p4v

Jed Liu
Bill Hallahan
Cole Schlesinger
Milad Sharif
Jeongkeun Lee
Robert Soulé
Han Wang
Calin Cascaval
Nick McKeown
Nate Foster

Università della Svizzera italiana

CORNELL UNIVERSITY FOUNDED A.D. 1865

p4v

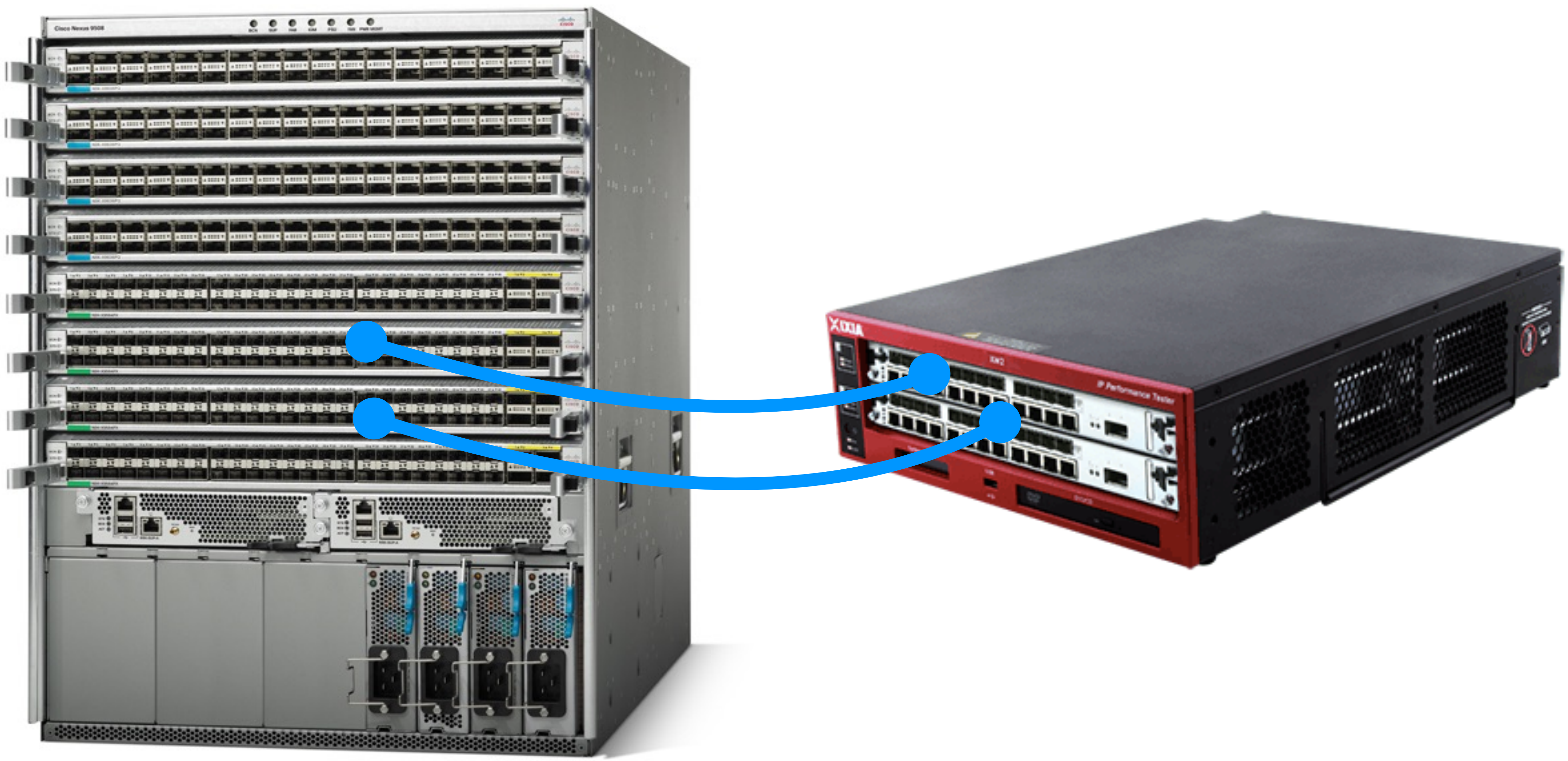
Or, how I learned to stop worrying and trust Z3

Jed Liu
Bill Hallahan
Cole Schlesinger
Milad Sharif
Jeongkeun Lee
Robert Soulé
Han Wang
Calin Cascaval
Nick McKeown
Nate Foster

Suppose you buy a router...

Question: How do you ensure that it works as expected?

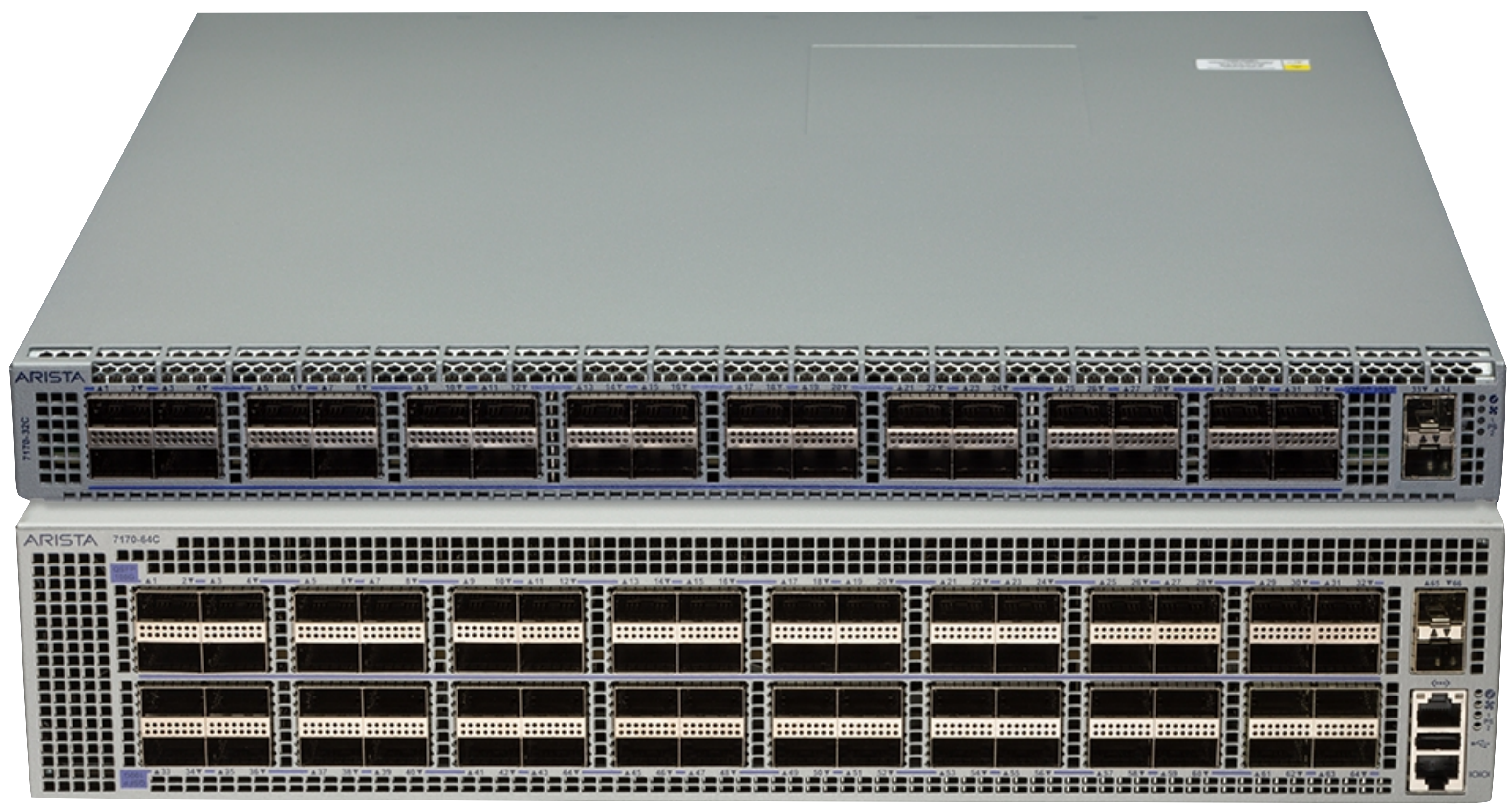
Suppose you buy a router...



Question: How do you ensure that it works as expected?

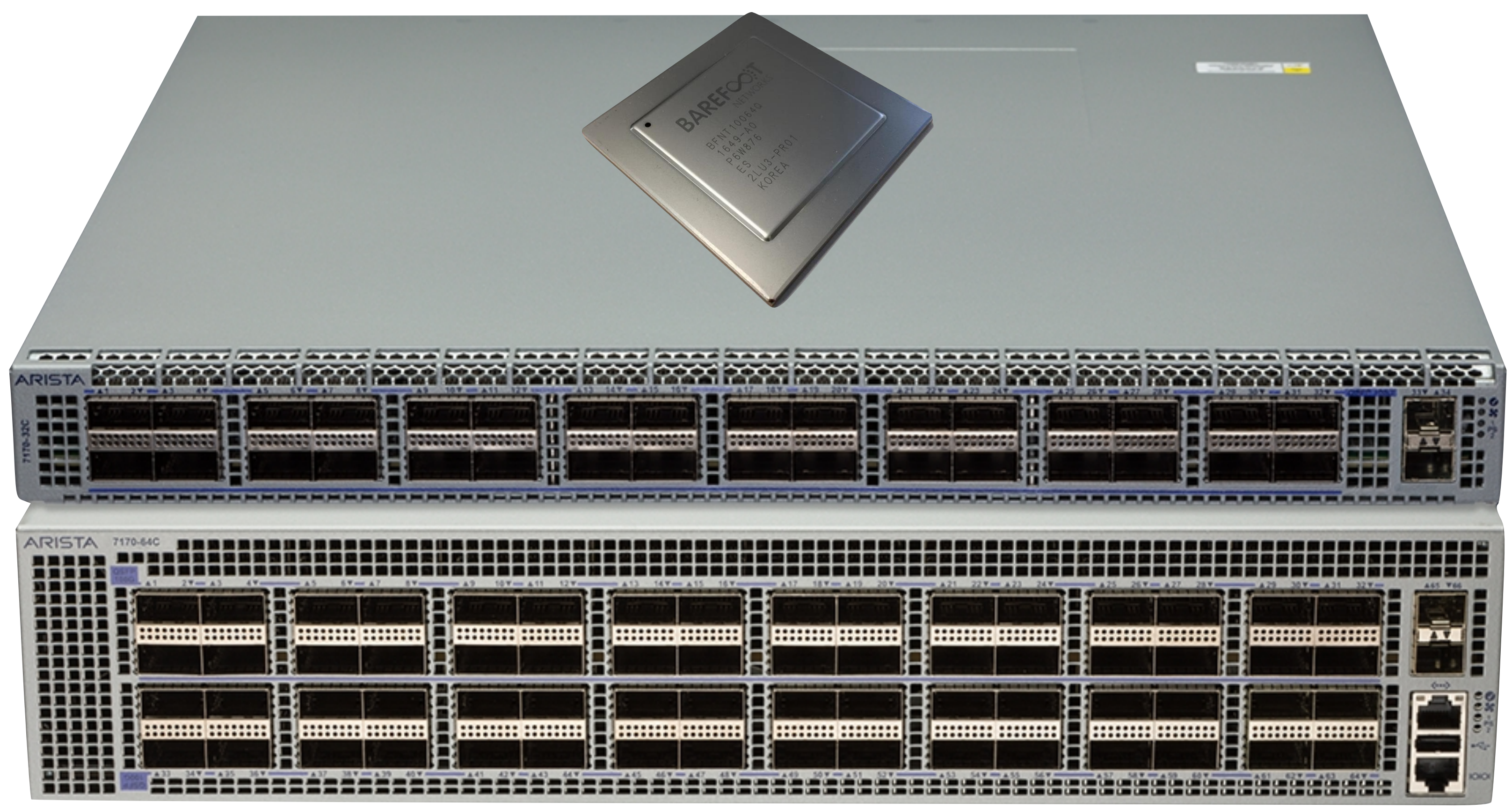
Answer: Test it!

What if it's a *programmable* router?



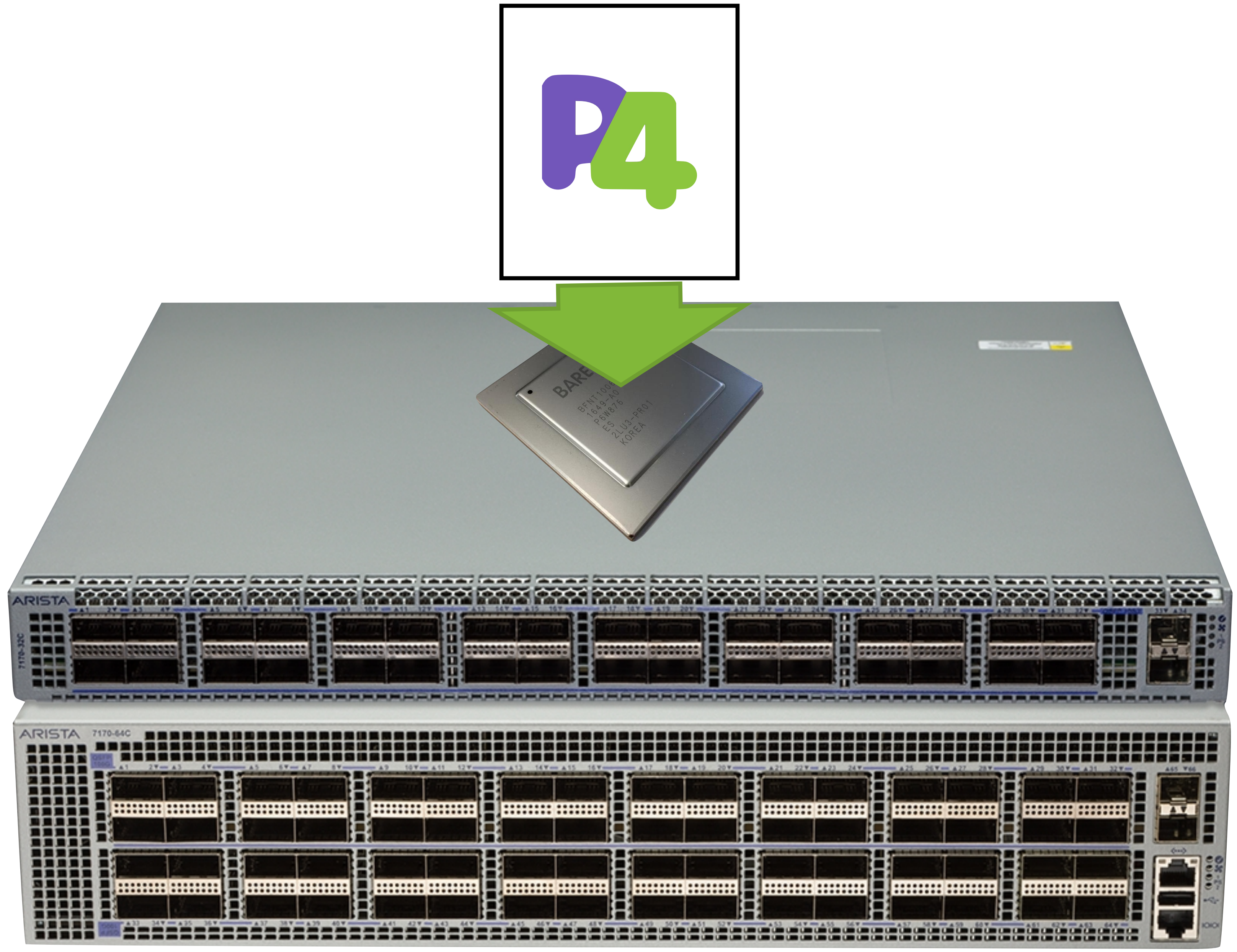
Question: Now what do you do?

What if it's a *programmable* router?



Question: Now what do you do?

What if it's a *programmable* router?



Question: Now what do you do?

Example: NetChain & NetCache

Example: NetChain & NetCache

NetCache: Balancing Key-Value Stores with Fast In-Network Caching

Xin Jin[1], Xiaozhou Li[2], Haoyu Zhang[3], Robert Soulé[2,4], Jeongkeun Lee[2], Nate Foster[2,5], Changhoon Kim[2], Ion Stoica[6]

[1]Johns Hopkins University, [2]Barefoot Networks, [3]Princeton University, [4]Università della Svizzera italiana, [5]Cornell University, [6] UC Berkeley

ABSTRACT

We present NetCache, a new key-value store architecture that leverages the power and flexibility of new-generation programmable switches to handle queries on hot items and balance the load across storage nodes. NetCache provides high aggregate throughput and low latency even under highly-skewed and rapidly-changing workloads. The core of NetCache is a packet-processing pipeline that exploits the capabilities of modern programmable switch ASICs to efficiently detect, index, cache and serve hot key-value items in the switch data plane. Additionally, our solution guarantees cache coherence with minimal overhead. We implement a NetCache prototype on Barefoot Tofino switches and commodity servers and demonstrate that a single switch can process 2+ billion queries per second for 64K items with 16-byte keys and 128-byte values, while only consuming a small portion of its hardware resources. To the best of our knowledge, this is the first time that a sophisticated application-level functionality, such as in-network caching, has been shown to run at line rate on programmable switches. Furthermore, we show that NetCache improves the throughput by 3-10× and reduces the latency of up to 40% of queries by 50%, for high-performance, in-memory key-value stores.

CCS CONCEPTS

• Information systems → Key-value stores; **• Networks → Programmable networks**; **In-network processing**; **• Computer systems organization → Cloud computing**;

KEYWORDS

Key-value stores; Programmable switches; Caching

ACM Reference Format:
Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 17 pages. https://doi.org/10.1145/3132747.3132764

1 INTRODUCTION

Modern Internet services, such as search, social networking and e-commerce, critically depend on high-performance key-value stores. Rendering even a single web page often requires hundreds or even thousands of storage accesses [34]. So, as these services scale to billions of users, system operators increasingly rely on *in-memory* key-value stores to meet the necessary throughput and latency demands [32, 36, 38].

