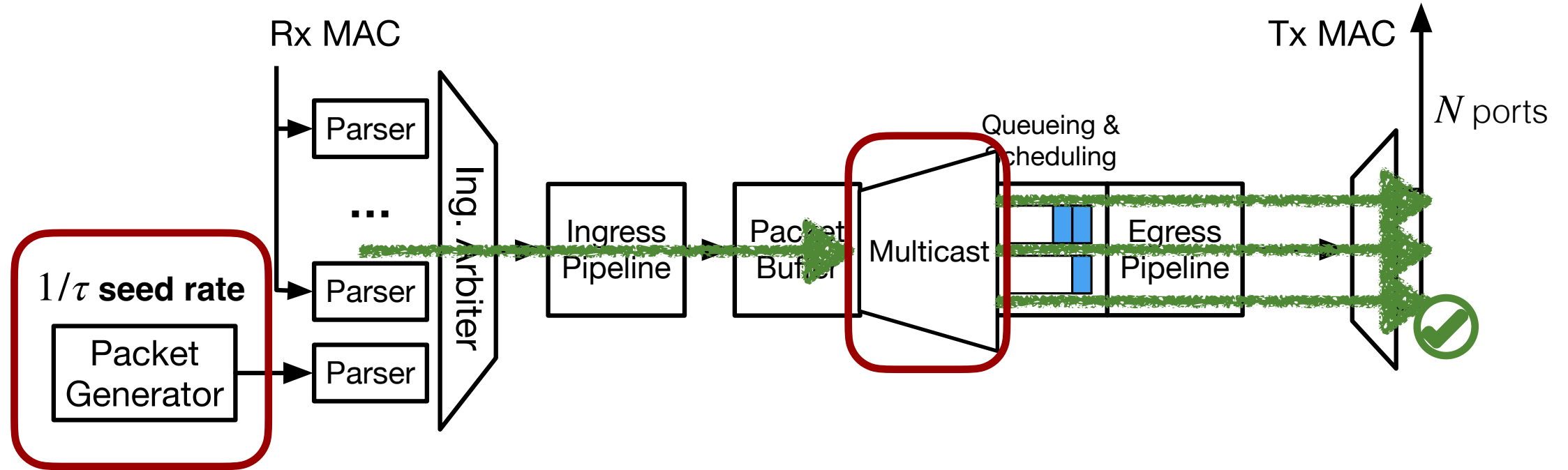
**#1 Scalability:** overwhelm generator capacity to satisfy target rate for all ports

**#2 Cross-traffic contention:** affect throughput, latency, or loss of **user traffic!**

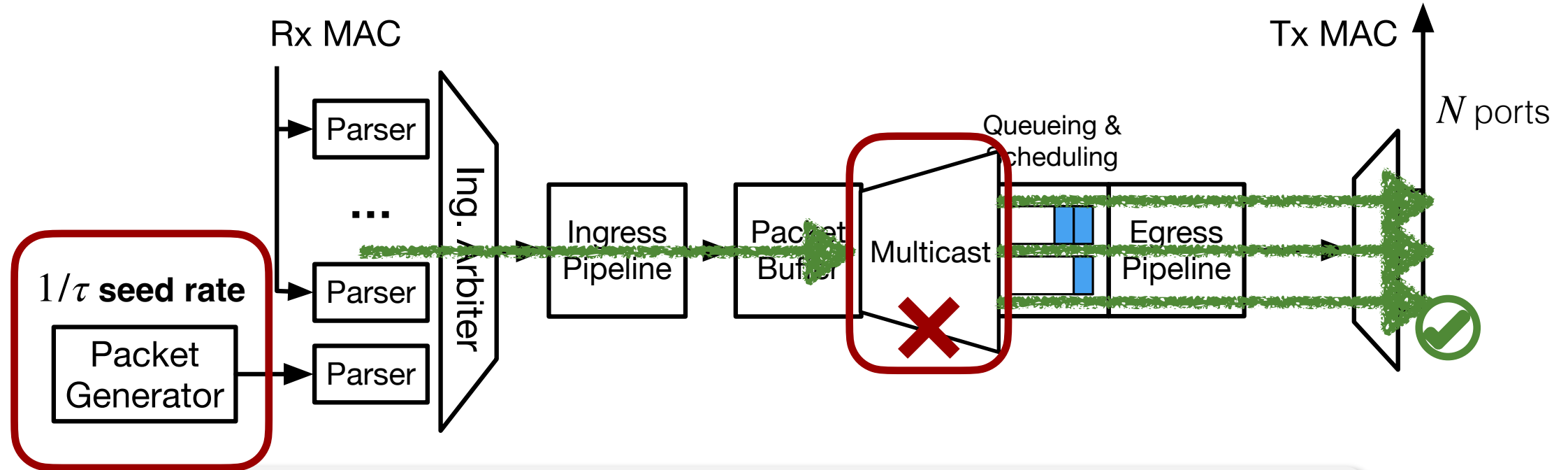


# Problem #1 : scalability

**Solution:** *seed stream amplification*



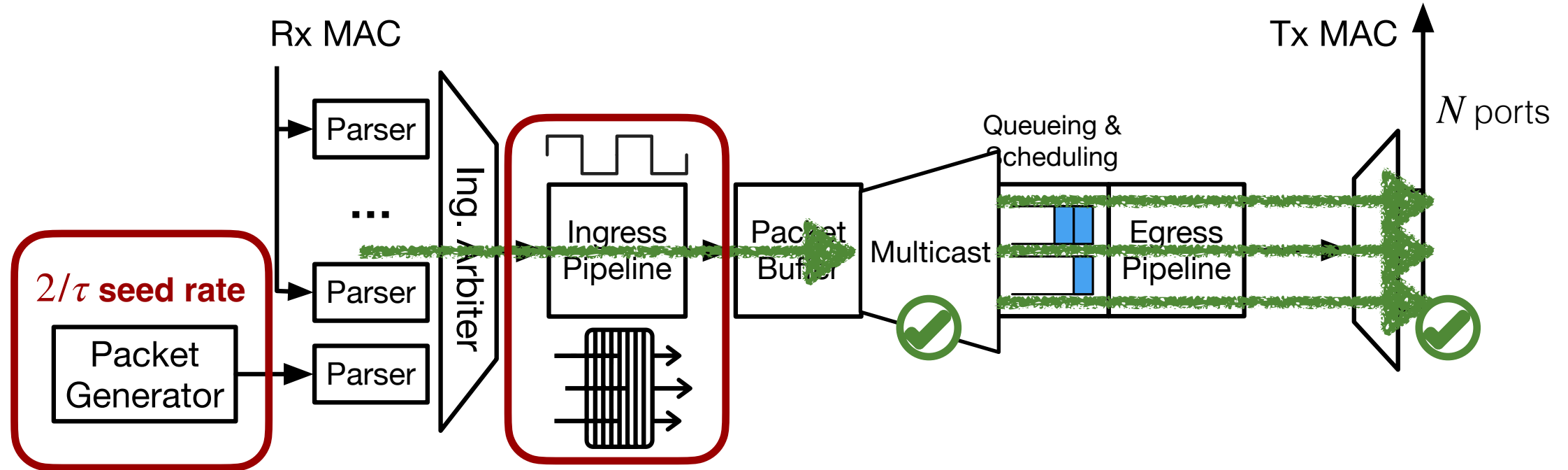
## Problem #2: cross-traffic contention at PRE



**Monopolize usage and waste PRE packet-level BW!**

# Problem #2: cross-traffic contention at PRE

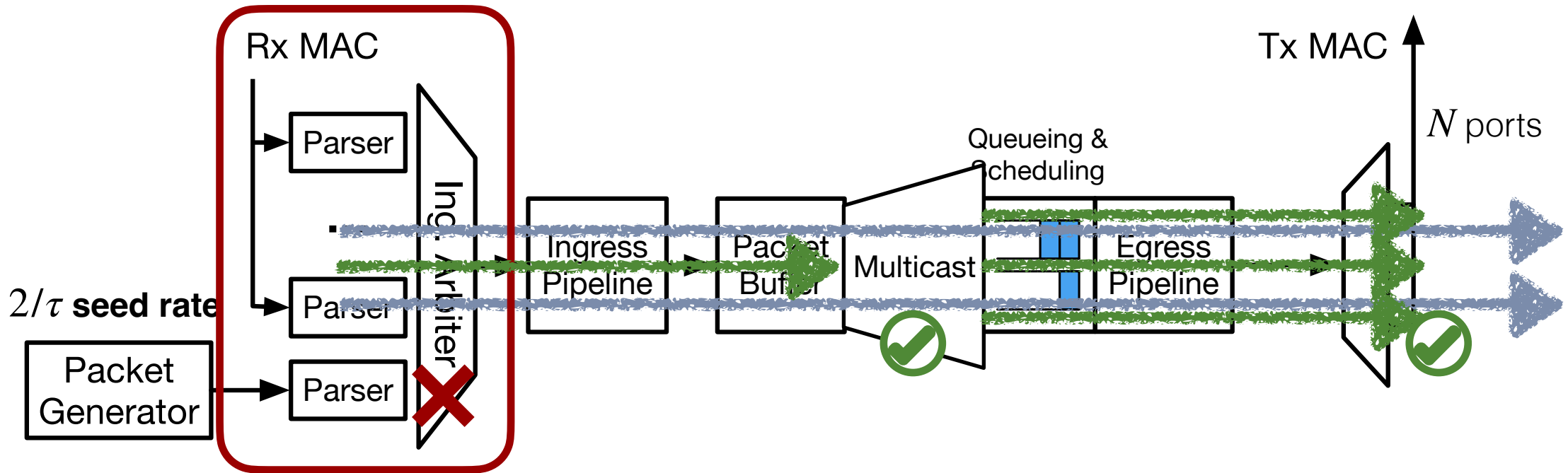
**Solution:** amplify seed stream **on-demand**



## **Selective filtering**

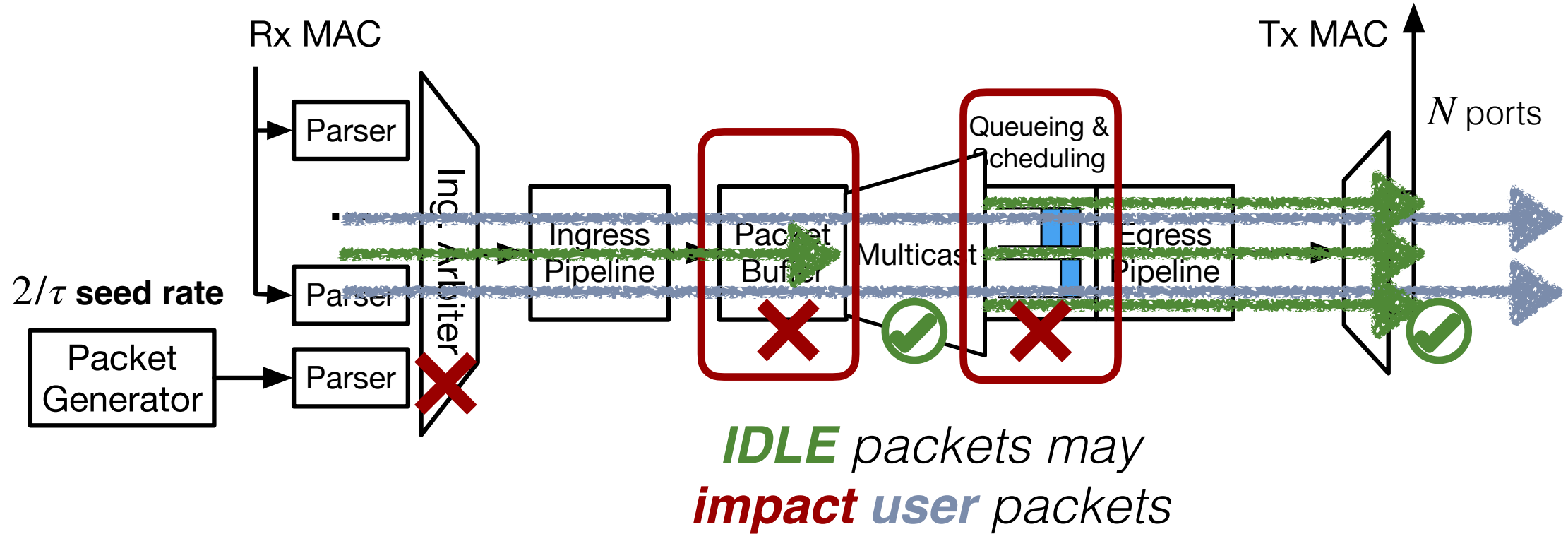
- Per-egress port bitmap indicating packet presence in the last  $\tau/2$  cycle
- If not, replicate an IDLE to the port

# Problem: other contention points



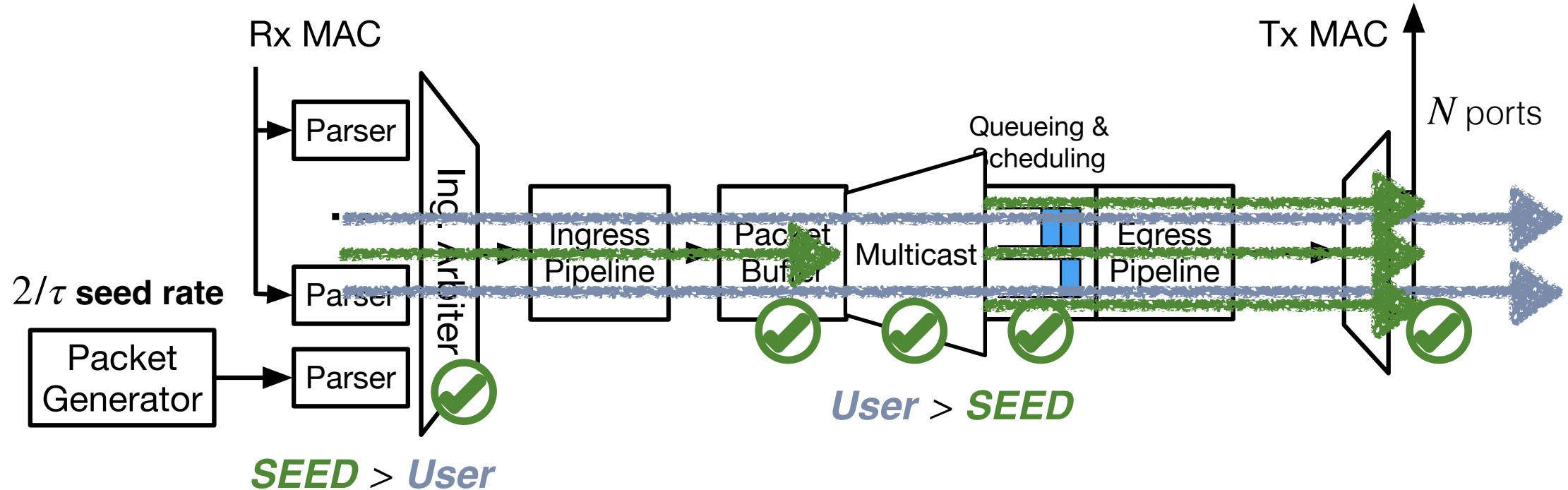
*User* packets may  
**starve** **SEED** packets

# Problem: other contention points



# Problem: other contention points

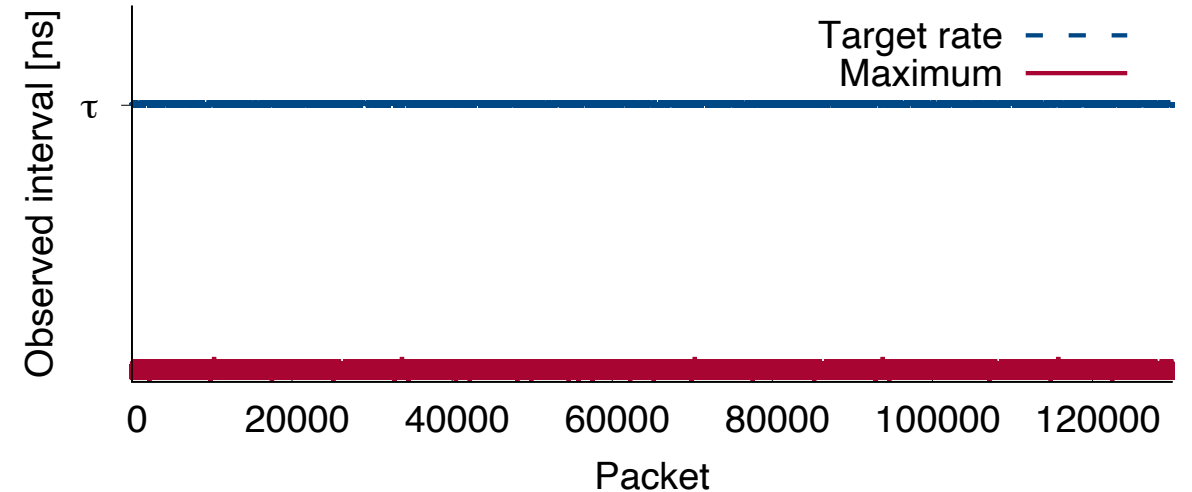
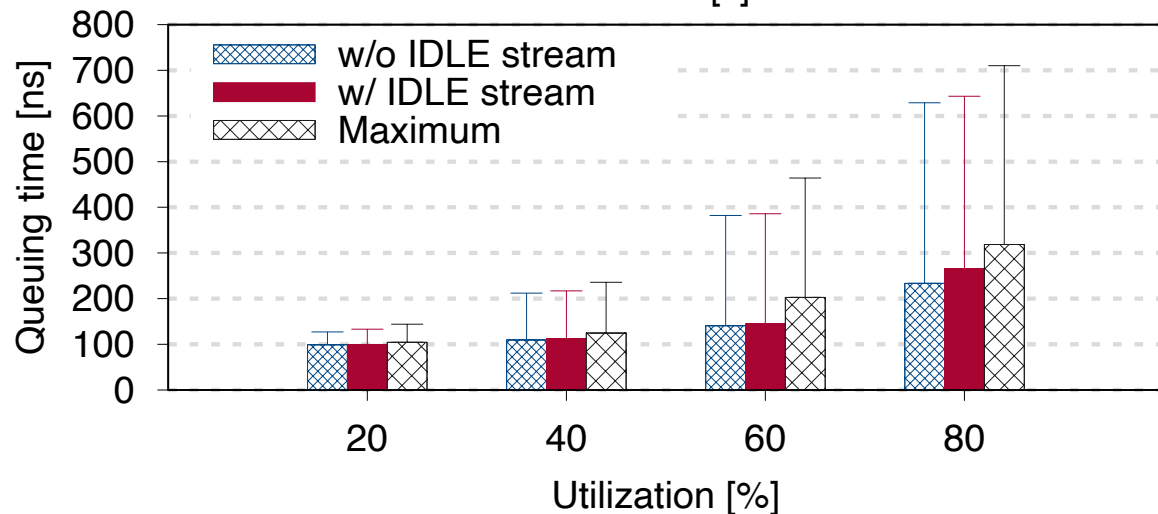
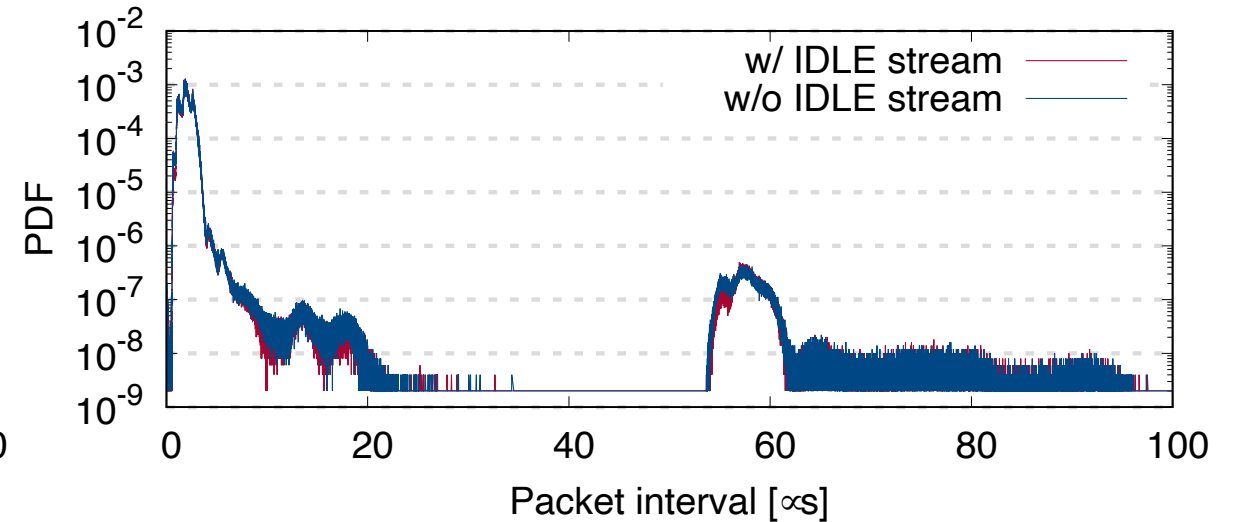
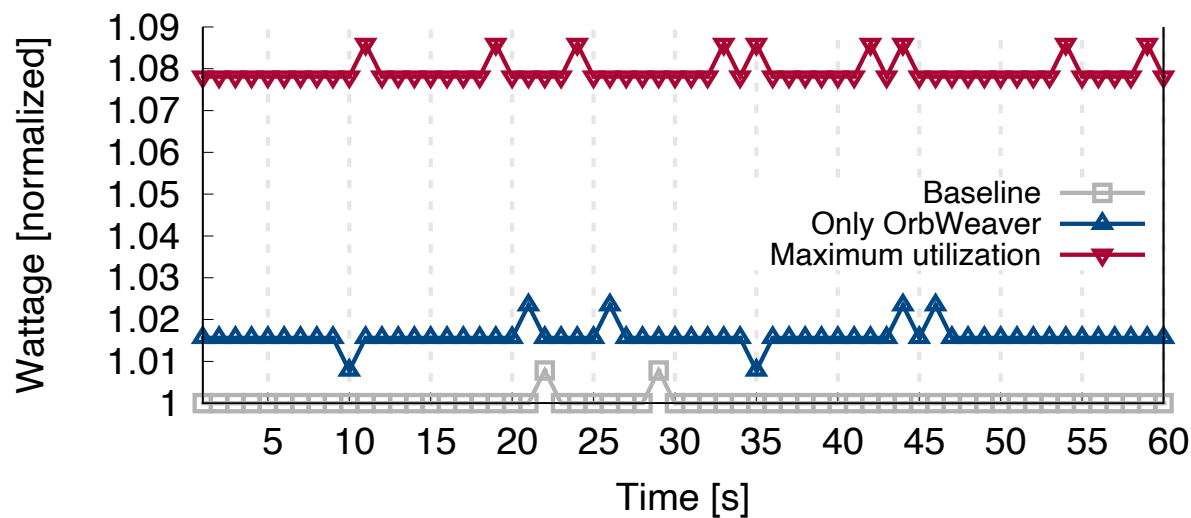
**Solution:** leverage rich configuration options for priorities and buffer management



