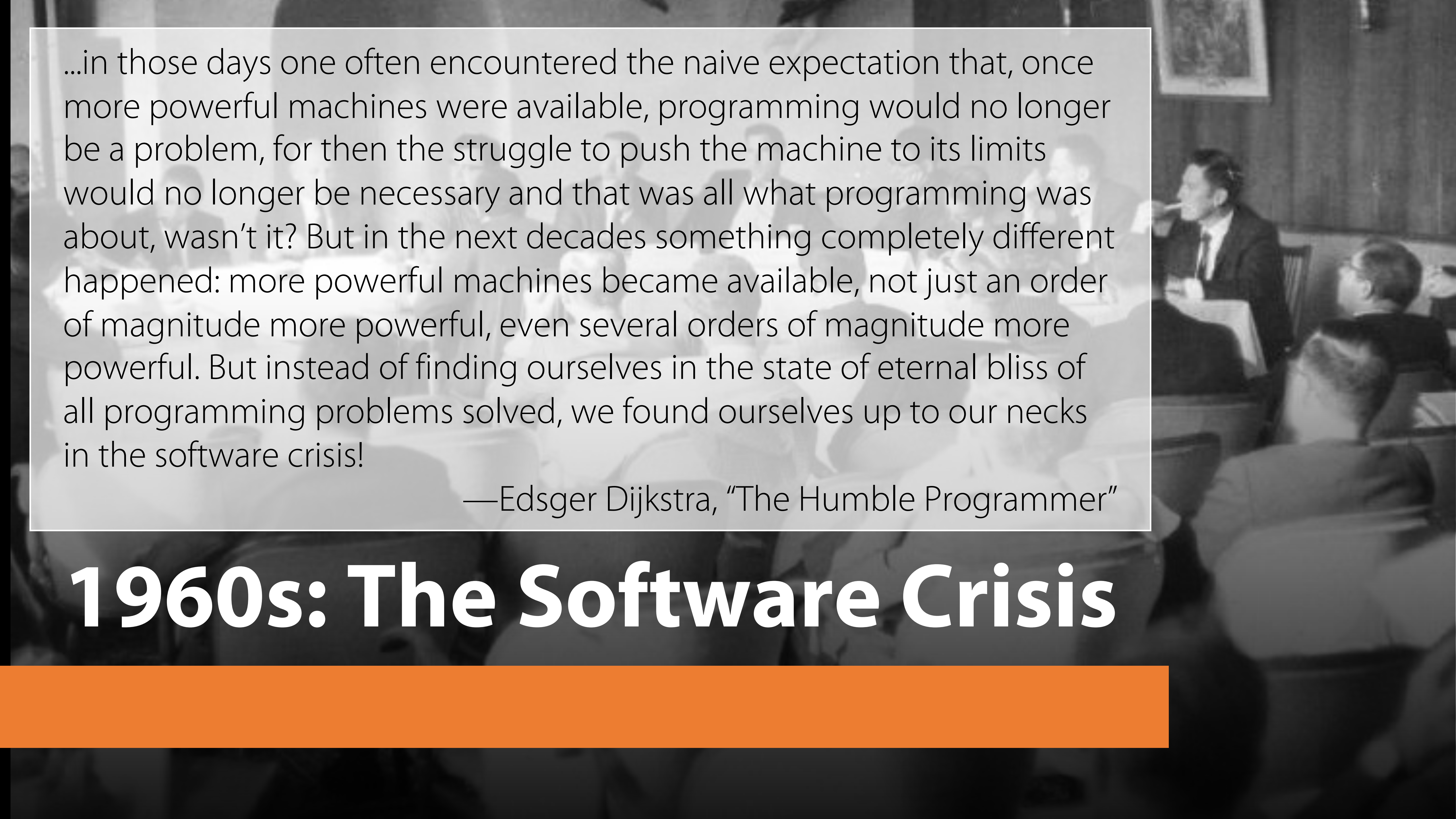
Deep Programmability: A New Lens on Networking

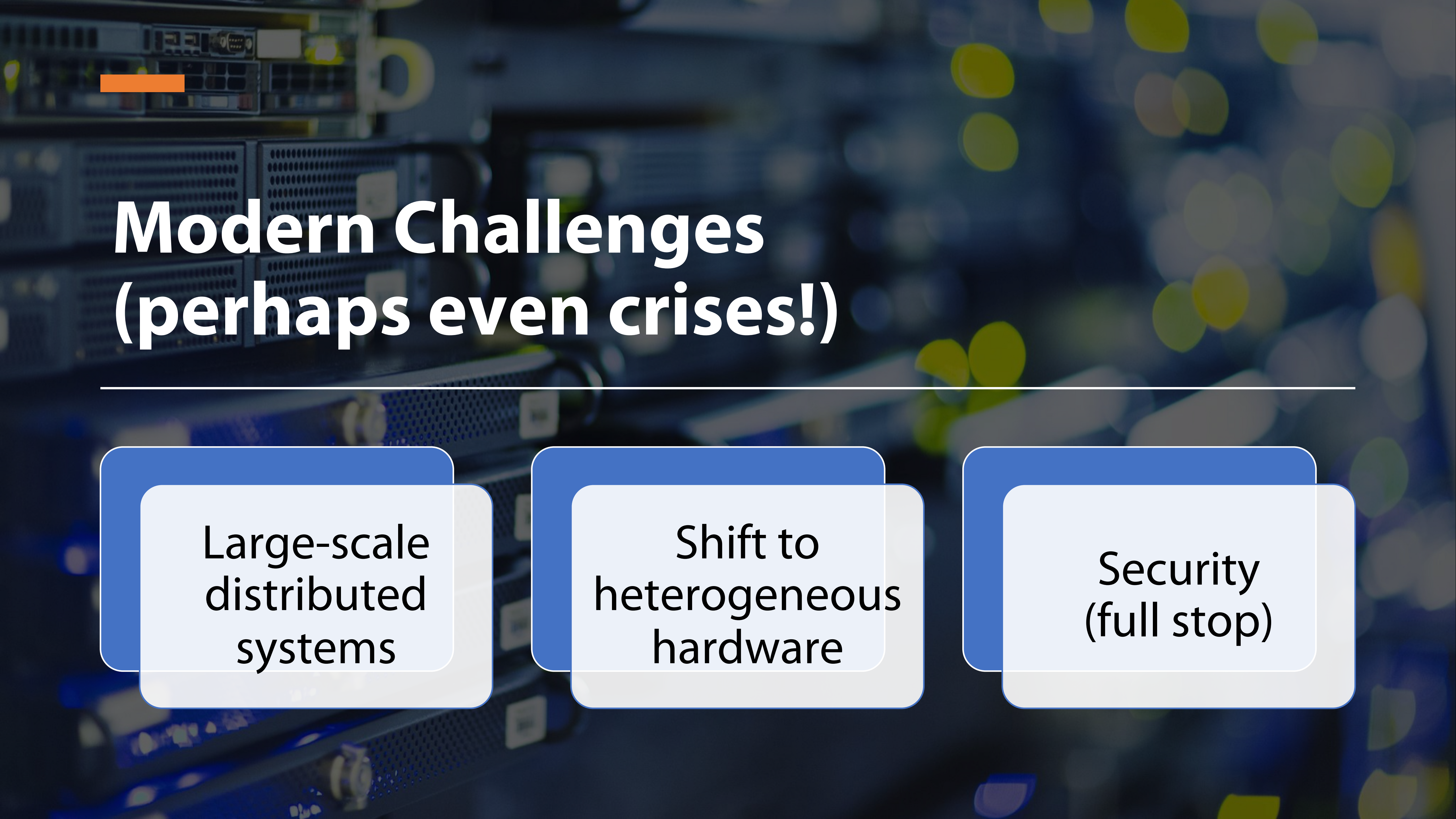
Nate Foster

Cornell & Intel



> ...in those days one often encountered the naive expectation that, once more powerful machines were available, programming would no longer be a problem, for then the struggle to push the machine to its limits would no longer be necessary and that was all what programming was about, wasn't it? But in the next decades something completely different happened: more powerful machines became available, not just an order of magnitude more powerful, even several orders of magnitude more powerful. But instead of finding ourselves in the state of eternal bliss of all programming problems solved, we found ourselves up to our necks in the software crisis!
>
> —Edsger Dijkstra, "The Humble Programmer"

1960s: The Software Crisis



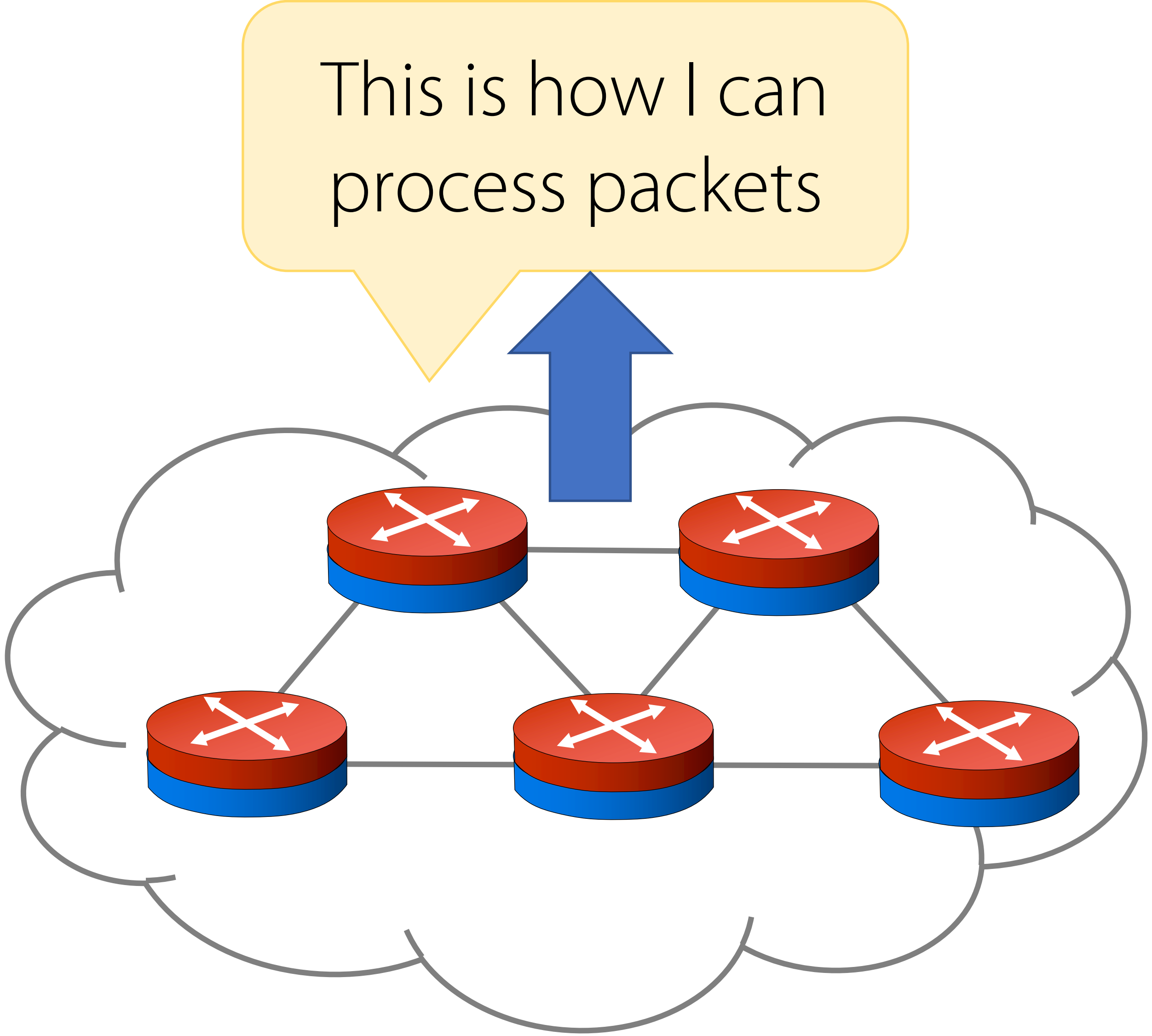
Modern Challenges (perhaps even crises!)

- Large-scale distributed systems
- Shift to heterogeneous hardware
- Security (full stop)

Networks play a central role in modern systems...

But if we can program them at all, we use the analogues of machine code!