One major challenge in scaling a key-value store—whether in memory or not—is coping with skewed, dynamic workloads. Popular items receive far more queries than others, and the set of “hot items” changes rapidly due to popular posts, limited-time offers, and trending events [2, 11, 19, 21]. For example, prior studies have shown that 10% of items account for 60-90% of queries in the Memcached deployment at Facebook [2]. This skew can lead to severe load imbalance, which results in significant performance degradations: servers are either over- or under-utilized, throughput is reduced, and response times suffer from long tail latencies [14]. This degradation can be further amplified when storage servers use per-core sharding to handle high concurrency [5].

The problem of load imbalance is particularly acute for high-performance, in-memory key-value stores. While tra-

Example: NetChain & NetCache

NetCache: Bala
with Fast I

Xin Jin[1], Xiaozhou L
Jeongkeun Lee[2], Nate F
[1]Johns Hopkins University
[4]Università della Svizzera

ABSTRACT

We present NetCache, a new key-value store architectu
leverages the power and flexibility of new-generatio
grammable switches to handle queries on hot items a
ance the load across storage nodes. NetCache provide
aggregate throughput and low latency even under h
skewed and rapidly-changing workloads. The core
Cache is a packet-processing pipeline that exploits
pabilities of modern programmable switch ASICs
ciently detect, index, cache and serve hot key-value it
the switch data plane. Additionally, our solution guar
cache coherence with minimal overhead. We impler
NetCache prototype on Barefoot Tofino switches an
modity servers and demonstrate that a single switch c
cess 2+ billion queries per second for 64K items with 1
keys and 128-byte values, while only consuming a sm
tion of its hardware resources. To the best of our know
this is the first time that a sophisticated applicatio
functionality, such as in-network caching, has been
to run at line rate on programmable switches. Furthe
we show that NetCache improves the throughput by
and reduces the latency of up to 40% of queries by 5
high-performance, in-memory key-value stores.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; • **Net
→ Programmable networks**; **In-network proces
Computer systems organization** → **Cloud comp**

NetChain: Scale-Free Sub-RTT Coordination

Xin Jin[1], Xiaozhou Li[2], Haoyu Zhang[3], Nate Foster[2,4],
Jeongkeun Lee[2], Robert Soulé[2,5], Changhoon Kim[2], Ion Stoica[6]

[1]*Johns Hopkins University,* [2]*Barefoot Networks,* [3]*Princeton University,*
[4]*Cornell University,* [5]*Università della Svizzera italiana,* [6] *UC Berkeley*

Abstract

Coordination services are a fundamental building block of modern cloud systems, providing critical functionalities like configuration management and distributed locking. The major challenge is to achieve low latency and high throughput while providing strong consistency and fault-tolerance. Traditional server-based solutions require multiple round-trip times (RTTs) to process a query. This paper presents NetChain, a new approach that provides scale-free sub-RTT coordination in datacenters. NetChain exploits recent advances in programmable switches to store data and process queries entirely in the network data plane. This eliminates the query processing at coordination servers and cuts the end-to-end latency to as little as half of an RTT—clients only experience processing delay from their own software stack plus network delay, which in a datacenter setting is typically much smaller. We design new protocols and algorithms based on chain replication to guarantee strong consistency and to efficiently handle switch failures. We implement a prototype with four Barefoot Tofino switches and four commodity servers. Evaluation results show that compared to traditional server-based solutions like ZooKeeper, our prototype provides orders of magnitude higher throughput and lower latency, and handles failures gracefully.

1 Introduction

Coordination services (e.g., Chubby [1], ZooKeeper [2] and etcd [3]) are a fundamental building block of modern cloud systems. They are used to synchronize access to shared resources in a distributed system, providing critical functionalities such as configuration management, group membership, distributed locking, and barriers. These various forms of coordination are typically implemented on top of a *key-value store* that is replicated with a consensus protocol such as Paxos [4] for *strong consistency* and *fault-tolerance*.

DrTM [6], which can process hundreds of millions of transactions per second with a latency of tens of microseconds, crucially depend on fast distributed locking to mediate concurrent access to data partitioned in multiple servers. Unfortunately, acquiring locks becomes a significant bottleneck which severely limits the transaction throughput [7]. This is because servers have to spend their resources on (*i*) processing locking requests and (*ii*) aborting transactions that cannot acquire all locks under high-contention workloads, which can be otherwise used to execute and commit transactions. This is one of the main factors that led to relaxing consistency semantics in many recent large-scale distributed systems [8, 9], and the recent efforts to avoid coordination by leveraging application semantics [10, 11]. While these systems are successful in achieving high throughput, unfortunately, they restrict the programming model and complicate the application development. A fast coordination service would enable high transaction throughput without any of these compromises.

Today's server-based solutions require multiple end-to-end round-trip times (RTTs) to process a query [1, 2, 3]: a client sends a request to coordination servers; the coordination servers execute a consensus protocol, which can take several RTTs; the coordination servers send a reply back to the client. Because datacenter switches provide sub-microsecond per-packet processing delay, the query latency is dominated by host delay which is tens to hundreds of microseconds for highly-optimized implementations [12]. Furthermore, as consensus protocols do not involve sophisticated computations, the workload is communication-heavy and the throughput is bottlenecked by the server IO. While state-of-the-art solutions such as NetBricks [12] can boost a server to process tens of millions of packets per second, it is still orders of magnitude slower than a switch.

We present NetChain, a new approach that leverages the power and flexibility of new-generation pro-

This Talk

An practical property-checking tool for P4-programmable data planes

This Talk

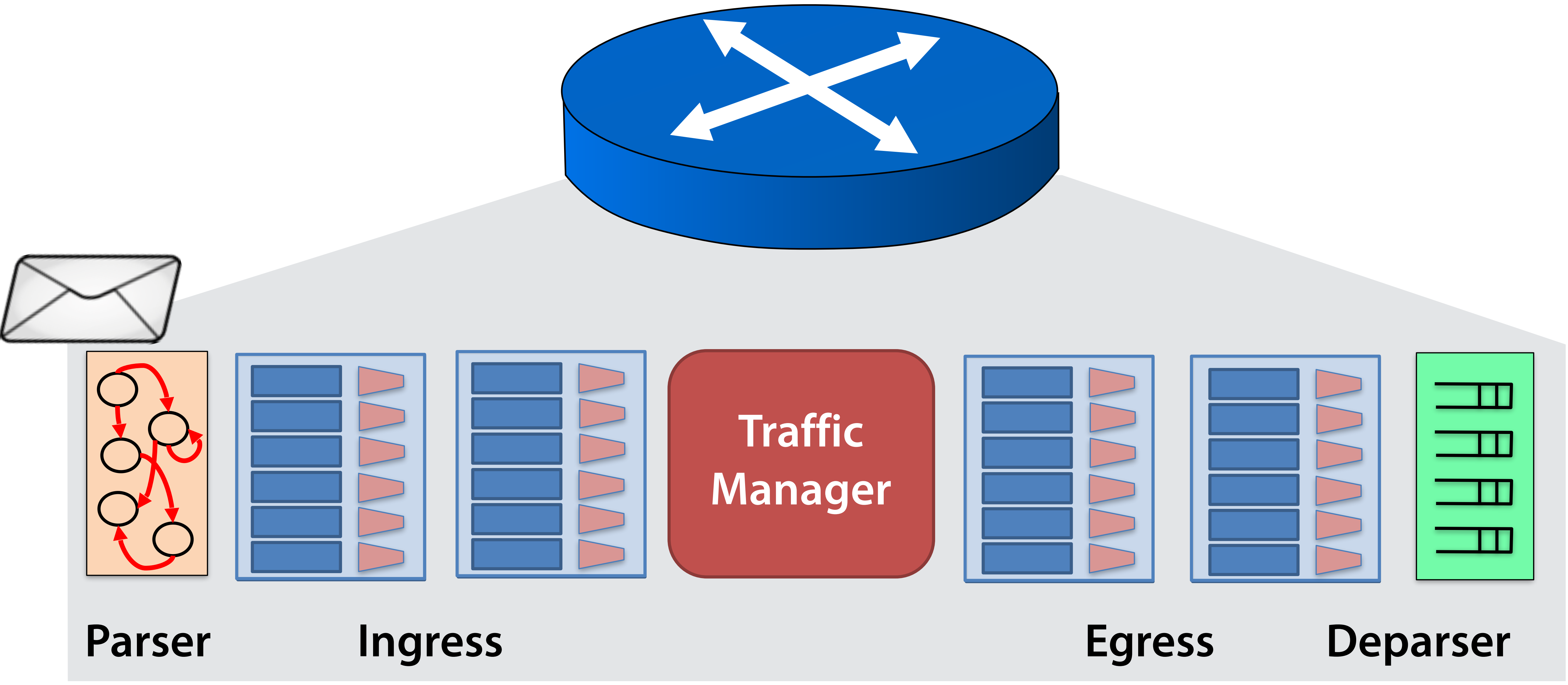
An practical property-checking tool for P4-programmable data planes

Plan:

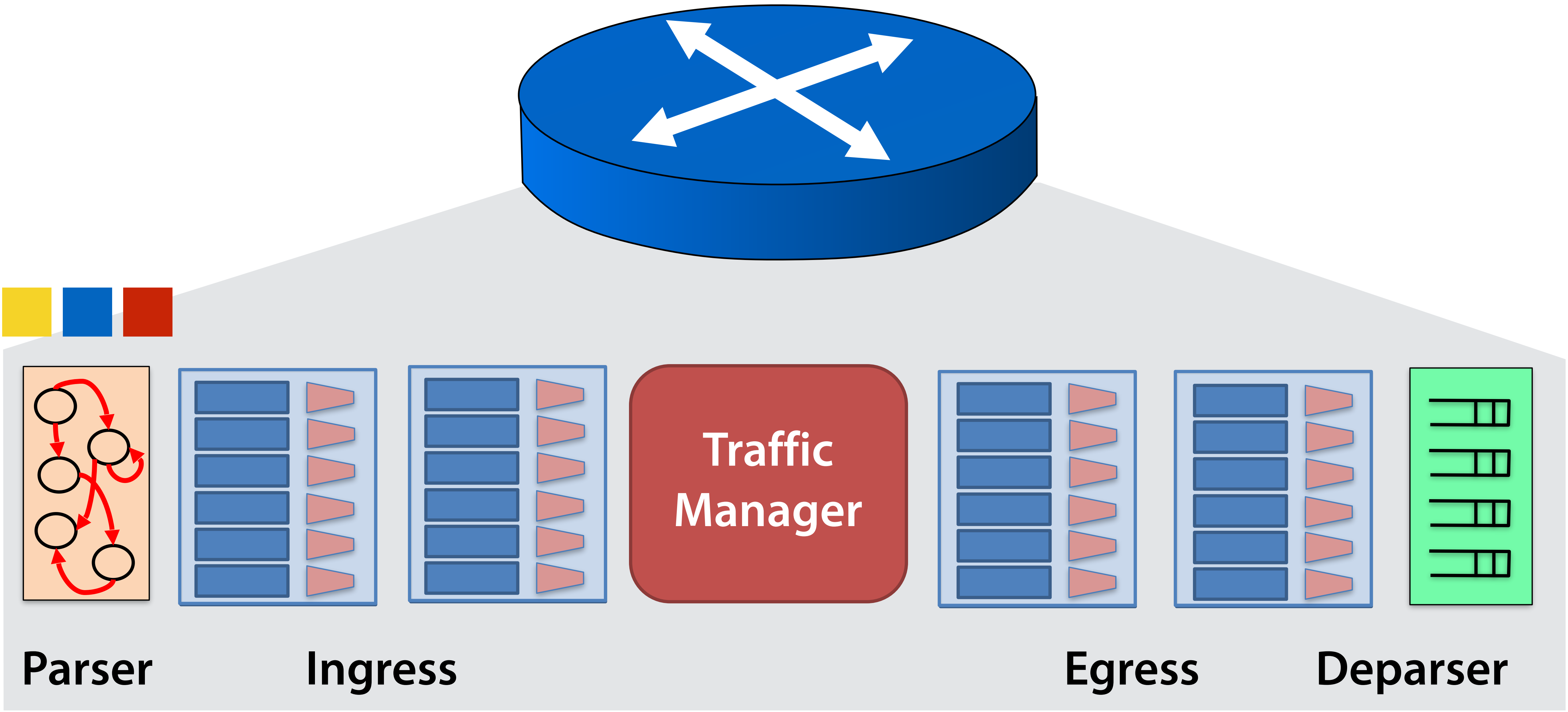
- Background on P4 language
- Verification approach
- Experience

Background

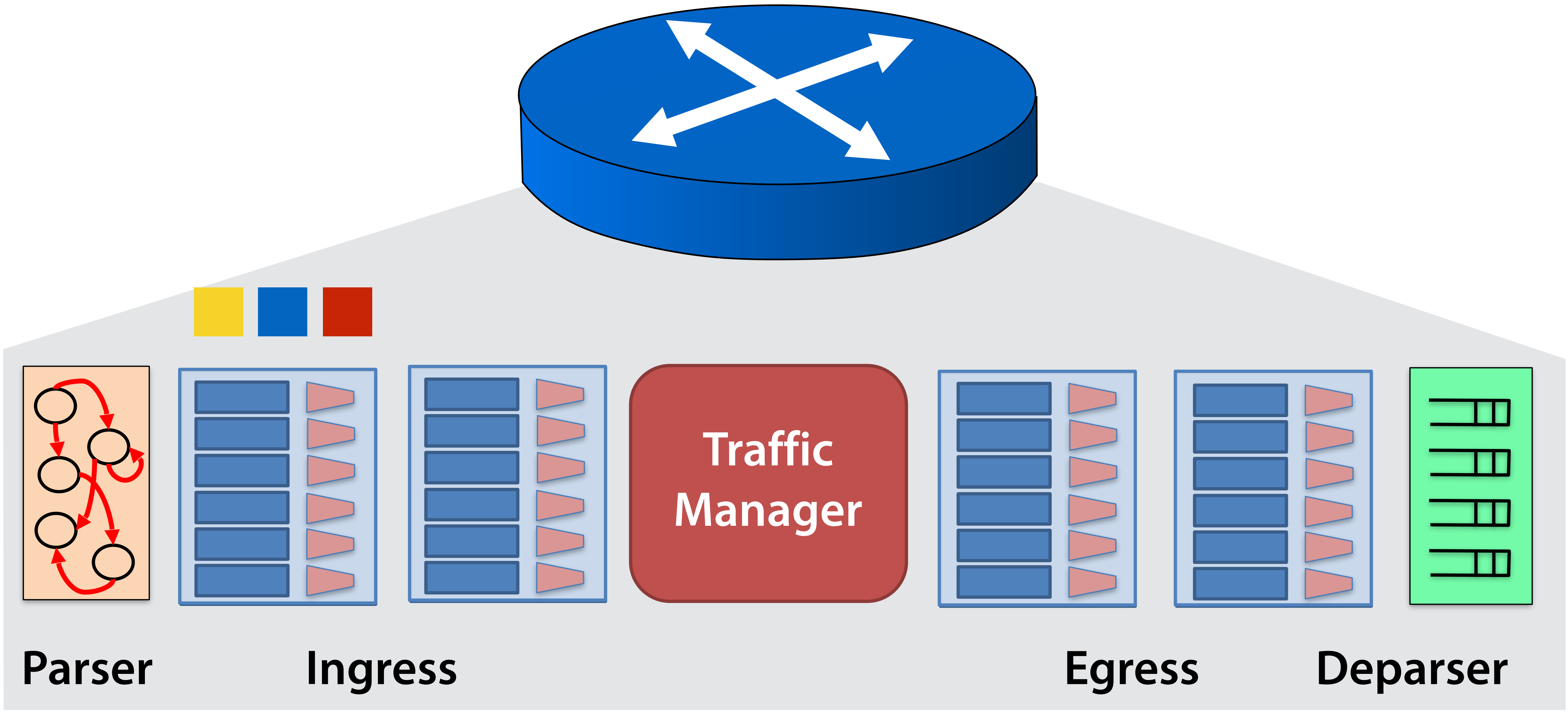
PISA [SIGCOMM '13]



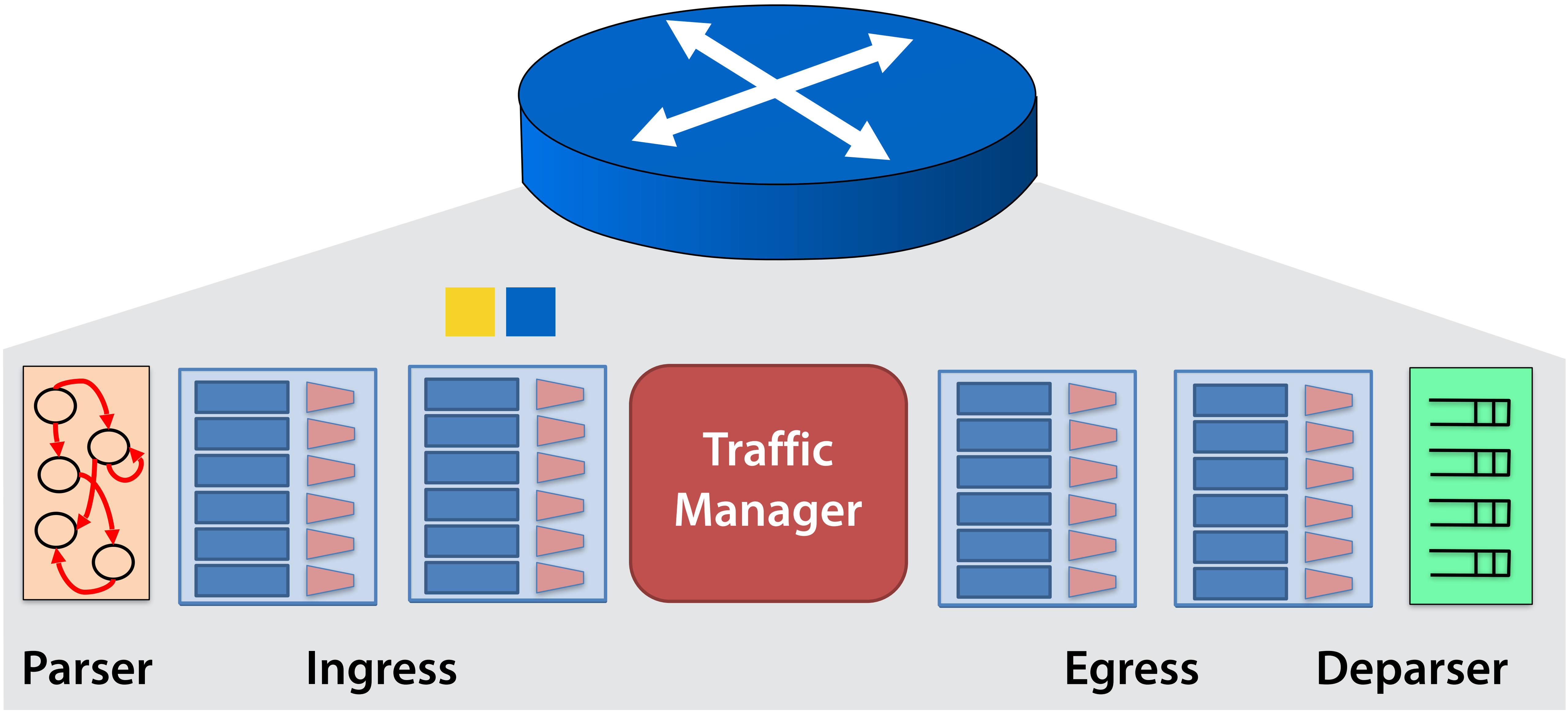
PISA [SIGCOMM '13]