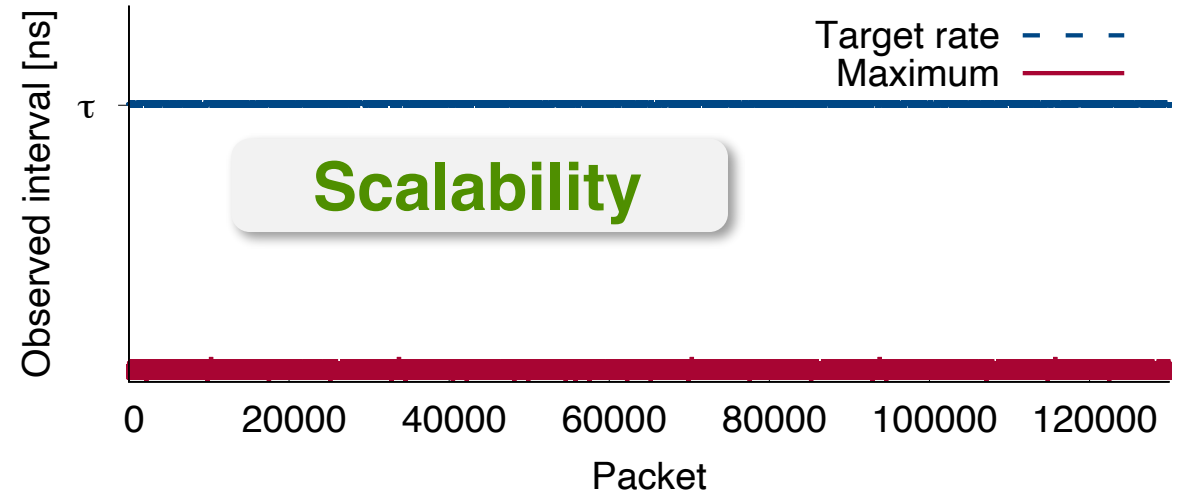
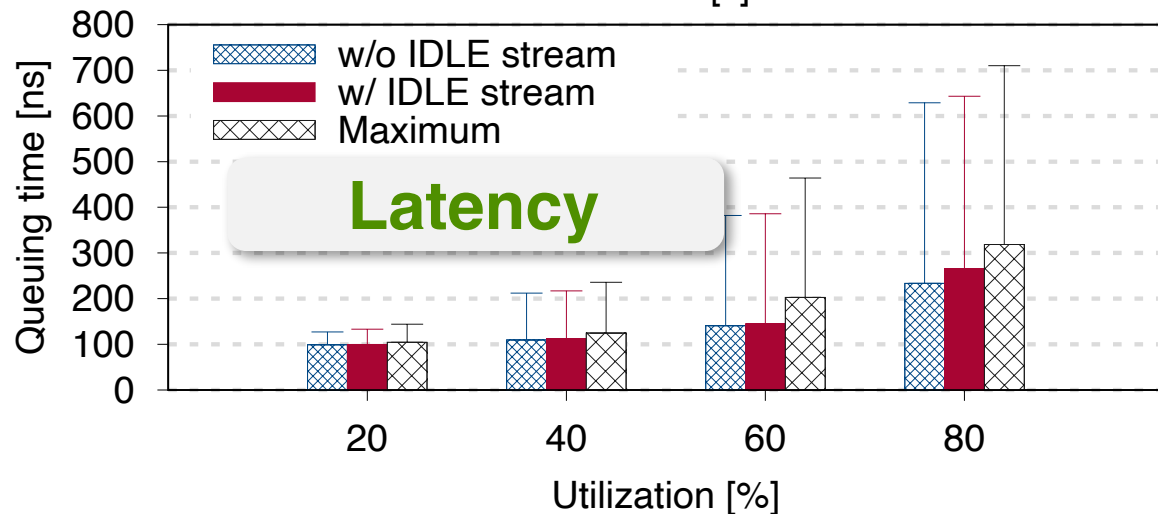
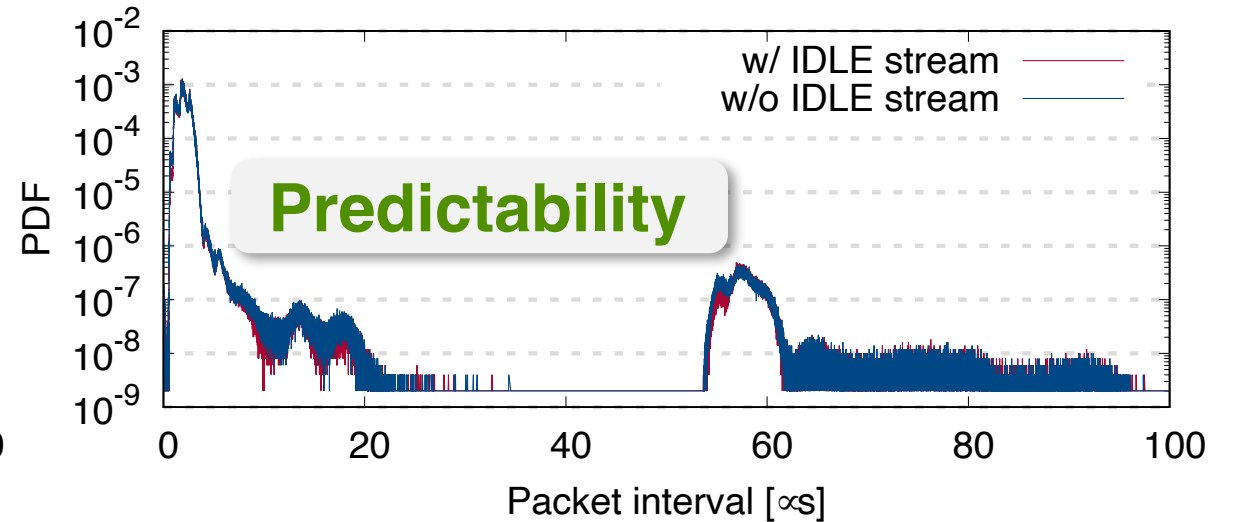
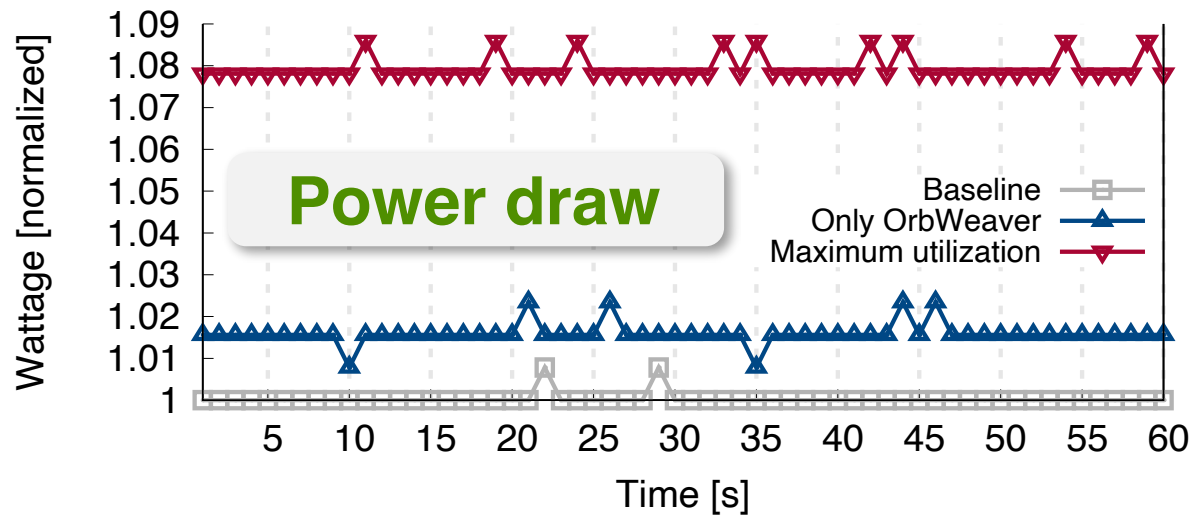
- Zero impact of weaved stream predictability
- Zero impact of **user traffic** throughput or buffer usage
- Negligible impact of latency of **user packets**

# Implementation and evaluation

*Hardware prototype on a pair of Wedge100BF-32X Tofino switches*

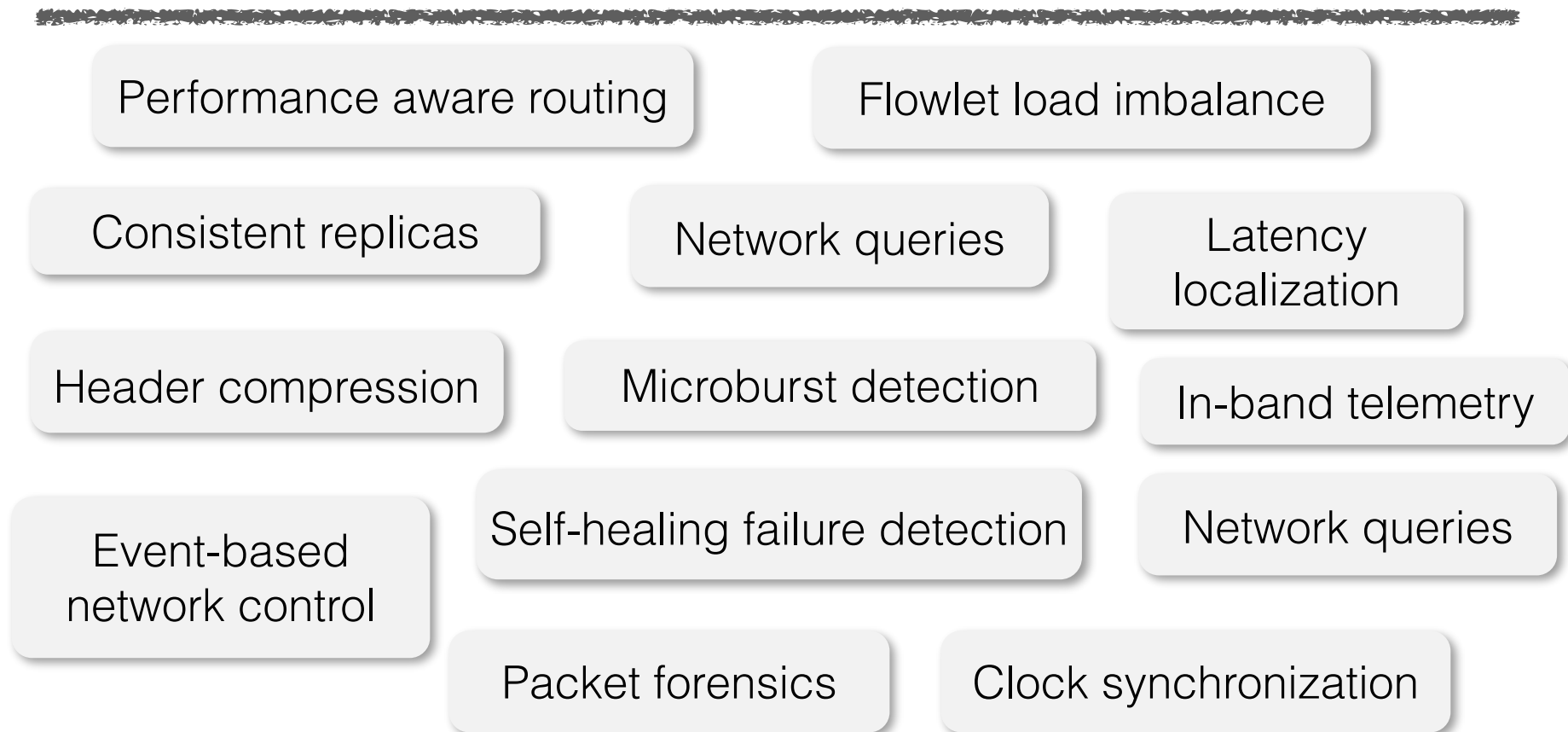


**Takeaway:** **Little-to-no impact** of power draw, latency, or throughput while guaranteeing **predictability** of the weaved stream!





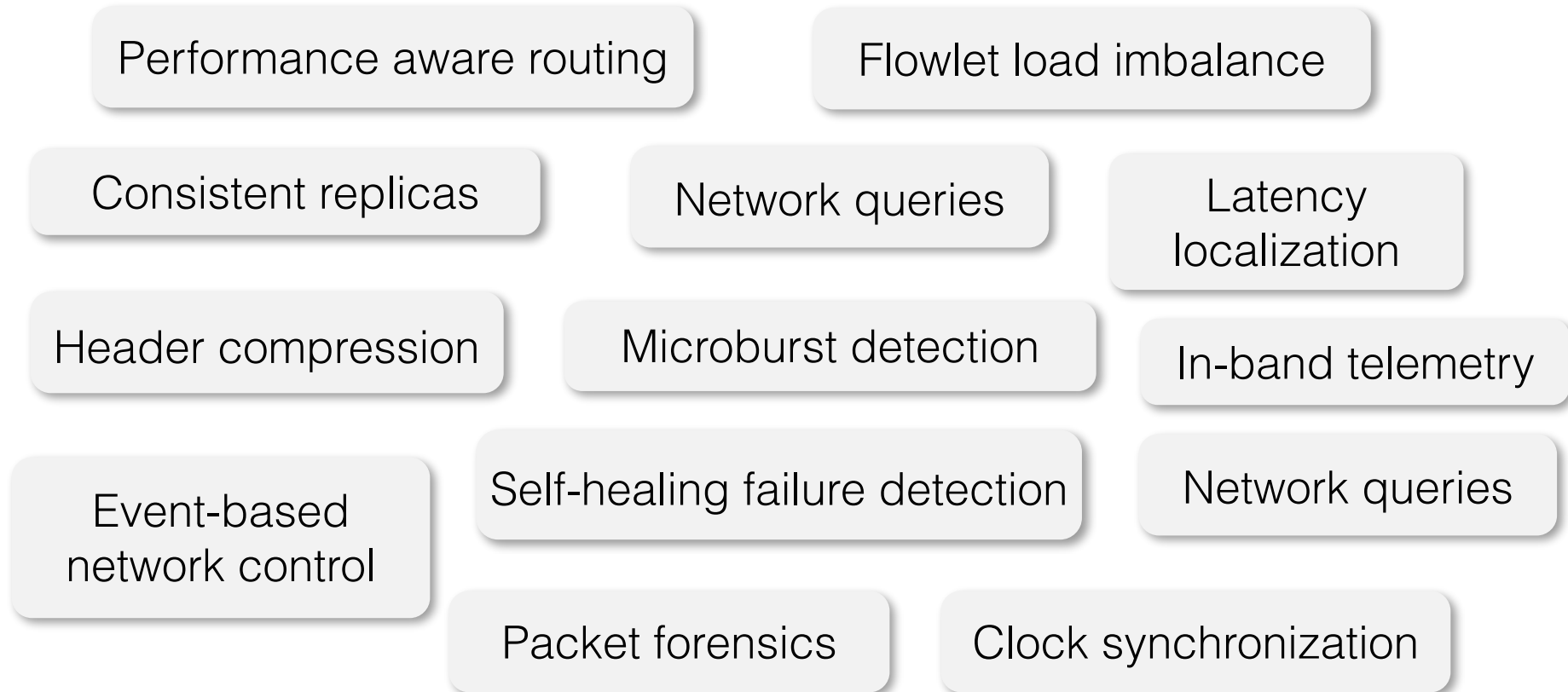
# OrbWeaver use cases



# OrbWeaver use cases



## ***Fine-grained network state inference [R1]***



# OrbWeaver use cases



***Free information  
dissemination [R2]***



***Fine-grained network  
state inference [R1]***

Performance aware routing

Flowlet load imbalance

Consistent replicas

Network queries

Latency  
localization

Header compression

Microburst detection

In-band telemetry

Event-based  
network control

Self-healing failure detection

Network queries

Packet forensics

Clock synchronization

# OrbWeaver use cases



***Free information  
dissemination [R2]***



***Fine-grained network  
state inference [R1]***

Performance aware routing

Flowlet load imbalance

Consistent replicas

Network queries

Latency  
localization

Header compression

Microburst detection

In-band telemetry

Event-based  
network control

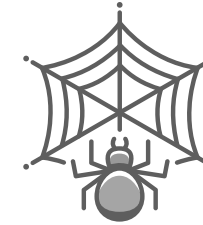
Self-healing failure detection

Network queries

Packet forensics

Clock synchronization

# Failure detection with OrbWeaver

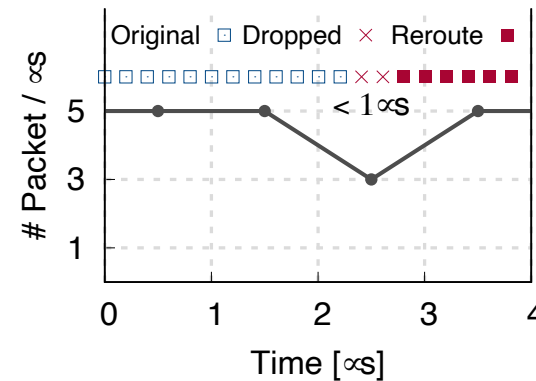
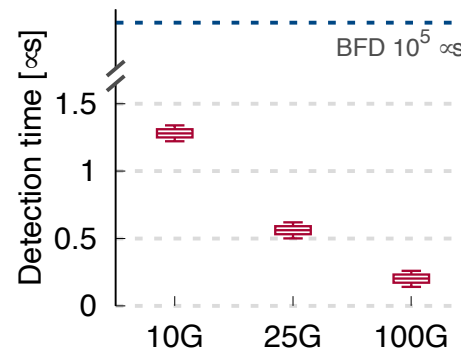