Status Quo: Bottom-Up Design



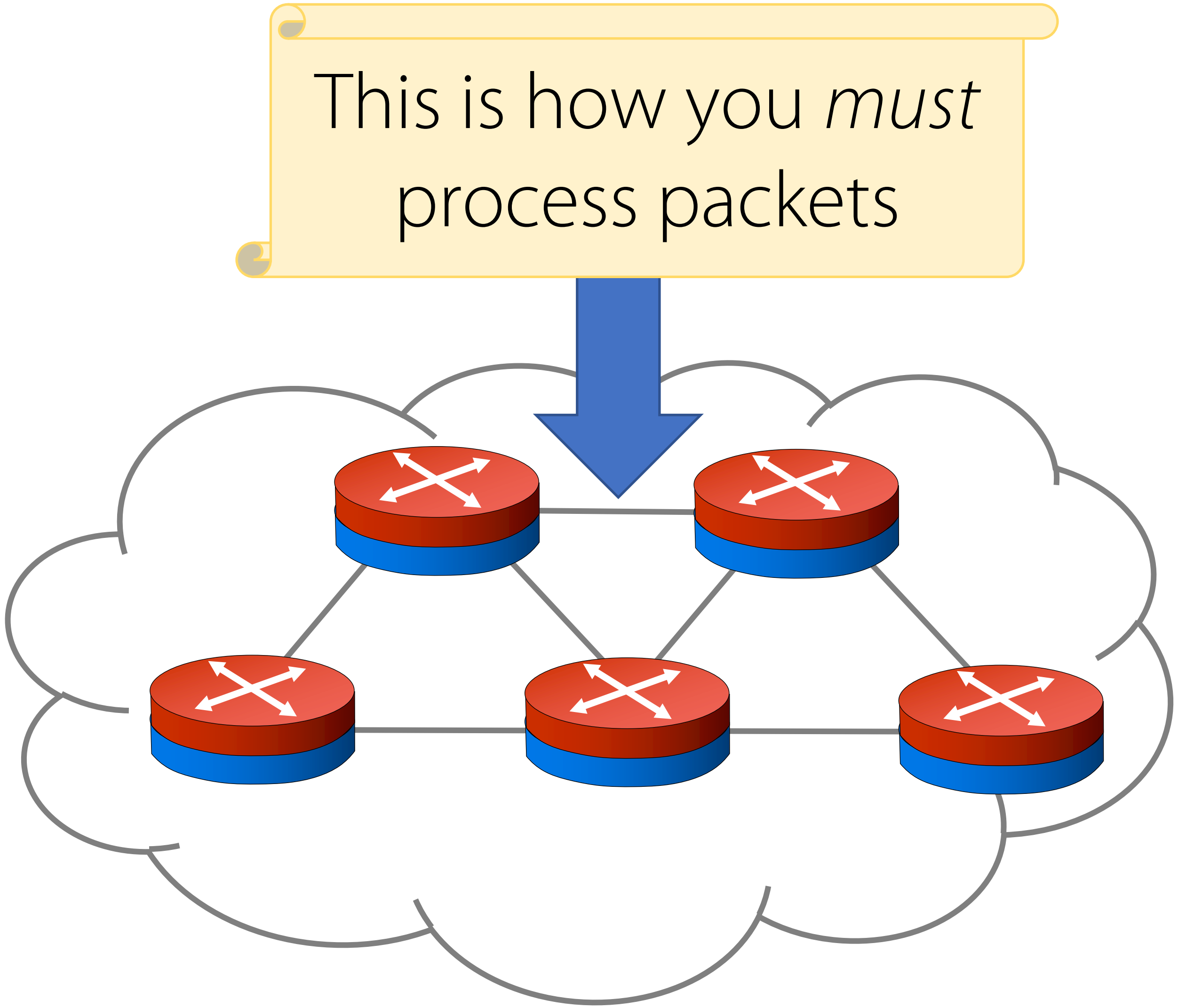
Network capabilities defined by:

- Standards bodies
- Distributed protocols
- Equipment vendors

Hard for *system owners* to build networks with the structure and properties they want

Custom behaviors must be encoded using low-level notions: IP addresses, VLANs, link weights, etc.

Emerging: Top-Down Design



This is how you *must* process packets

Network capabilities defined by system owners as programs!

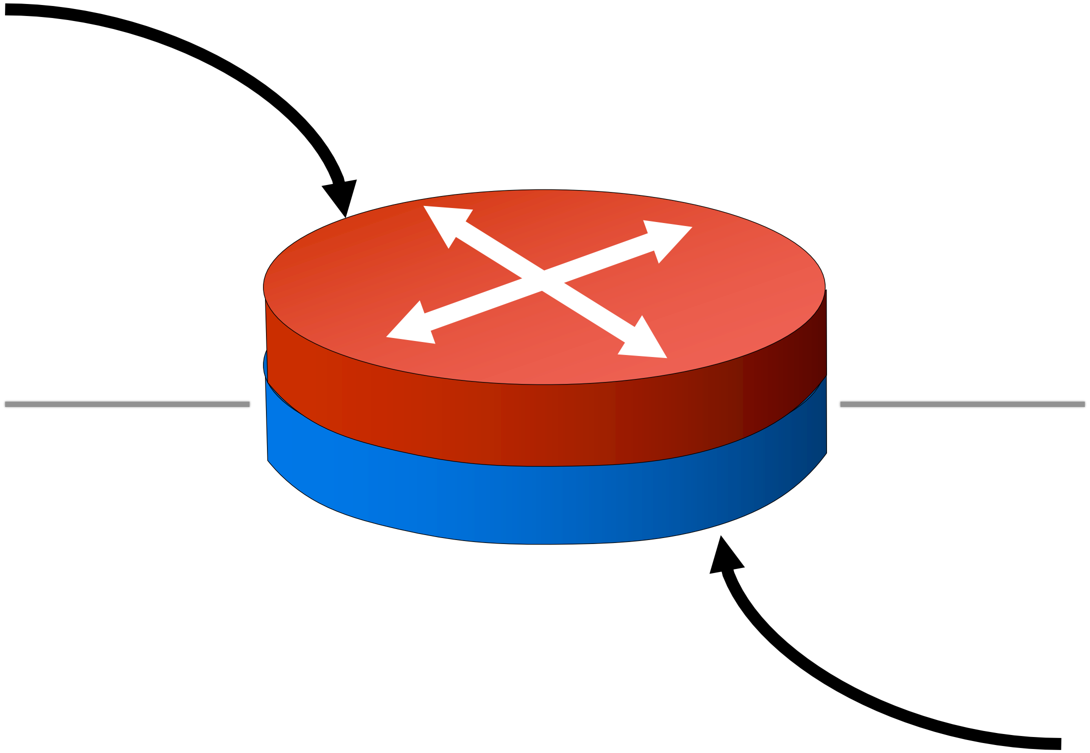
Key ingredients:

- Programmable hardware
- Domain-specific languages
- Compilers, verification tools, etc.

Deep Programmability

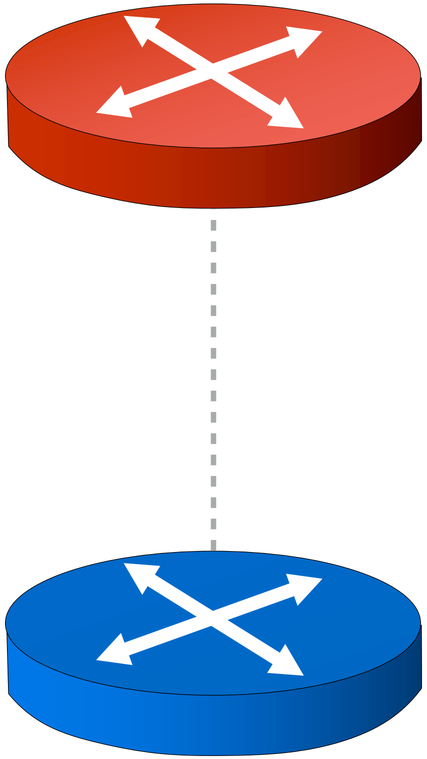
Conventional Network

Control Plane
discovers topology,
computes routes,
manages policy, etc.



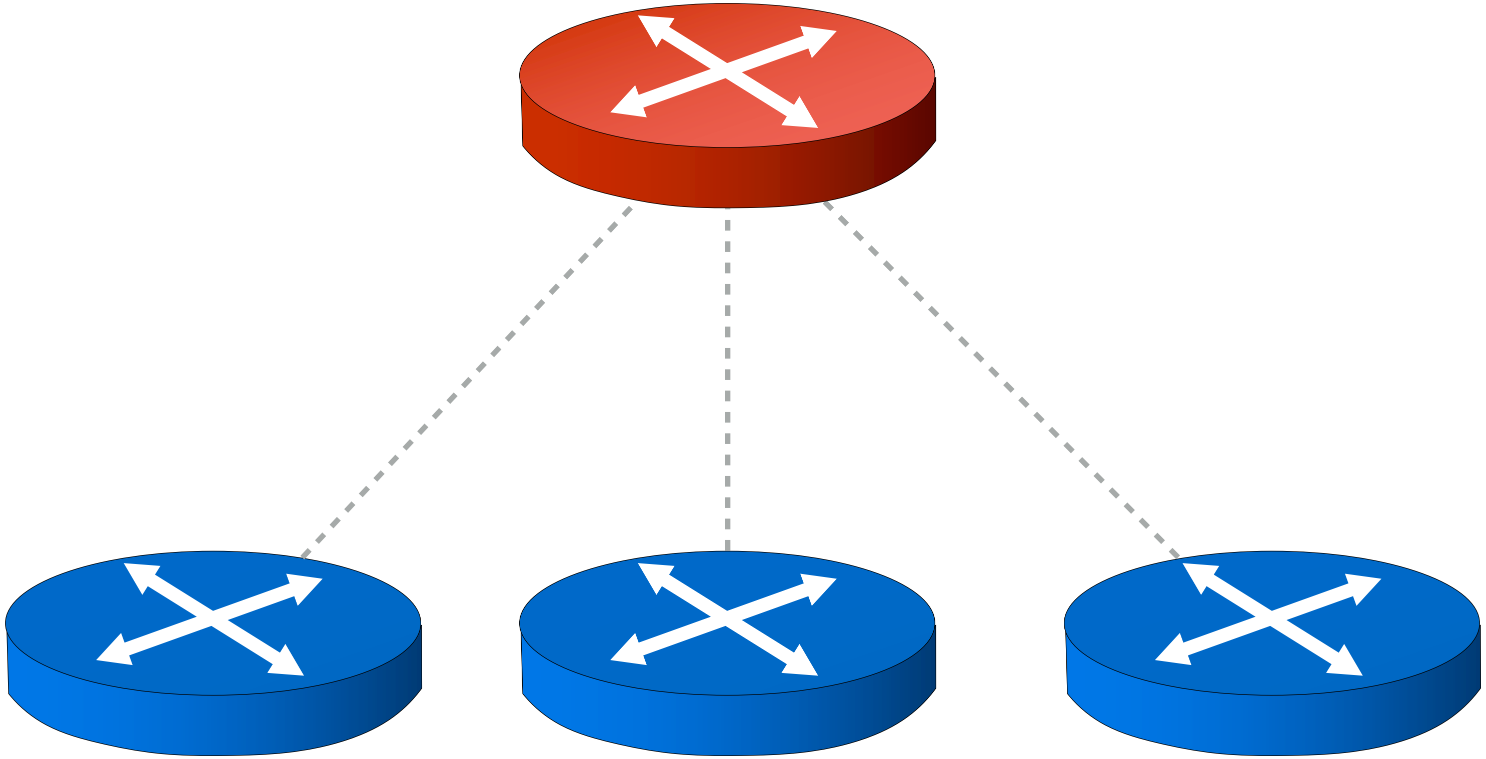
Data plane
forwards packets,
enforces access control,
monitors flows, etc.

Software-Defined Network



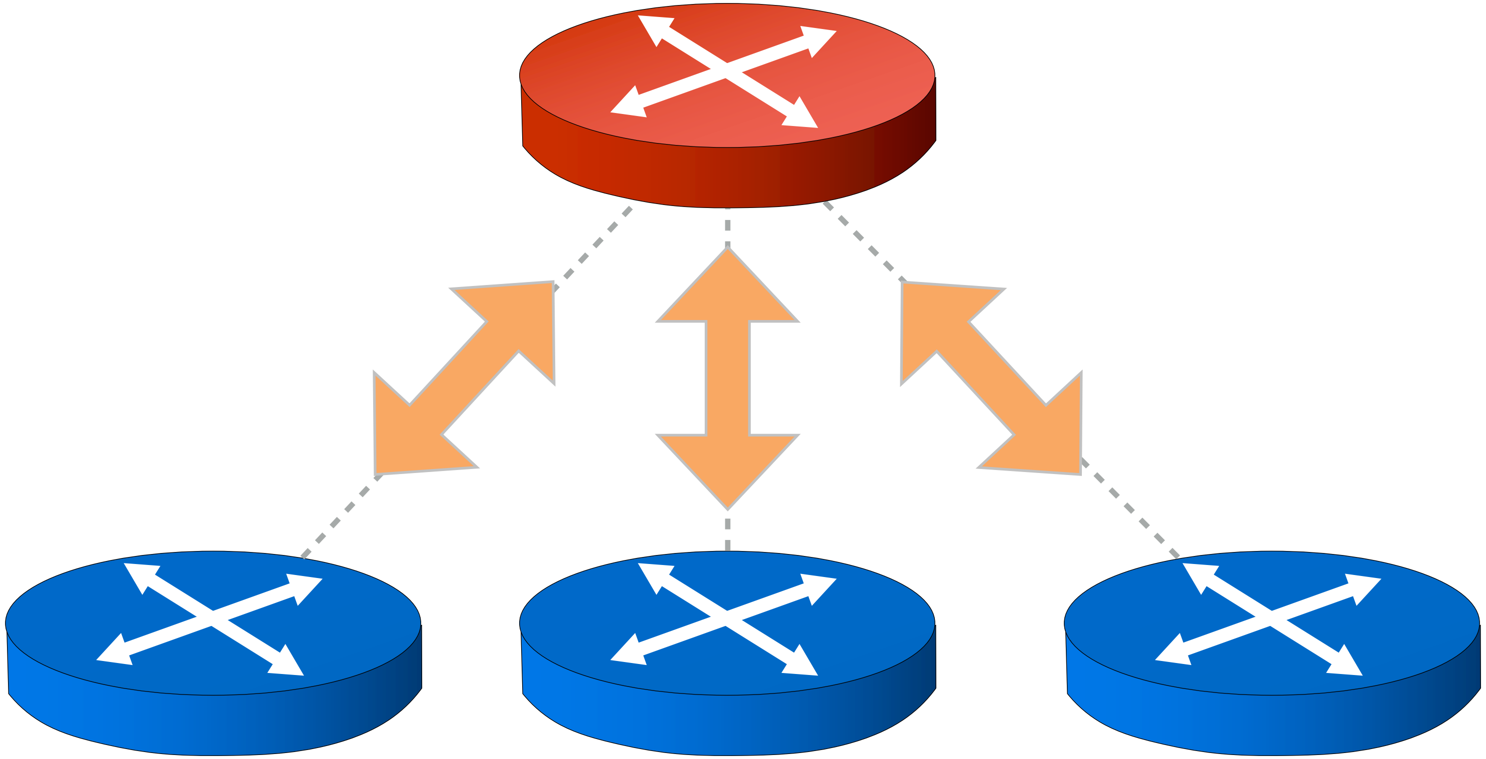
1. Separate control plane and data plane