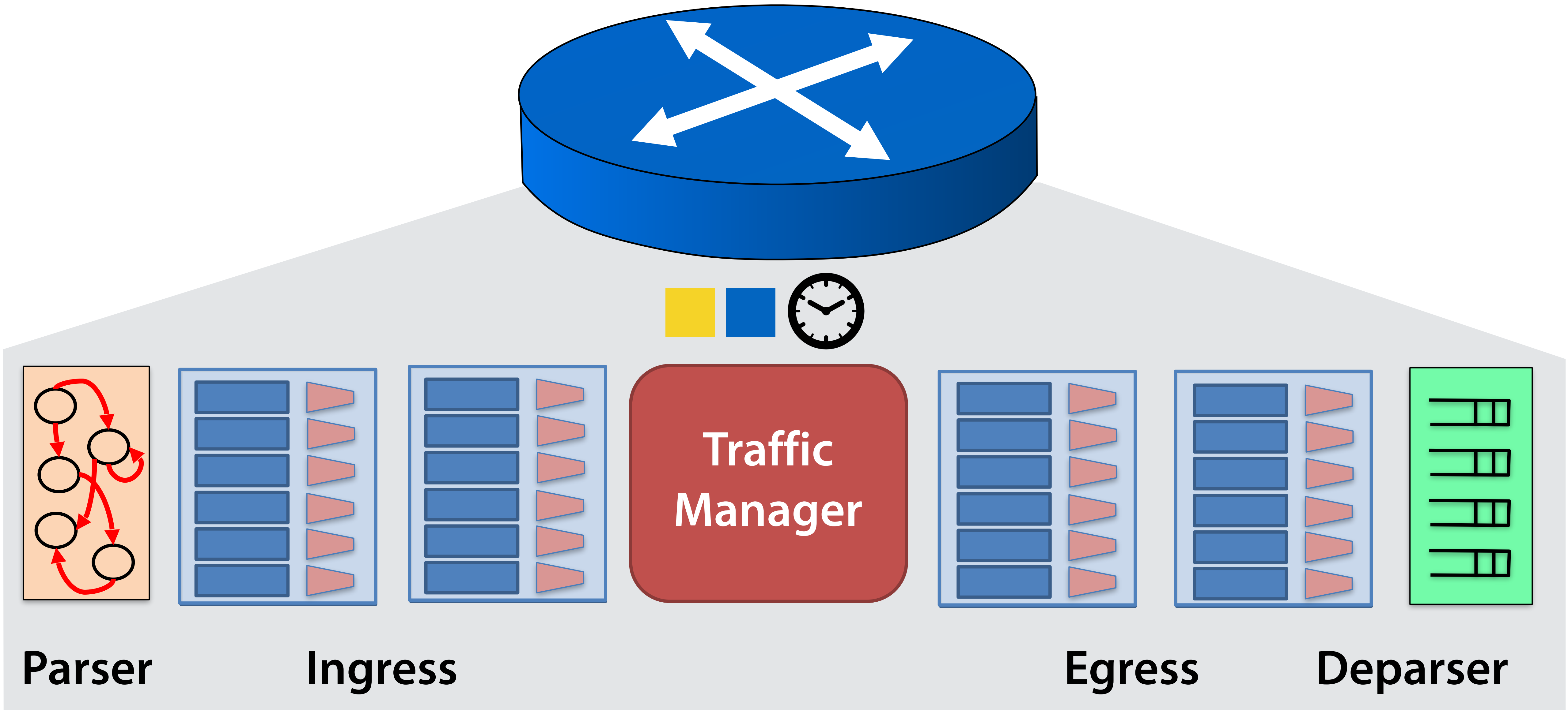
PISA [SIGCOMM '13]



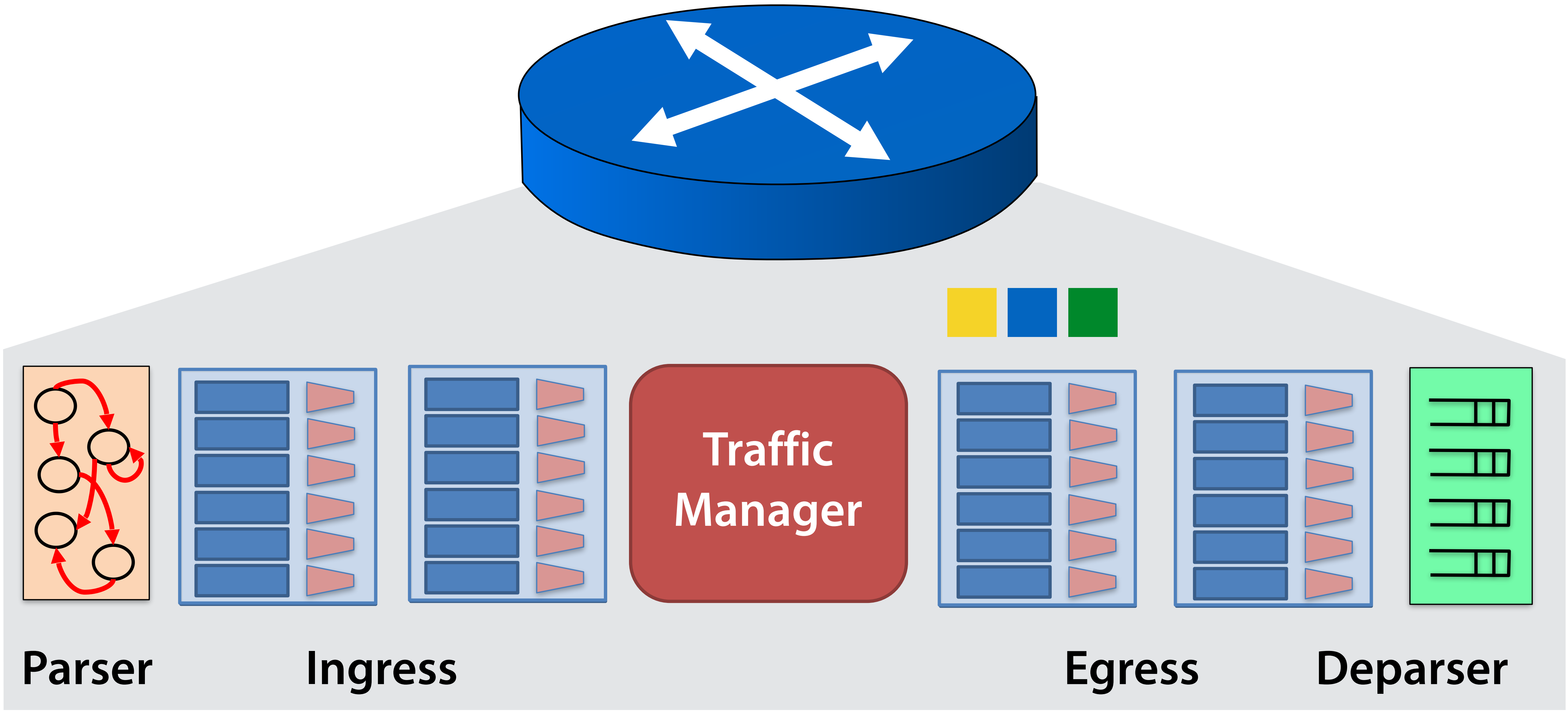
PISA [SIGCOMM '13]



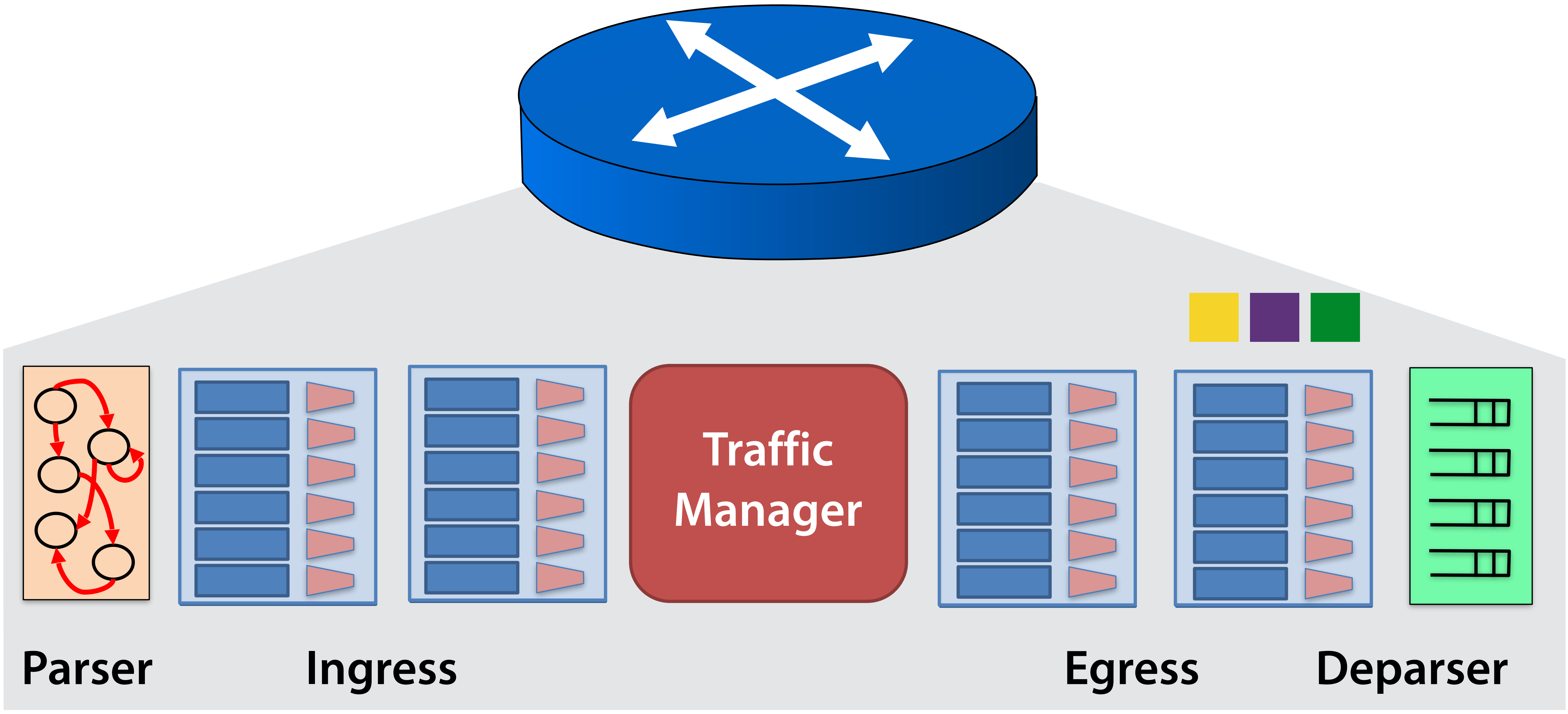
PISA [SIGCOMM '13]



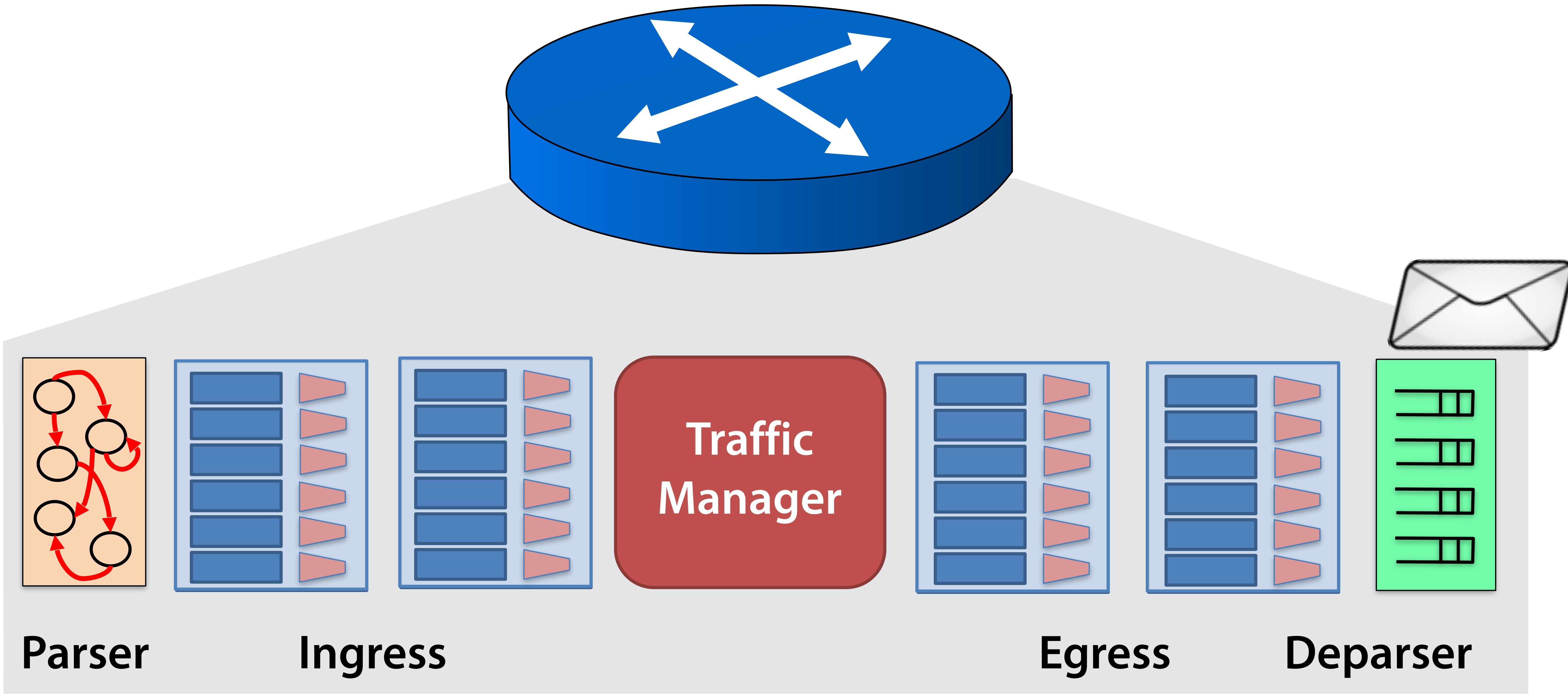
PISA [SIGCOMM '13]



PISA [SIGCOMM '13]



PISA [SIGCOMM '13]



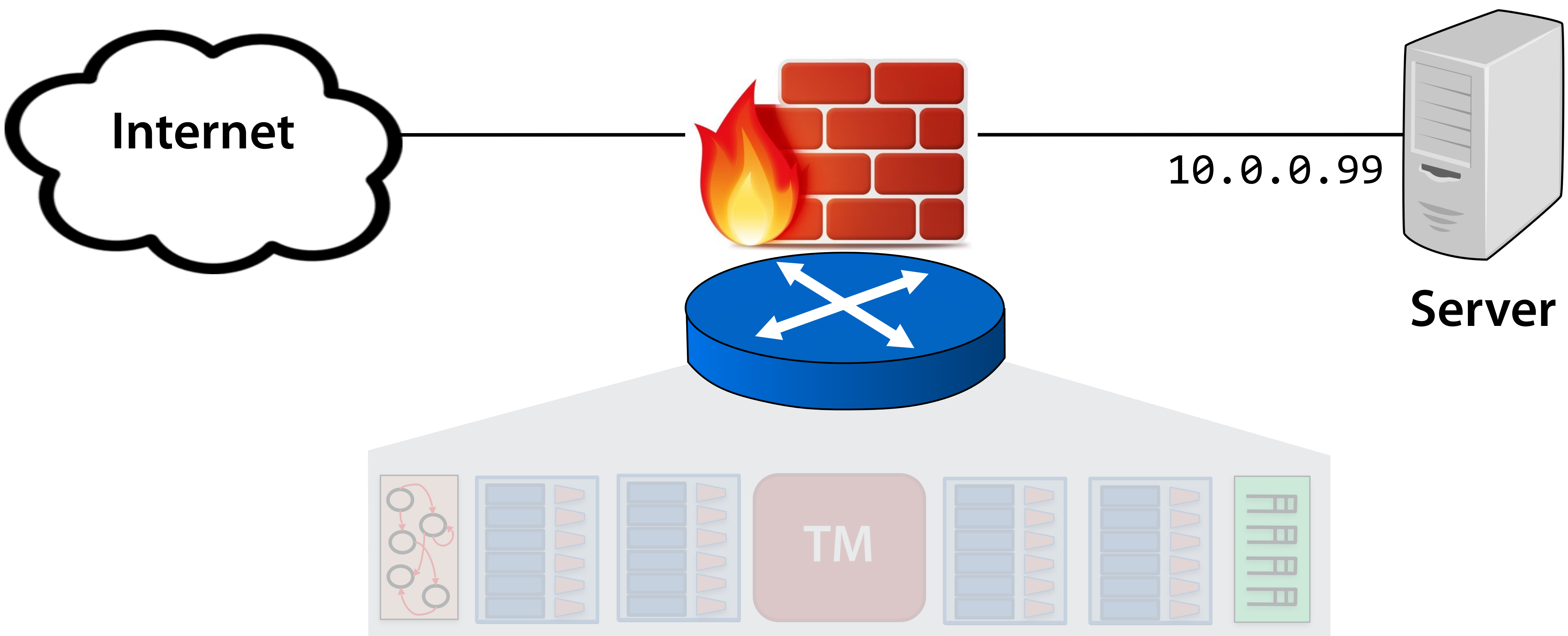
P4 Language

- Domain-specific parsers and match-action tables
- Standard imperative features (types and control flow)
- Slogan: "constant work in constant time"
  - Bounded state
  - No loops