*Before: Weak guarantee of the messaging channel*

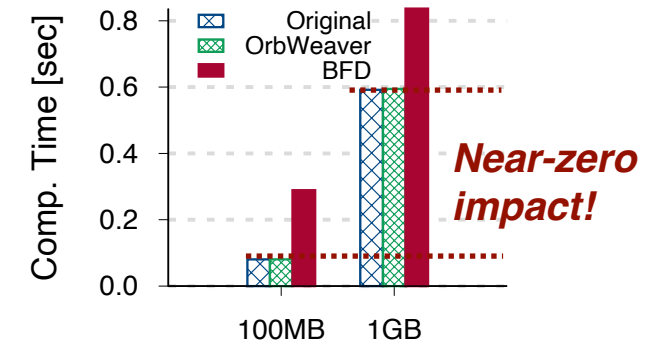
*After: OrbWeaver's weaved stream abstraction guarantees maximum inter-packet gap (120ns for 100 GbE)*



*Emulated failures with optical attenuators tested under varying link speeds*



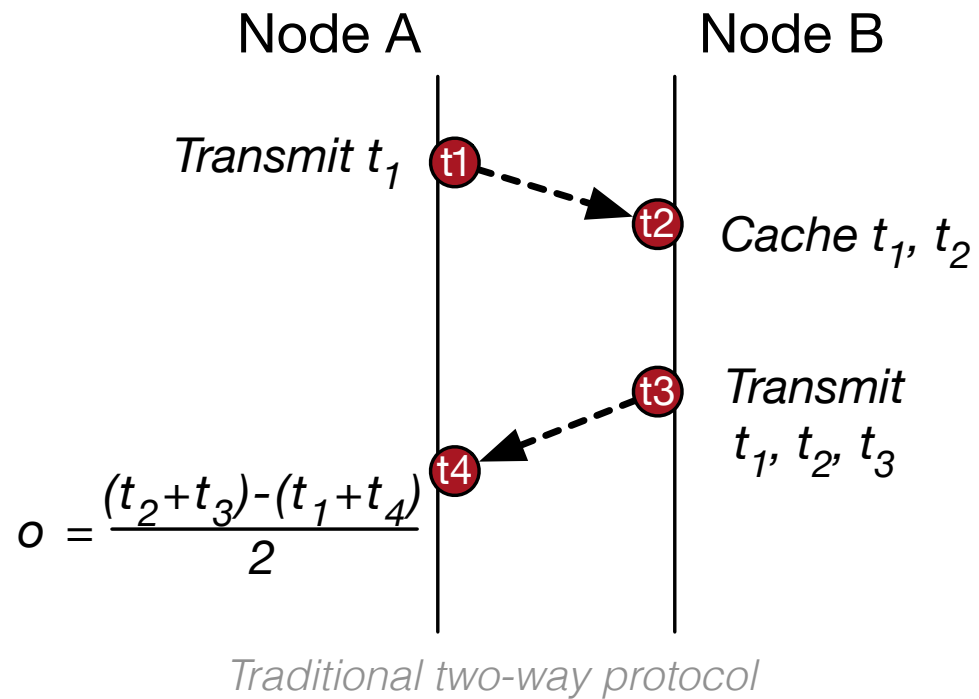
*Combining it with data-plane reroute*



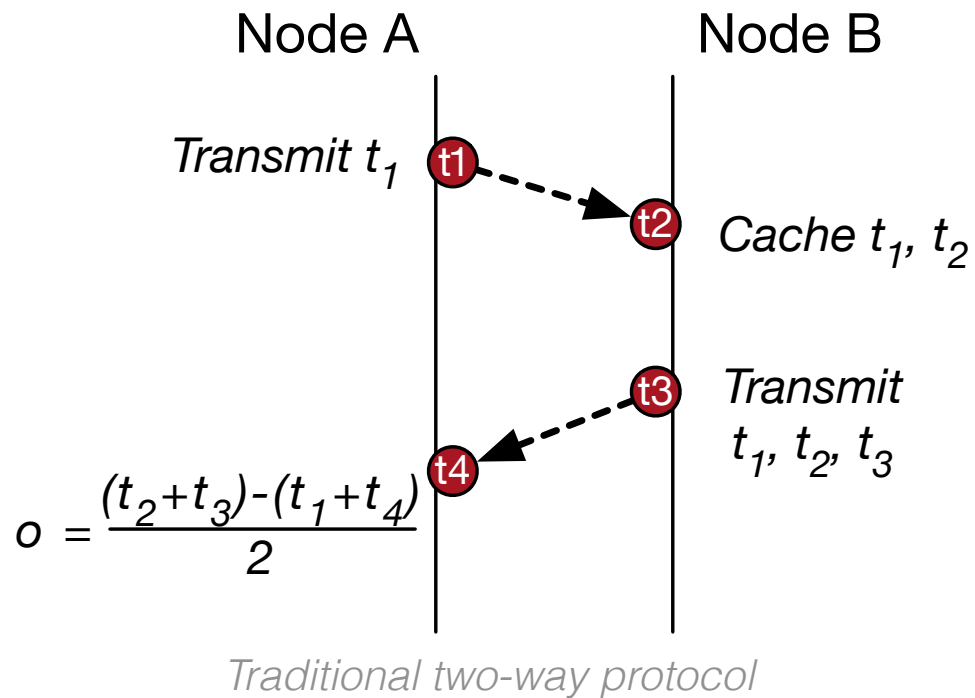
**Near-zero impact!**

Push the detection speed to its **limits** toward instantaneous, self-healing failure mitigation

# Example: time synchronization



# Example: time synchronization

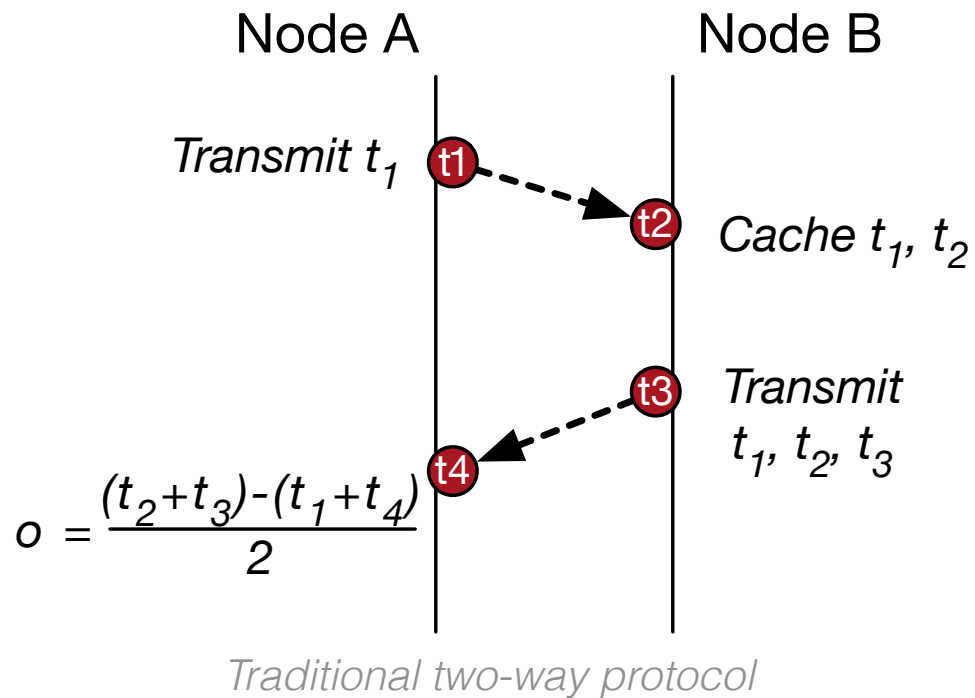


## *Existing approaches for high precision*

- Require special hardware (such as DTP)
- Require messaging overheads (such as DPTP)



# Example: time synchronization



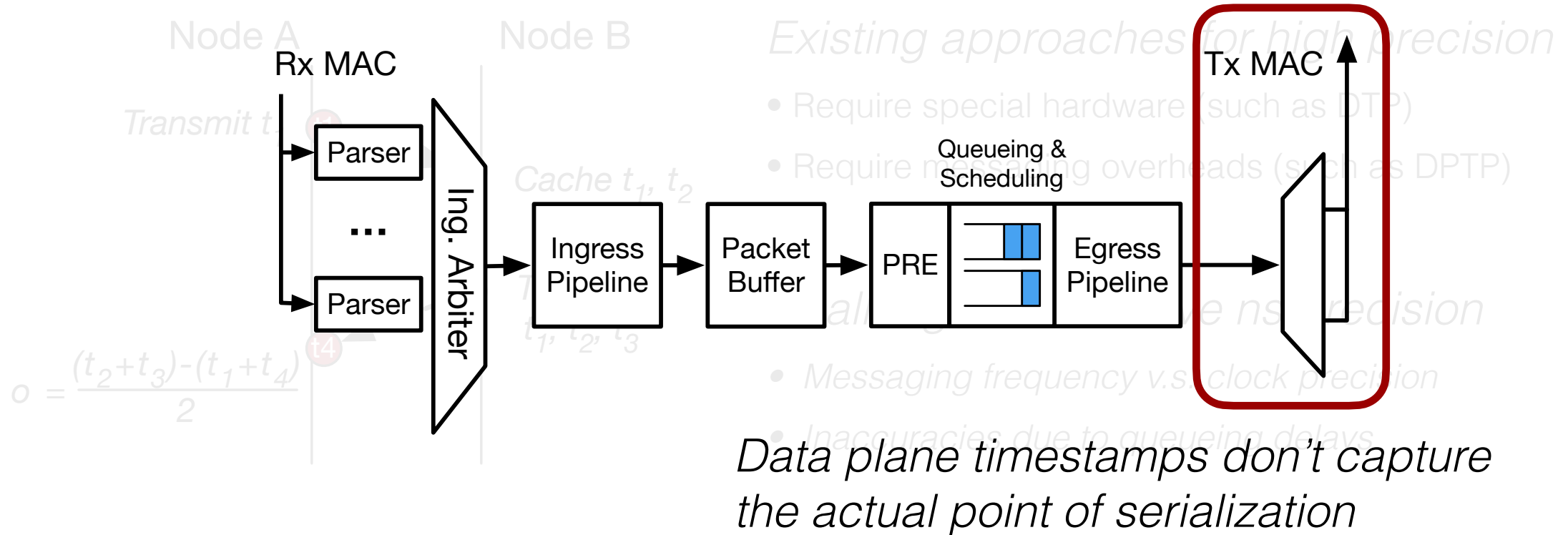
## *Existing approaches for high precision*

- Require special hardware (such as DTP)
- Require messaging overheads (such as DPTP)

## *Challenges to achieve ns precision*

- Messaging frequency v.s. clock precision
- Inaccuracies due to queueing delays

# Example: time synchronization



# OrbWeaver redesign

## Key ideas:

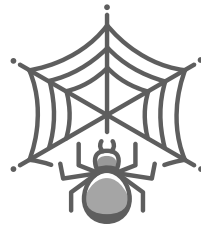
1. Embed timestamp information in **free IDLE packets** [R2]

# OrbWeaver redesign

## Key ideas:

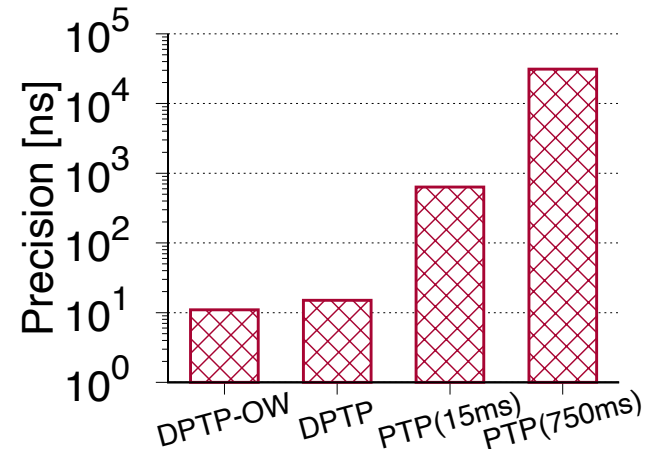
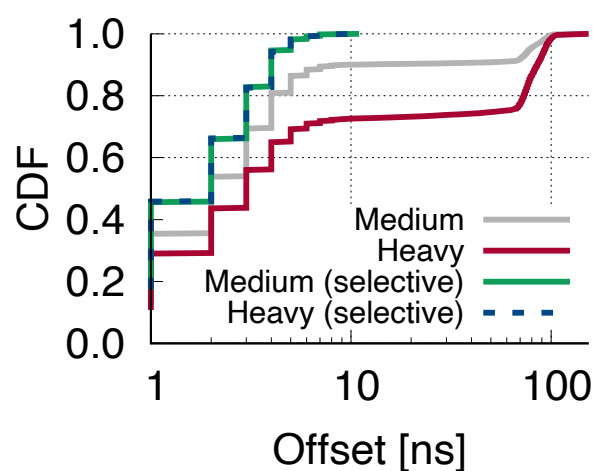
1. Embed timestamp information in **free IDLE packets** [R2]
2. Selective synchronization: **infer queue delay** from IDLE gaps and filter out **unreliable messages** [R1]

# OrbWeaver redesign



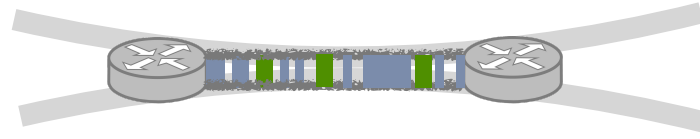
## Key ideas:

1. Embed timestamp information in **free IDLE packets** [R2]
2. Selective synchronization: **infer queue delay** from IDLE gaps and filter out **unreliable messages** [R1]



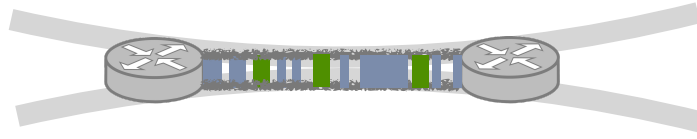
Achieve same or better performance with close-to-zero overheads

# OrbWeaver: summary



- **Weaved stream abstraction** to harvest IDLE cycles
  - Push the utilization of IDLE cycles to its *limits*
  - Guarantee predictability with little-to-zero overhead

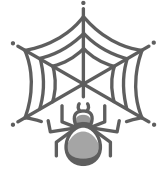
# OrbWeaver: summary



- **Weaved stream abstraction** to harvest IDLE cycles
  - Push the utilization of IDLE cycles to its *limits*
  - Guarantee predictability with little-to-zero overhead
- Generic support of a wide range of data plane applications for free
  - *Don't* need to choose between coordination fidelity and bandwidth overhead
  - **Broader implications:** rethink the design of distributed coordination protocols



# Outline

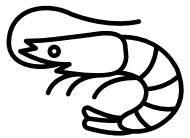


## **OrbWeaver** (*NSDI 2022*)

Reusing IDLE link cycles for in-band control communication

**Reuse**

Zero-waste  
designs



## **Mantis** (*SIGCOMM 2020*)

Recycling switch resources for flexible, sub-RTT reactions

**Recycle**



## **Beaver** (*OSDI 2024*)

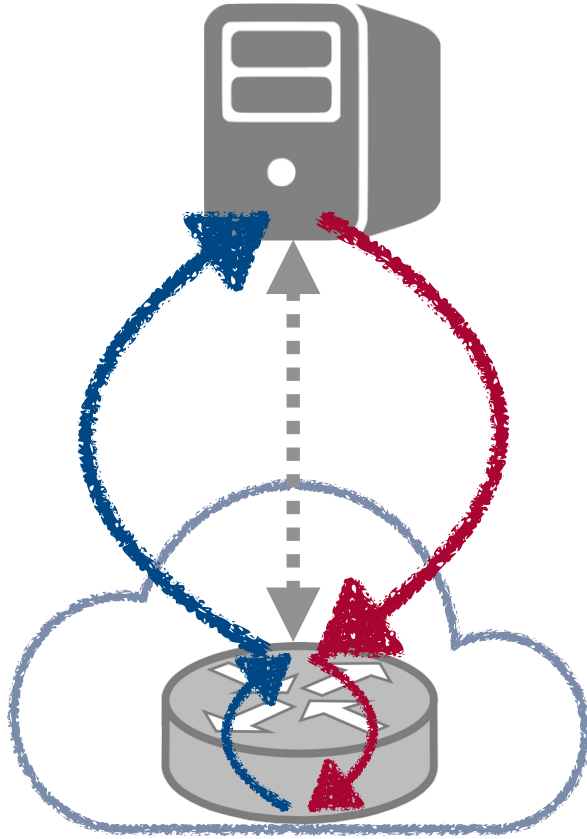
Reducing 'tax' of partial snapshots for distributed cloud services

**Reduce**

# Today's networks react

- A common task: ***reacting*** to current network conditions
  - Detecting failures and then rerouting
  - Identifying malicious flows and then filtering
  - Recognizing load imbalance and then adjusting
- In data centers, reactions need be fast

# Today's primitives for reaction



SDNs or conventional control loops

*Flexible but slow*

Built-in data plane primitives

*Fast but restrictive*

Programmable switches?

Constraints on operations in actions, number of stages, SRAM accesses, egress/ingress communication, in-band match-action updates...

# Today's primitives for reaction

Can we enable fine-grained reactions with minimum **latency** and maximum **flexibility**?



Built-in data plane primitives  
*Fast but restrictive*

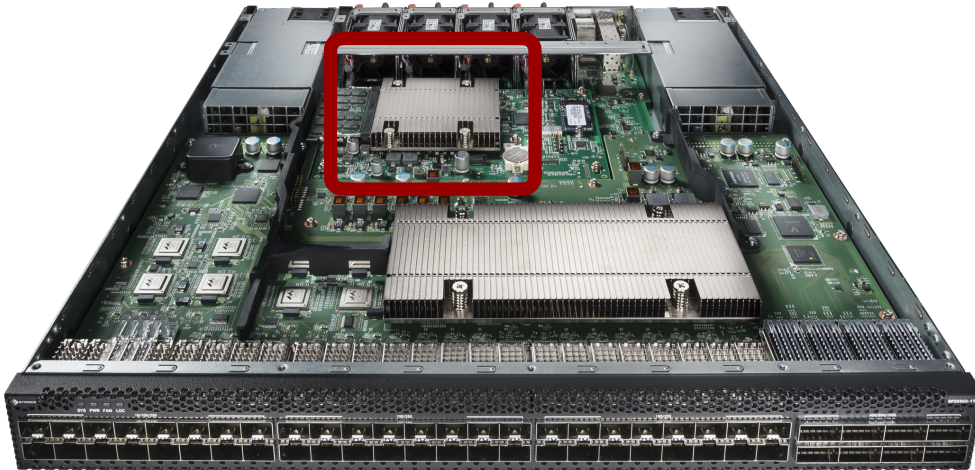
Programmable switches?

Constraints on operations in actions, number of stages, SRAM accesses, egress/ingress communication, in-band match-action updates...

# A peek inside a switch chassis...

## On-board CPU

ONIE, Debian/ONL, SONiC



- More **capable** with higher BW switching ASICs
  - Physical cores: 2→4→8
- Underlying workloads involve **out-of-band, infrequent** executions, e.g., IS-IS, BGP, RSVP, DHCP, LLDP, SNMP

Not part of the general compute pool, **underutilized!**

# Approach

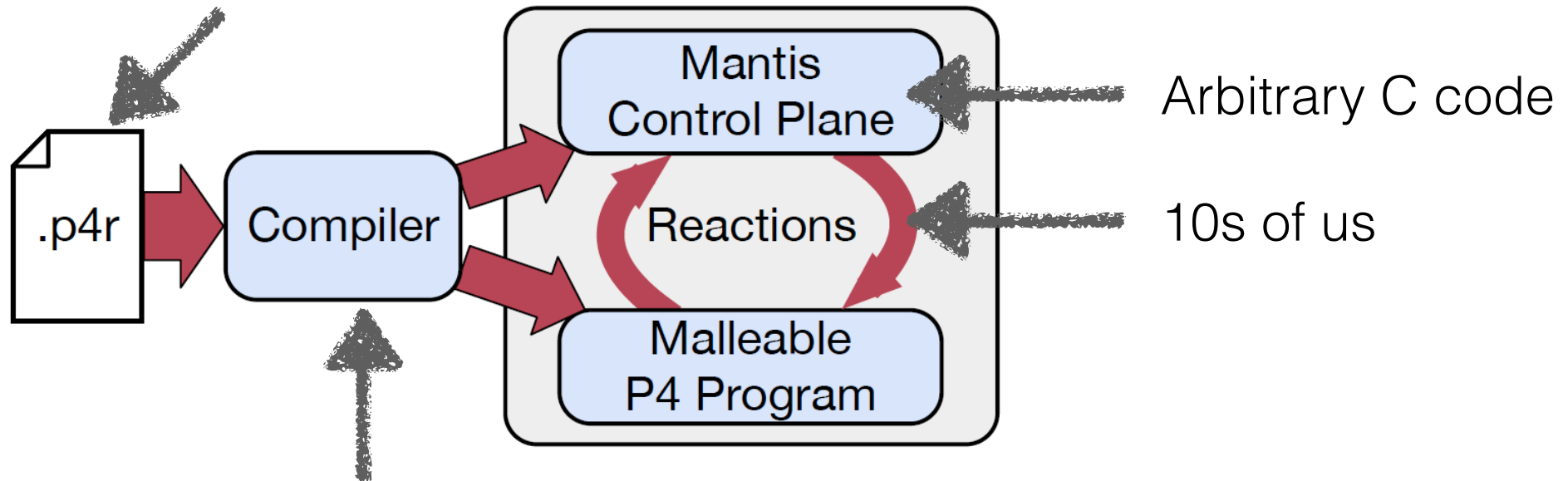
Can we enable fine-grained reactions with minimum ***latency*** and maximum ***flexibility***?