Software-Defined Network



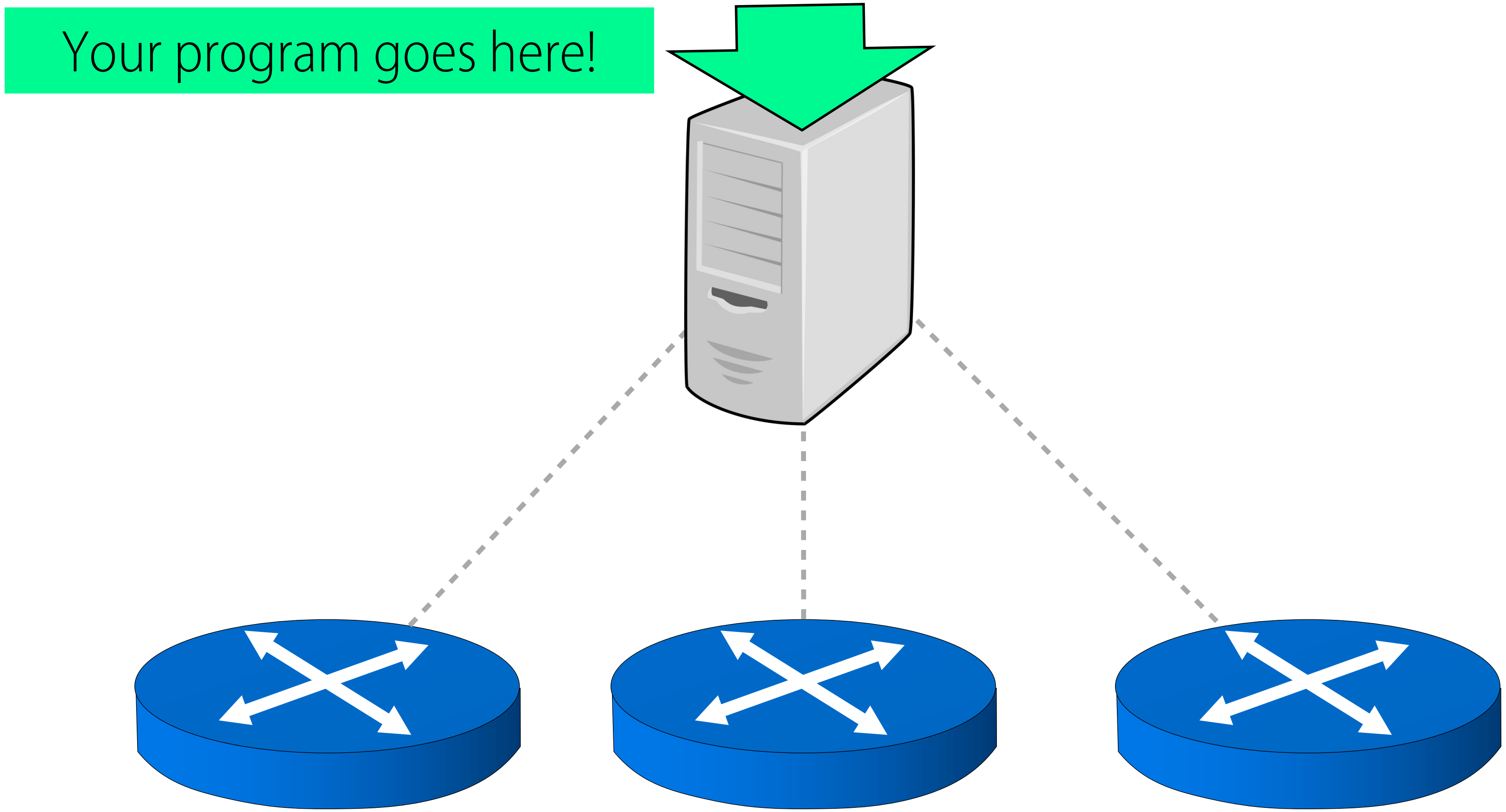
2. Pick the right "unit of abstraction" for control plane

Software-Defined Network



3. Standardize run-time configuration APIs

Software-Defined Network

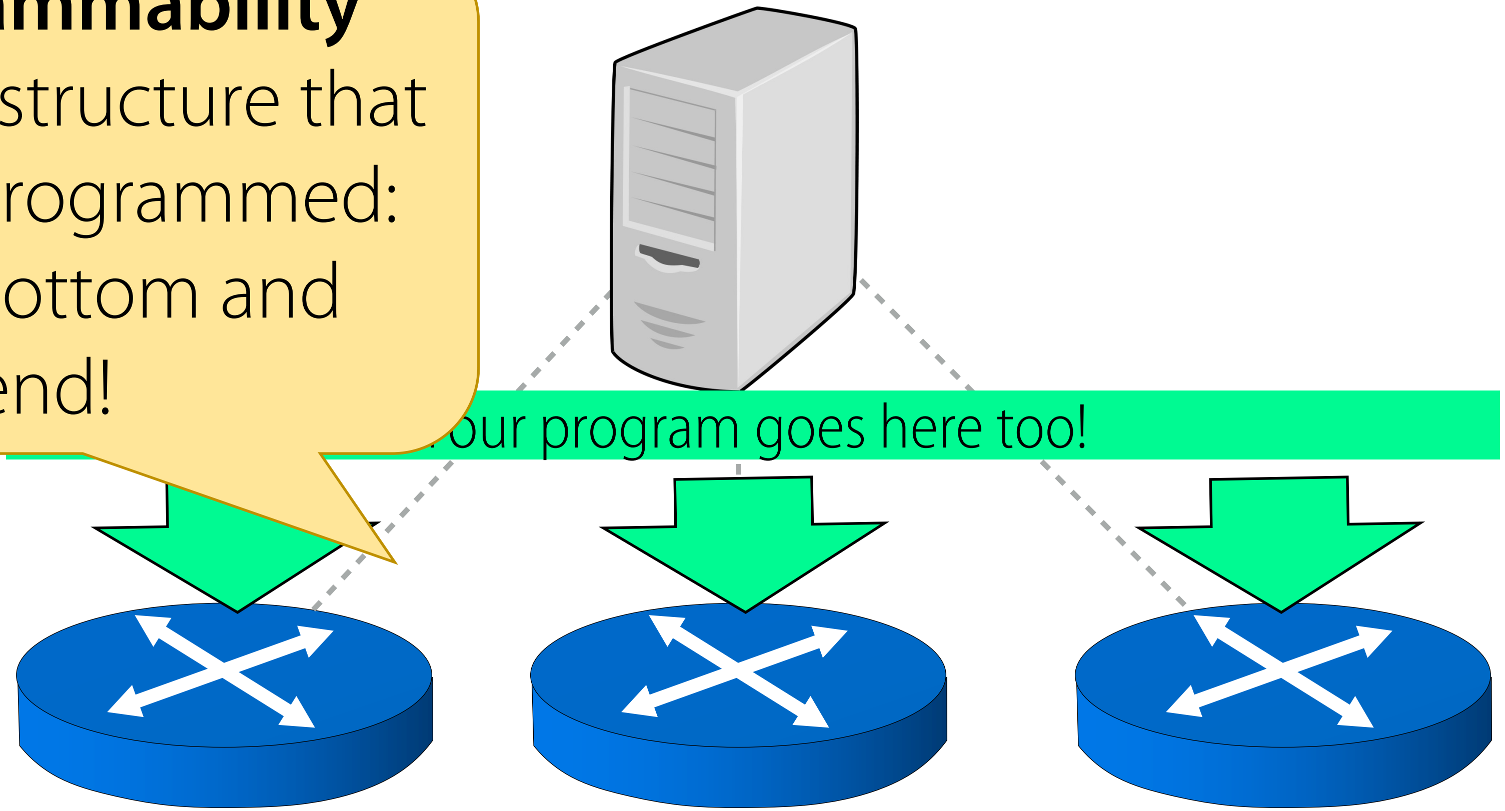


Your program goes here!

4. Replace control plane with general-purpose machine

Software-Defined Network

Deep programmability
network infrastructure that can be fully programmed: from top to bottom and from end to end!



5. Replace the data plane with programmable hardware

Killer Applications (so far...)

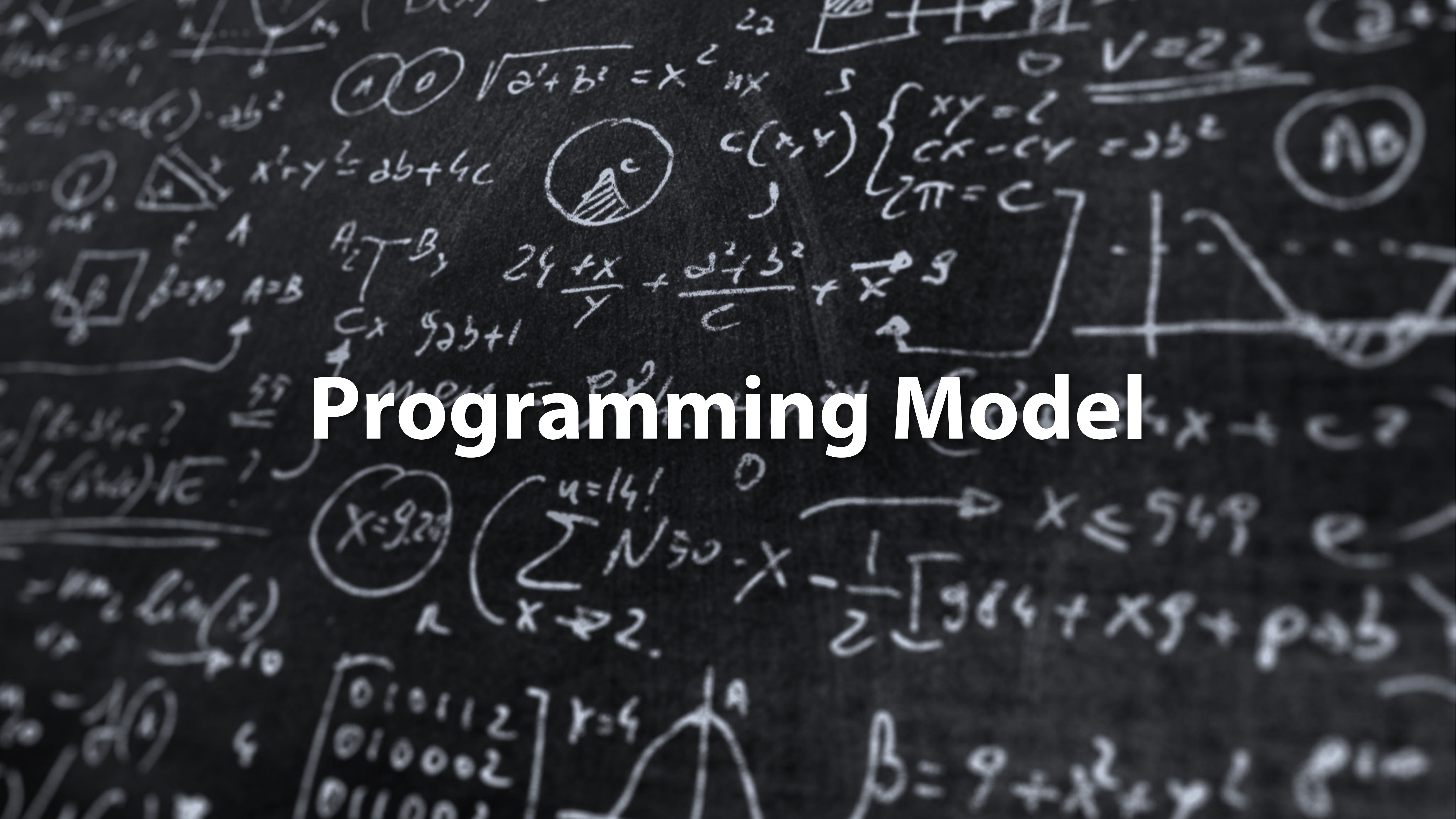
Network Virtualization
Virtualize a private network, enabling running in cloud environments

Traffic Engineering
Optimize network paths, reducing cost, latency, congestion, etc.

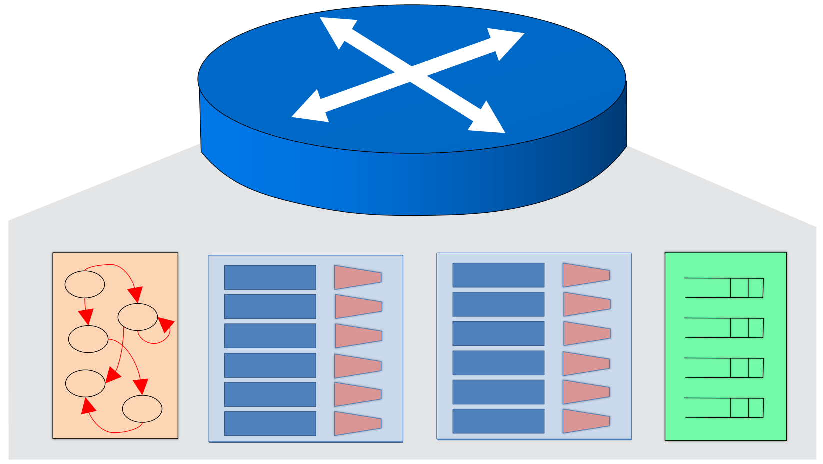
Network Monitoring
Implement per-packet monitoring that tracks paths, delay, causality, etc.

In-Network Computing
Offload services like caching, coordination, failure detection, etc.