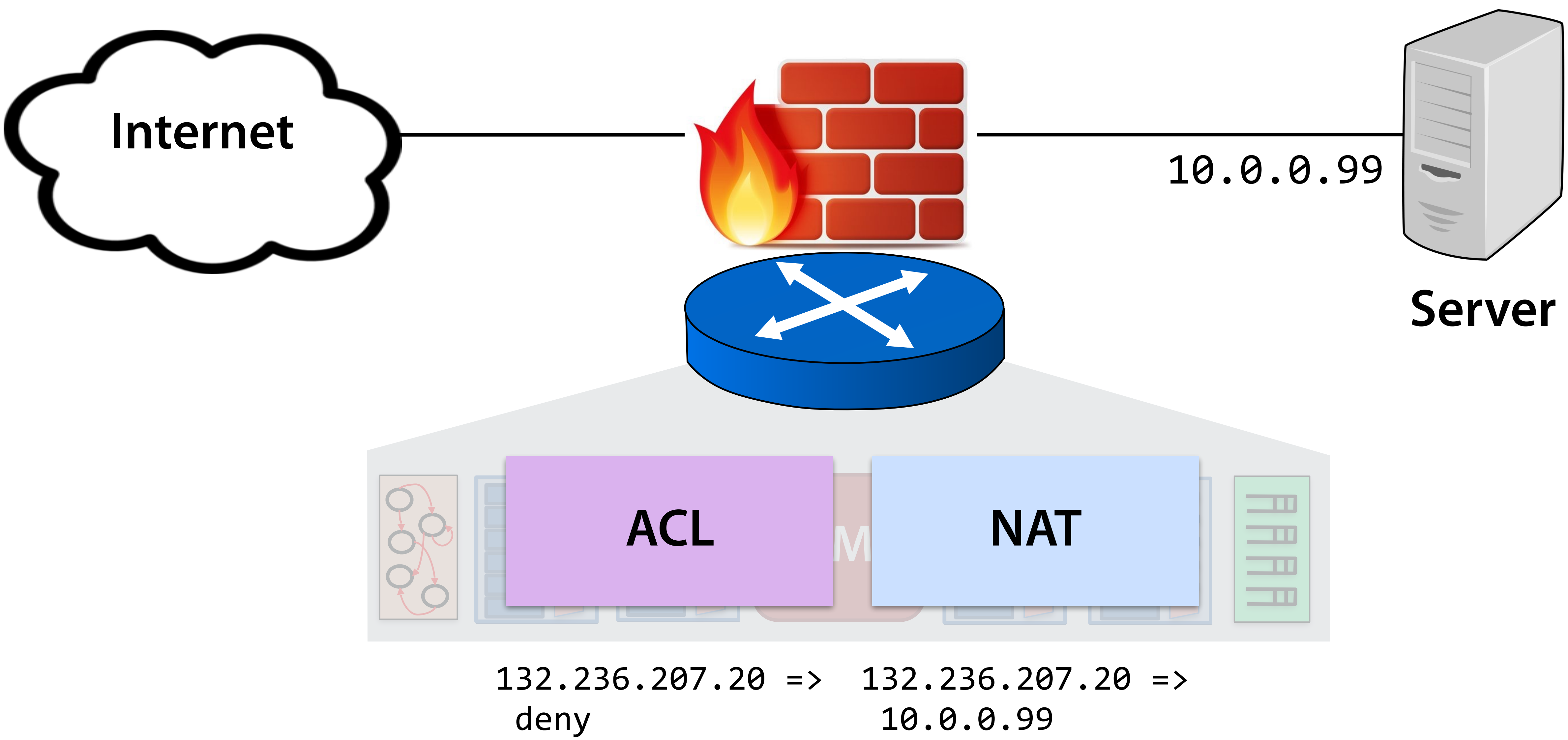
```
action allow() {
  modify_field(std_meta.egress_spec,1);
}
action deny() {
  drop ();
}
table acl {
  reads {
    ipv4.srcAddr : lpm;
    ipv4.dstAddr : lpm;
  }
  actions { allow; deny; }
  default_action: deny;
}
```