1. Push the reactions as close to the switch ASIC as possible
2. Co-design the data plane program with local CPUs for fine-grained malleability and ease of use

# Mantis overview

*Usable, fast, and expressive in-network reactions on today's RMT switches*

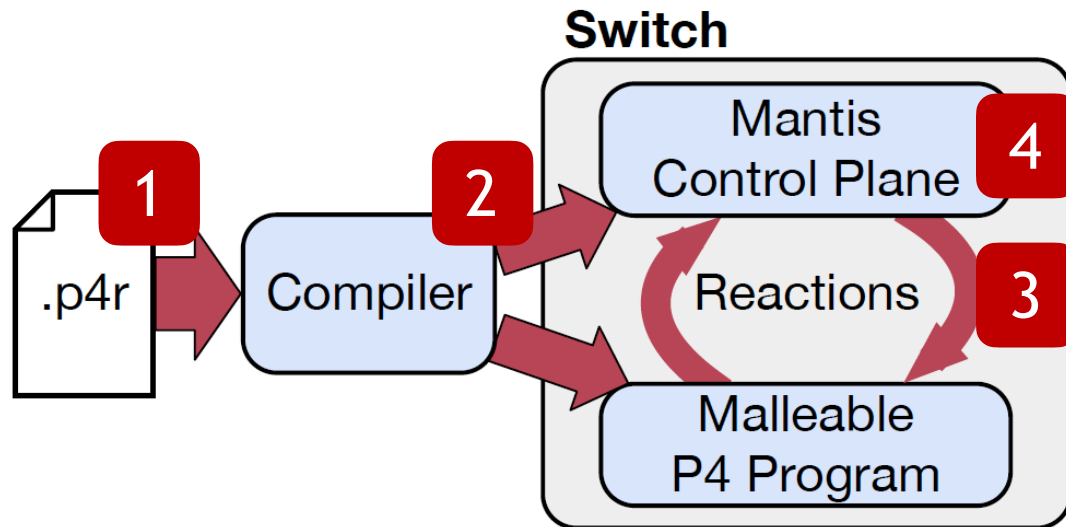
Simple extension to P4



Generates code for 'runtime' reconfigurability/serializability



# Anatomy of Mantis



***M1* Language**

***M2* Translation**

*M3* Isolation

*M4* Execution

# Abstraction

## 1. Malleable entities

- Amenable to fine-grained reconfiguration at runtime

## 2. Reactions

- Package reaction logic into a C-like function

# M1 : start with P4 code

foo.p4

```
table my_table {  
    reads { ipv4.dst : ternary; }  
    actions { my_action; drop; }  
}  
action my_action() {  
    modify_field(priority, 1);  
}
```

How to make it run time reconfigurable?

# M1 : P4R example

foo.p4*r*

```
table my_table {  
    reads { ipv4.dst : ternary; }  
    actions { my_action; drop; }  
}  
action my_action() {  
    modify_field(priority, 1);  
}
```

# M1 : P4R example

foo.p4**r**

```
malleable value prio_var {  
    width : 16; init : 1;  
}  
  
table my_table {  
    reads { ipv4.dst : ternary; }  
    actions { my_action; drop; }  
}  
  
action my_action() {  
    modify_field(priority, ${prio_var});  
}
```

Declaring malleable entities

Previous P4 code with references  
to malleable entities

# M1 : P4R example

foo.p4r

```
malleable value prio_var {  
    width : 16; init : 1;  
}  
  
table my_table {  
    reads { ipv4.dst : ternary; }  
    actions { my_action; drop; }  
}  
  
action my_action() {  
    modify_field(priority, ${prio_var});  
}  
  
reaction my_reaction(reg re_qdepths[1:10]){  
    uint16_t cur_max = 0;  
    for (int i = 1; i <= 10; ++i)  
        if (re_qdepths[i] > cur_max) {  
            cur_max = re_qdepths[i];  
        }  
    if (cur_max > THRESHOLD) {  
        ${prio_var} = 5;  
    }  
}
```

Declaring malleable entities

Previous P4 code with references  
to malleable entities

Specifying reaction arguments

Reaction with arbitrary C

Reconfiguration

# M1: P4R example

foo.p4r

```
malleable value prio_var {
    width : 16; init : 0;
}

table my_table {
    reads { ipv4.dst : 1; }
    actions { my_action; }
}

action my_action() {
    modify_field(prio_var, 5);
}

reaction my_reaction() {
    uint16_t cur_max = 0;
    for (int i = 1; i <= 10; i++) {
        if (re_qdepths[i] > cur_max) {
            cur_max = re_qdepths[i];
        }
    }
    if (cur_max > THRESHOLD) {
        ${prio_var} = 5;
    }
}
```

## Malleable entities

- Malleable value
- Malleable field (table match, action...)
- Malleable table

## Reaction function arguments

- Register
- Field
- Malleable field

Malleable entities

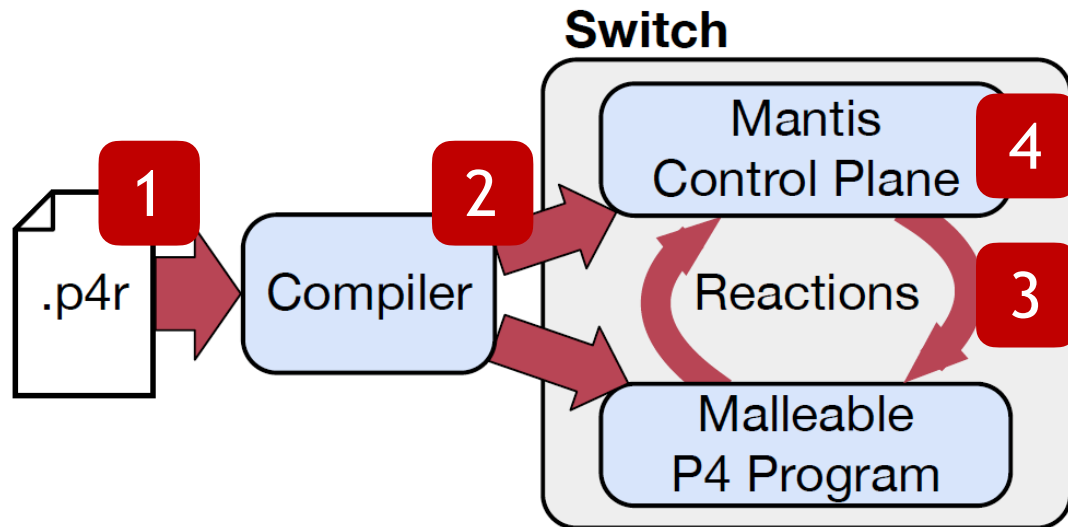
Table with references to entities

Reaction arguments

Reaction with arbitrary C

Reconfiguration

# Anatomy of Mantis



- M1* Language
- M2* Translation
- M3* Isolation
- M4* Execution



# M2: P4R transformation

foo.p4r

```
malleable value prio_var {
    width : 16; init : 1;
}

table my_table {
    reads { ipv4.dst : ternary; }
    actions { my_action; drop; }
}

action my_action() {
    modify_field(priority, ${prio_var});
}

reaction my_reaction(reg re_qdepths[1:10]){
    uint16_t cur_max = 0;
    for (int i = 1; i <= 10; ++i)
        if (re_qdepths[i] > cur_max) {
            cur_max = re_qdepths[i];
        }
    if (cur_max > THRESHOLD) {
        ${prio_var} = 5;
    }
}
```

Preparing registers for a  
***pull-based model***

# M2: P4R transformation

foo.p4r

```
malleable value prio_var {  
    width : 16; init : 1;  
}  
table my_table {  
    reads { ipv4.dst : ternary; }  
    actions { my_action; drop; }  
}  
action my_action() {  
    modify_field(priority, ${prio_var});  
}
```

Generalize user-specified knobs for “**hitless**” reconfiguration

# M2: P4R transformation

foo.p4r

```
malleable value prio_var {  
  width : 16; init : 1;  
}  
table my_table {  
  reads { ipv4.dst : ternary; }  
  actions { my_action; drop; }  
}  
action my_action() {  
  modify_field(priority, ${prio_var}p4r_meta_.prio_var);  
}  
header_type p4r_meta_t_ {  
  field {prio_var : 16;}  
}  
metadata p4r_meta_t_ p4r_meta_;
```

Replacing the malleable  
value

# M2: P4R transformation

foo.p4r

```
malleable value prio_var {  
  width : 16; init : 1;  
}  
table my_table {  
  reads { ipv4.dst : ternary; }  
  actions { my_action; drop; }  
}  
action my_action() {  
  modify_field(priority, #{prio_var}p4r_meta_.prio_var);  
}
```

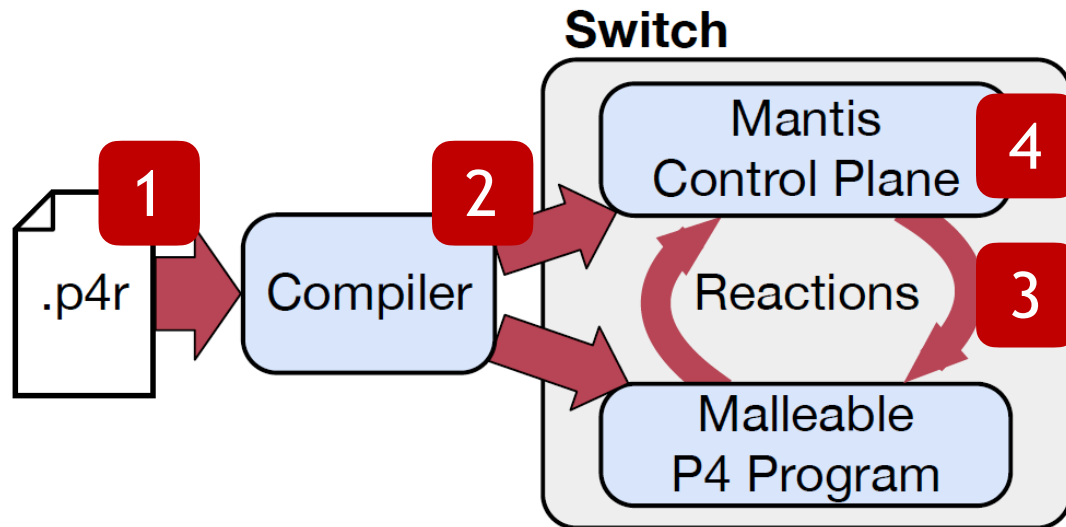
```
header_type p4r_meta_t_ {  
  field {prio_var : 16;}  
}  
metadata p4r_meta_t_ p4r_meta_;
```

```
table p4r_init_ {  
  actions {p4r_init_action_;}  
  size : 1;  
}  
action p4r_init_action_(prio_var) {  
  modify_field(p4r_meta_.prio_var, prio_var);  
}
```

Replacing the malleable  
value

Multi-purpose initialization table

# Anatomy of Mantis



*M1* Language

*M2* Translation

*M3* Isolation

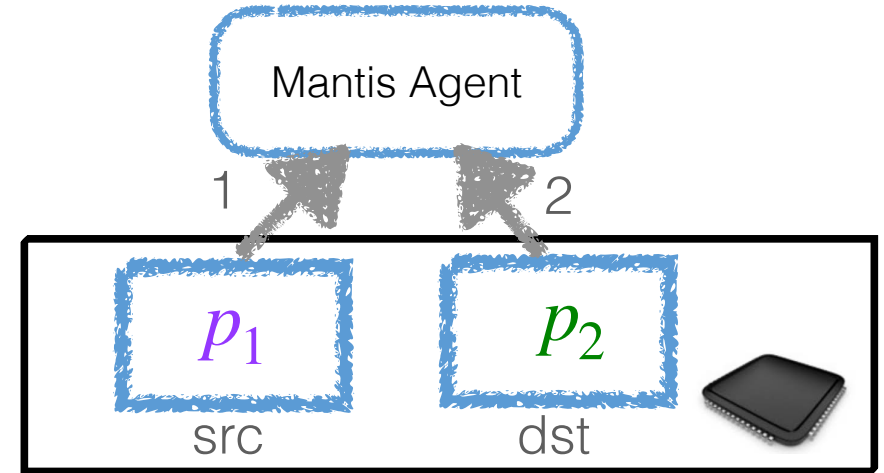
*M4* Execution

# M3: Isolation (ACID)

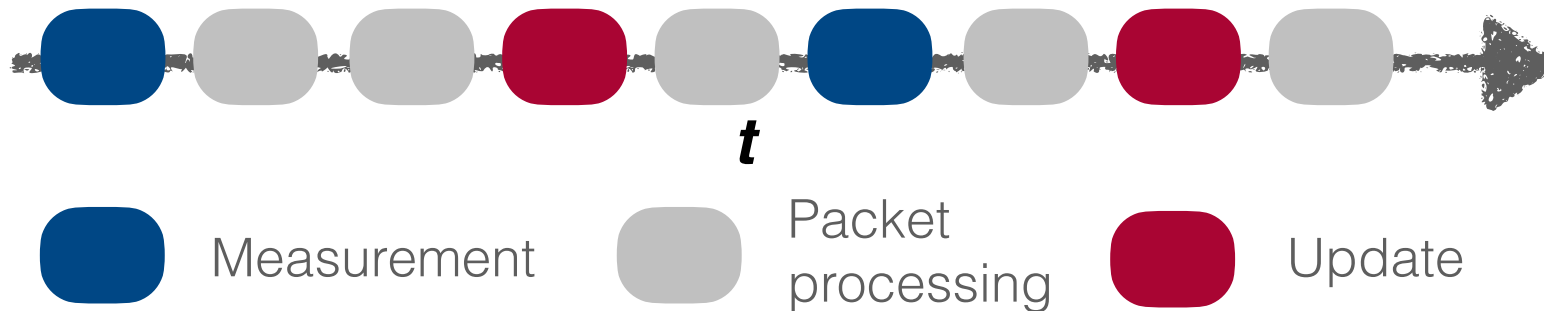
Isolation **matters**, consider

```
reaction my_reaction(reg src, reg dst){}
```

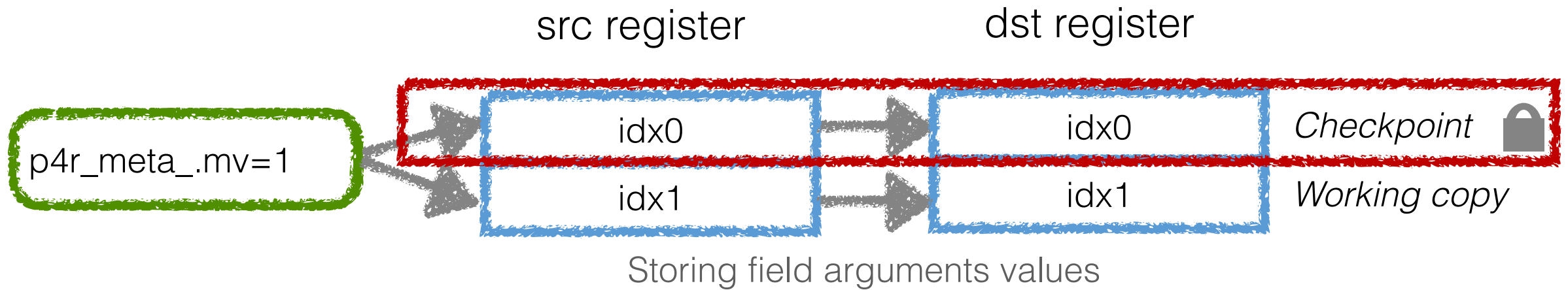
- Expectation:  $src \leftarrow p_1, dst \leftarrow p_1$
- Without isolation:  $src \leftarrow p_1, dst \leftarrow p_2$



Mantis enforces *per-pipeline, per-reaction* serializable isolation

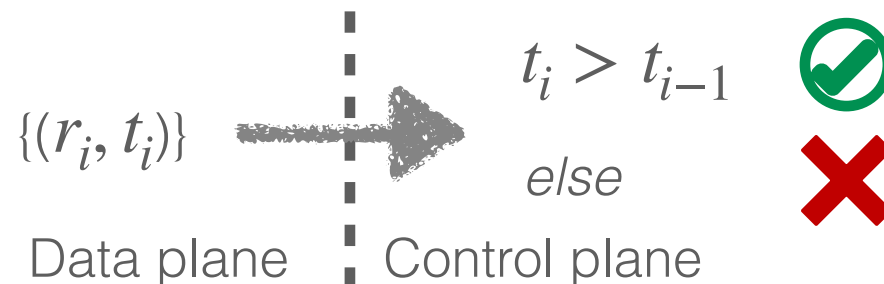


# M3: Isolating measurement



For a register, **at most** one element will be updated on a packet thread  
*Stale values* may appear in the current checkpoint for register arguments


Timestamps  $t_i$  appended  
to the duplicate buffer



# M3: Isolating updates

Three-phase updates for isolating *fast*, *repeated*, *partial* updates


vv=0 (exact match)



Match	Action
hdr.a=0, vv=0	my_action(0)
hdr.a=0, vv=1	my_action(0)
hdr.a=1, vv=0	my_action(1)
hdr.a=1, vv=1	my_action(1)

From previous mirror phase


vv=0




Match	Action
hdr.a=0, vv=0	my_action(0)
hdr.a=0, vv=1	my_action(0)
hdr.a=1, vv=0	my_action(1)
hdr.a=1, vv=1	my_action( <b>2</b> )

**Prepare** updates in vv=1 copy for malleable entities

vv=1



Match	Action
hdr.a=0, vv=0	my_action(0)
hdr.a=0, vv=1	my_action(0)
hdr.a=1, vv=0	my_action( <b>2</b> )
hdr.a=1, vv=1	my_action(2)



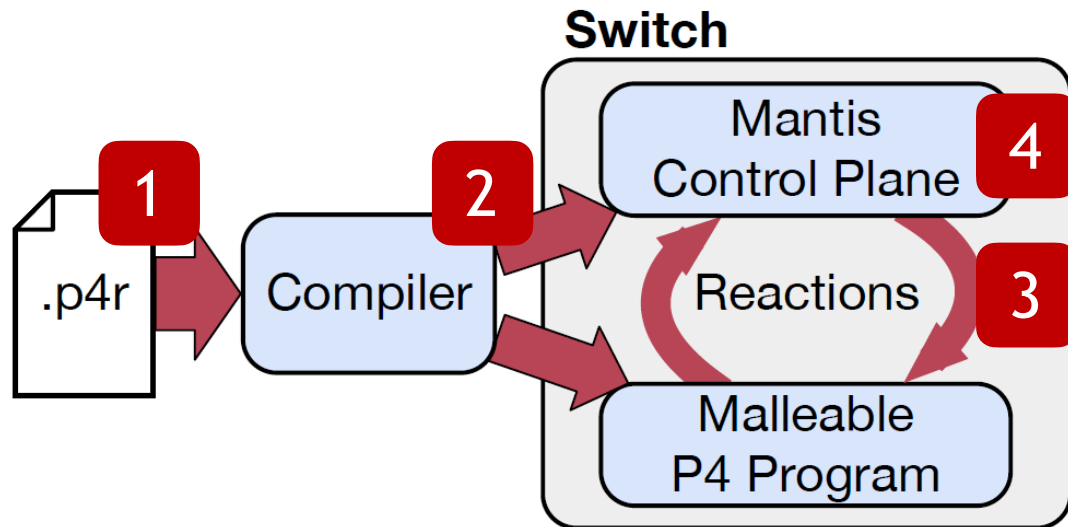
$vv \oplus 1$   
**Commit**

**Mirror** the changes to the shadow copy for amortization

*Bounded* memory overhead and *predictable* latency



# Anatomy of Mantis



*M1* Language

*M2* Translation

*M3* Isolation

*M4* Execution

# M4: Mantis control plane

Traditionally data/control plane interactions are treated as *one-off, isolated* events, i.e., assumed to be “*on the slow path*”