Programming Model



Dataplane Model



```
00000001 00000010 00000000 00000011
00000000 00000011 00000001 10101010
10111011 01010000 01100101 01110100
01110010 00110100 00100000 01101001
01110011 00100000 01100001 01110111
01100101 01110011 01101111 01101101
01100101 00100001
```

Match	Action
ip.dst = h1	forward 1
ip.dst = h2	forward 2
*	drop

```
00000001 00000011 00000000 00000011
00000001 10101010 10111011 01010000
01100101 01110100 01110010 00110100
00100000 01101001 01110011 00100000
01100001 01110111 01100101 01110011
01101111 01101101 01100101 00100001
```

1. Parse
Extract structured packet representation

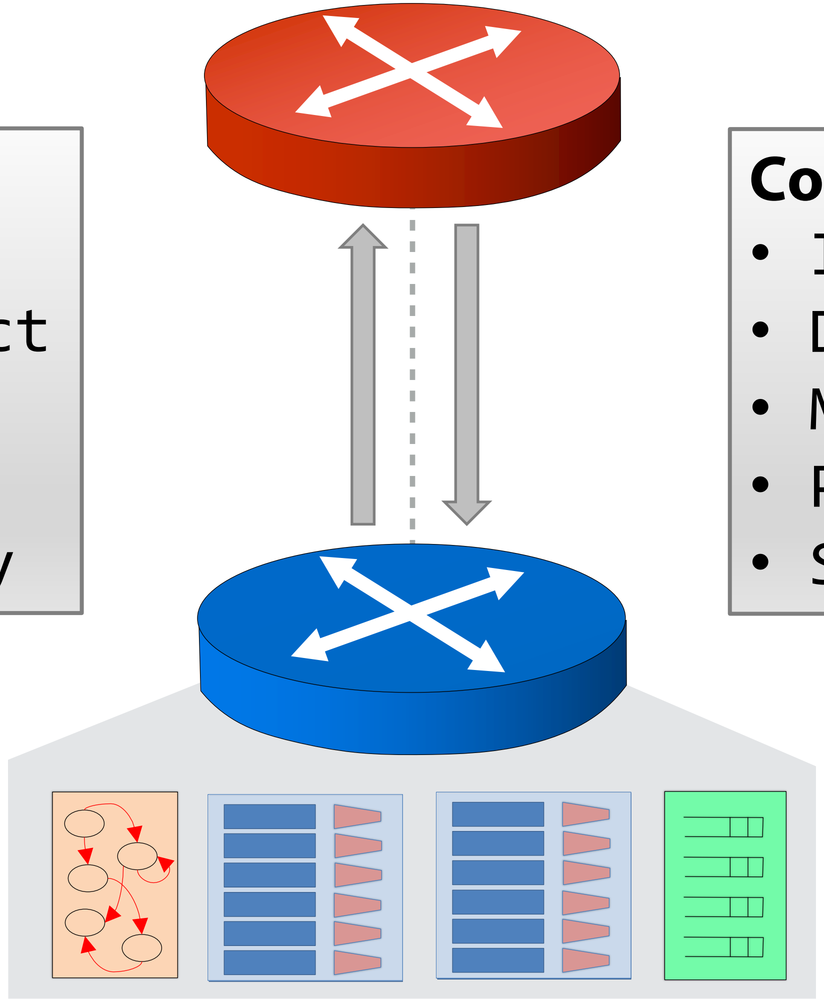
2. Process
Looking headers in routing tables, make forwarding decision

3. Deparse
Transform packet back into bits and forward along to next hop(s)

Control-Plane API

Switch to controller

- `SwitchConnect`
- `SwitchDisconnect`
- `PortStatus`
- `PacketIn`
- `StatisticsReply`



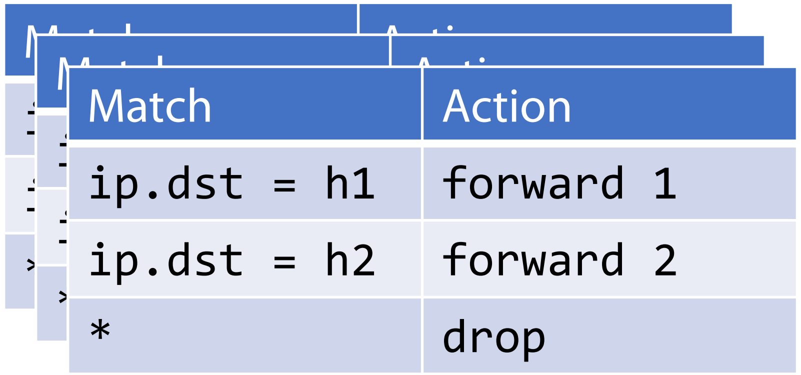
Controller to switch

- `InstallRule`
- `DeleteRule`
- `ModifyRule`
- `PacketOut`
- `StatisticsRequest`

From Pipelines to Functions

Built-In

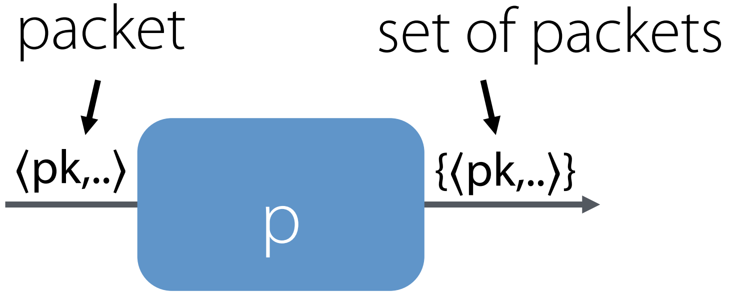
SDN's built-in programming model describes behavior in terms of device-level constructs like pipelines of match-action tables on single switches



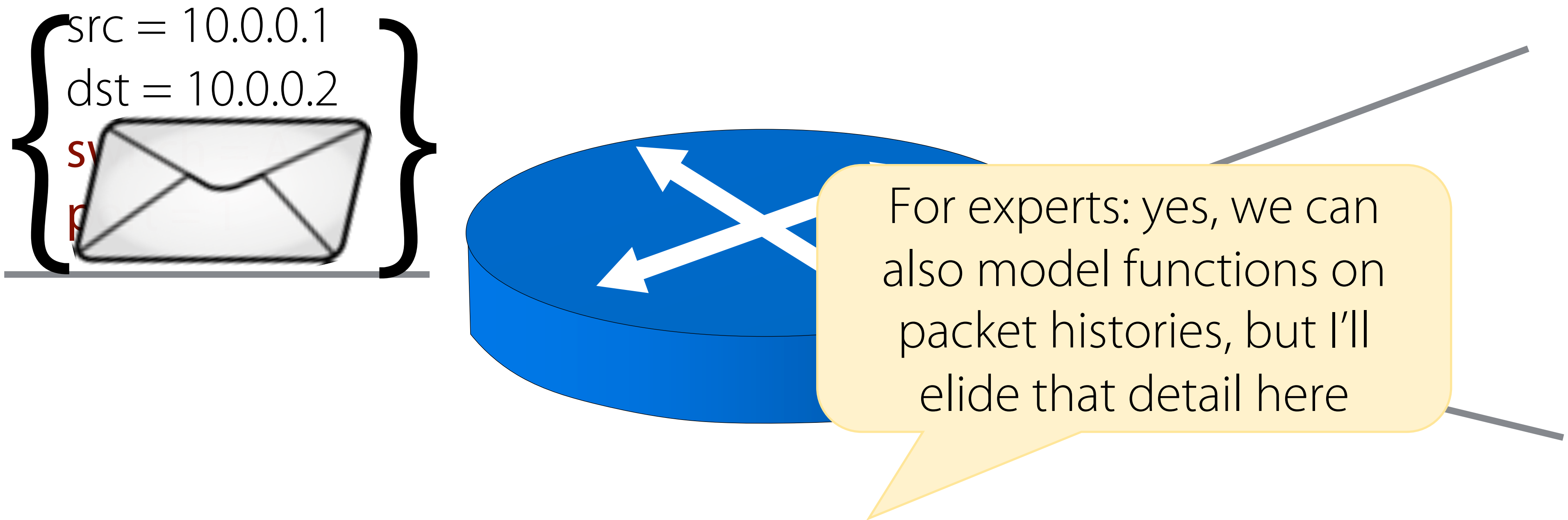
Match	Action
`ip.dst = h1`	`forward 1`
`ip.dst = h2`	`forward 2`
`*`	`drop`

Functional

A better approach is to use a domain-specific model that describes behavior using simple, composable programming abstractions



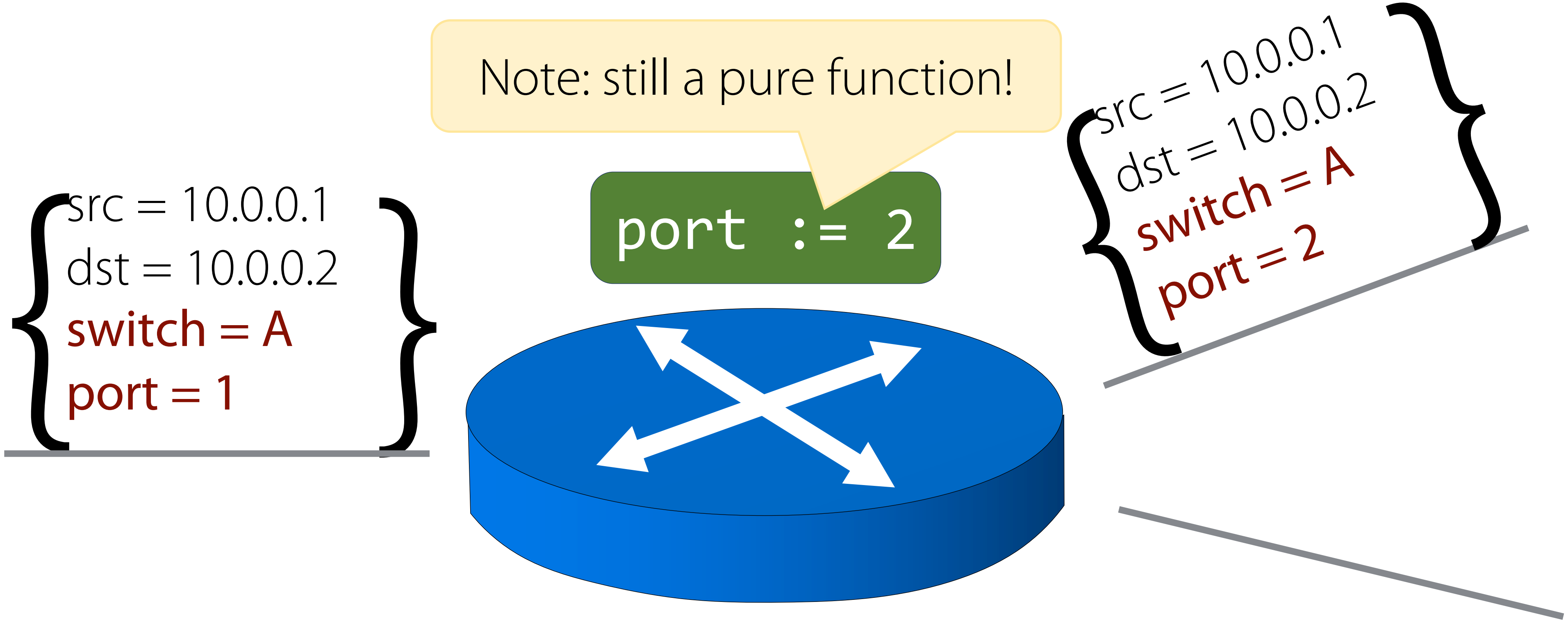
DSL Design



For experts: yes, we can also model functions on packet histories, but I'll elide that detail here

Packets → Packet Set

DSL Design



Note: still a pure function!