Match	Actions
1.2.3.4	deny
*	allow

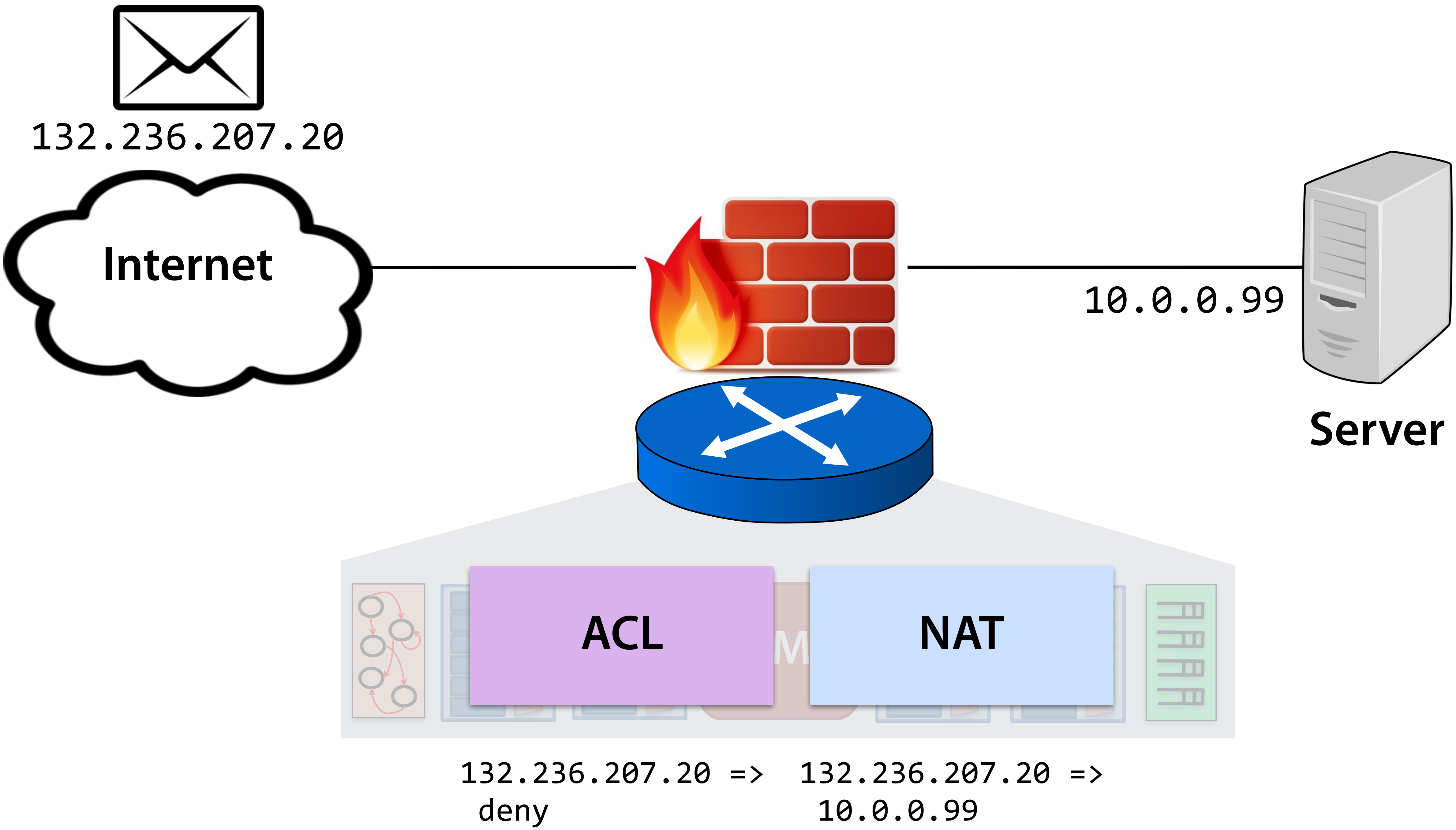
Example: Firewall



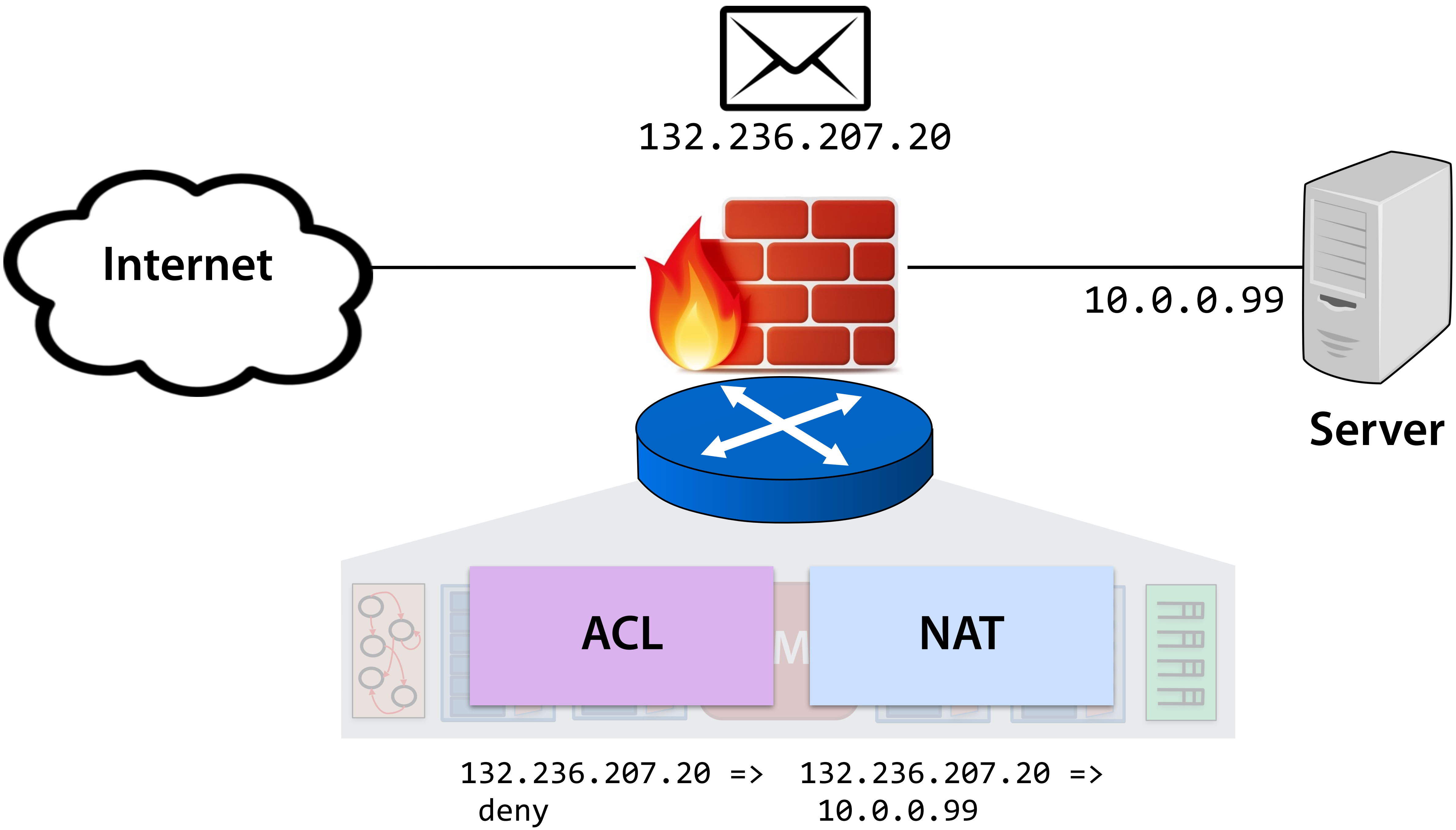
Example: Firewall



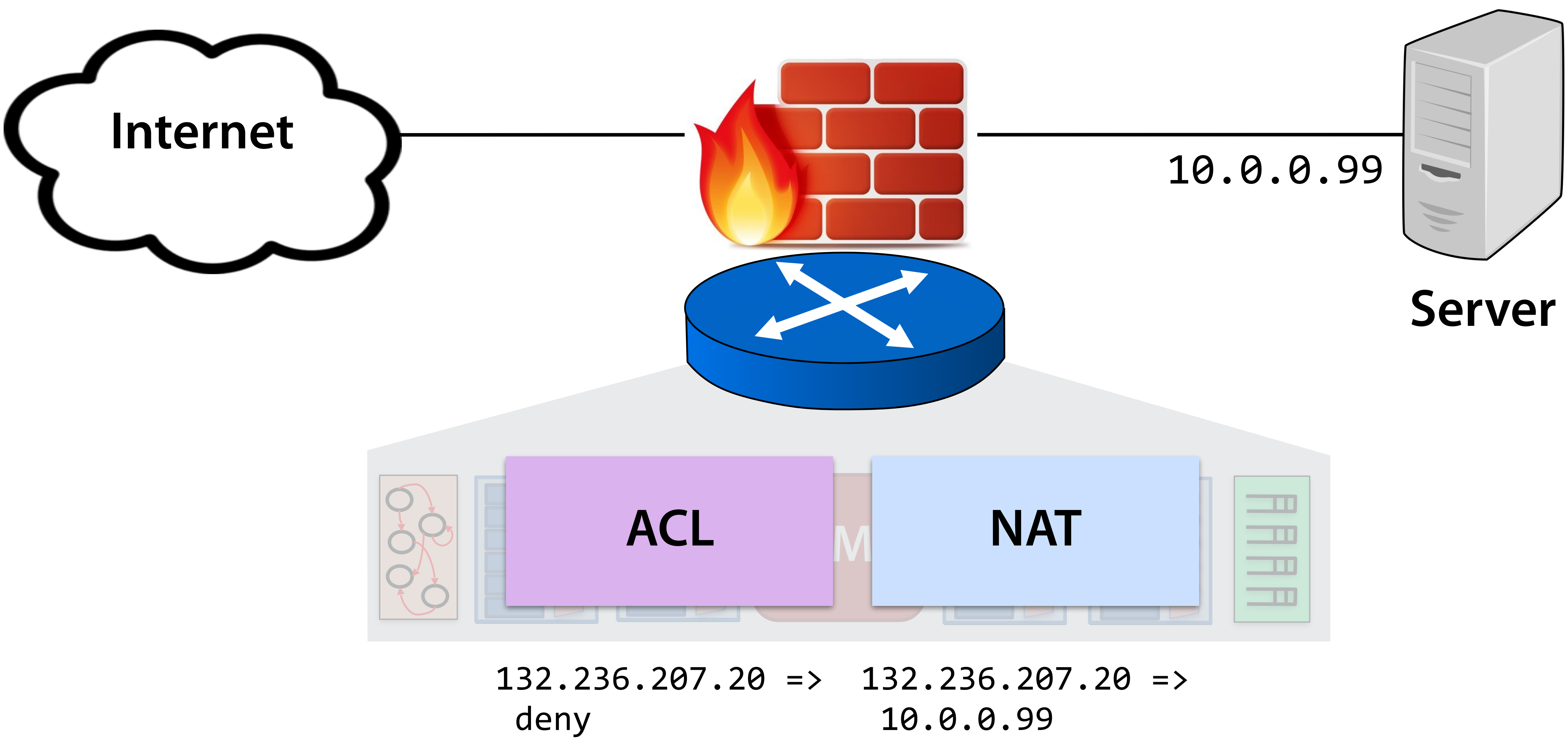
Example: Firewall



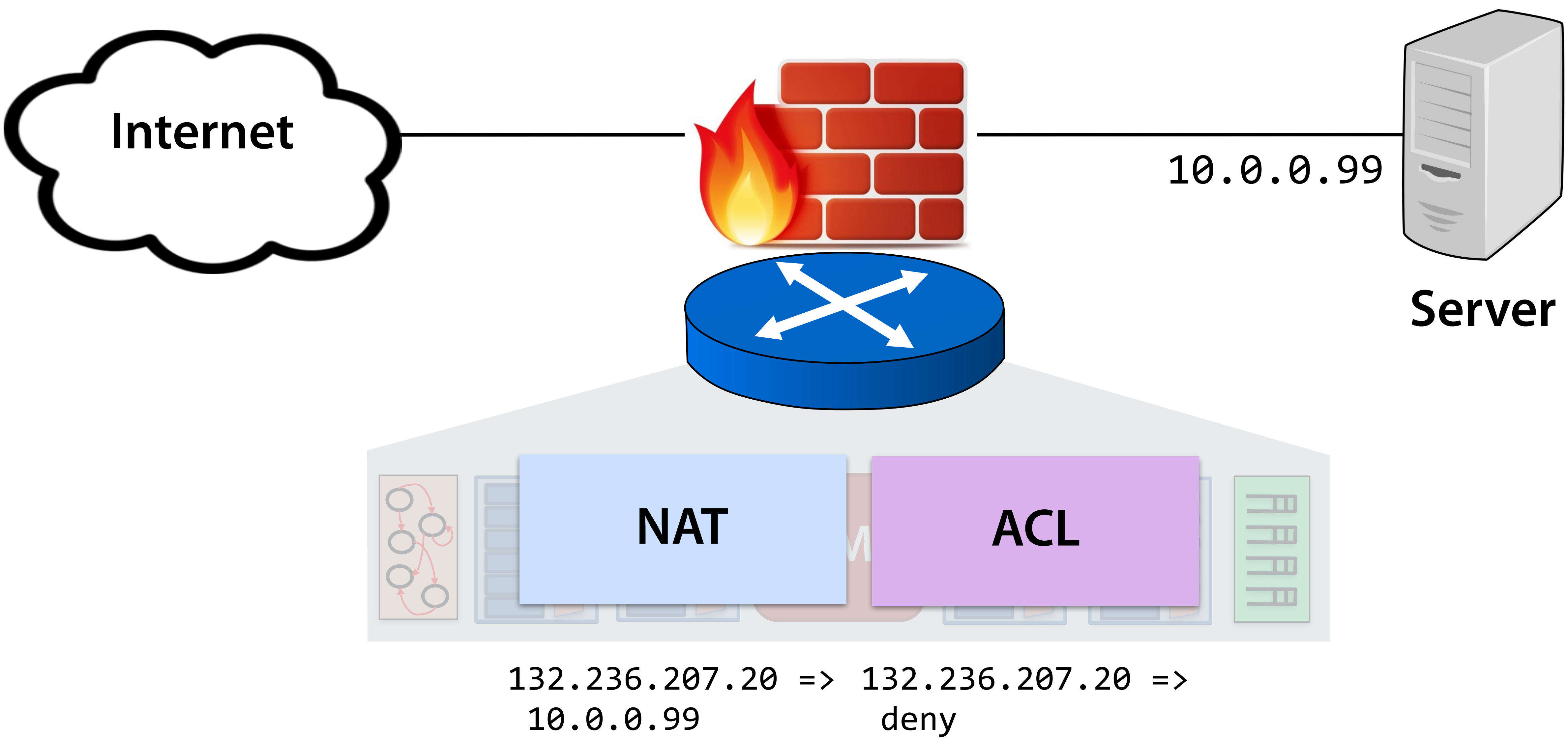
Example: Firewall



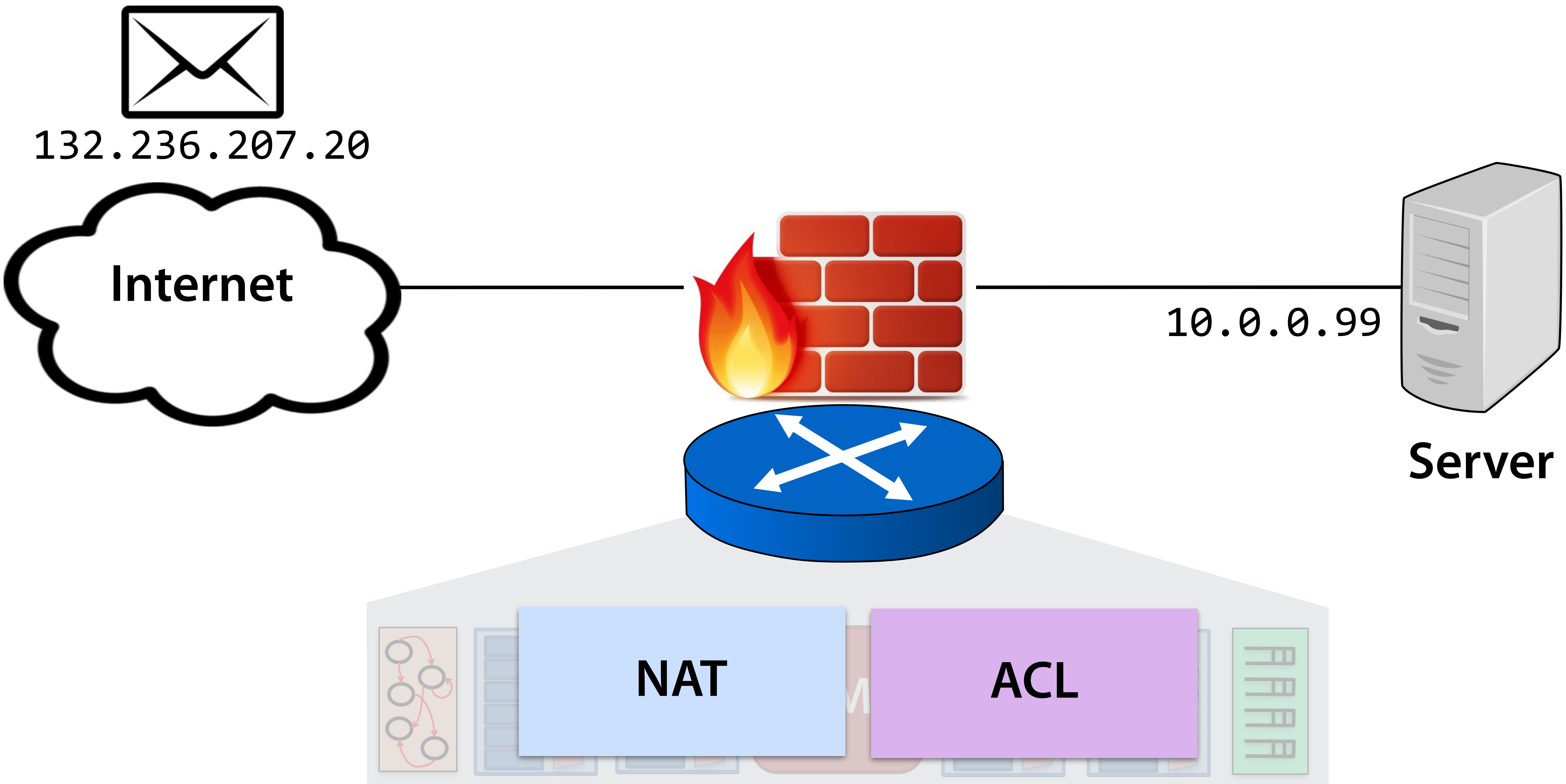
Example: Firewall



Example: Firewall



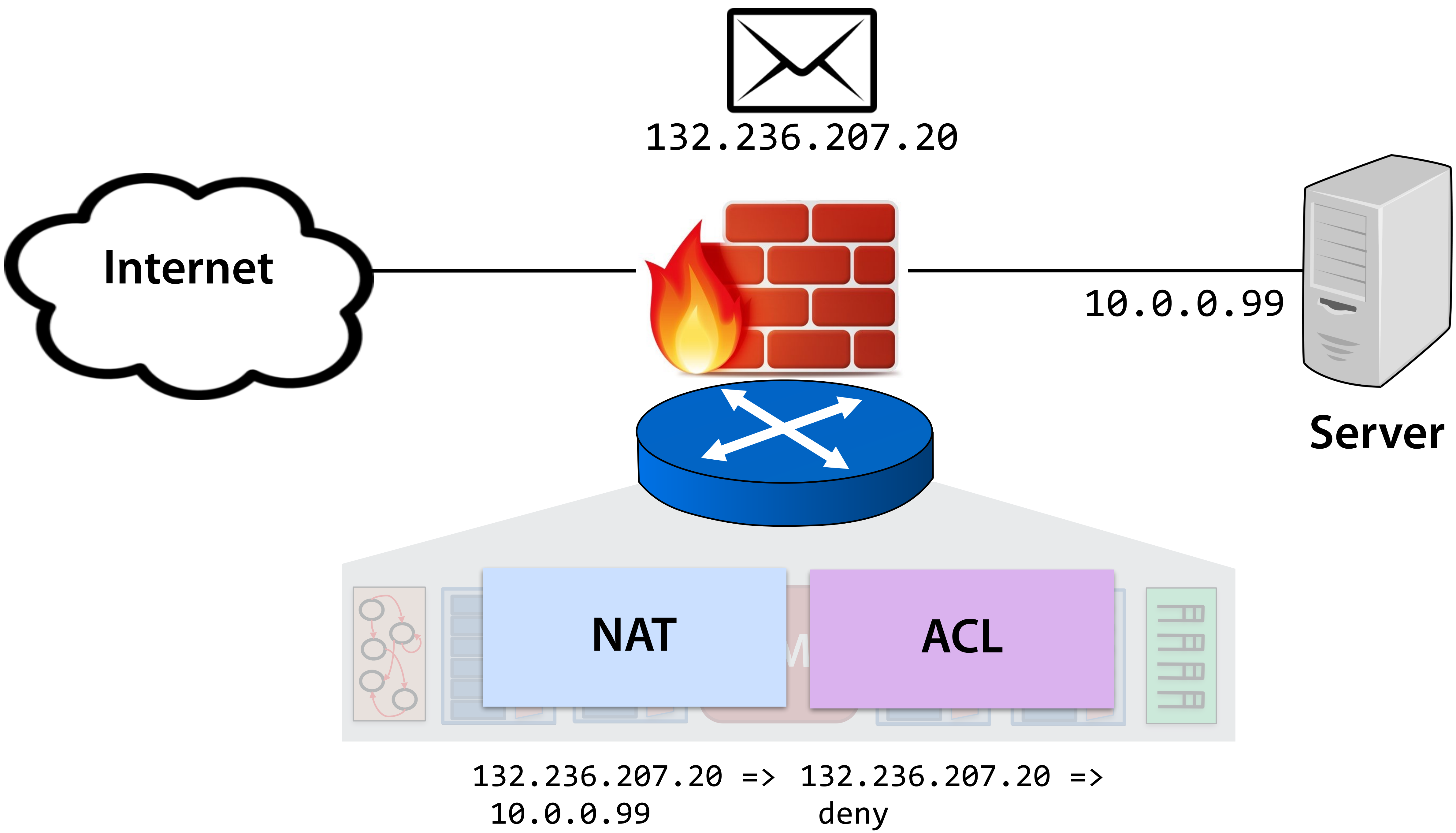
Example: Firewall



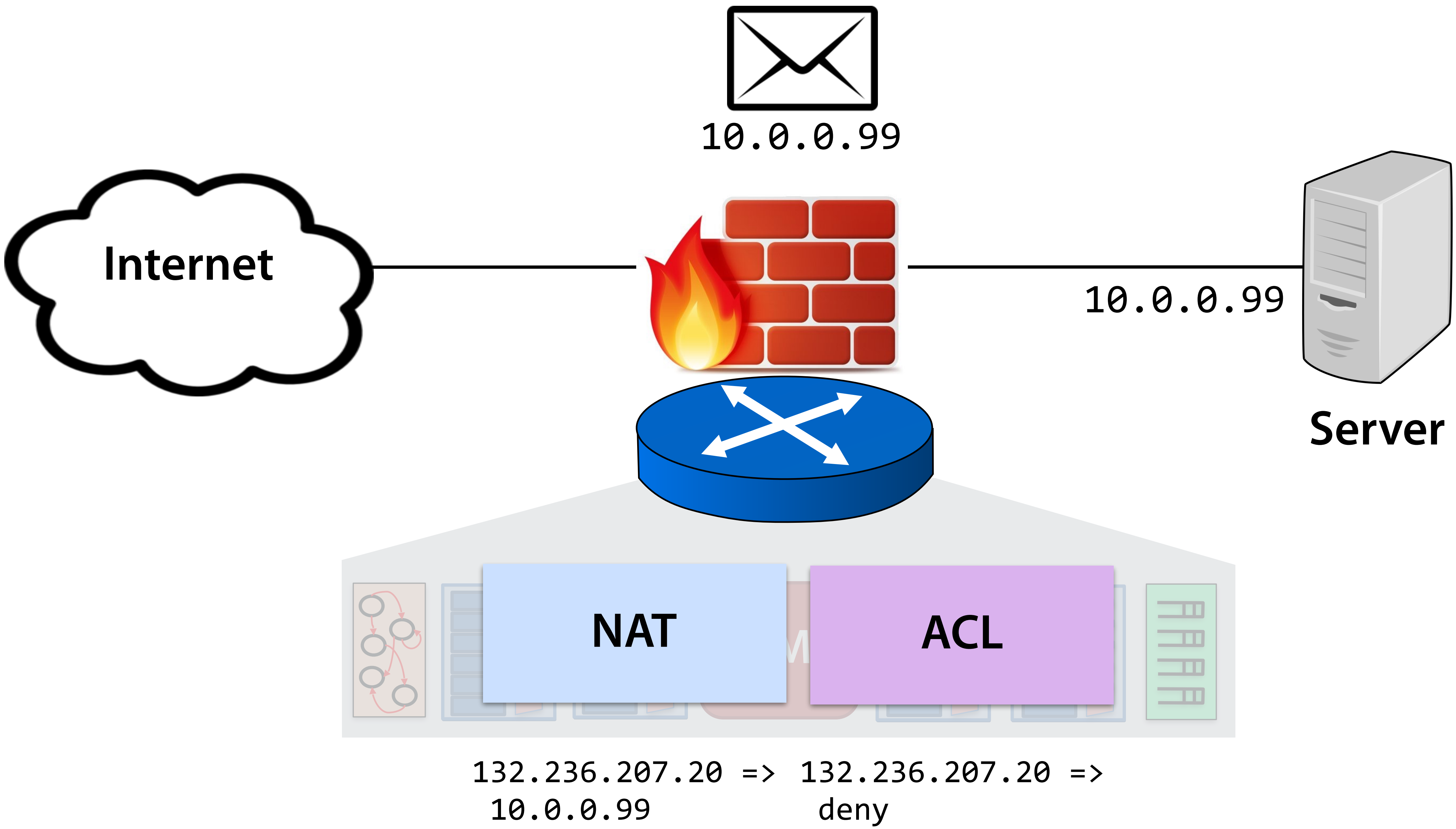
132.236.207.20 => 10.0.0.99

132.236.207.20 => deny

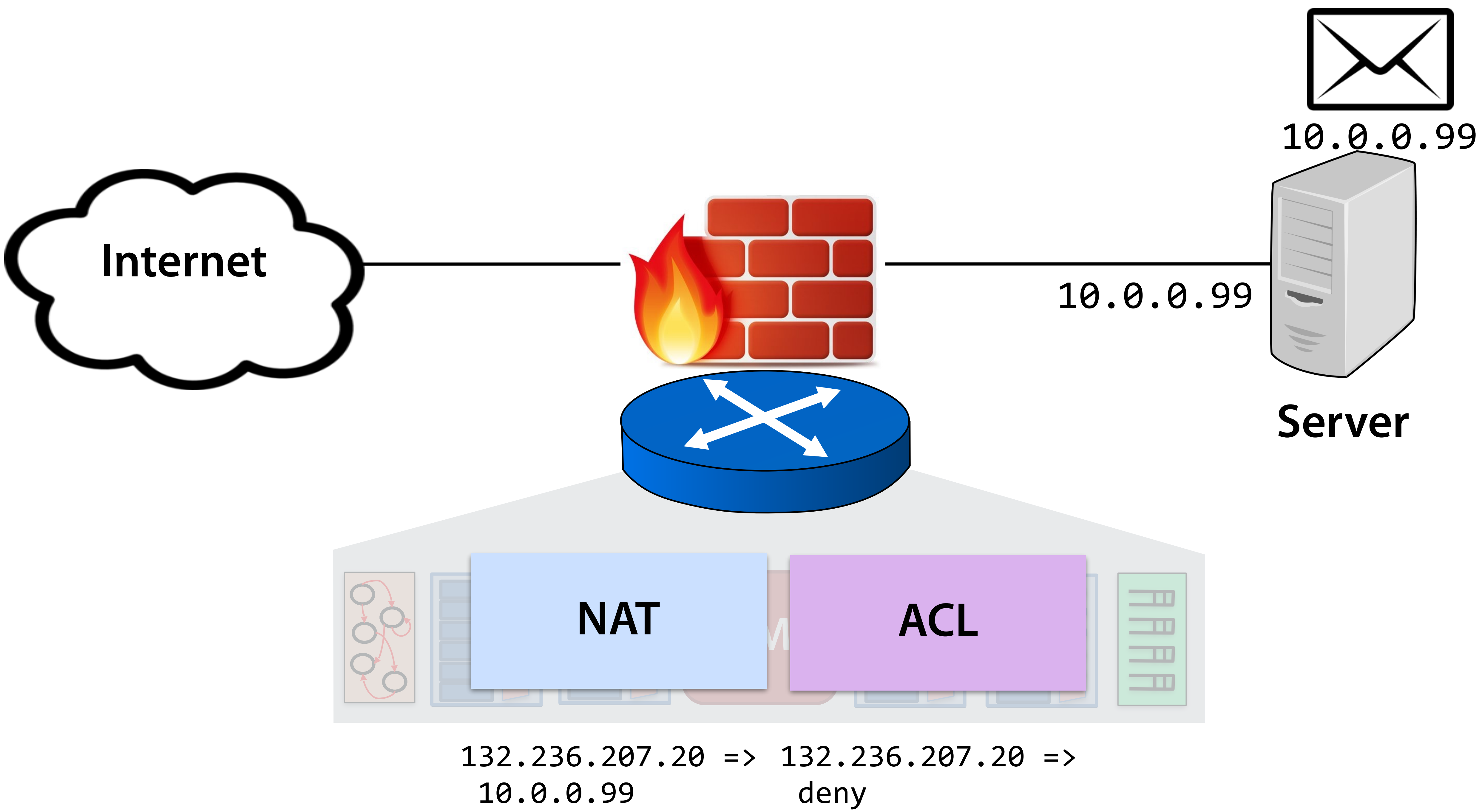
Example: Firewall



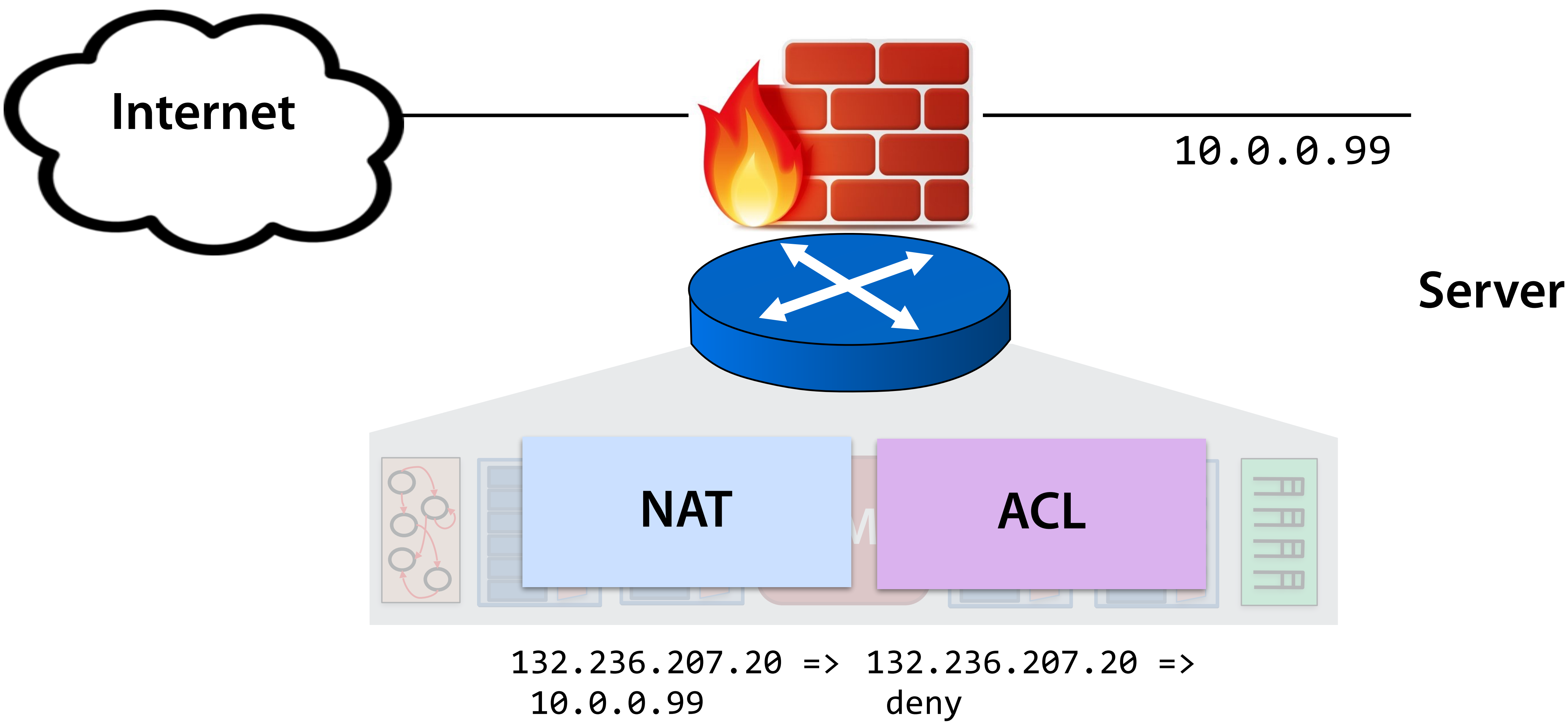
Example: Firewall



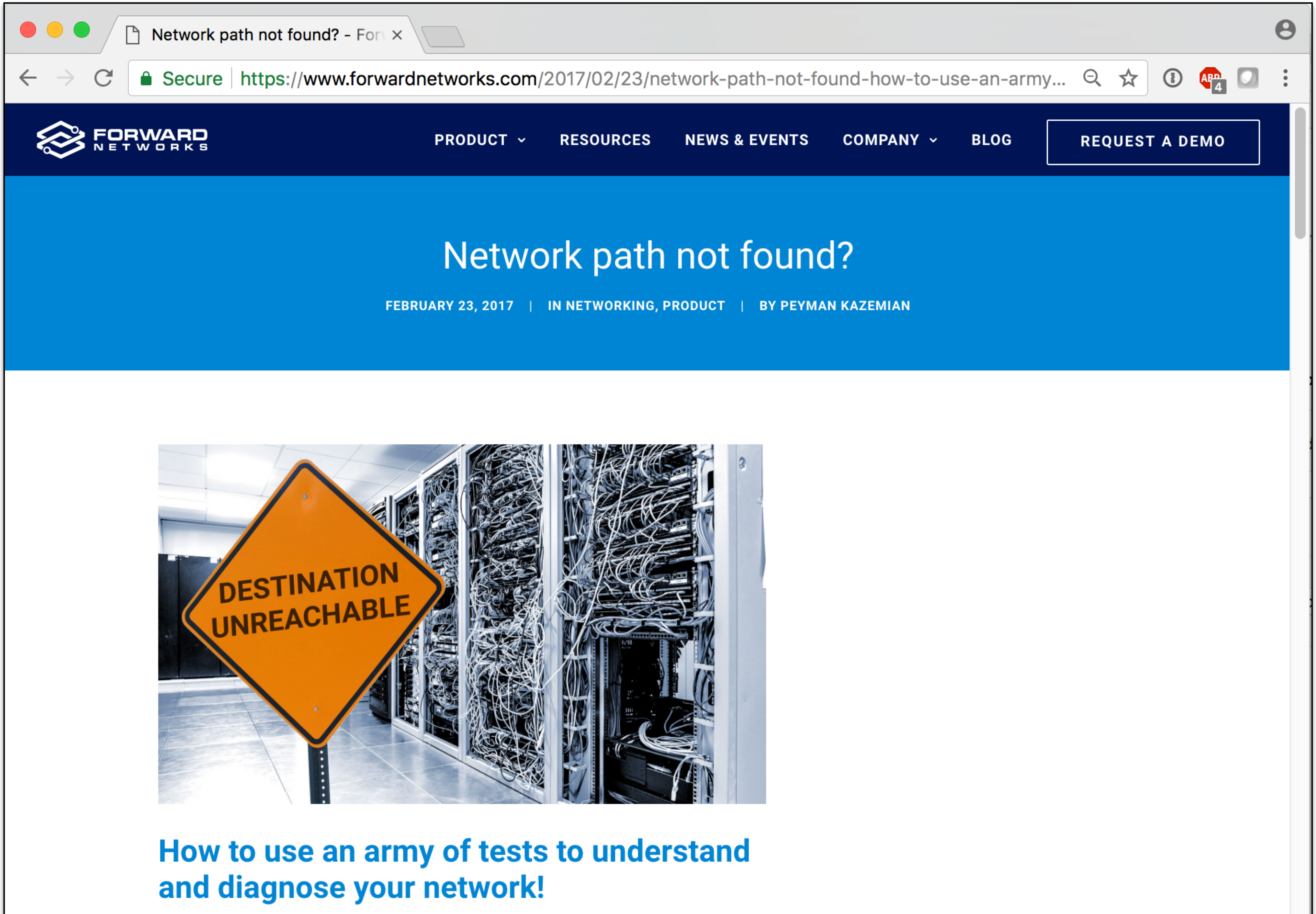
Example: Firewall



Example: Firewall



This is not a toy example!



Demo: Firewall

Types

```
/* Headers and Instances */
header_type ethernet_t {
 fields {
  dst_addr:48;
  src_addr:48;
  ether_type:16;
 }
}
header_type ipv4_t {
 fields {
  pre_tcl:64;
  ttl:8;
  protocol:8;
  checksum: 16;
  src_addr:32;
  dst_addr: 32;
 }
}
header ethernet_t ethernet;
header ipv4_t ipv4;
```

Parsers

```
/* Parsers */
parser start
  extract(ethernet);
  return select(ethernet.ether_type) {
    0x800: parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return ingress;
}
```

Actions

```
/* Actions */
action allow() {
 modify_field(standard_metadata.egress_spec,1);
}
action deny() { drop(); }
action nop() { }
action rewrite(src_addr,dst_addr) {
  modify_field(ipv4.dst_addr,src_addr);
  modify_field(ipv4.dst_addr,dst_addr);
}
```

Tables

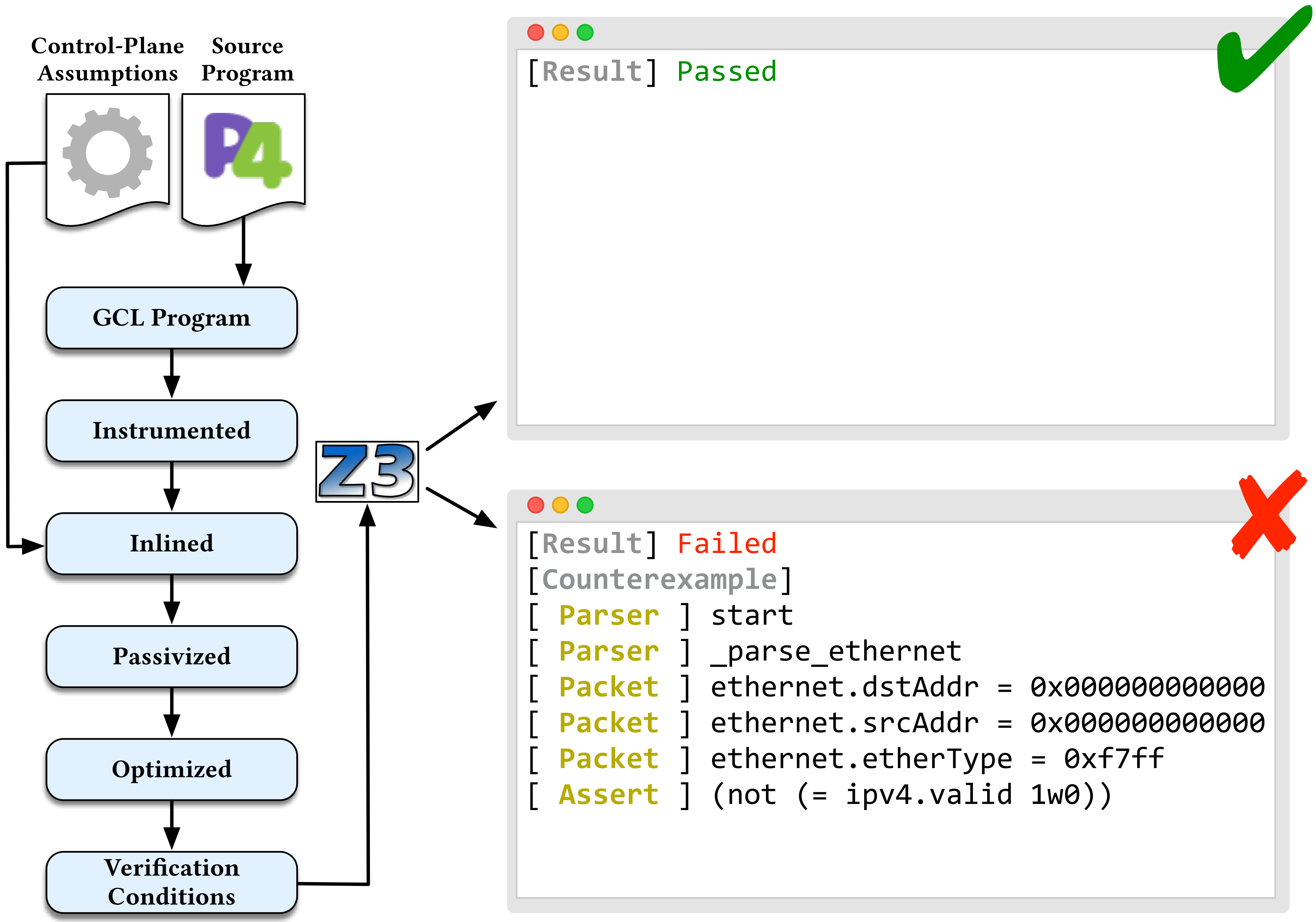
```
/* Tables */
table acl {
 reads {
  ipv4.src_addr:lpm;
  ipv4.dst_addr:lpm;
 }
 actions { allow; deny; }
 size: 1024;
}

table nat {
  reads { ipv4.dst_addr:lpm; }