Mantis control plane is instead ***reaction-centric***

```
helper_state = precompute_metadata();  
memo = setup_cache(helper_state);  
run_user_initialization(helper_state, memo);
```

Prologue

```
while(!stopped) {  
    updateTable(memo, "p4r_init_", {measure_ver : mv ^ 1});  
    read_measurements(memo, mv); mv ^= 1;  
    run_user_reaction(memo, helper_state, vv ^ 1);  
    updateTable(memo, "p4r_init_", {config_ver : vv ^ 1});  
    fill_shadow_tables(memo, vv); vv ^= 1;  
}
```

Dialogue

~PCIe latency of the underlying system

# Implementation and evaluation

Prototype implementation on a Wedge100BF-32X Tofino switch

- P4R frontend: Flex/Bison based, ~5000 lines of C++ and grammar
- Mantis agent: dynamic (re)loading of user reaction (.so object)

# Implementation and evaluation




Prototype implementation on a Wedge100BF-32X Tofino switch

- P4R frontend: Flex/Bison based, ~5000 lines of C++ and grammar
- Mantis agent: dynamic (re)loading of user reaction (.so object)

## Evaluation

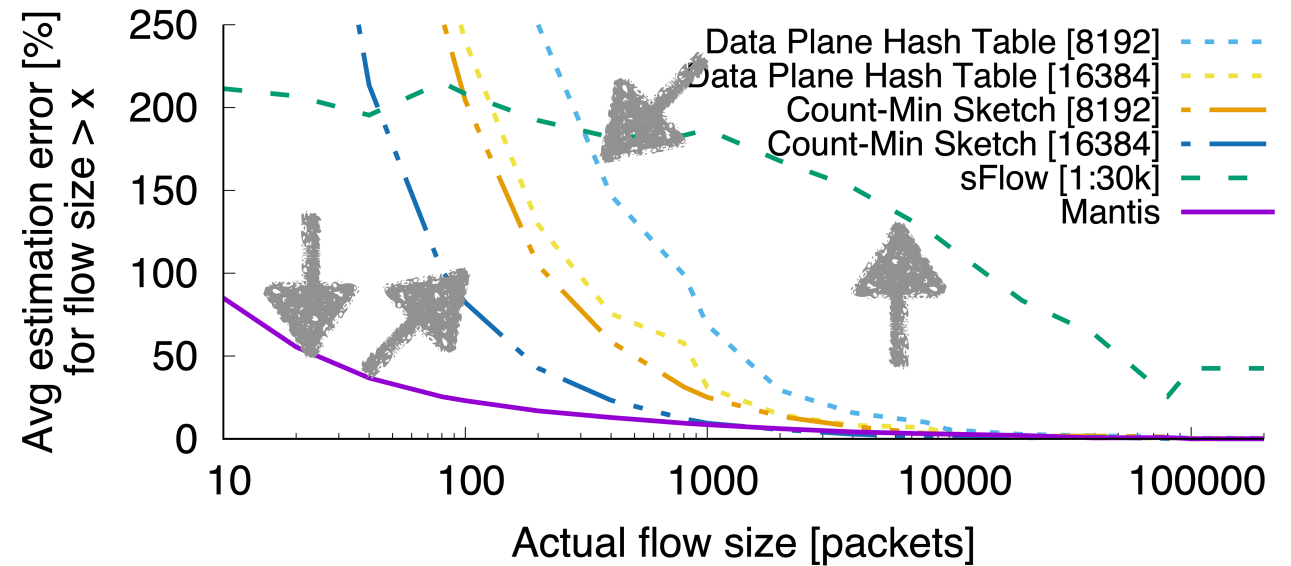
- How fast is Mantis's reaction time?
- What is the overhead?
- What are the applications of Mantis?
- How does Mantis compare to existing alternatives?

# Use cases

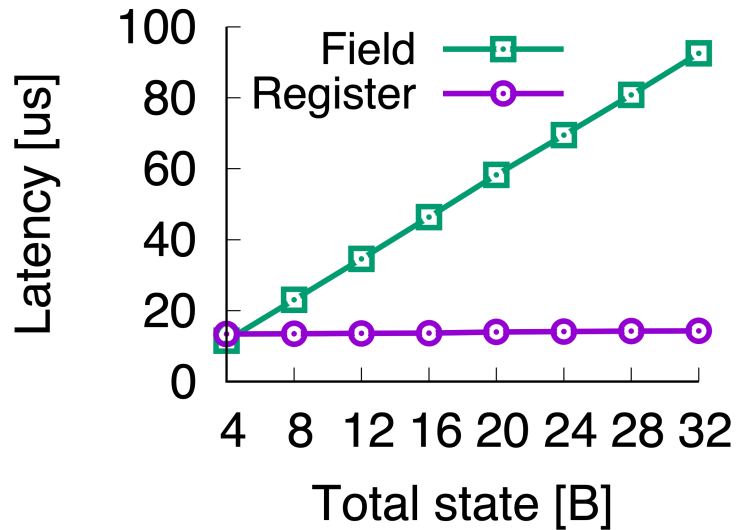
	<i>DoS mitigation</i>	<i>Route Recomputation</i>	<i>Hash polarization mitigation</i>	<i>Reinforcement Learning</i>
 <i>Measurement</i>	Flow signature, packet count	Heartbeat counts, timestamp	Queue depths of ECMP ports	Packet counts and queue depths
 <i>Control logic</i>	Block the sender if the estimated flow size exceeds a threshold	Mark the failed link if received heartbeat number is small than expected after consecutive K confirmations	Change ECMP hashing input to another permutation if found a persistent imbalance of port utilization	Use a Q-learning algorithm to calculate the optimal ECN threshold based on rewards
 <i>Reconfiguration</i>	Drop the malicious traffic for the blocked senders	Reroute traffic towards the affected link	Reconfigure the malleable fields for another 5-tuple permutation	Change ECN malleable value

# Flow size estimation

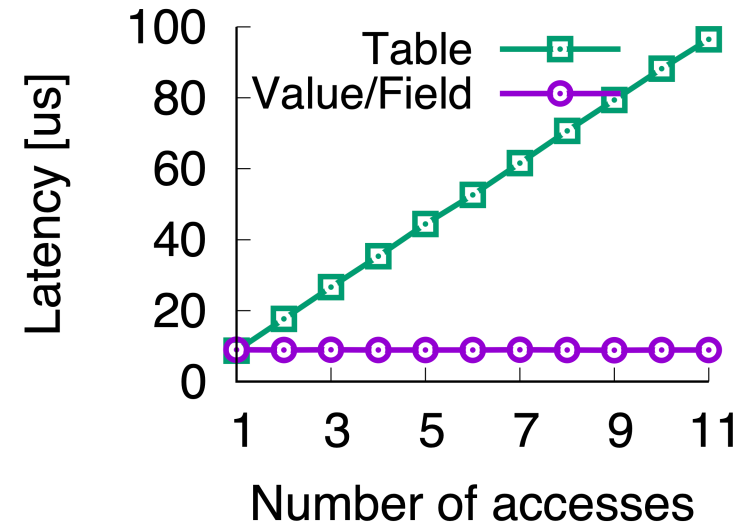
- Evaluation setting
  - CAIDA traces, 20s chunk, 10Gbps link of ISP backbone
- Arguments
  - packet source IP and packet counter
- Algorithm
  - Estimation formula  $\frac{\hat{f}_t - \hat{f}_{t_0}}{t - t_0}$
  - $t_0$ : timestamp when first observe the flow
  - Mantis sampling rate: every 10us, ~1 in 5 packets



# Mantis achieves fast reaction times



a: Reaction argument



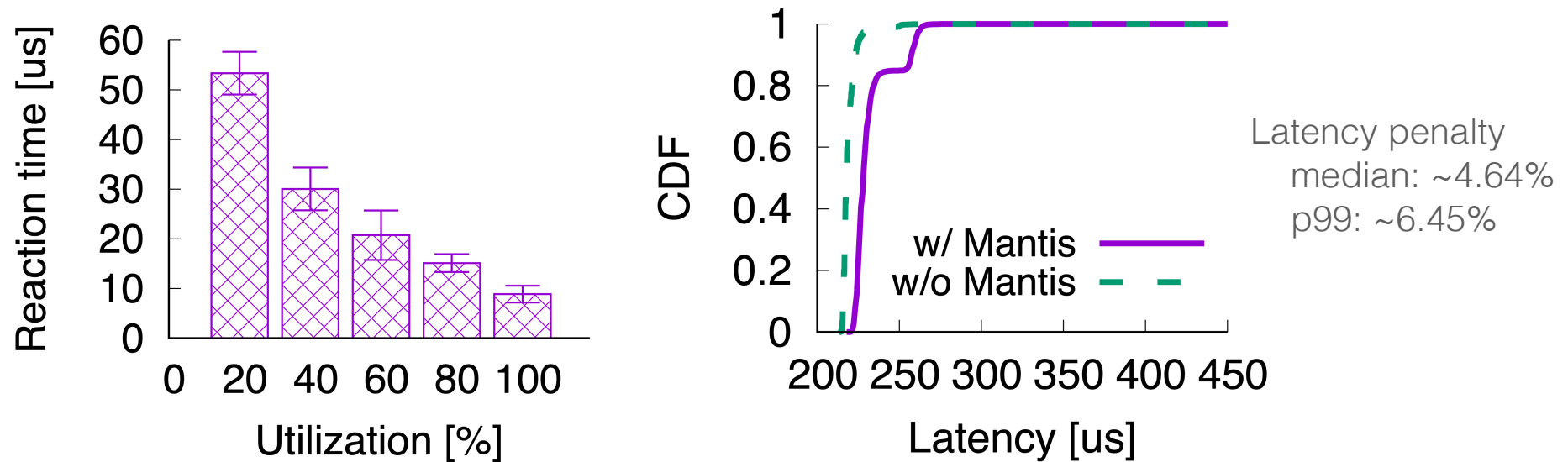
b: Malleable entity update

$$F_b(1 \text{ tblMod}) + \sum_{a \in \text{args}} \left( F_a(a) \right) + C + \sum_{t \in \text{tblMods}} \left( 2F_b(t) \right) + 2F_b(N_{init} - 1) + F_b(1 \text{ tblMod})$$

End-to-end reaction time: **10s of us**

# Mantis CPU overhead

A dialogue loop occupies up to a single core but can be throttled



Overall, Mantis can **co-exist** with other functionalities

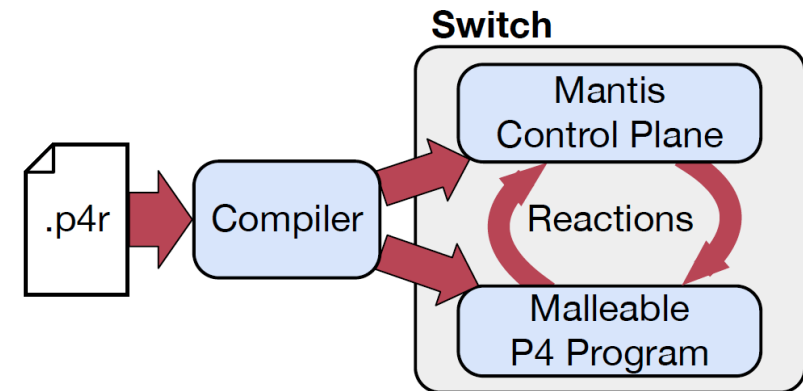


# Summary

- Fine-grained reaction to network statistics as first class citizen
- P4R interface to simplify the encoding of serializable reaction
- Generic support of sub-RTT reactive behaviors

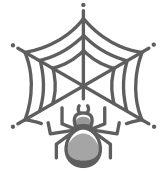
Mantis can be used for...

- Encoding flexible control logic
- Workarounds of current limitations
- Reducing memory overhead via offloading
- Data/control plane co-design



<https://github.com/eniac/Mantis>

# Outline

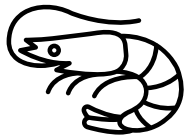


## **OrbWeaver** (*NSDI 2022*)

Reusing IDLE link cycles for in-band control communication

**Reuse**

Zero-waste  
designs



## **Mantis** (*SIGCOMM 2020*)

Recycling switch resources for flexible, sub-RTT reactions

**Recycle**



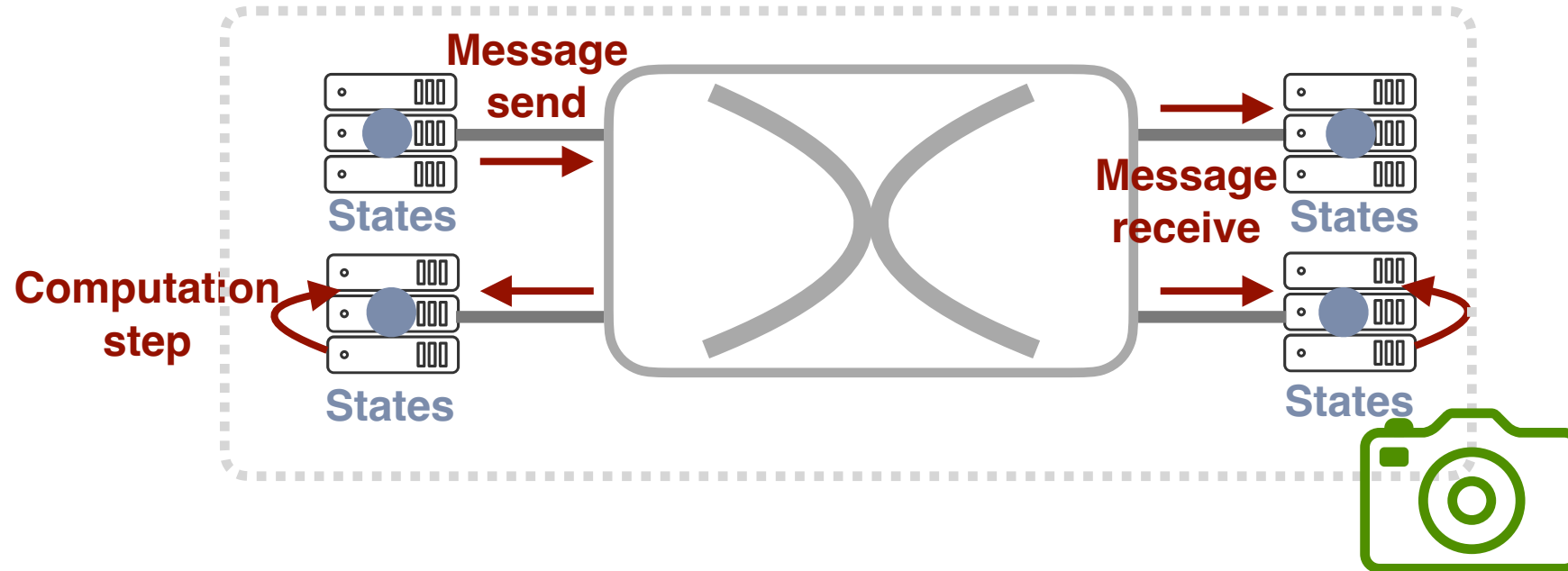
## **Beaver** (*OSDI 2024*)

Reducing 'tax' of partial snapshots for distributed cloud services

**Reduce**

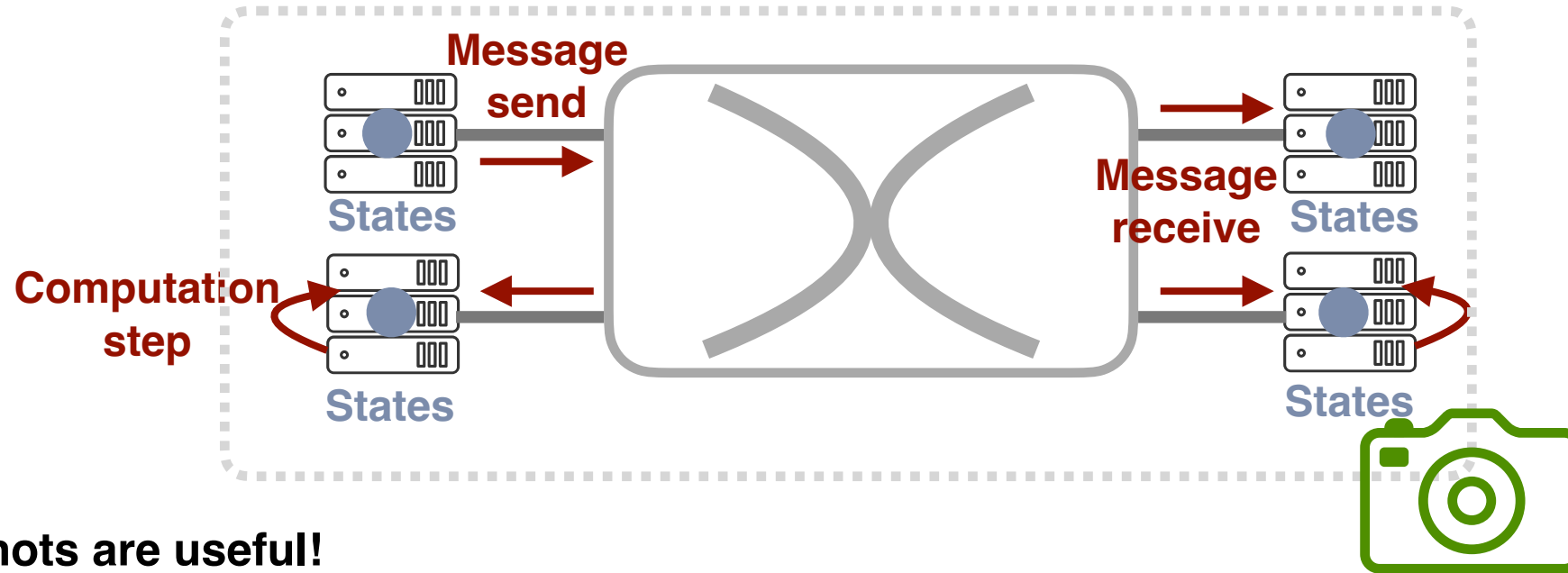
# Let's talk about snapshots

**Distributed snapshots:** a class of distributed algorithms to capture **consistent, global view** of **states**



# Let's talk about snapshots

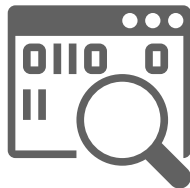
**Distributed snapshots:** a class of distributed algorithms to capture **consistent, global view** of **states**



**Snapshots are useful!**



*Network telemetry*



*Distributed software  
debugging*



*Deadlock detection*



*Checkpointing and  
failure recovery*

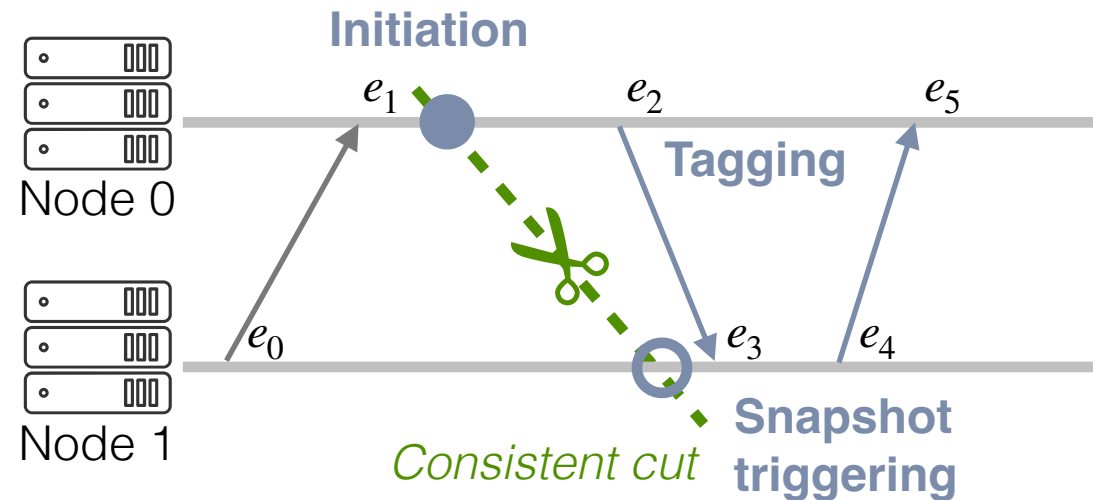
.....