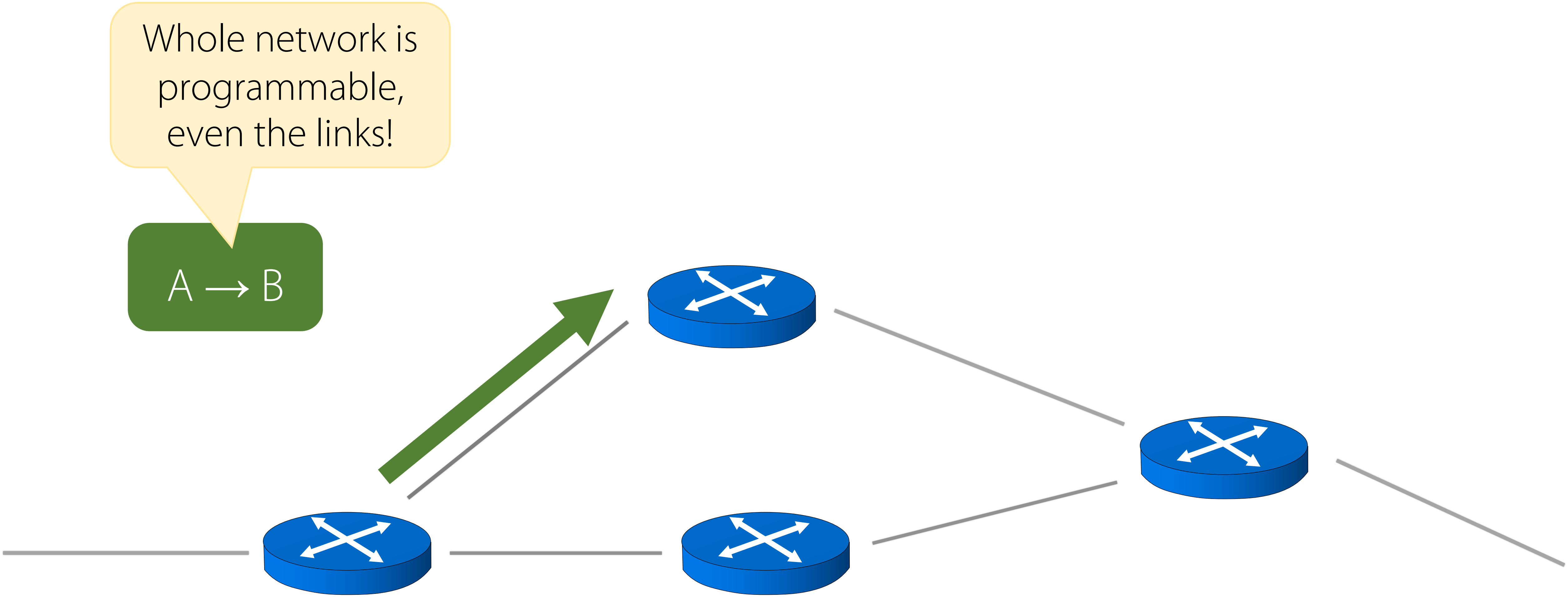
`port := 2`

{ src = 10.0.0.1
dst = 10.0.0.2
switch = A
port = 1 }

{ src = 10.0.0.1
dst = 10.0.0.2
switch = A
port = 2 }

Packets → Packet Set

DSL Design



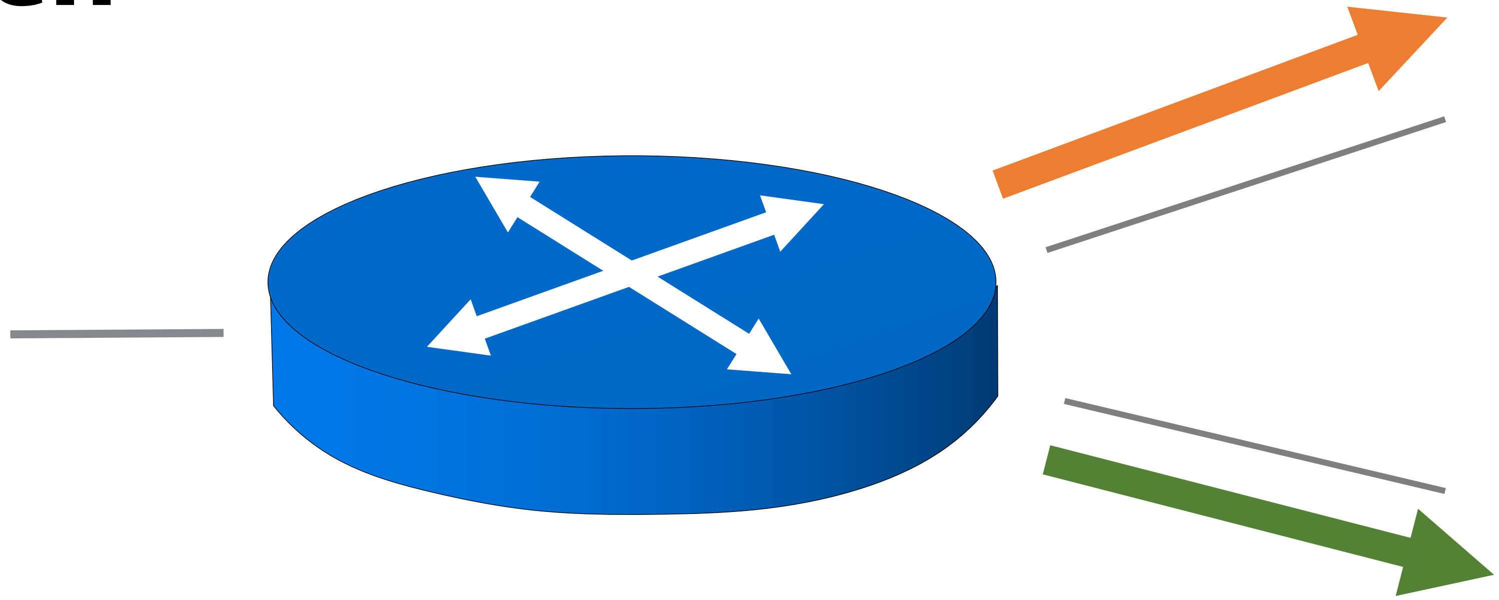
Whole network is programmable, even the links!

A → B

DSL Design

Conditionals: classify traffic and apply different policies

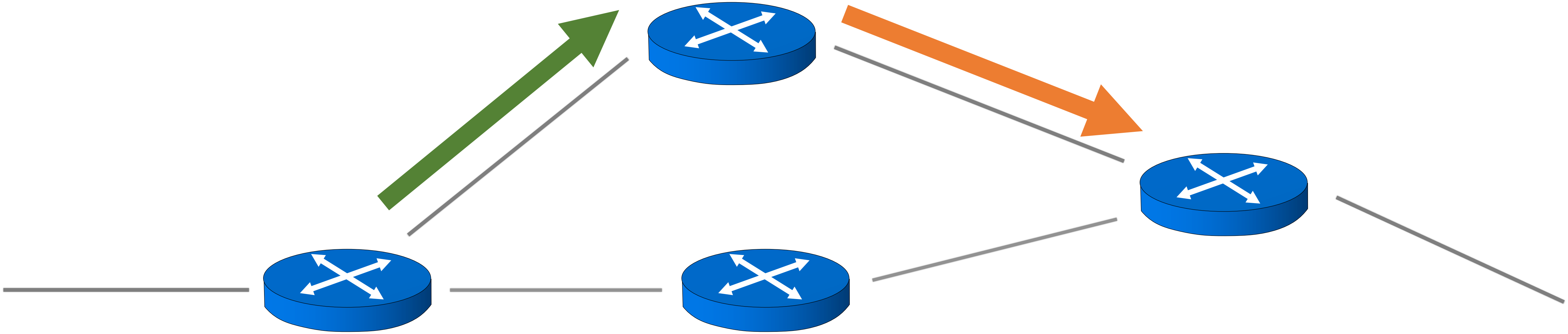
```
if f = n then
  p
else
  q
```



DSL Design

Composition: combine functionality specified by different program pieces

f ; g

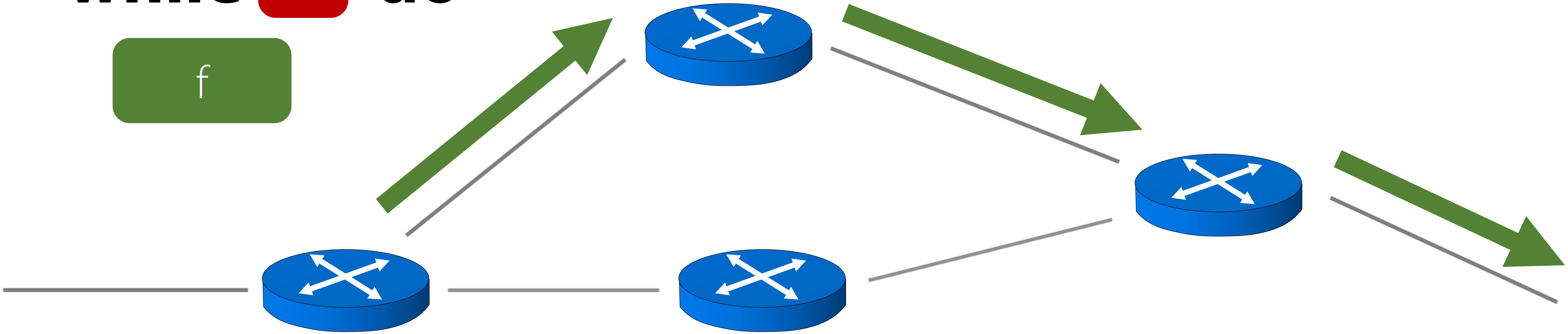


DSL Design

Loops: specify network-wide processing in terms of iterated steps

while a **do**

f



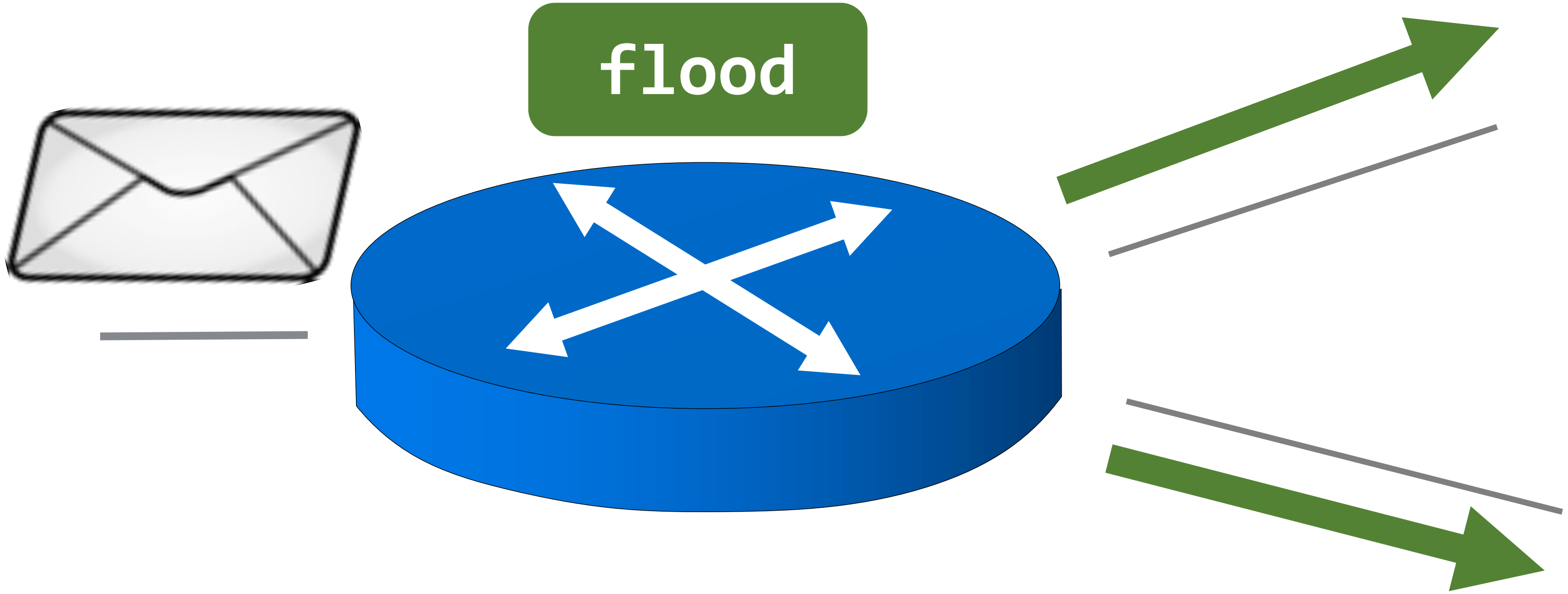
Data Plane DSL: Take I

```
a,b,c ::=
    | true
    | false
    | f = n
    | not a
    | a and b
    | a or b
```

```
p,q,r ::=
    | id
    | drop
    | f := n
    | p ; q
    | if a then p else q
    | while b do p
    | A → B
```

⚠ **Problem:** impossible to write a program that produces multiple packets!

Add a broadcast primitive?



⚠ **Puzzle:** how many packets should `flood;flood` produce?

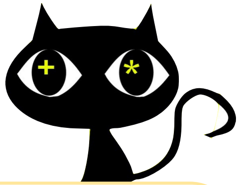
Data Plane DSL: Take II

```
p,q,r ::=
   | true
   | false
   | f = n
   | !p
   | f := n
   | p + q
   | p ; q
   | p*
   | A → B
```

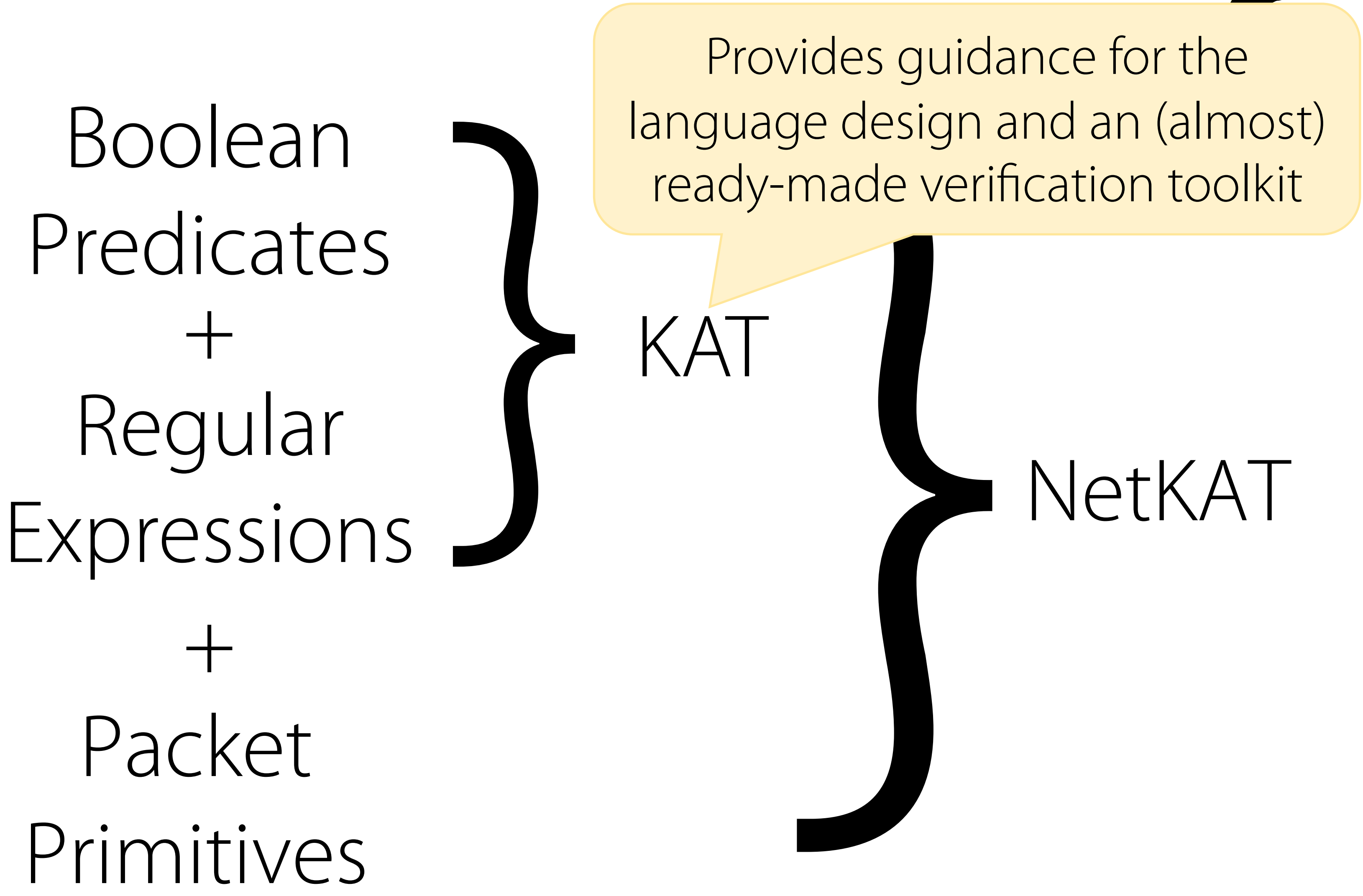
Key changes:

- Added union (`+`) operator, which duplicates packets
- Added iteration (`*`) operator
- Combined tests (`a,b`) and programs (`p,q`) into a single syntactic category (`and` is `;`, `or` is `+`)
- Loops, conditionals, and trivial programs (`id, drop`) can be derived
- `flood` can also be encoded using `+`

DSL is a KAT!



```
p,q,r ::=
    | true
    | false
    | f = n
    | !p
    | f := n
    | p + q
    | p ; q
    | p*
    | A → B
```



Boolean Predicates

+

Regular Expressions

KAT

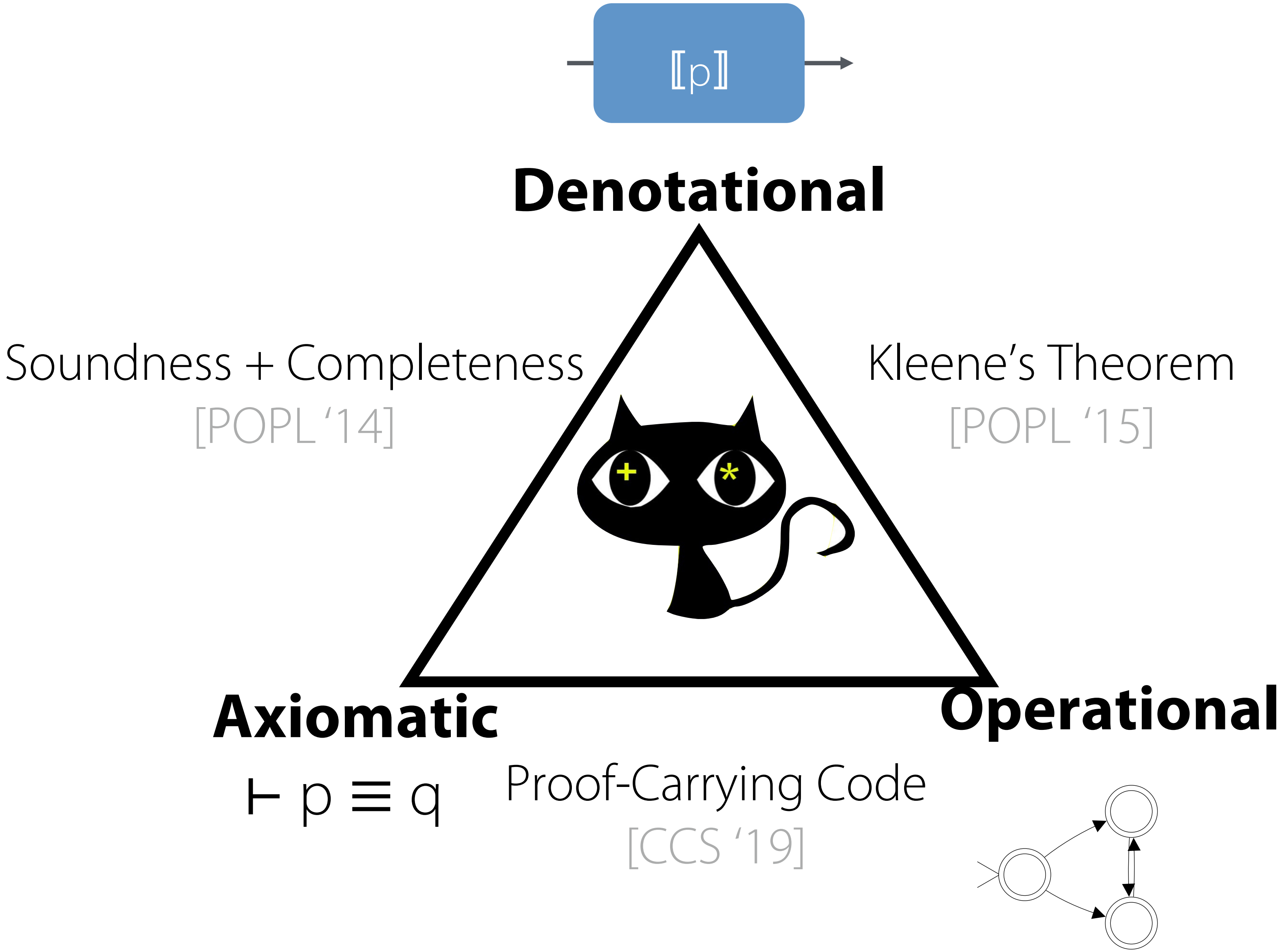
+

Packet Primitives

NetKAT

Provides guidance for the language design and an (almost) ready-made verification toolkit

Formal Semantics



$[\![p]\!]$

Denotational

Soundness + Completeness

[POPL ‘14]

Kleene’s Theorem

[POPL ‘15]

Axiomatic

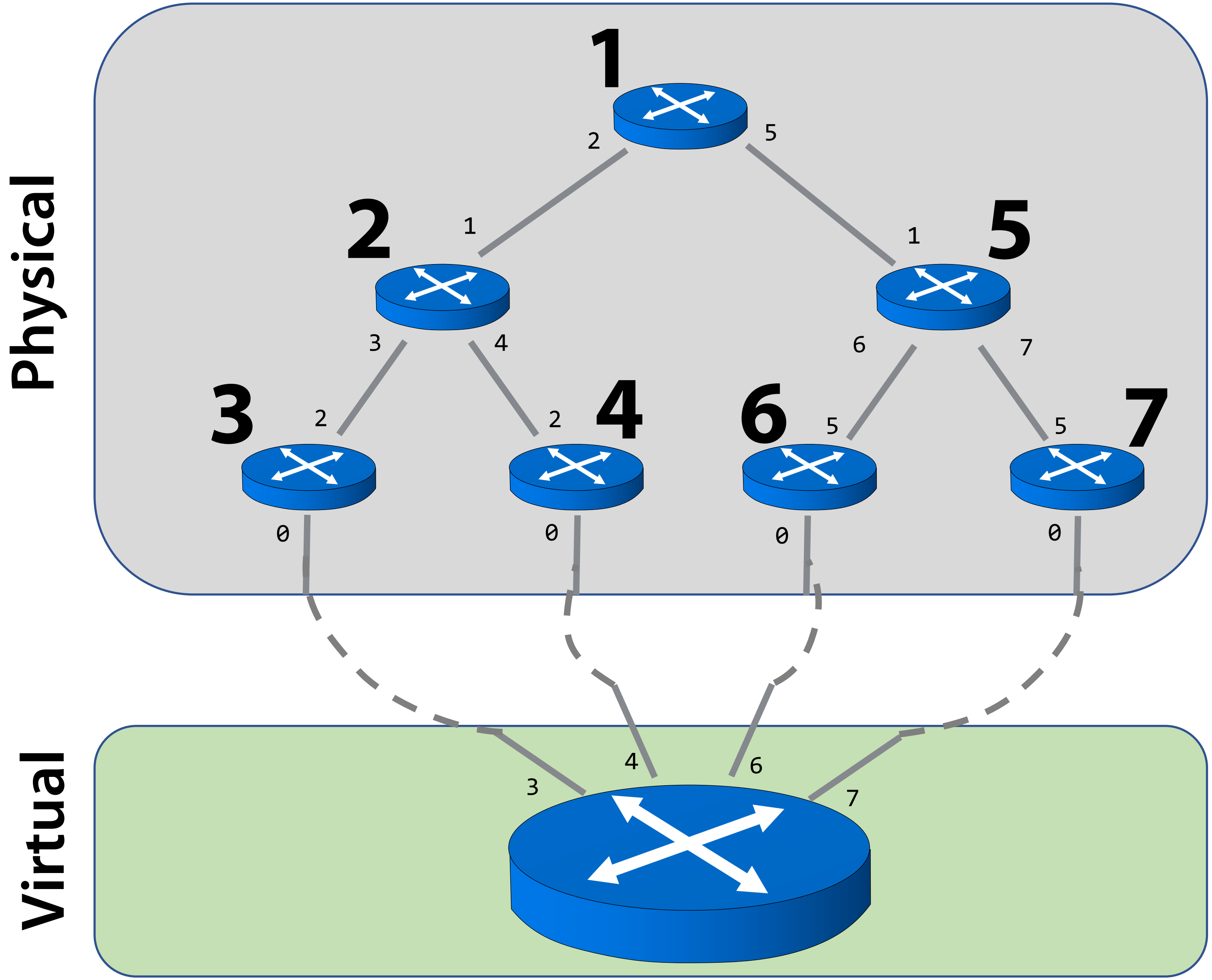
$\vdash p \equiv q$

Operational

Proof-Carrying Code

[CCS ‘19]