  actions { rewrite; nop; }
  default_action: nop();
  size: 8192;
}
```

Controls

```
/* Controls */
control ingress {
 apply(acl);
 apply(nat);
}
```

Verification Approach

Overview



Control-Plane Assumptions
Source Program
GCL Program
Instrumented
Inlined
Passivized
Optimized
Verification Conditions
Z3
[Result] Passed
[Result] Failed
[Counterexample]
[Parser] start
[Parser] _parse_ethernet
[Packet] ethernet.dstAddr = 0x000000000000
[Packet] ethernet.srcAddr = 0x000000000000
[Packet] ethernet.etherType = 0xf7ff
[Assert] (not (= ipv4.valid 1w0))

Guarded Commands

```
e ::= n | x | e1 + e2 | ... (* Expressions *)

ϕ ::= e1 = e2 | ϕ1 ∧ ϕ2 | ...  (* Formulas *)

c ::= assume ϕ                  (* Commands *)
    | assert ϕ
    | c1; c2
    | x := e
    | c1 [] c2
```

Verification Condition Generation

```
wlp(assume ψ, ϕ) ≜ ψ ⇒ ϕ

wlp(assert ψ, ϕ) ≜ ψ ∧ ϕ

wlp(c1; c2, ϕ) ≜ wlp(c1, wlp(c2, ϕ))

wlp(x := e, ϕ) ≜ ϕ[e / x]

wlp(c1 [] c2, ϕ) ≜ wlp(c1, ϕ) ∧ wlp(c2, ϕ)
```

Verification Condition Generation

$$\mathrm{wlp}(\mathtt{assume}\ \psi,\ \phi) \triangleq \psi \Rightarrow \phi$$

$$\mathrm{wlp}(\mathtt{assert}\ \psi,\ \phi) \triangleq \psi \wedge \phi$$

$$\mathrm{wlp}(c1;\ c2,\ \phi) \triangleq \mathrm{wlp}(c1,\ \mathrm{wlp}(c2,\ \phi))$$

$$\mathrm{wlp}(x := e,\ \phi) \triangleq \phi[e\ /\ x]$$

$$\mathrm{wlp}(c1\ [\,]\ c2,\ \phi) \triangleq \mathrm{wlp}(c1,\ \phi) \wedge \mathrm{wlp}(c2,\ \phi)$$

Can generate *efficient* preconditions using the translation due to Saxe and Flanagan [POPL '01]

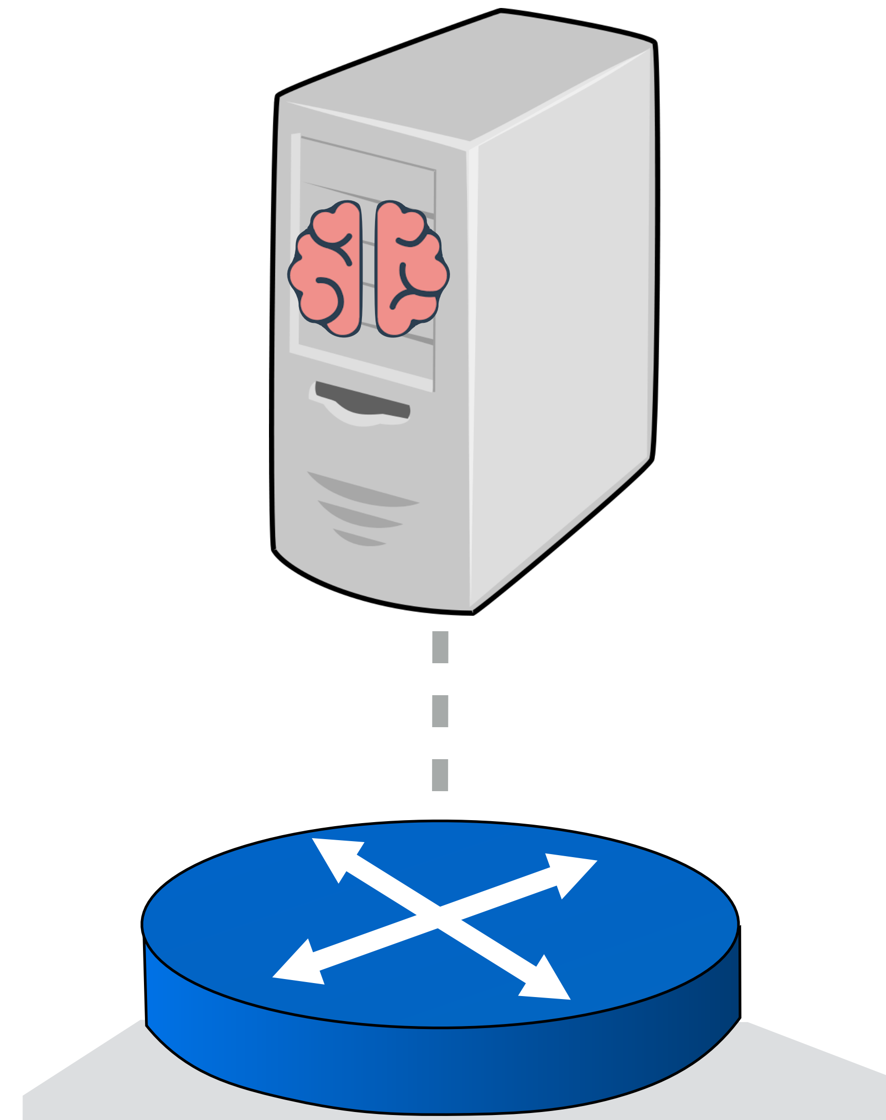
Challenge: Modeling Control Plane

A P4 program is really only half of a program...

The match-action tables are populated by the control plane which is unknown!