# Classic distributed snapshots

*e.g., Chandy-Lamport (TOCS 1985)*

# Classic distributed snapshots

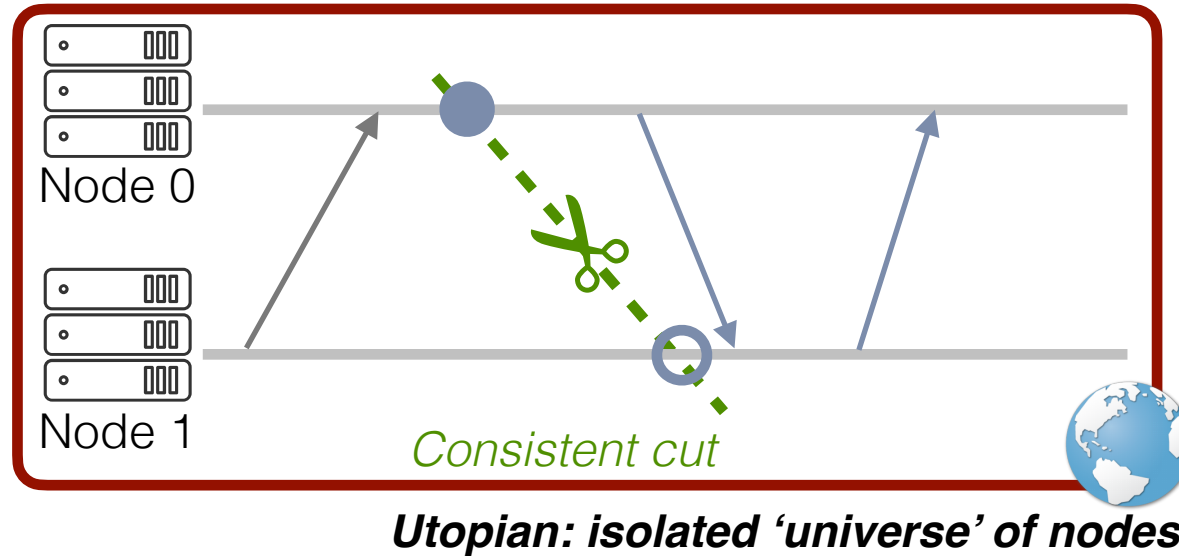
*e.g., Chandy-Lamport (TOCS 1985)*



## Guarantee of causal consistency

For **any** event  $e$  in the cut, if  $e' \rightarrow e$  (Lamport's 'happened before'),  $e'$  is in the cut.

# Classic snapshots operate in an isolated universe



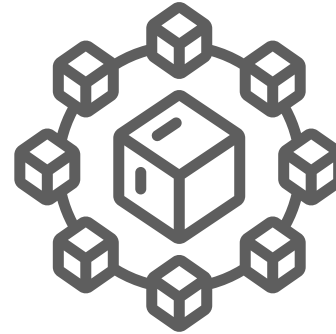
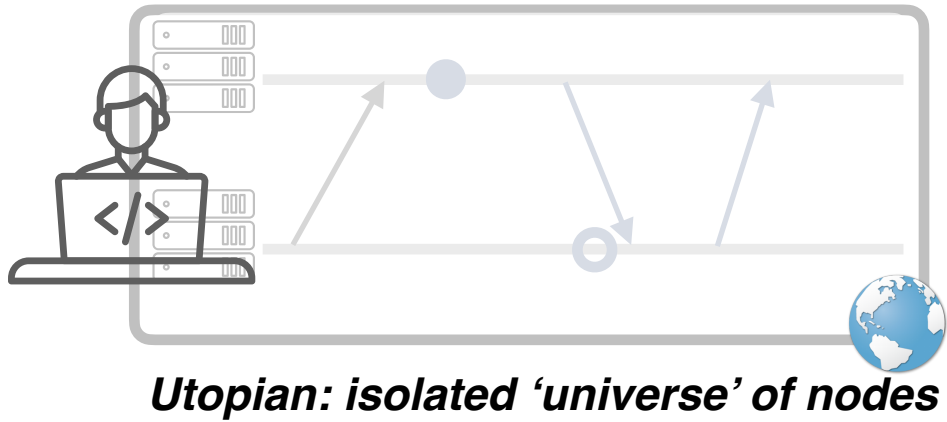
## Fundamental assumption:

The set of participants are **closed** under causal propagation.

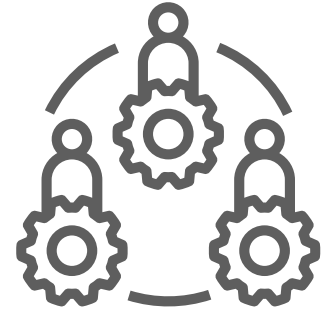


*Unfortunately, the assumption mismatches the real-world scenarios!*

# The assumption rarely matches **reality**!



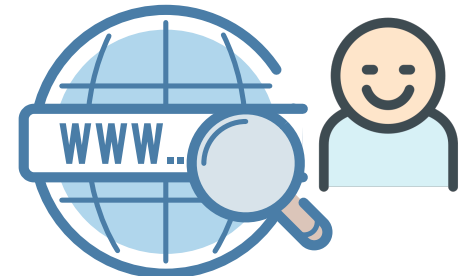
**Modular services**



**Instrumentation  
constraints**



**Costs and  
overheads**



**Hidden causality  
due to human**



# The assumption mismatches **the reality!**



**Unrealistic** to assume *zero* external interaction  
**Impractical** to instrument *all* processes

*Utopian: isolated 'universe' of nodes*

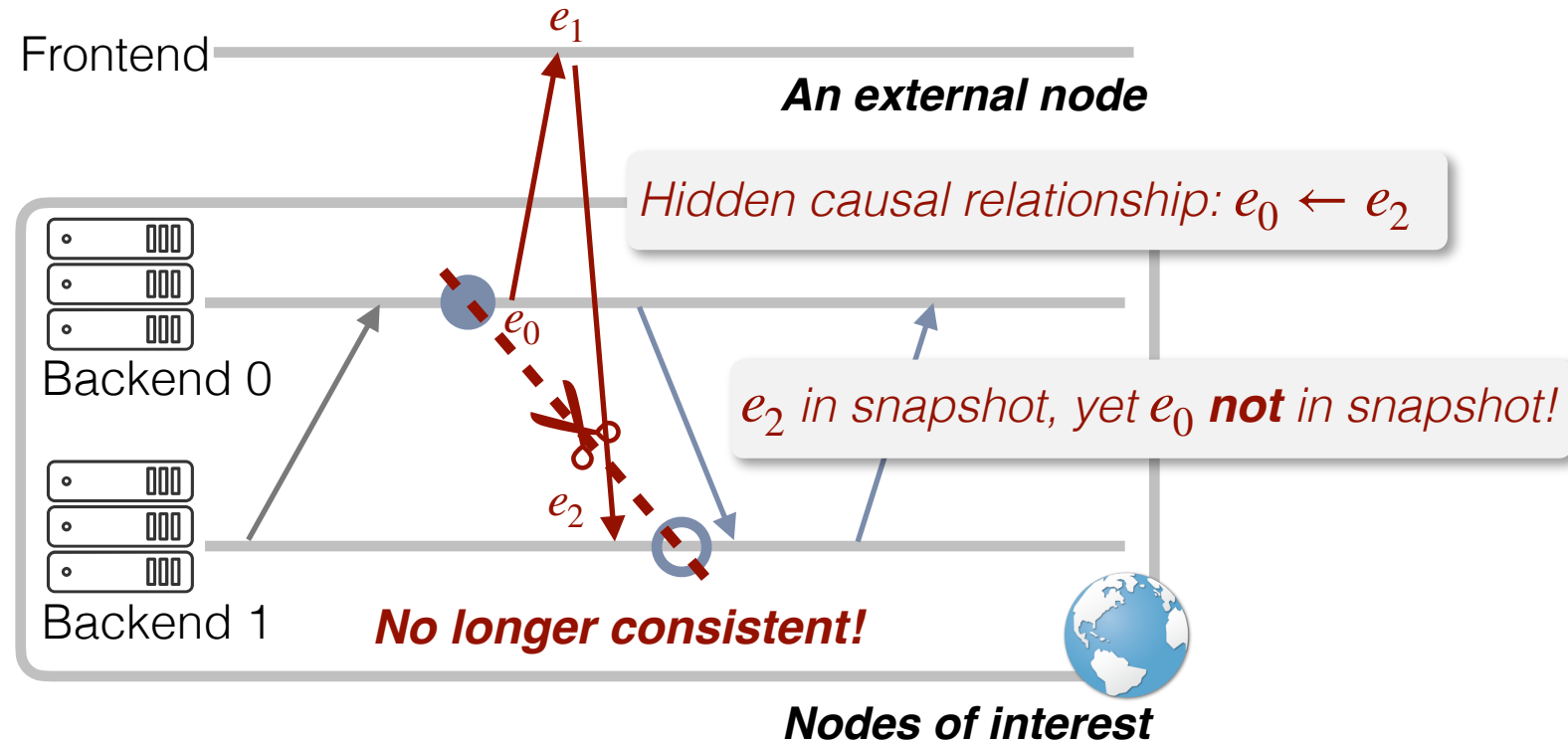


**Costs and  
overheads**



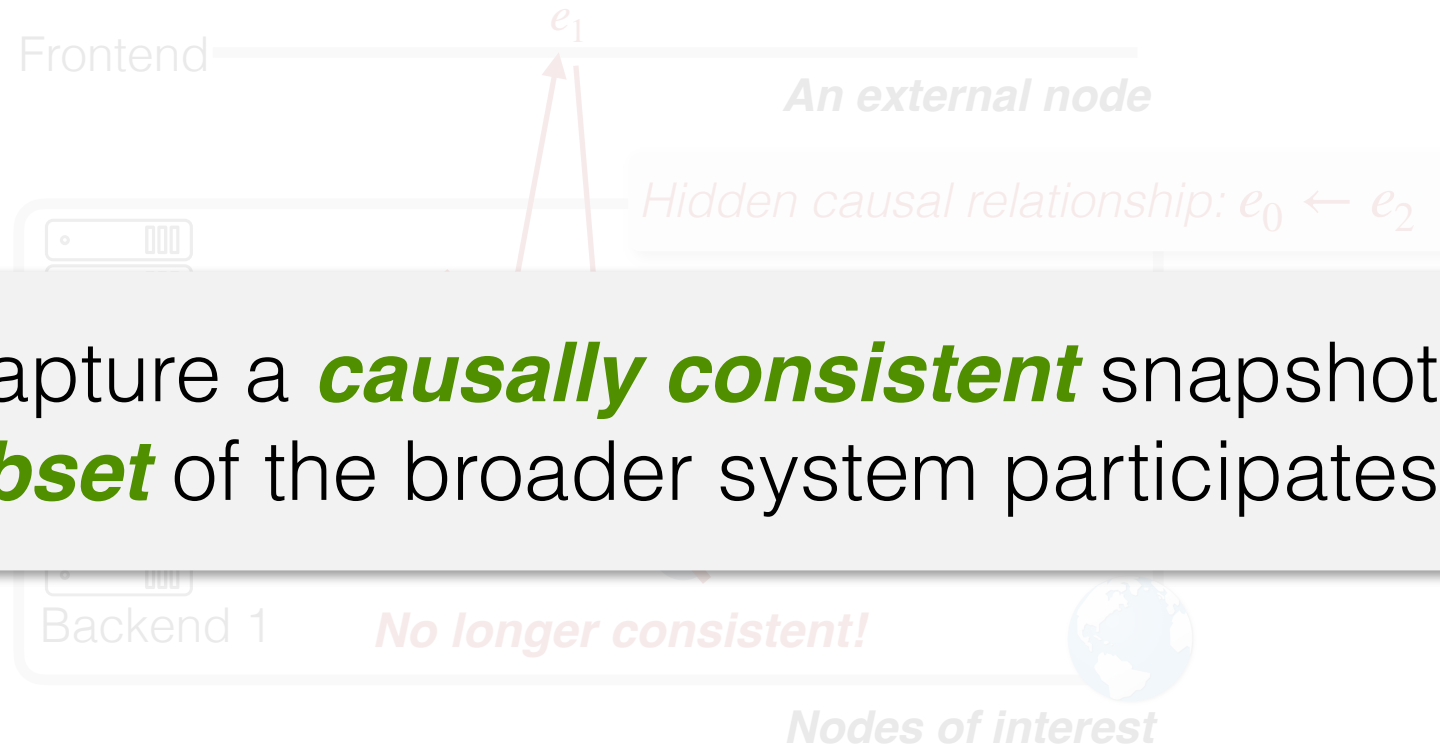
**Hidden causality  
due to human**

# Consequences?



*A single external node can break the guarantee!*

# Consequences?

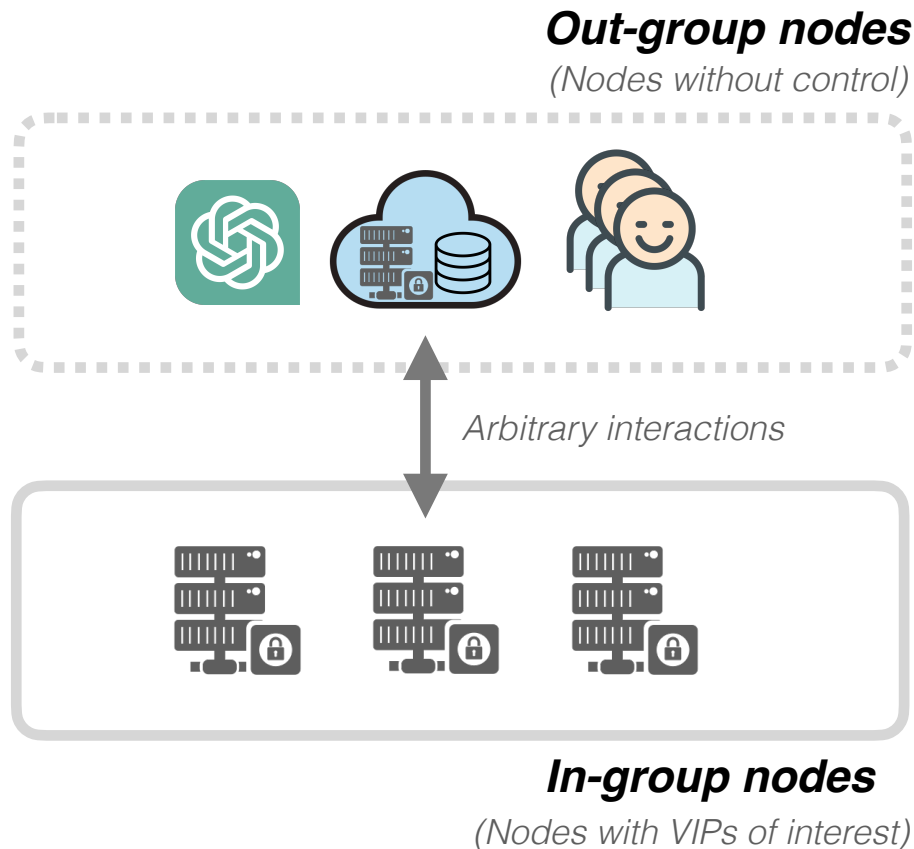


Can we capture a **causally consistent** snapshot when a **subset** of the broader system participates?



*A single external node can break the guarantee!*

# Beaver: practical partial snapshots



## The same causal consistency abstraction

Even when the target service interact with **external, black box services** (arbitrary number, scale, placement, or semantics) via **arbitrary pattern** (including multi-hop propagation of causal dependencies)



## Zero impact over existing service traffic

That is, **absence of blocking or any form of delaying operations** during distributed coordination

A photograph of a beaver standing on a dam made of sticks and logs in a river. The beaver is in the center, looking towards the camera. The background is a calm river with some reeds and a hazy sky.

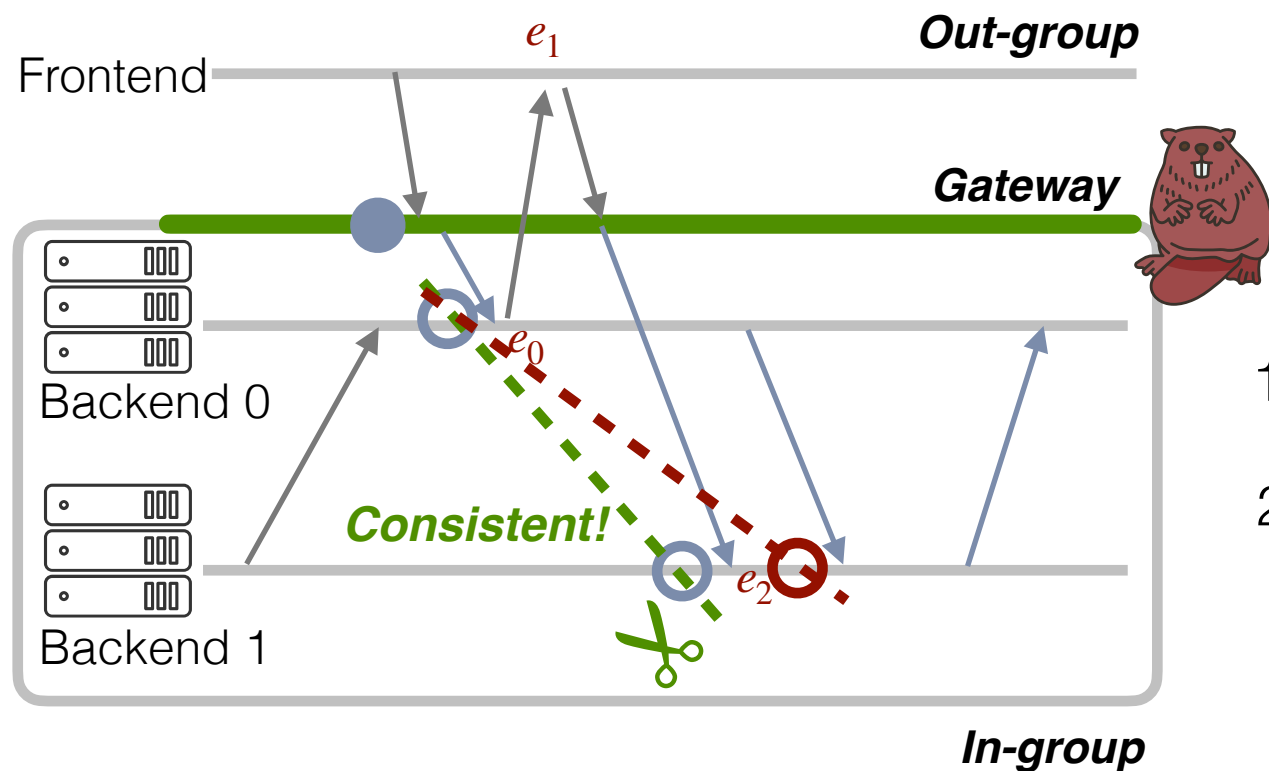
How is it even possible ***without*** coordinating machines external to those of interest?



**Build a dam like a Beaver!**



# Idea 1: Gateway (GW) indirection



Beaver's gateway (GW) indirection:

1. Initiate GW to enter snapshot out-of-band
2. Mark **inbound** packets correspondingly

Before: **inconsistent** cut at  (after  $e_2$ )

With GW: **consistent** cut at  (before  $e_2$ )

# Formalizing idea 1 : Monolithic Gateway Marking

**Theorem 1.** With MGM, a partial snapshot  $C_{part}$  for  $P^{in} \subseteq P$  is causally consistent, that is,  $\forall e \in C_{part}$ , if  $e' \cdot p \in P^{in} \wedge e' \rightarrow e$ , then  $e' \in C_{part}$ .