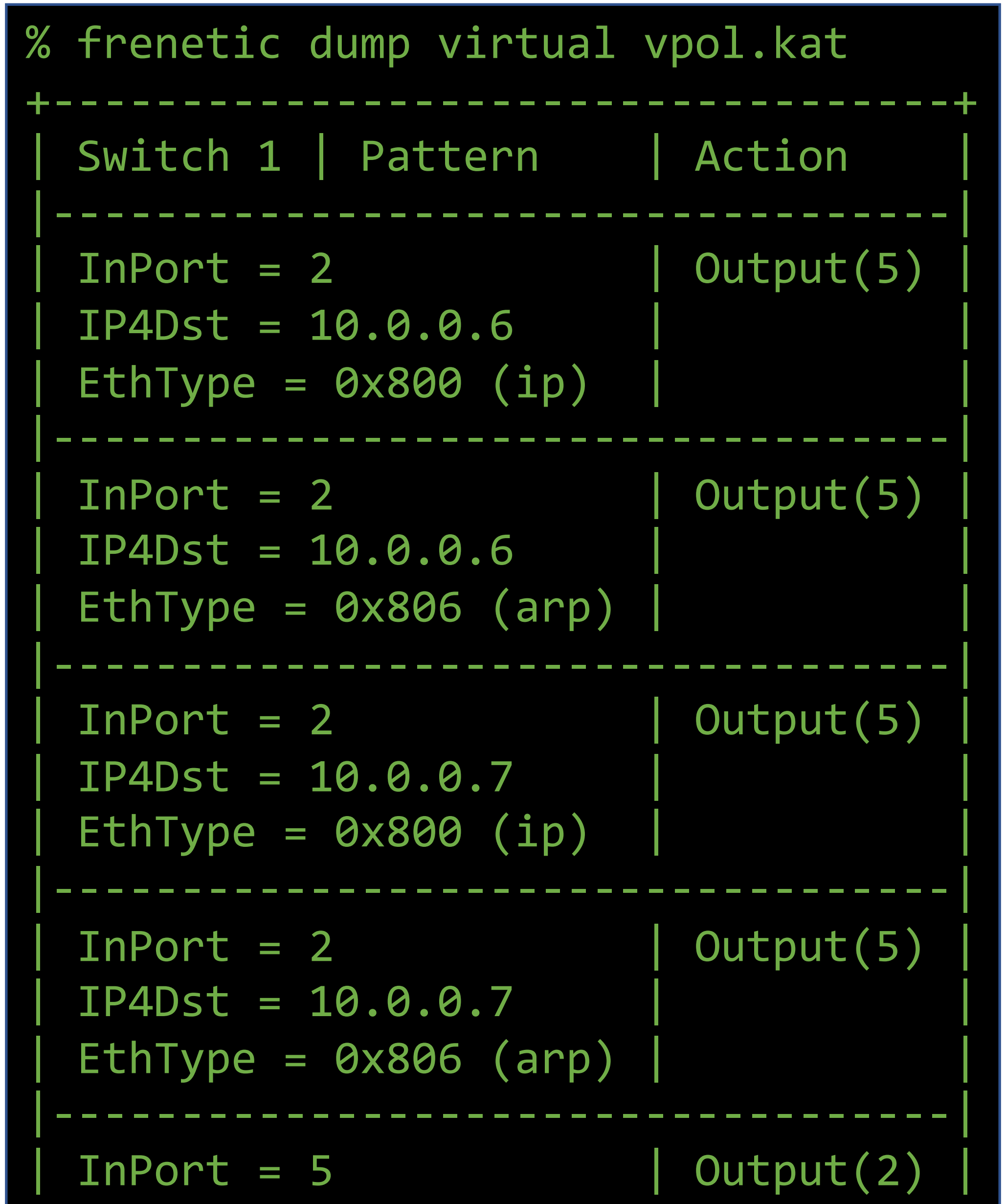
Virtual Compilation



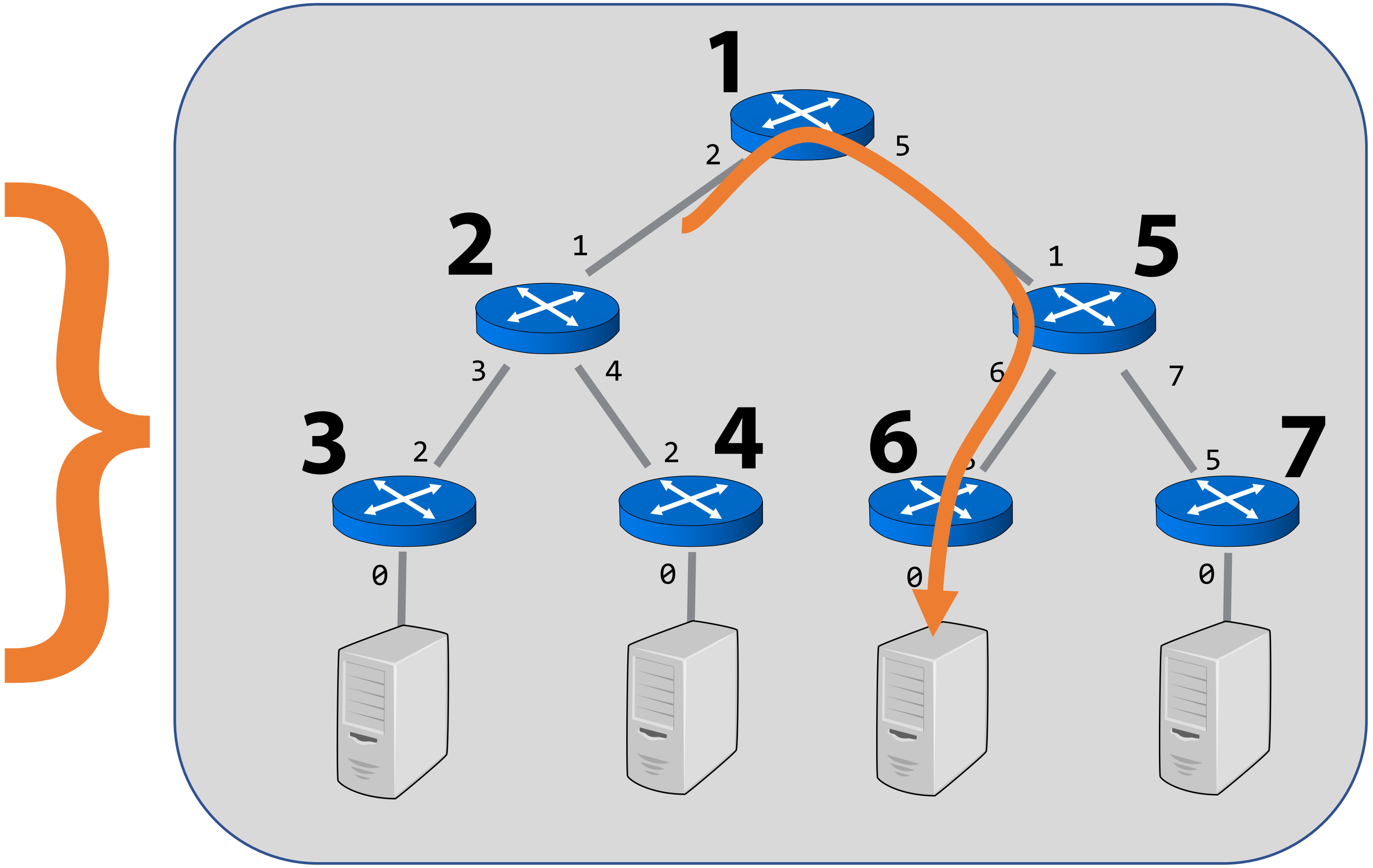
Virtual Policy

```
if ip4Dst=10.0.0.3 then
  vport:=3
else if ip4Dst=10.0.0.4 then
  vport:=4
else if ip4Dst=10.0.0.6 then
  vport:=6 else
if ip4Dst=10.0.0.7 then
  vport:=7
else
  drop
```

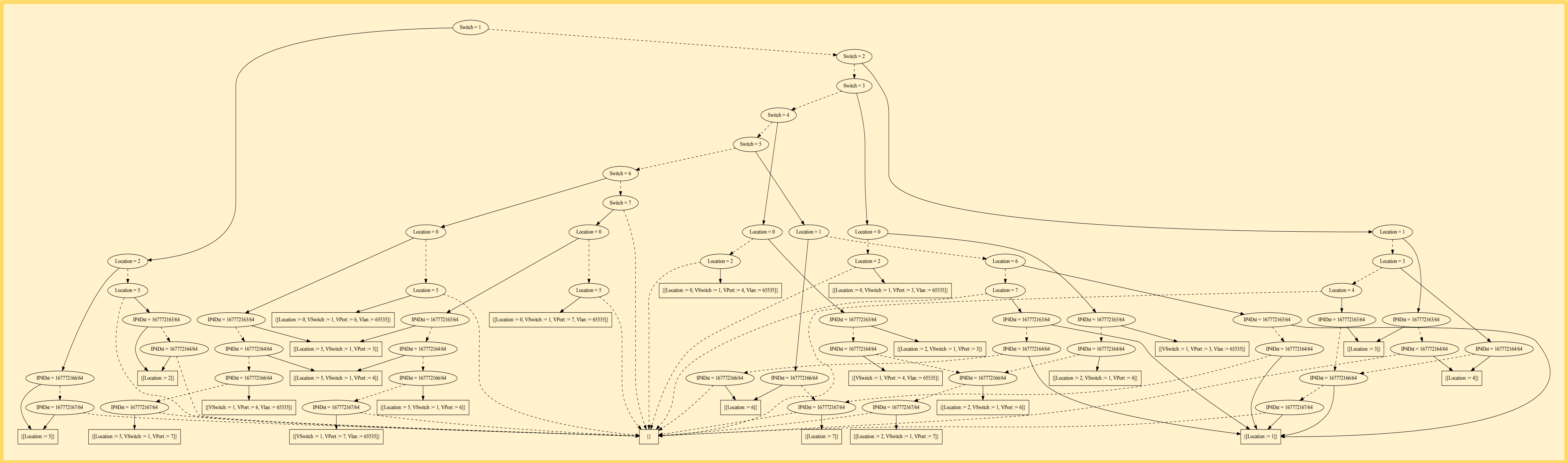
Compiler Demo



```
% frenetic dump virtual vpol.kat
+-----------------------------------+
| Switch 1 | Pattern    | Action    |
|-----------------------------------|
| InPort = 2            | Output(5) |
| IP4Dst = 10.0.0.6     |           |
| EthType = 0x800 (ip)  |           |
|-----------------------------------|
| InPort = 2            | Output(5) |
| IP4Dst = 10.0.0.6     |           |
| EthType = 0x806 (arp) |           |
|-----------------------------------|
| InPort = 2            | Output(5) |
| IP4Dst = 10.0.0.7     |           |
| EthType = 0x800 (ip)  |           |
|-----------------------------------|
| InPort = 2            | Output(5) |
| IP4Dst = 10.0.0.7     |           |
| EthType = 0x806 (arp) |           |
|-----------------------------------|
| InPort = 5            | Output(2) |
```



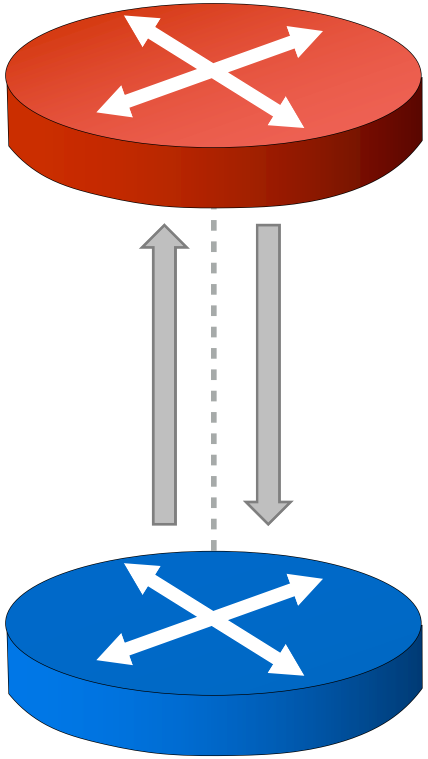
NetKAT Automaton



Internally, the compiler exploits the semantic foundation provided by KAT to convert the program to an automaton, which then guides the generation of match-action forwarding rules

Dynamic Network Updates

So, what about the control plane?

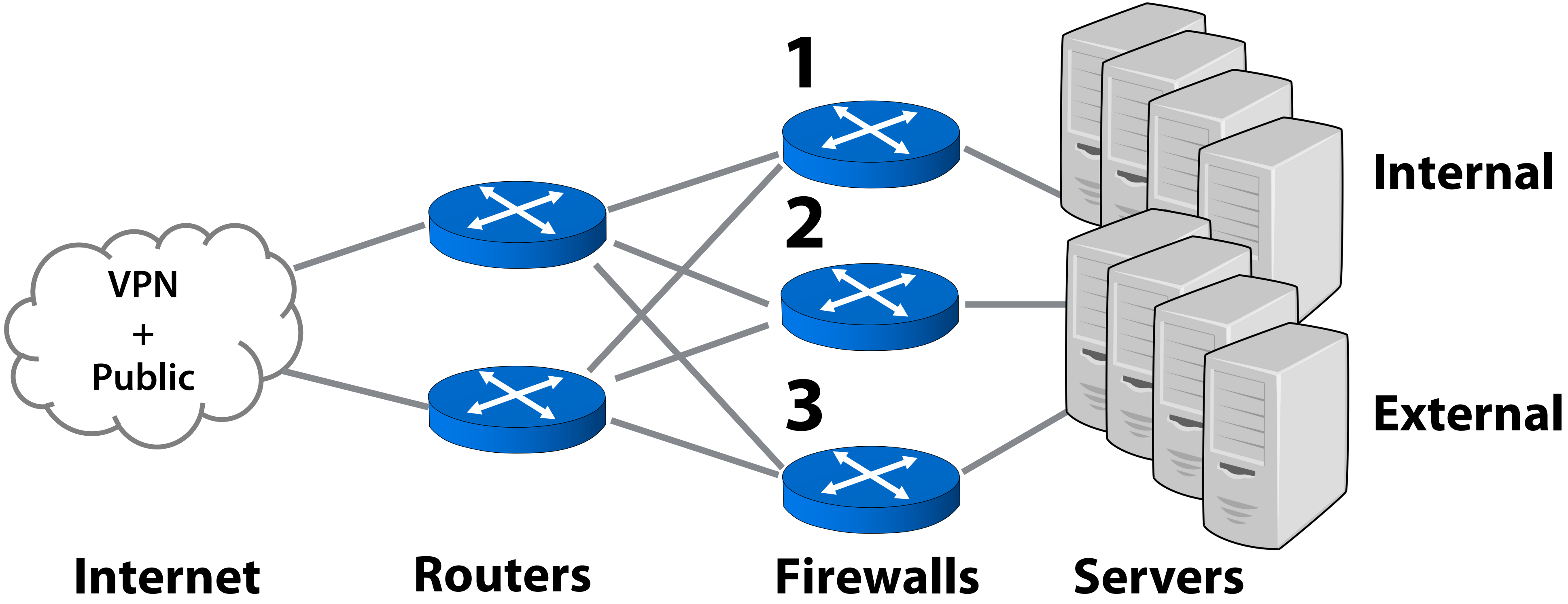


We've seen how to raise the level of abstraction, going from match-action tables to network-wide forwarding functions

But the control plane often needs to make *changes*, in response to events such as:

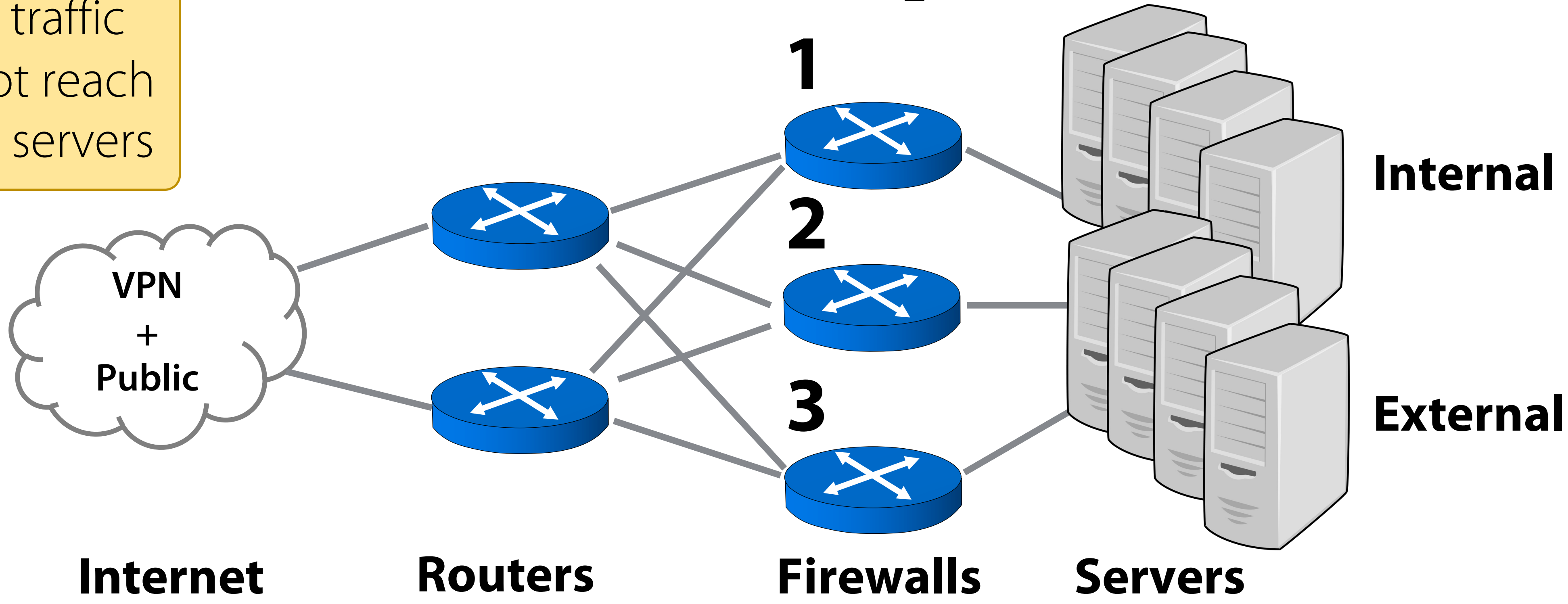
- Topology changes
- Shifts in traffic demands
- Device or link failures
- Operator-initiated maintenance