Formally, table application is translated to a non-deterministic choice between actions or miss

In general, to verify realistic programs we need to model the behavior of the control plane



Match	Action
1.2.3.4	deny
*	accept

Solution: Ghost State

```
$p4v_zombie.reach$acl := 1w0;
$p4v_zombie.hit$acl := 1w0;
$p4v_zombie.action$acl := 2w0;
$p4v_zombie.reads$acl$0 := 32w0;
$p4v_zombie.reads$acl$1 := 32w0;
```

Solution: Ghost State

```
 $p4v_zombie.reach$acl := 1w0;
 $p4v_zombie.hit$acl := 1w0;
 $p4v_zombie.action$acl := 2w0;
 $p4v_zombie.reads$acl$0 := 32w0;
 $p4v_zombie.reads$acl$1 := 32w0;
```

```
apply(acl);
```

```
$p4v_zombie.reach$acl := 1w1;
$p4v_zombie.reads$acl$0 := ipv4.src_addr;
$p4v_zombie.reads$acl$1 := ipv4.dst_addr;
@[ Match ] acl;
{($p4v_zombie.hit$acl := 1w1;
  { (@[ Action ] acl <hit> (allow);
     $p4v_zombie.action$acl := 2w1;
     standard_metadata.egress_spec := 9w1;
  [](@[ Action ] acl <hit> (deny);
     $p4v_zombie.action$acl := 2w2;
     standard_metadata.egress_spec := 9w511) })
[]($p4v_zombie.hit$acl := 1w0)
   @[ Action ] acl <miss>) };
```

Solution: Ghost State

```
$p4v_zombie.reach$acl := 1w0;
$p4v_zombie.hit$acl := 1w0;
```

```
assume
  reads(acl, ip4v.dst_addr) == 132.236.207.20
implies
  action(acl) == deny
```

```
$p4v_zombie.reach$acl := 1w1;
$p4v_zombie.reads$acl$0 := ipv4.src_addr;
$p4v_zombie.reads$acl$1 := ipv4.dst_addr;
@[ Match ] acl;
{($p4v_zombie.hit$acl := 1w1;
  { (@[ Action ] acl <hit> (allow);
     $p4v_zombie.action$acl := 2w1;
     standard_metadata.egress_spec := 9w1;
  [](@[ Action ] acl <hit> (deny);
     $p4v_zombie.action$acl := 2w2;
     standard_metadata.egress_spec := 9w511) })
[]($p4v_zombie.hit$acl := 1w0)
   @[ Action ] acl <miss>) };
```

Solution: Ghost State

```
$p4v_zombie.reach$acl := 1w0;
```

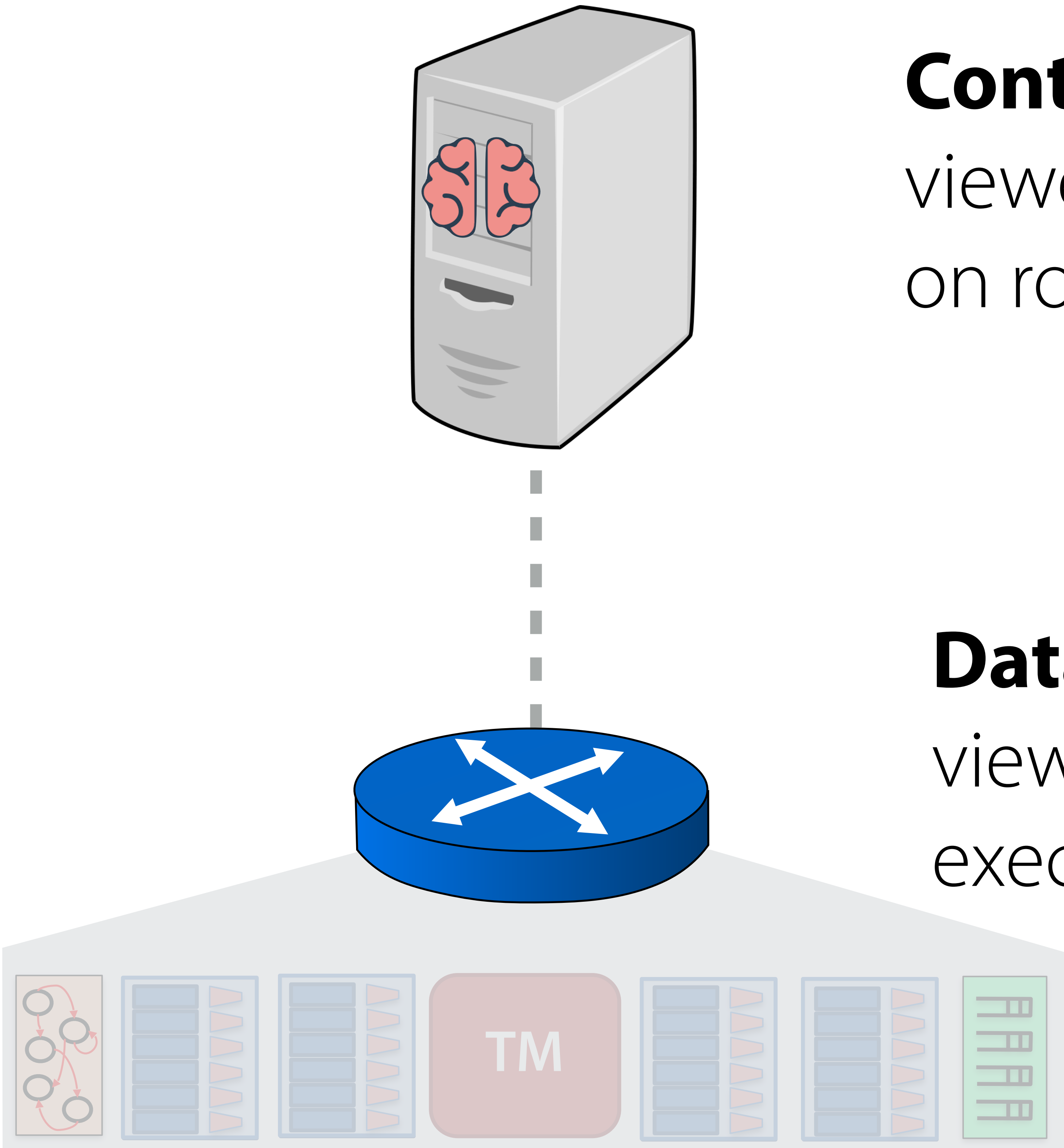
```
assume
  reads(acl, ip4v.dst_addr) == 132.236.207.20
implies
  action(acl) == deny
```

```
$p4v_zombie.reach$acl := 1w1;
$p4v_zombie.reads$acl$0 := ipv4.src_addr;
```

```
assume
  $p4v_zombie.reads$acl$1 == 32w2230112020
implies
  $p4v_zombie.action$acl == 2w2
```

```
    standard_metadata.egress_spec := 9w511) })
[]($p4v_zombie.hit$acl := 1w0)
  @[ Action ] acl <miss>) };
```

Subtlety: How to bridge two views?



Control-plane: behavior viewed in terms of an invariant on router configuration

Data-plane: behavior viewed through the lens of an execution of the P4 program

Subtlety: How to bridge two views?

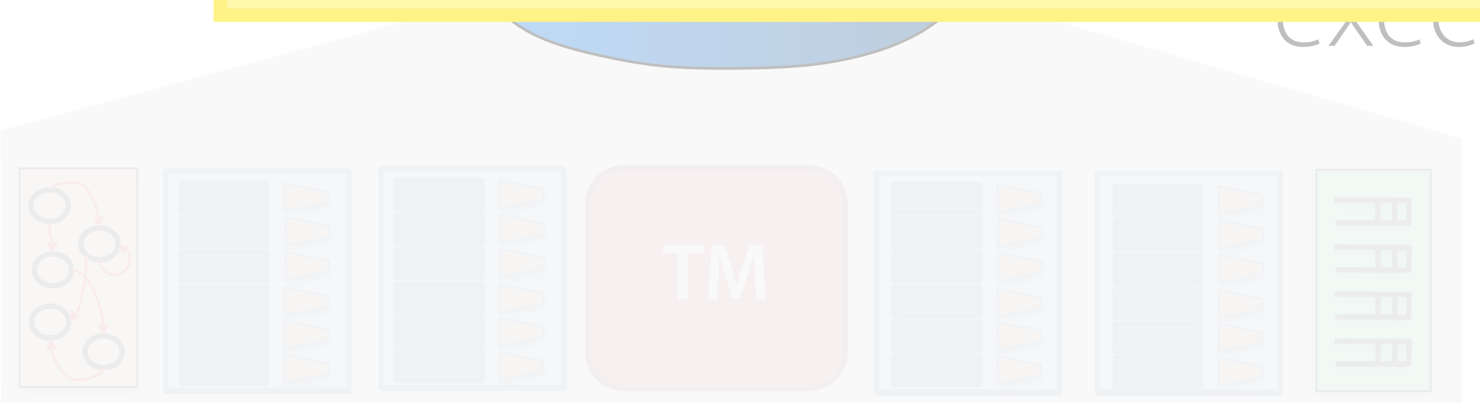
Control-plane: behavior viewed in terms of an invariant

Our solution:

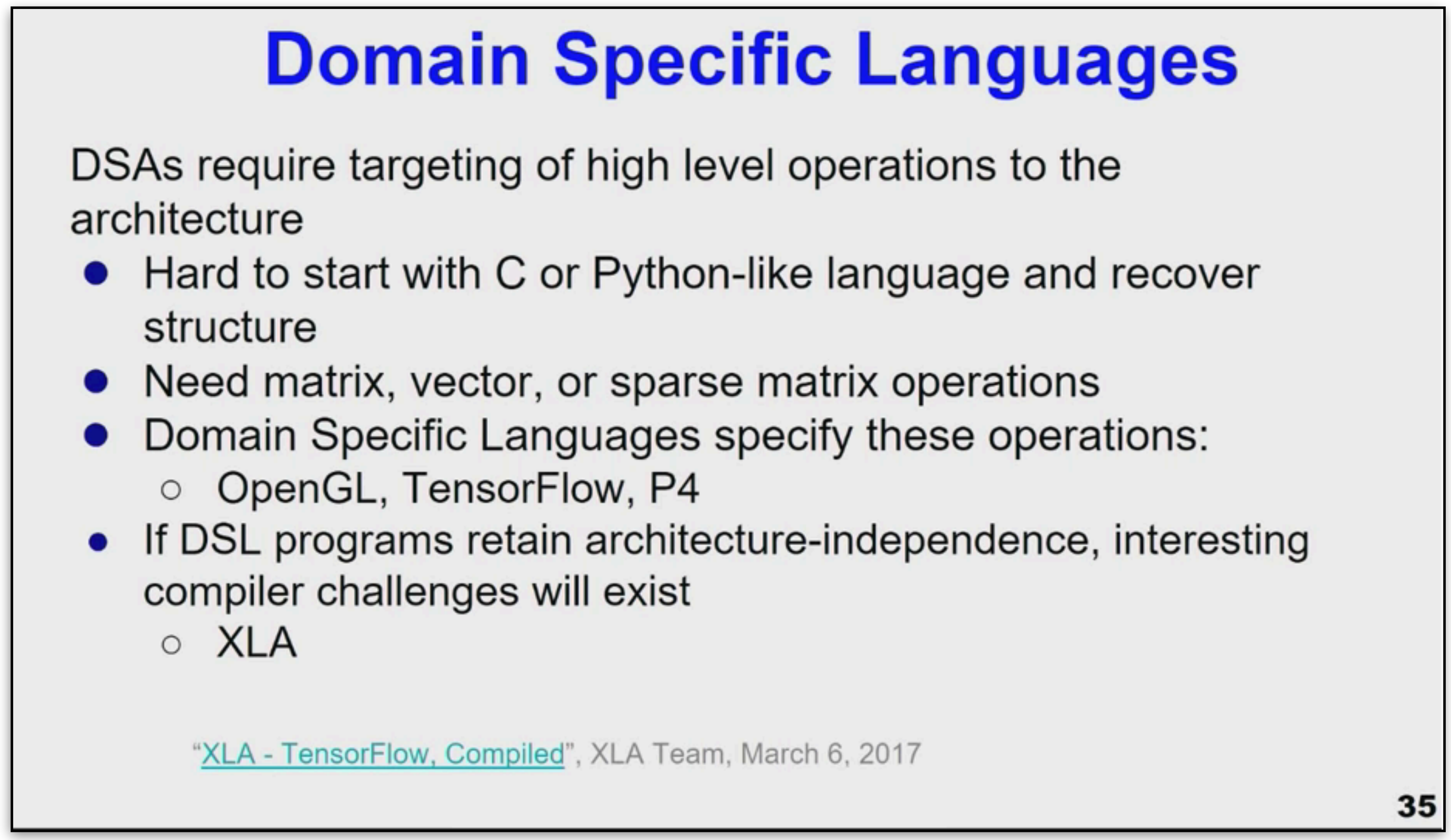
- Write the control-plane invariant in terms of data plane state (`reads`, `action`, etc.)
- Predicate *every* assertion in the data plane on the control-plane invariant
- This means that control-plane invariants must be location-independent!

of an

execution of the P4 program



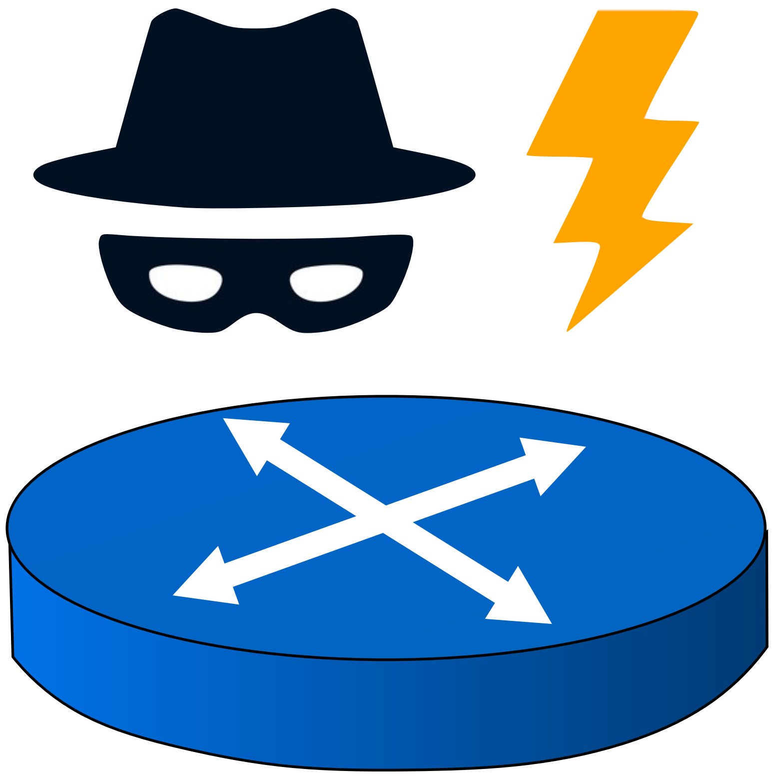
Aside: we may see more of this...



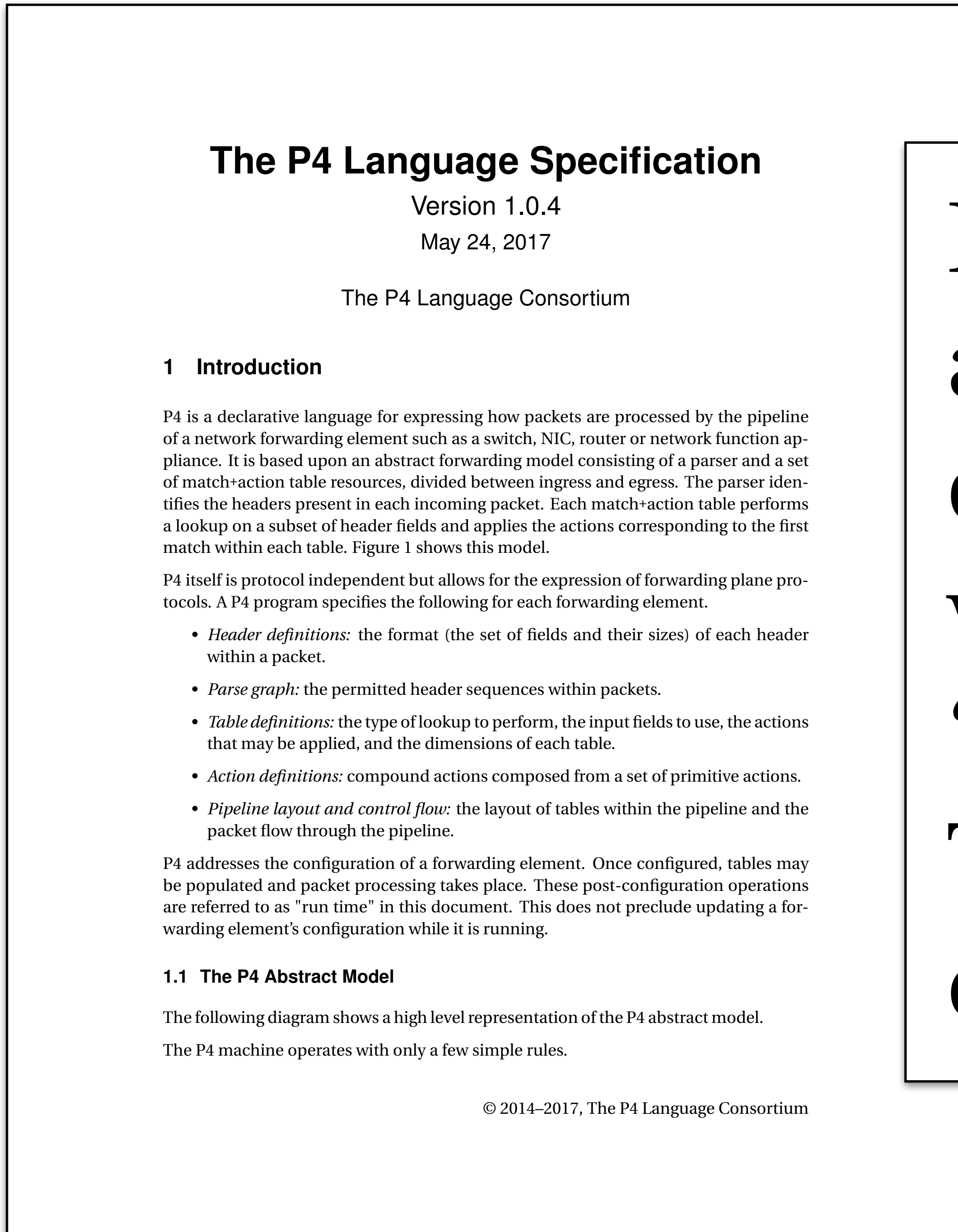
Experience

Data Plane Errors

- Reading/writing invalid headers
- Unhanded exceptions
- Incorrect use of packet metadata
- Malformed parsers/deparsers
- Unintended control flows



Header Validity



The P4 Language Specification

Version 1.0.4

May 24, 2017

The P4 Language Consortium

1 Introduction

P4 is a declarative language for expressing how packets are processed by the pipeline of a network forwarding element such as a switch, NIC, router or network function appliance. It is based upon an abstract forwarding model consisting of a parser and a set of match+action table resources, divided between ingress and egress. The parser identifies the headers present in each incoming packet. Each match+action table performs a lookup on a subset of header fields and applies the actions corresponding to the first match within each table. Figure 1 shows this model.