*Proof.* Let  $e \cdot p = p_i^{in}$  and  $e' \cdot p = p_j^{in}$ . There are 3 cases:

1. Both events occur in the same process, i.e.,  $i = j$ .
2.  $i \neq j$  and the causality relationship  $e' \rightarrow e$  is imposed purely by in-group messages.
3. Otherwise, the causality relationship  $e' \rightarrow e$  involves at least one  $p \in P^{out}$ .

In cases (1) and (2), the theorem is trivially true using identical logic to proofs of traditional distributed snapshot protocols. We prove (3) by contradiction.

Assume  $(e \in C_{part}) \wedge (\exists e' \rightarrow e)$  but  $(e' \notin C_{part})$ . With (3),  $e' \rightarrow e$  means that there must exist some  $e^{out}$  (at an out-group process) satisfying  $e' \rightarrow e^{out} \rightarrow e$ . Now, because  $e' \notin C_{part}$ , we know  $e_{p_j^{in}}^{ss} \rightarrow e'$  or  $e_{p_j^{in}}^{ss} = e'$ , that is,  $p_j^{in}$ 's local snapshot happened before or during  $e'$ . Combined with the fact that the gateway is the original initiator of the snapshot protocol, we know that  $e_g^{ss} \rightarrow e' \rightarrow e^{out} \rightarrow e$ .

We can focus on a subset of the above causality chain:  $e_g^{ss} \rightarrow e$ . From the properties of the in-group snapshot protocol,  $e_g^{ss} \rightarrow e$  implies that  $e \notin C_{part}$ .

This contradicts our original assumption that  $e \in C_{part}$ !  $\square$

**Formal proof in paper**



Holds even if treating the out-group nodes as black boxes



Sufficient to **only** observe the inbound messages

# Key ideas in Beaver



*How to ensure consistency without coordinating external machines?*

**Idea 1: Indirection through Monolithic Gateway Marking (MGM)**

*How to enforce MGM practically in today's network?*

**Challenge 1** How to instantiate GW?

**Challenge 2** How to handle asynchronous GWs?



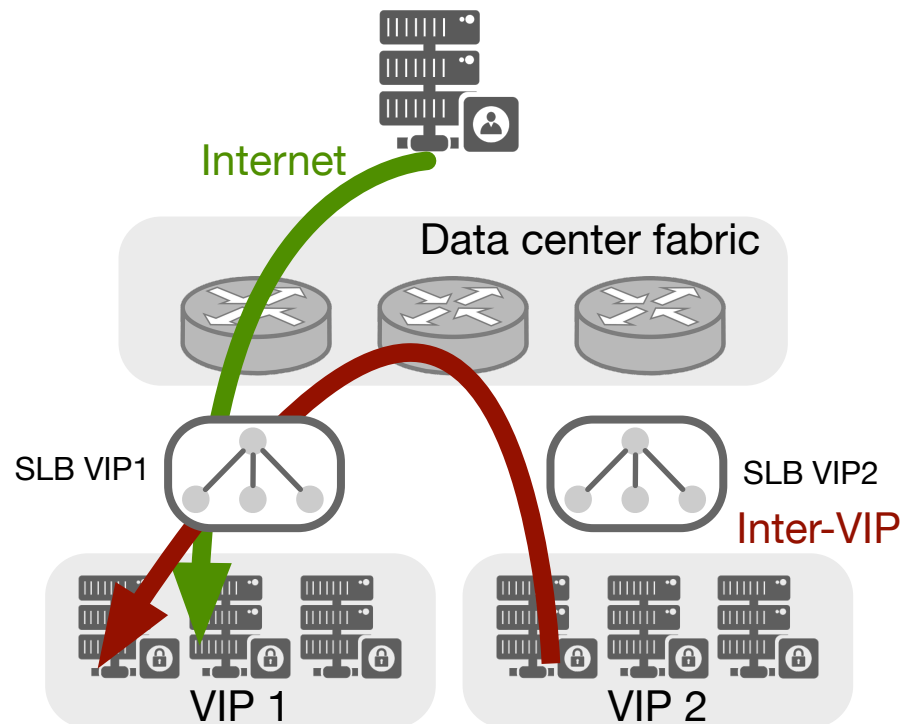
# Challenge 1 : instantiating GWs



Rerouting all inbound traffic through the GW is **costly**



Cloud data centers already place layer-4 load balancers (SLBs)



Repurpose SLBs for in-situ marking

# Key ideas in Beaver

*How to ensure consistency without coordinating external machines?*

**Idea 1: Indirection through Monolithic Gateway Marking (MGM)**

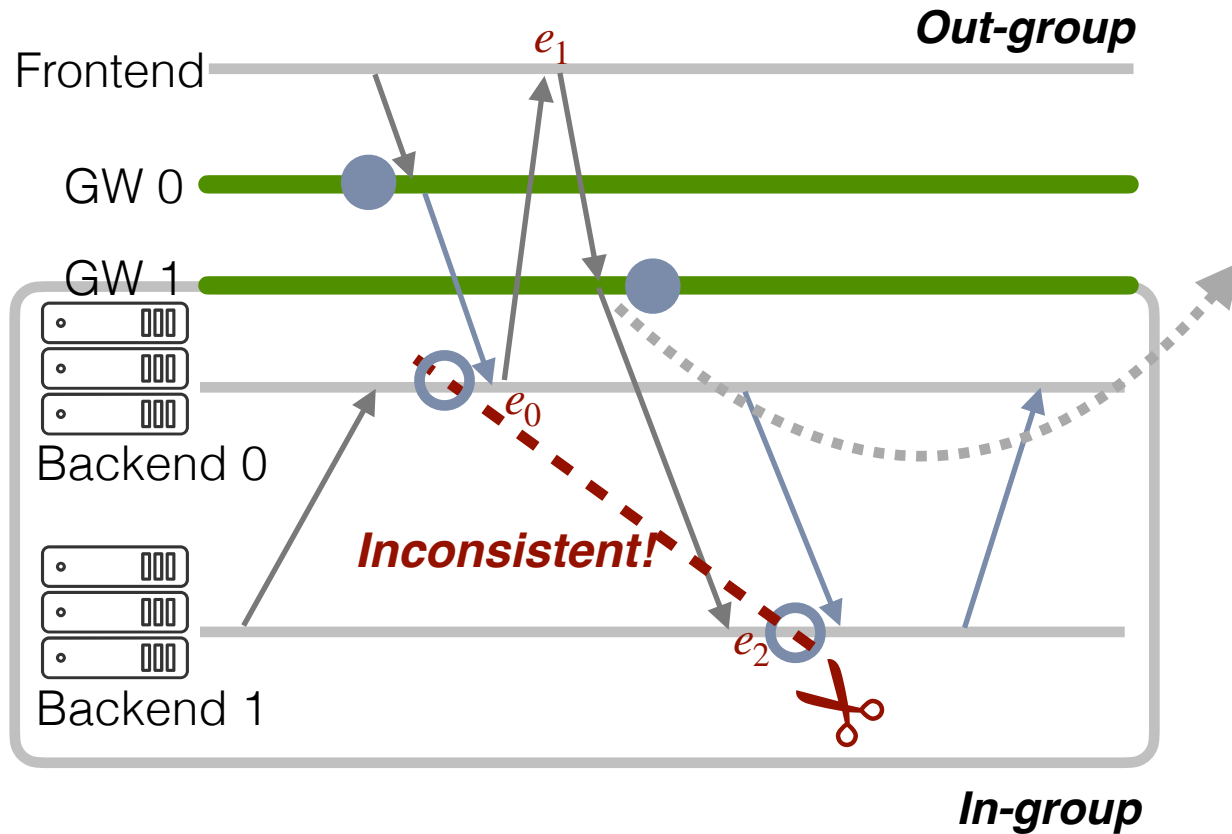
*How to enforce MGM practically in today's network?*

**Challenge 1** How to instantiate GW?

**Idea 2: Reuse existing SLBs with unique locations**

**Challenge 2** How to perform atomic snapshot initiation for asynchronous GWs?

# Implications of multiple SLBs

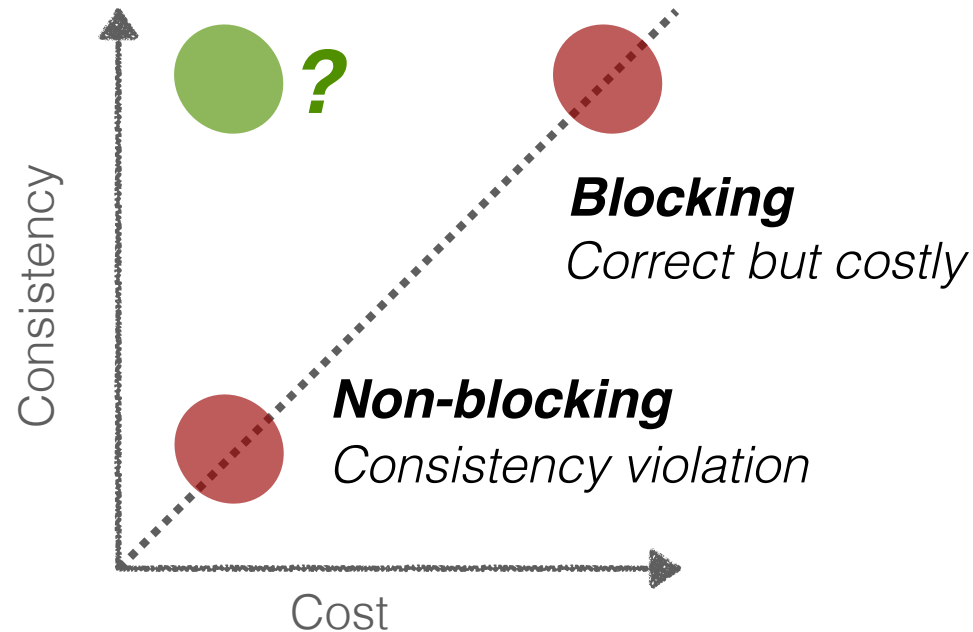


GW 1 hasn't initiated the new snapshot mode to mark it, triggering the **violation**

**$e_2$  in snapshot, yet  $e_0$  that leads to it is not, inconsistent!**

# Handling multiple GWs: design space

*How about blocking messages to 'atomically' trigger all SLBs?*



Can we get both **consistency** and **zero cost**?



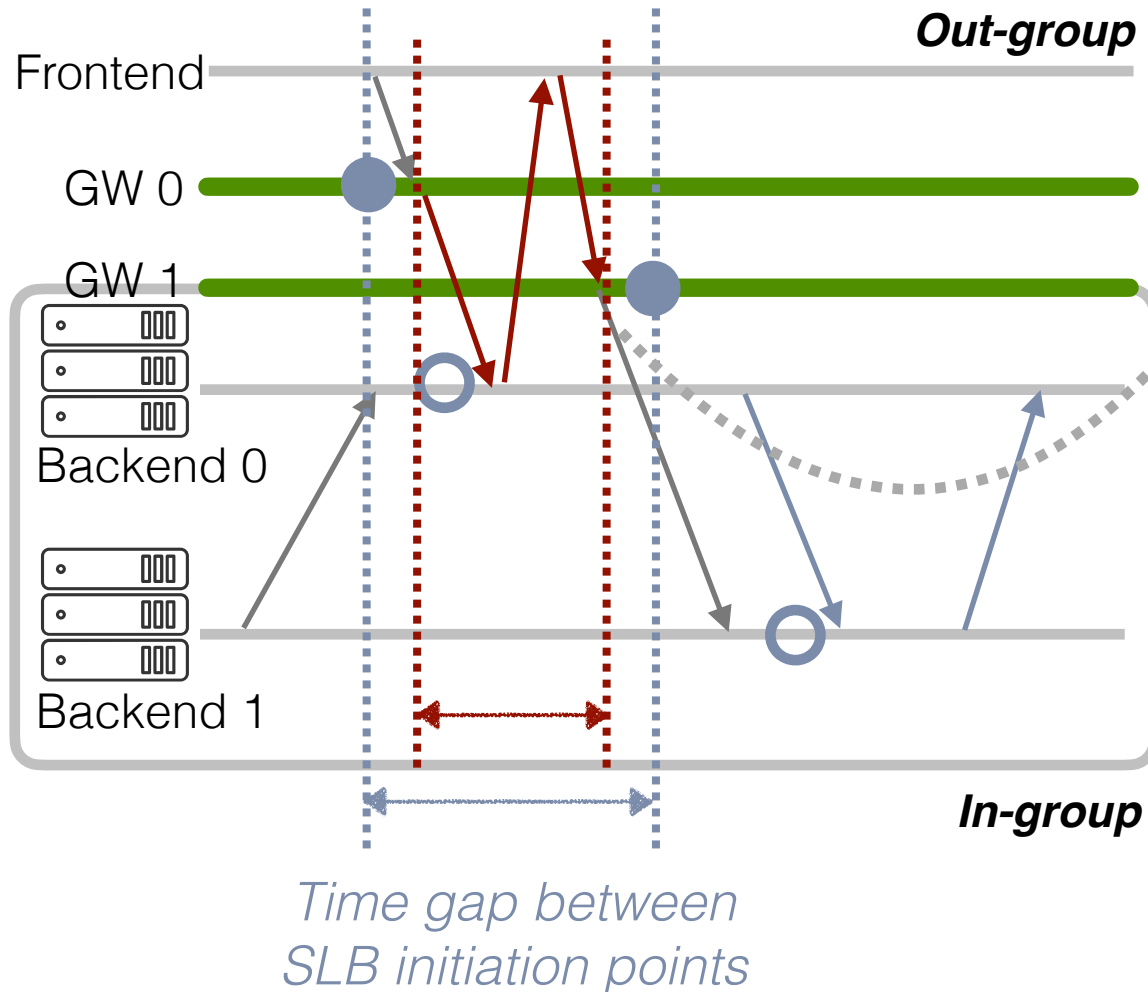
**Optimistic Gateway Marking (OGM)**

*Intuition & formalism*

*Mechanism*

# Challenge 2: handling multiple SLBs

**Reflection:** Beyond worst cases, when and how often does the violation occur?



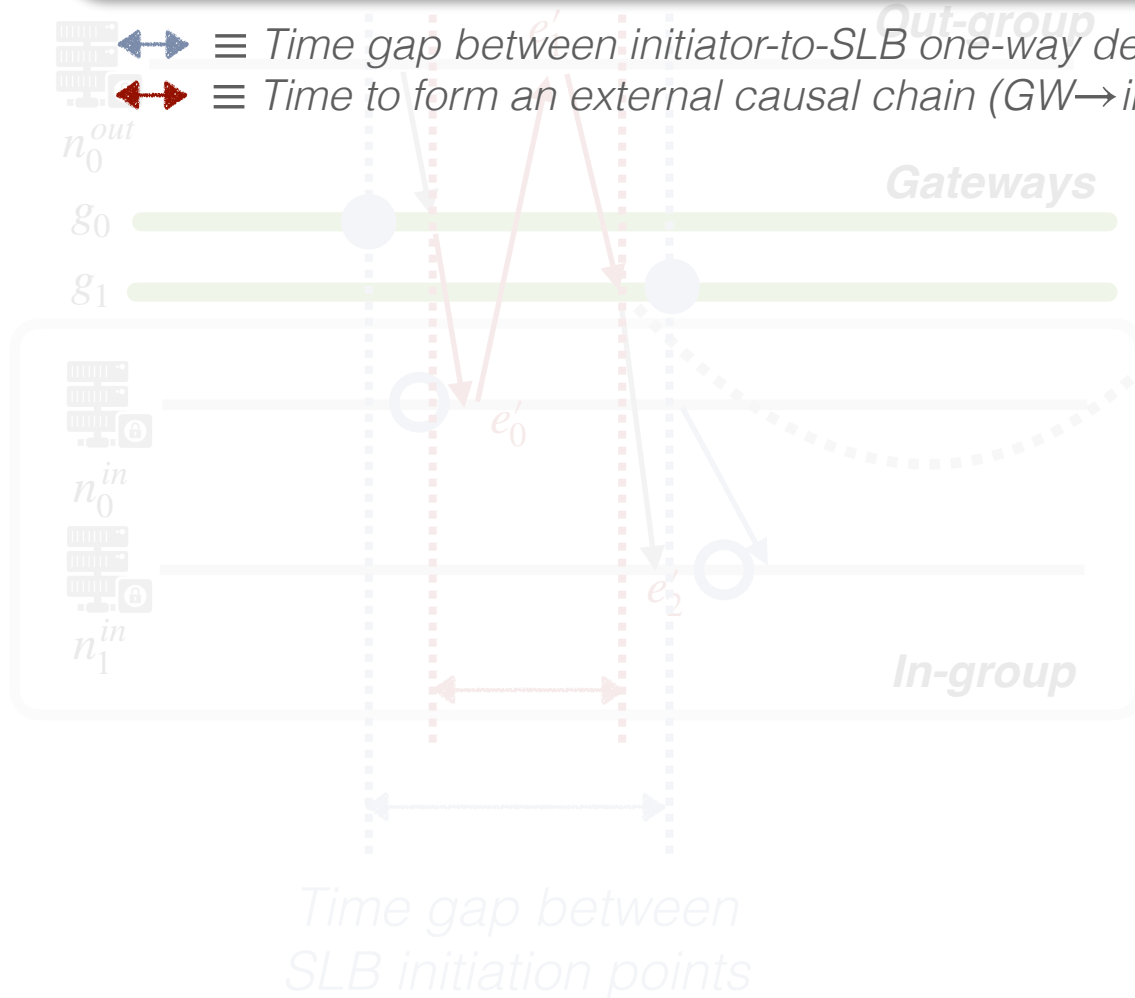
**Observation:**

**Causally relevant messages** are rare!  
GW → in-group → out-group → GW (external causal chain)

**Intuition:** the resulting snapshot is consistent

1. if  $\longleftrightarrow$  is **large enough**
2. or if  $\longleftrightarrow$  is **'close' enough**

**Theorem:** if  $\leftrightarrow < \rightleftarrows$ , the partial snapshot is consistent!



**Observation:**

Causally relevant n  
GW  $\rightarrow$  in-group  $\rightarrow$  out-  
causal chain)

**Theorem 2.** In a system with multiple asynchronous gateways, let the wall-clock time of the first and last gateway snapshots be  $e_{gmin}^{ss} = \min_{g \in G} (e_g^{ss}.t)$  and  $e_{gmax}^{ss} = \max_{g \in G} (e_g^{ss}.t)$ , respectively. Also let  $\forall g \in G, \tau_{min} = \min(d(g, g'; \{p, q\}))$ , where  $g, g' \in G, p \in P^{in}$ , and  $q \in P^{out}$ . If  $e_{gmax}^{ss}.t - e_{gmin}^{ss}.t < \tau_{min}$ , then the partial snapshot is causally consistent.

**Proof.** We extend the proof of Theorem 1 to a distributed setting. Similar to Theorem 1, there are three cases, with (3) being the one that differs. We again prove it by contradiction. Assume  $(e \in C_{part}) \wedge (\exists e' \rightarrow e)$  but  $(e' \notin C_{part})$ . As before, there must be some chain  $e' \rightarrow e^{out} \rightarrow e^s \rightarrow e$ . Because  $e' \notin C_{part}$ , we have  $e_{p_j^n}^{ss} \rightarrow e'$  or  $e_{p_j^n}^{ss} = e'$ , that is,  $p_j^n$  must have been triggered directly or indirectly by an inbound message. Denote the arrival of this inbound message at its marking gateway as  $e^s$ . By the definition of  $\tau_{min}$ , we have  $e^s.t - e^{s'}.t \geq \tau_{min} > e_{gmax}^{ss}.t - e_{gmin}^{ss}.t$ . Thus, at event  $e^s$ , the gateway must have already initiated the snapshot and will mark  $e^s.m$  before forwarding. This results in  $e \notin C_{part}$ , a contradiction!  $\square$

**Formal proof in paper**

**Intuition:** the resulting snapshot is consistent

1. if  $\rightleftarrows$  is **large enough**
2. or if  $\leftrightarrow$  is **'close' enough**

**Theorem:** if  $\leftrightarrow < \rightleftarrows$ , the partial snapshot is consistent!

$\leftrightarrow \equiv$  Time gap between initiator-to-SLB one-way delays  
 $\rightleftarrows \equiv$  Time to form an external causal chain (GW  $\rightarrow$  in-group  $\rightarrow$  out-group  $\rightarrow$  GW)

**Observation: condition holds in most cases anyway!**

$\leftrightarrow$  can **approximate zero**

- SLBs share the same region
- Proper placement of controller

$\rightleftarrows$  is relatively high

- $\geq 3$  trips through the fabric
- Higher when the out-group is in another DC or Internet

**Optimistic Gateway Marking (OGM)**

Time gap between SLB initiation points

Optimistic execution in common cases

Verification/rejection of snapshots under worst cases

**Theorem 2.** In a system with multiple asynchronous gateways, let the wall-clock time of the first and last gateway snapshots be  $e_{gmin}^{ss} = \min_{e_g^{ss}}(e_g^{ss}.t)$  and  $e_{gmax}^{ss} = \max_{e_g^{ss}}(e_g^{ss}.t)$ , respectively. Also let  $\forall g \in G, \tau_{min} = \min(d(g, g'; \{p, q\}))$ , where  $g, g' \in G, p \in P^{in}$ , and  $q \in P^{out}$ . If  $e_{gmax}^{ss}.t - e_{gmin}^{ss}.t < \tau_{min}$ , then the partial snapshot is causally consistent.