P4 itself is protocol independent but allows for the expression of forwarding plane protocols. A P4 program specifies the following for each forwarding element.

- *Header definitions:* the format (the set of fields and their sizes) of each header within a packet.
- *Parse graph:* the permitted header sequences within packets.
- *Table definitions:* the type of lookup to perform, the input fields to use, the actions that may be applied, and the dimensions of each table.
- *Action definitions:* compound actions composed from a set of primitive actions.
- *Pipeline layout and control flow:* the layout of tables within the pipeline and the packet flow through the pipeline.

P4 addresses the configuration of a forwarding element. Once configured, tables may be populated and packet processing takes place. These post-configuration operations are referred to as "run time" in this document. This does not preclude updating a forwarding element's configuration while it is running.

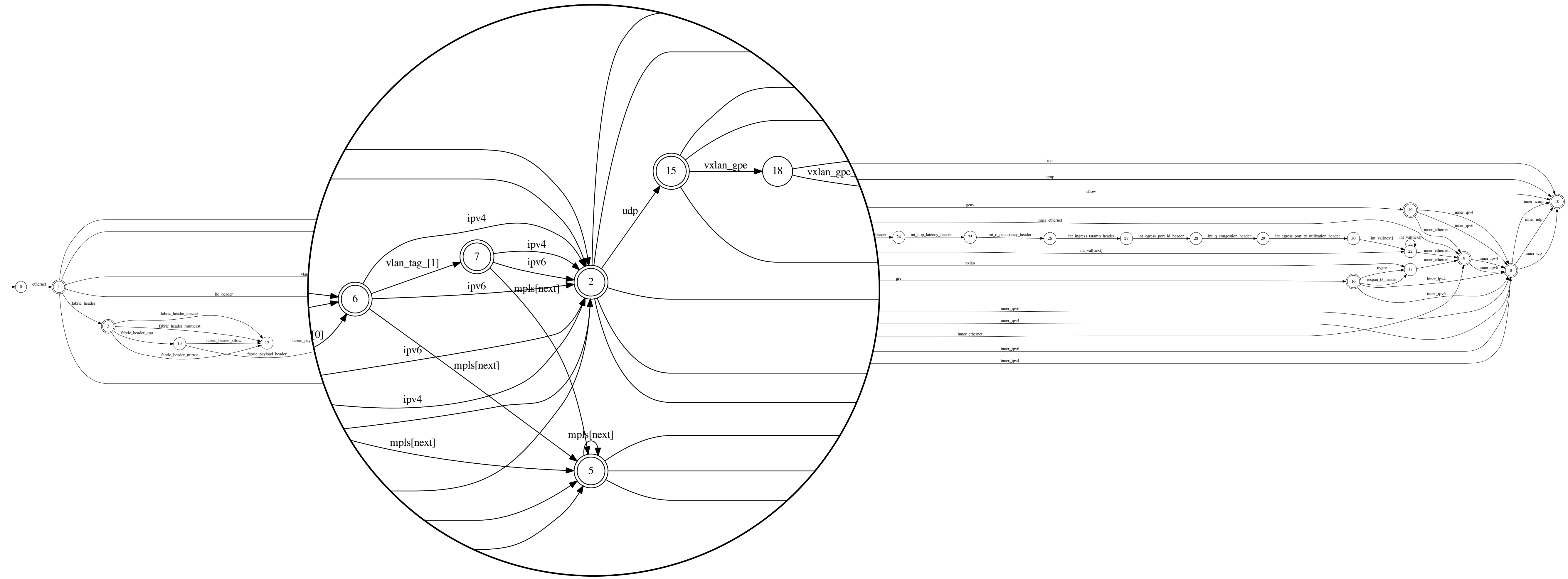
1.1 The P4 Abstract Model

The following diagram shows a high level representation of the P4 abstract model.

The P4 machine operates with only a few simple rules.

References at run time to a header instance (or one of its fields) which is not valid results in a special "undefined" value.
The implications of this depend on the context.

Example: switch.p4 Parser



The analog of Hoare's "$1B mistake" leads to significant complications in real-world programs like **`switch.p4`**

switch.p4 Validity

Statistics

- 7KLoC
- 58 parse states
- 120 match-action tables

Control-Plane Annotations

- 758 LoC
- ~2 days of programmer effort
- Default actions (30)
- Fabric wellformedness (14)
- Table actions (66)
- Guarded reads (10)
- Action data (14)

Bugs

- Parser bugs (2)
- Action flaws (4)
- Infeasible control-plane (3)
- Invalid read (1)

switch.p4 Validity

Statistics

- 7KLoC
- 58 parse states
- 120 match-action tables

Control-Plane Annotations

- 758 LoC
- ~2 days of programmer effort
- Default actions (30)
- Fabric wellformedness (14)
- Table actions (66)
- Guarded reads (10)
- Action data (14)

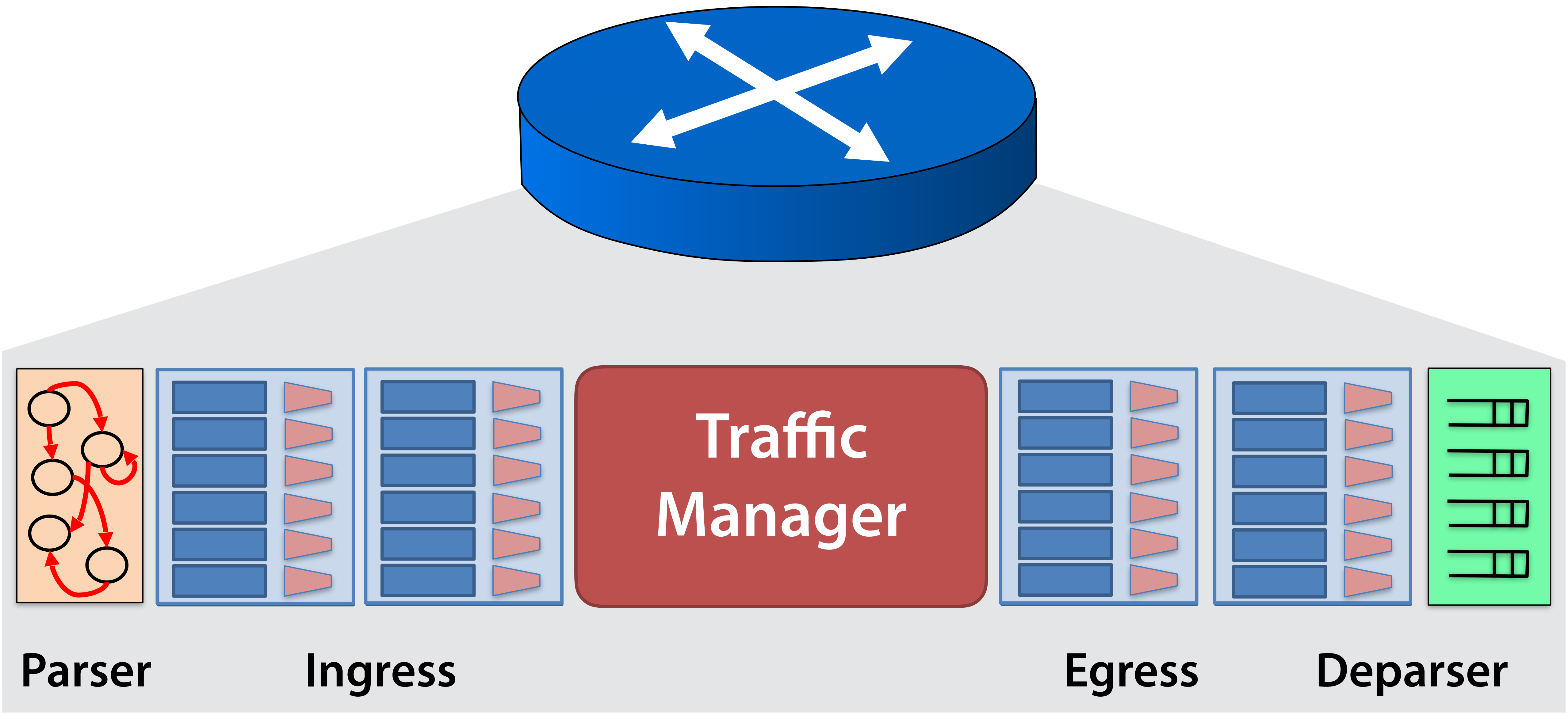
Bugs

- Parser bugs (2)
- Action flaws (4)
- Infeasible control-plane (3)
- Invalid read (1)

```
// For a fabric unicast/multicast packet whose ingressTunnelType is IP-in-IP,
// ipvX and inner_ipvX must be valid.
assume
  ((fabric_ingress_dst_lkp_valid_fabric_header_unicast == 1 and
    fabric_ingress_dst_lkp_fabric_header_unicast_ingressTunnelType == 3) or
  (fabric_ingress_dst_lkp_valid_fabric_header_multicast == 1 and
   fabric_ingress_dst_lkp_fabric_header_multicast_ingressTunnelType == 3))
implies
 ((fabric_ingress_dst_lkp_valid_ipv4 == 1 or
   fabric_ingress_dst_lkp_valid_ipv6 == 1) and
  (fabric_ingress_dst_lkp_valid_inner_ipv4 == 1 or
   fabric_ingress_dst_lkp_valid_inner_ipv6 == 1))
```

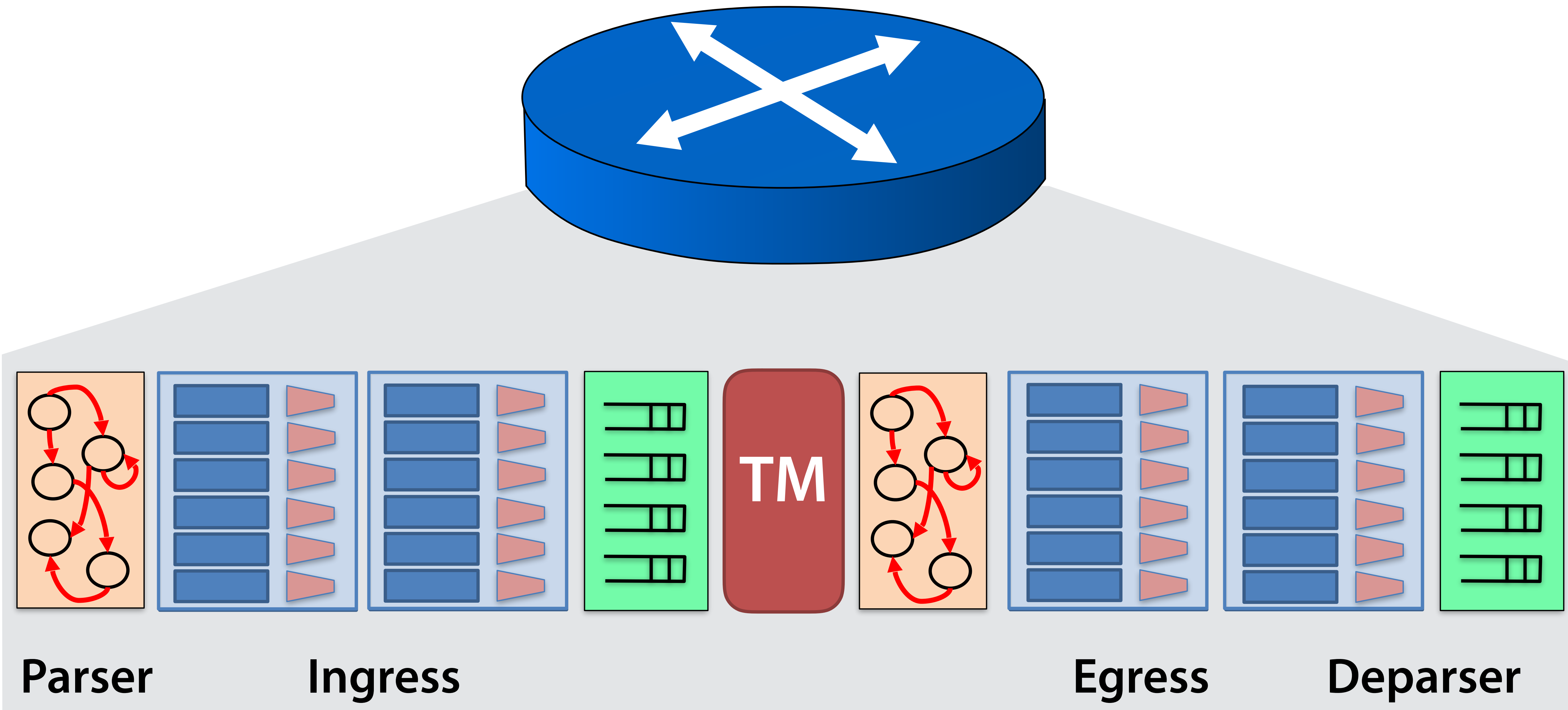
Parser and Deparser Compatbility

In PISA, state is copied verbatim from ingress to egress...



Parser and Deparser Compatbility

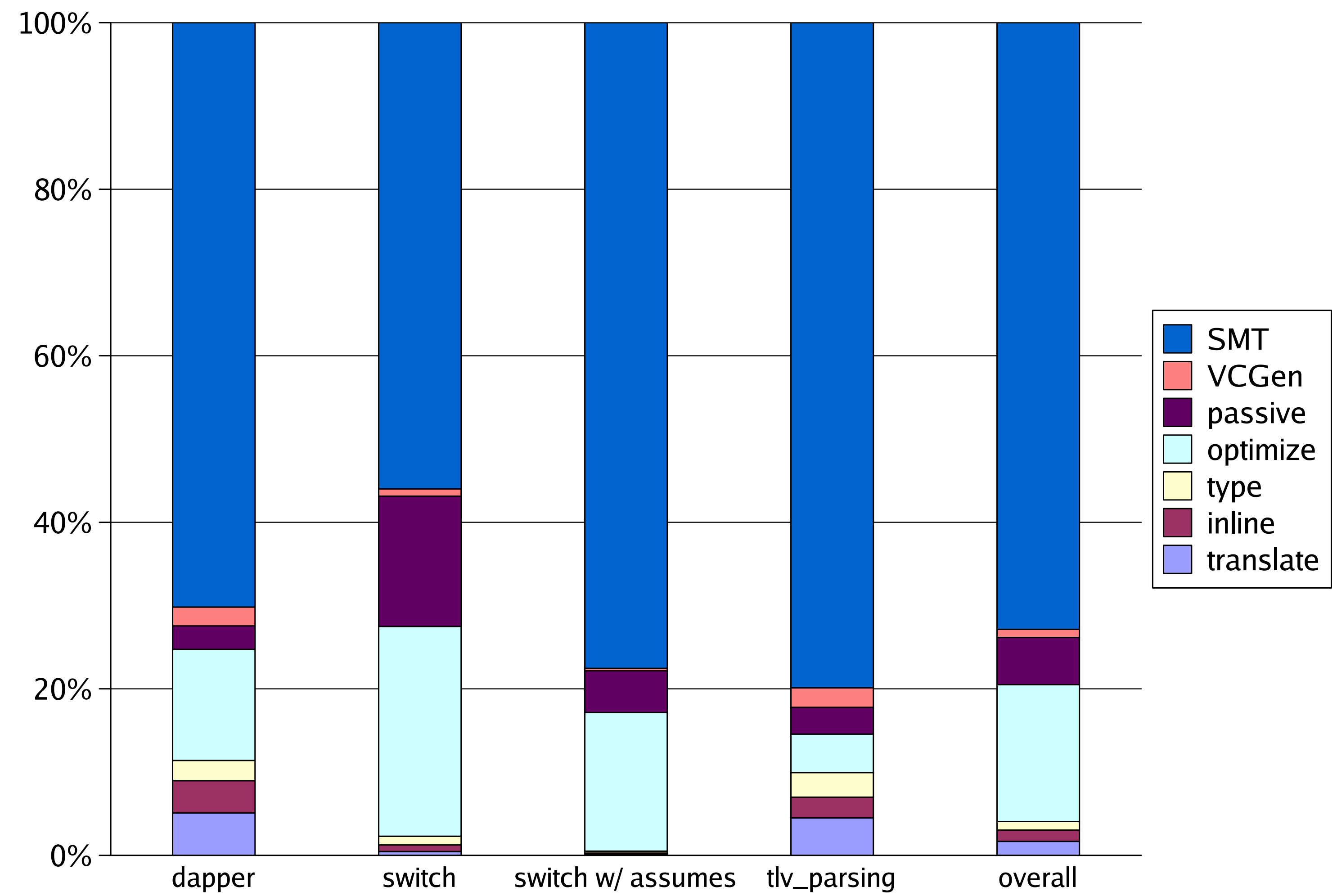
In PISA, state is copied verbatim from ingress to egress...



In reality, the parser and deparser are used to (de)serialize state...

Experiments

Program	LOC	Time (ms)
axon	100	313
dapper	618	34691
easyroute	53	32
flowlet_swit	251	32
linear_road	883	76
nat	294	50
ndn	525	685
paxos	205	41
simple_rout	64	33
switch	7304	15,579
tor	472	76
vpc	278	42



Conclusions

- The intersection between networking and formal methods has gotten *very* interesting in recent years
- The P4 language offers a unique opportunity to shape how networks are built for decades to come
- Many challenging problems remain:
  - Domain-specific annotation language
  - Synthesis of control-plane annotations
  - Verifying control-plane annotations
  - Usability of tools by non-experts
  - Extending to networks of P4 routers