**Proof.** We extend the proof of Theorem 1 to a distributed setting. Similar to Theorem 1, there are three cases, with (3) being the one that differs. We again prove it by contradiction.

Assume  $(e \in C_{part}) \wedge (\exists e' \rightarrow e)$  but  $(e' \notin C_{part})$ . As before, there must be some chain  $e' \rightarrow e^{out} \rightarrow e^s \rightarrow e$ . Because  $e' \notin C_{part}$ , we have  $e_{p_j^n}^{ss} \rightarrow e'$  or  $e_{p_j^n}^{ss} = e'$ , that is,  $p_j^n$  must have been triggered directly or indirectly by an inbound message. Denote the arrival of this inbound message at its marking gateway as  $e^s$ . By the definition of  $\tau_{min}$ , we have  $e^s.t - e^{s'}.t \geq \tau_{min} > e_{gmax}^{ss}.t - e_{gmin}^{ss}.t$ . Thus, at event  $e^s$ , the gateway must have already initiated the snapshot and will mark  $e^s.m$  before forwarding. This results in  $e \notin C_{part}$ , a contradiction!  $\square$

**Formal proof in paper**

**Intuition:** the resulting snapshot is consistent

1. if  $\leftrightarrow$  is large enough

# How does Beaver detect a snapshot violation?

**Theorem:** if  $\leftrightarrow < \rightleftarrows$ , the partial snapshot is consistent

$\leftrightarrow$   $\equiv$  Time gap between initiator-to-SLB one-way delays

$\rightleftarrows$   $\equiv$  Time to form an external causal chain ( $GW \rightarrow \text{in-group} \rightarrow \text{out-group} \rightarrow GW$ )



1. Determine the lower bound of  $\rightleftarrows$  statically
2. Measure a safe upper bound for  $\leftrightarrow$  online using a single clock



***False positives is fine as one can always retry!***



# Key ideas in Beaver

*How to ensure consistency without coordinating external machines?*

## **Idea 1: Indirection through Monolithic Gateway Marking (MGM)**

*How to enforce MGM practically in today's network?*

**Challenge 1** How to instantiate GW?

## **Idea 2: Reuse existing SLBs with unique locations**

**Challenge 2** How to perform atomic snapshot initiation for asynchronous GWs?

## **Idea 3: Optimistic Gateway Marking (OGM)**

- Optimistic execution *in common cases*
- Verification/rejection of snapshot *under worst cases*

# Key ideas in Beaver



*How to ensure consistency without coordinating external machines?*

## **More details about Beaver's protocol...**

- Synchronization-free snapshot verification
- Supporting parallel snapshots
- Handling failures
- Handling packet loss, delay, and reordering
- ...

*Challenge 2: How to handle asynchronous GWS?*

## **Idea 3: Optimistic Gateway Marking (OGM)**

- Optimistic execution *in common cases*
- Verification/rejection of snapshot *under worst cases*

# Implementation and evaluation

## SLB-associated workflow

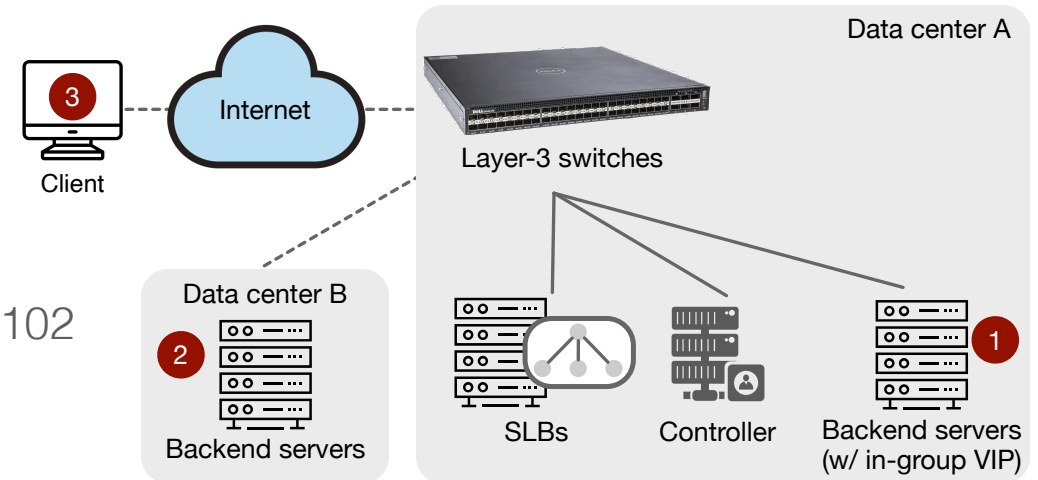
- Layer-3 ECMP forwarding per service VIPs: DELL EMC PowerSwitch S4048-ON
- Core SLB functions in DPDK: ~1860 LoC
- Backend server functions in XDP and tc: ~1040 LoC

## Beaver protocol integration

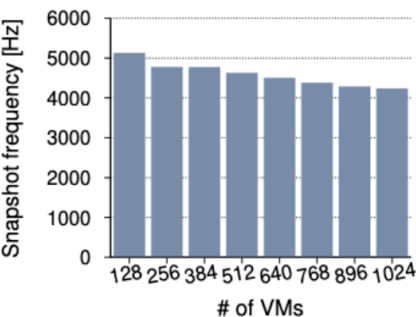
- Minimal logic: (1) 68 LoC for SLB DPDK data path logic (2) 102 LoC for eBPF at in-group VMs

## Topology

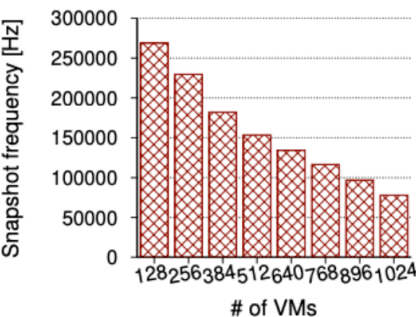
- Support typical communication patterns
- Possible out-group locations: within the same DC, DC at a different region, or on the Internet
- Scale up to 16 SLB servers and 1024 backend applications



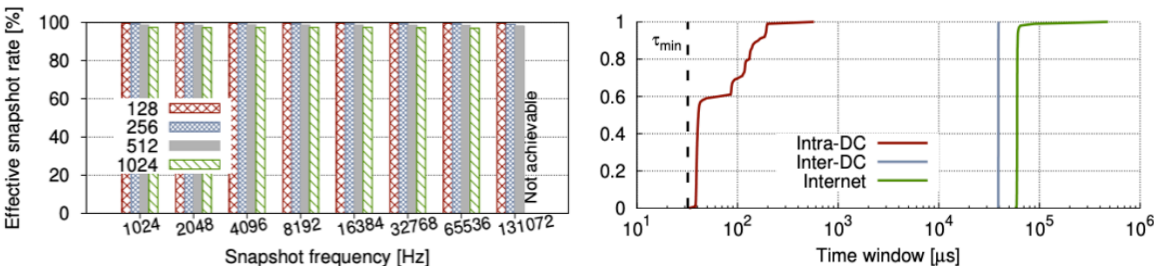
# Details in the paper...



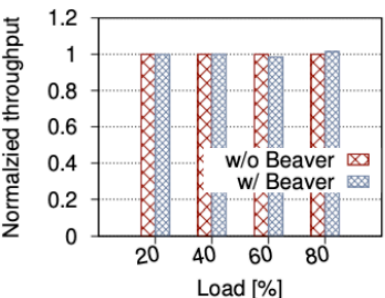
(a) w/o parallelism



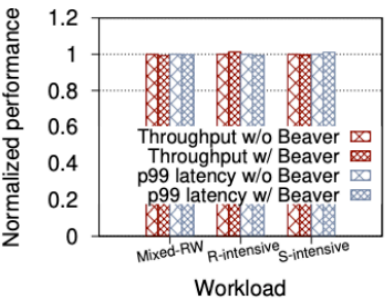
(b) w/ parallelism



## Beaver supports fast snapshot rates



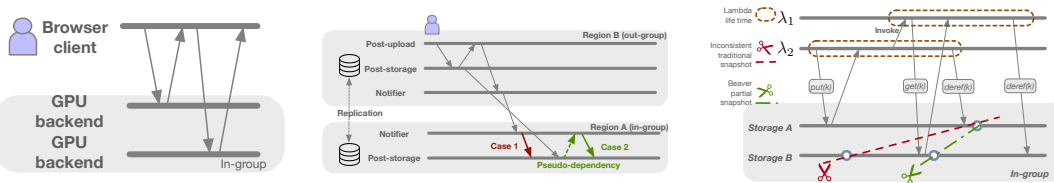
(a) Stressed workloads



(b) YCSB benchmarks

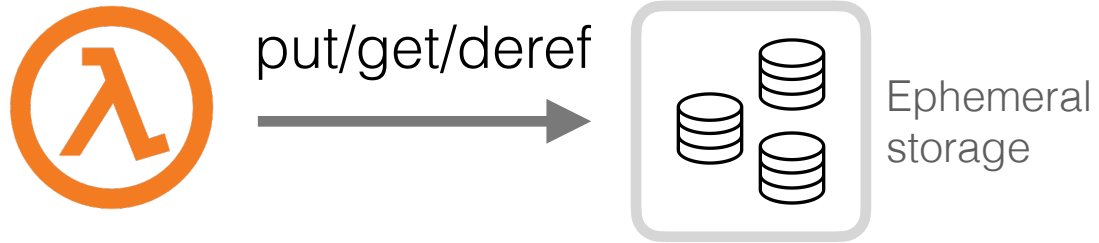
## Beaver incurs zero impact

## Beaver rejects snapshots infrequently

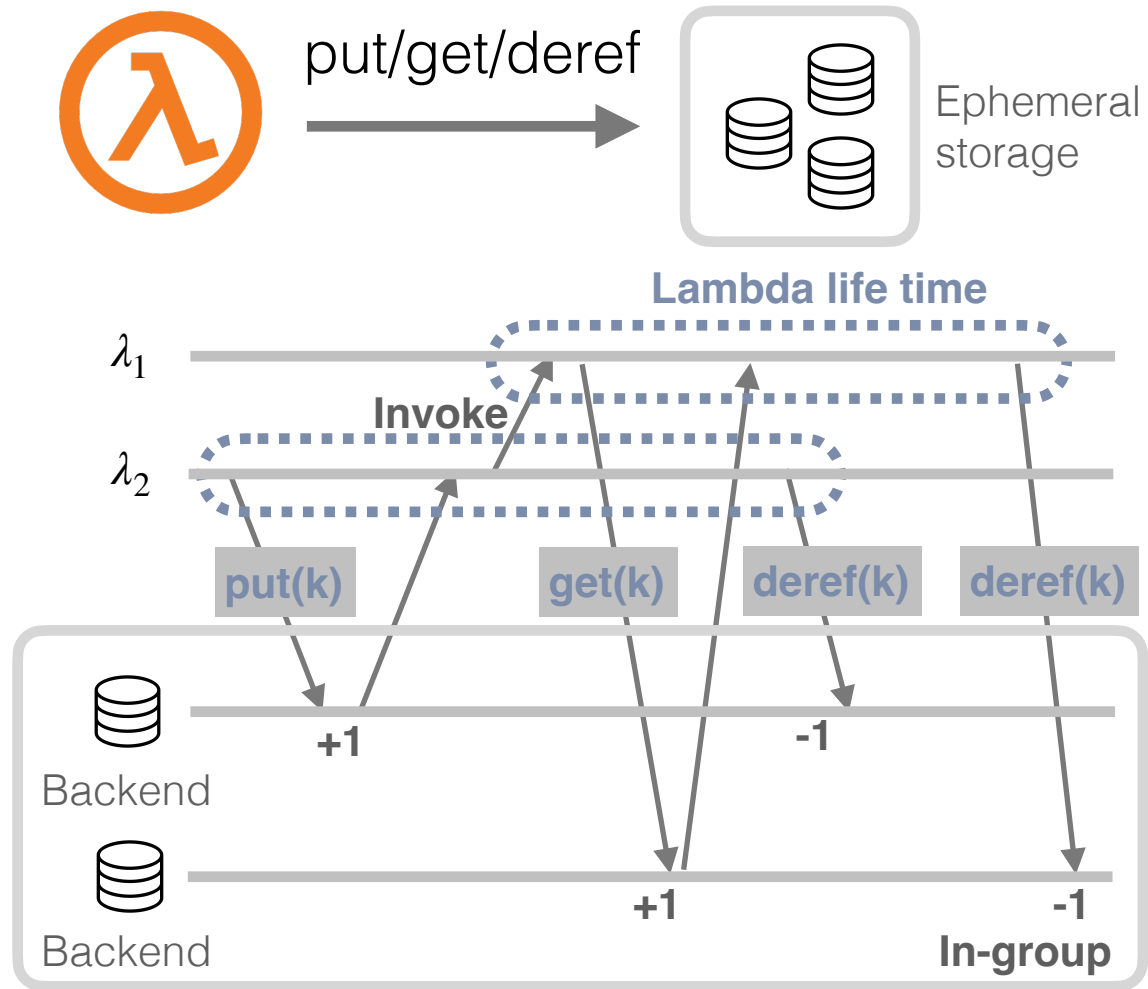


## Use cases: integration testing, service analytics, deadlock detection, garbage collection...

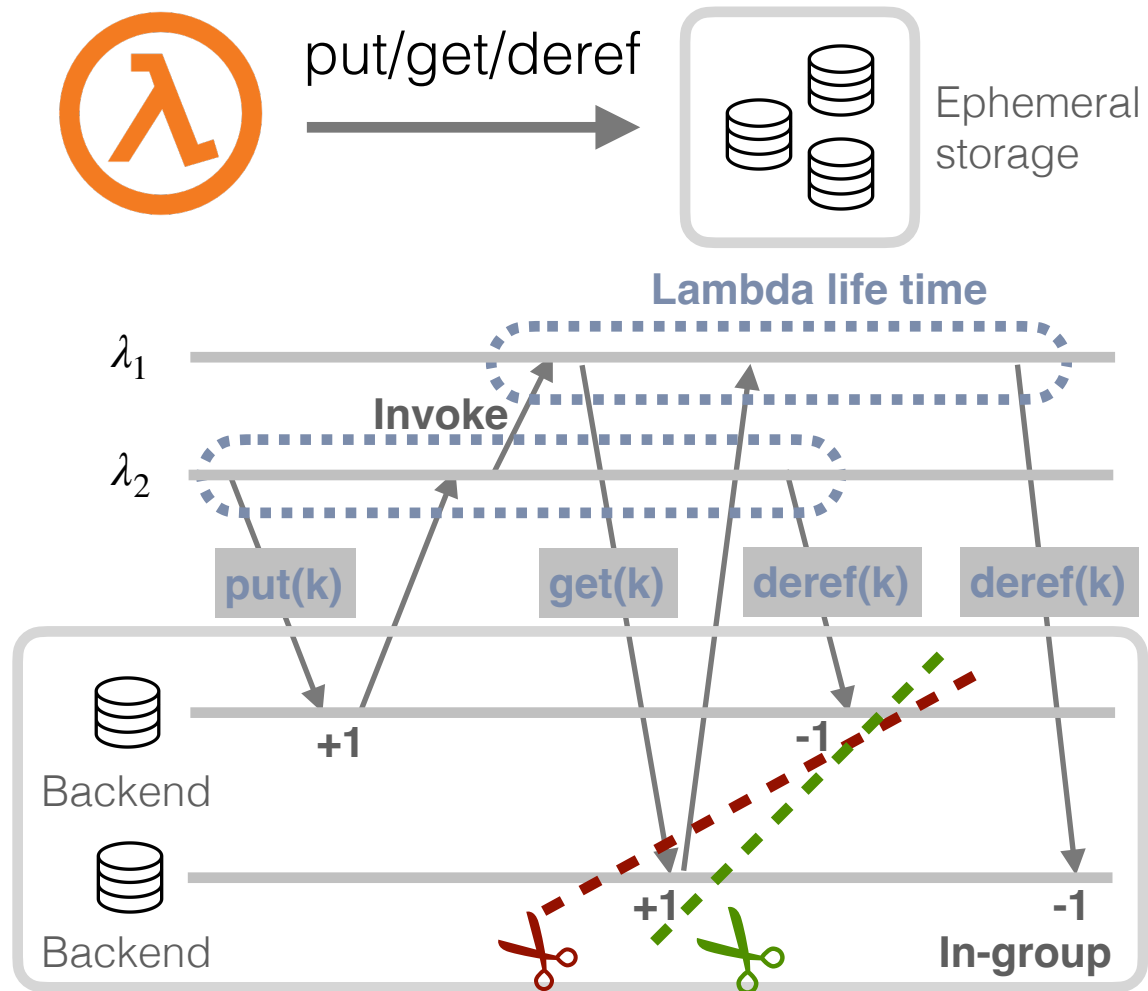
# Example: garbage collection for ephemeral storage



# Example: garbage collection for ephemeral storage



# Example: garbage collection for ephemeral storage



## Strawman

Reference count = 0, unsafe recycle decision of  $k$ !



Reference count = 1, safe decision recognizing open reference to  $k$



# Beaver: summary

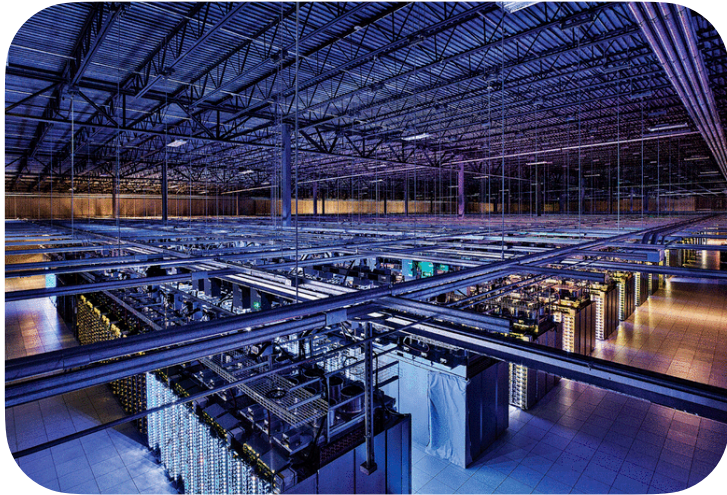
The first partial snapshot protocol that extends classic distributed snapshots in **practical cloud settings**

Guarantees **causal consistency** while incurring **minimal changes and overheads**

Key idea: Exploit data center characteristics (e.g., unique topologies)



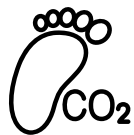
# Vision: toward **zero-waste** networked systems



Insatiable application demand



Increasing energy consumption



*Embodied carbon is also a major contributor!*



**Grand challenge:** *Push the wastes in computing infrastructure to their limits*

# Vision: toward **zero-waste** networked systems



## **Tight coupling IDLE resources, e.g., for performant network control**

- *Can we repurpose the underutilized resources for integrating network tasks?*
- *Or, how to reduce the wasted consumption to its limits (e.g., power)?*



## **Restructuring systems stacks for efficient ‘tax’ functions**

- *Can we enable an asynchronous IDLE channel for executing tax functions?*
- *How to exploit the growing heterogeneity in hardware accelerators?*



## **Rethinking classic layering principle for a clean-slate redesign**

- *How to specialize the stack leveraging the predictability in emerging workloads/primitives?*
- *Can we simplify and break the current layering architecture while ensuring modularity?*
- *Beyond cross-layer design, what does that ‘post-layering’ architecture look like?*