Network Updates



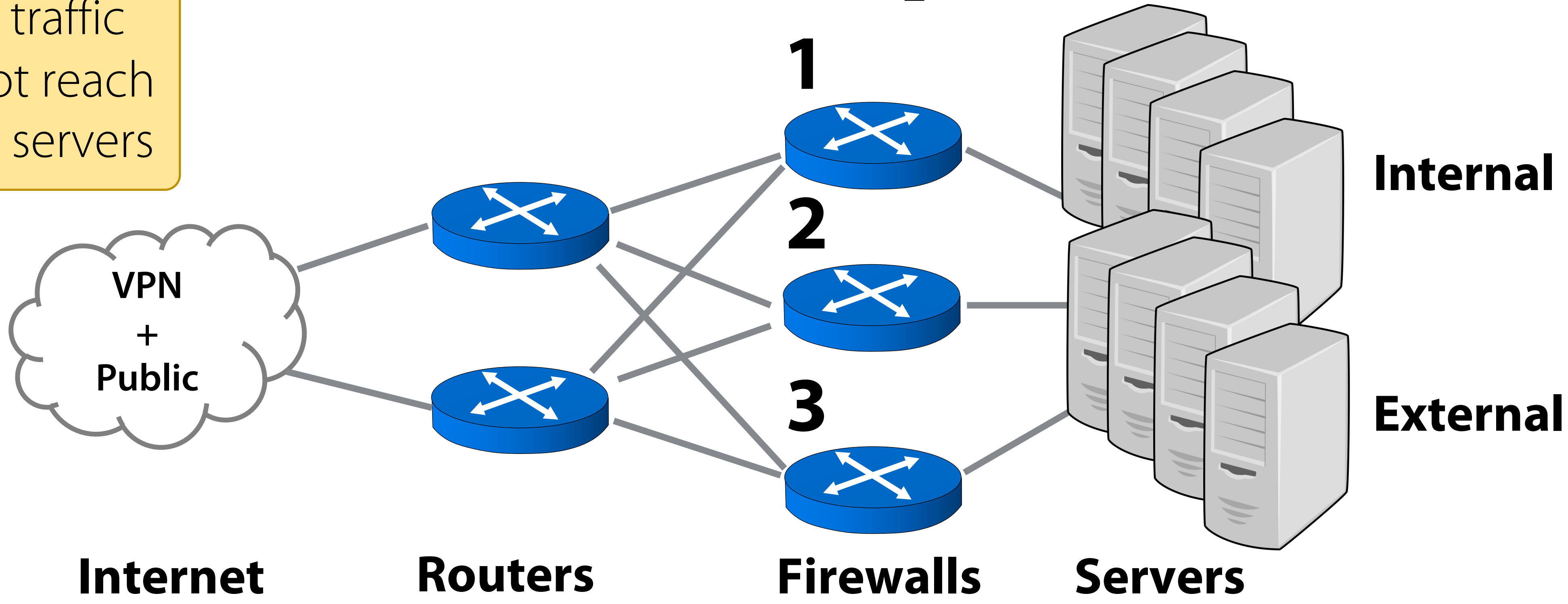
Network Updates

Public traffic must not reach internal servers



Network Updates

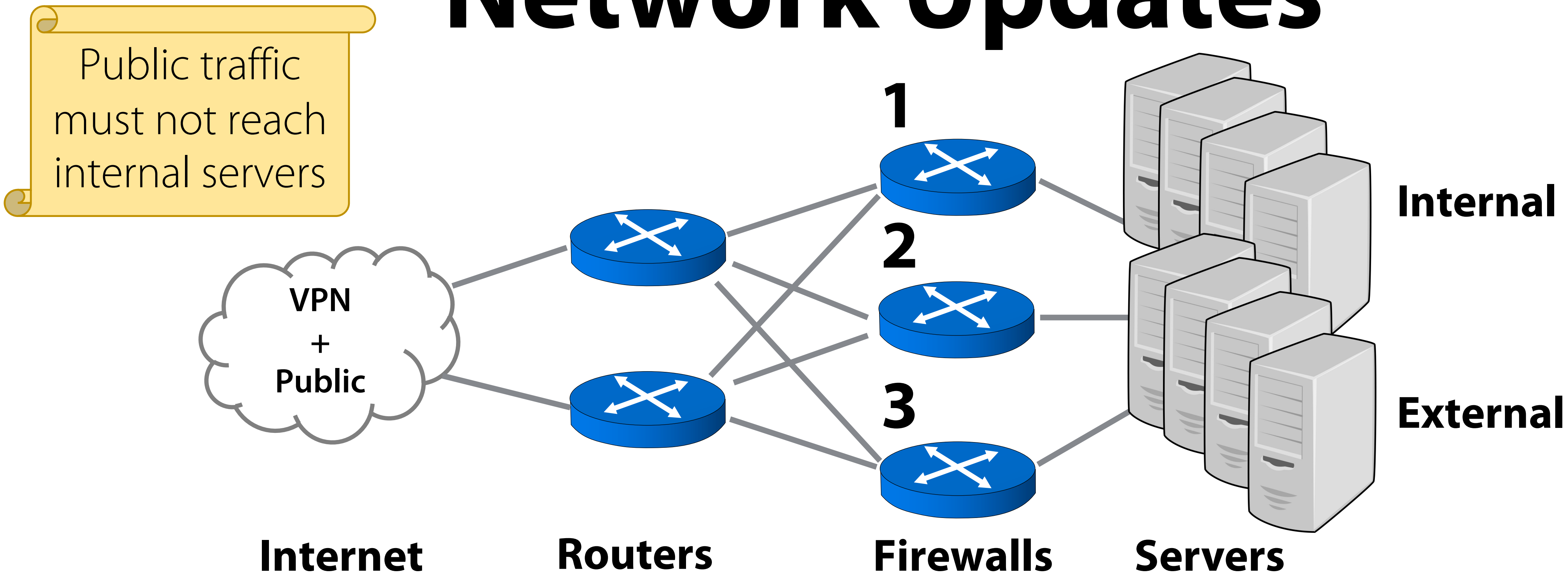
Public traffic must not reach internal servers



Configuration A:

- VPN via Firewall #1
- Public via Firewalls #2-3

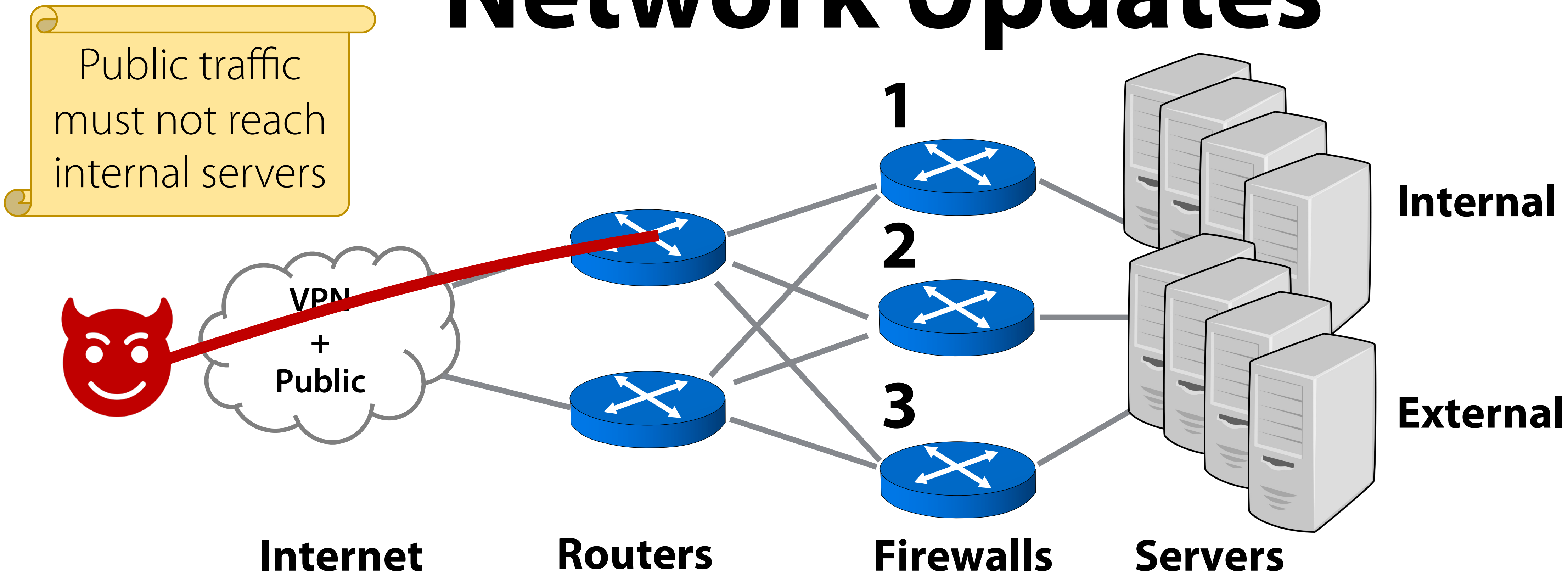
Network Updates



Configuration A:
- VPN via Firewall #1
- Public via Firewalls #2-3

Configuration B:
- VPN via Firewalls #1-2
- Public via Firewall #3

Network Updates



Configuration A:

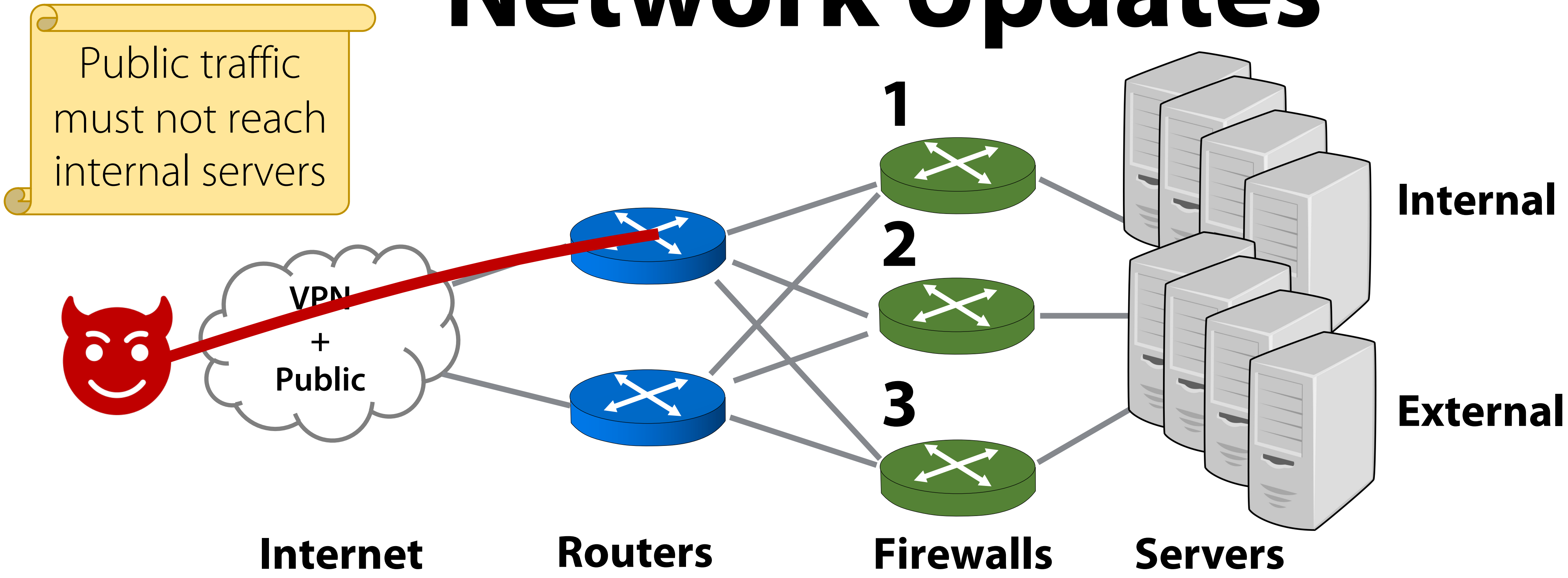
- VPN via Firewall #1
- Public via Firewalls #2-3

→

Configuration B:

- VPN via Firewalls #1-2
- Public via Firewall #3

Network Updates



Configuration A:

- VPN via Firewall #1
- Public via Firewalls #2-3

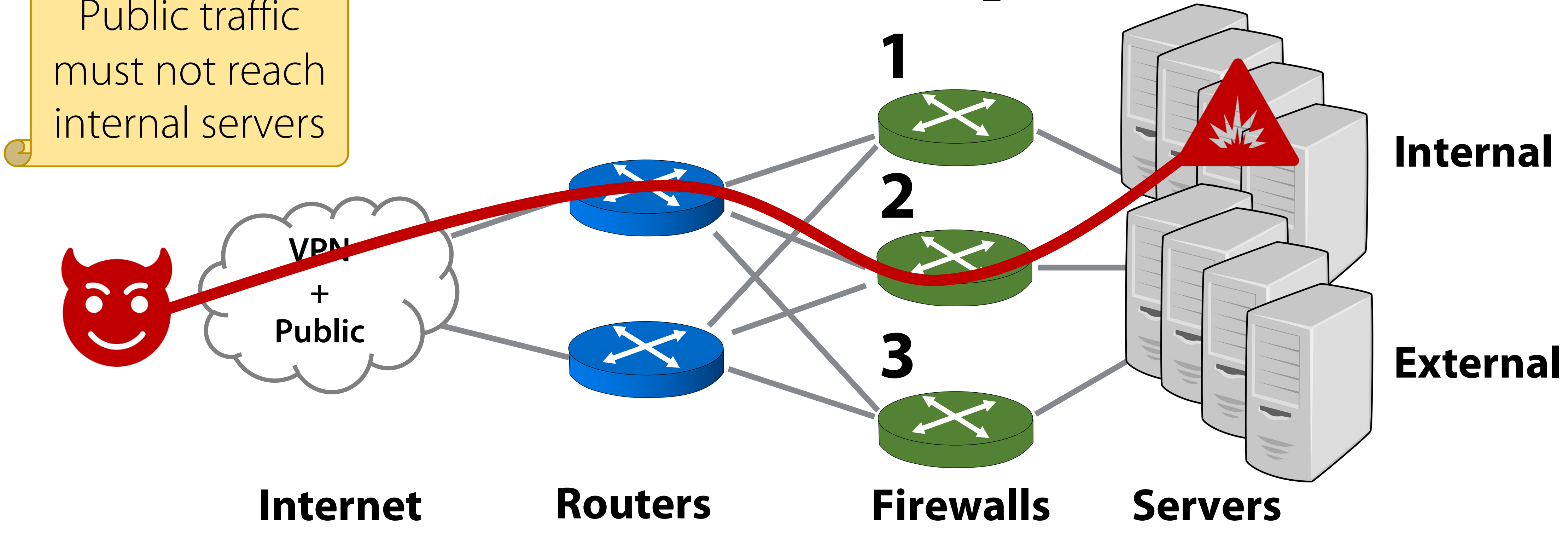
→

Configuration B:

- VPN via Firewalls #1-2
- Public via Firewall #3

Network Updates

Public traffic must not reach internal servers



Configuration A:

- VPN via Firewall #1
- Public via Firewalls #2-3

→

Configuration B:

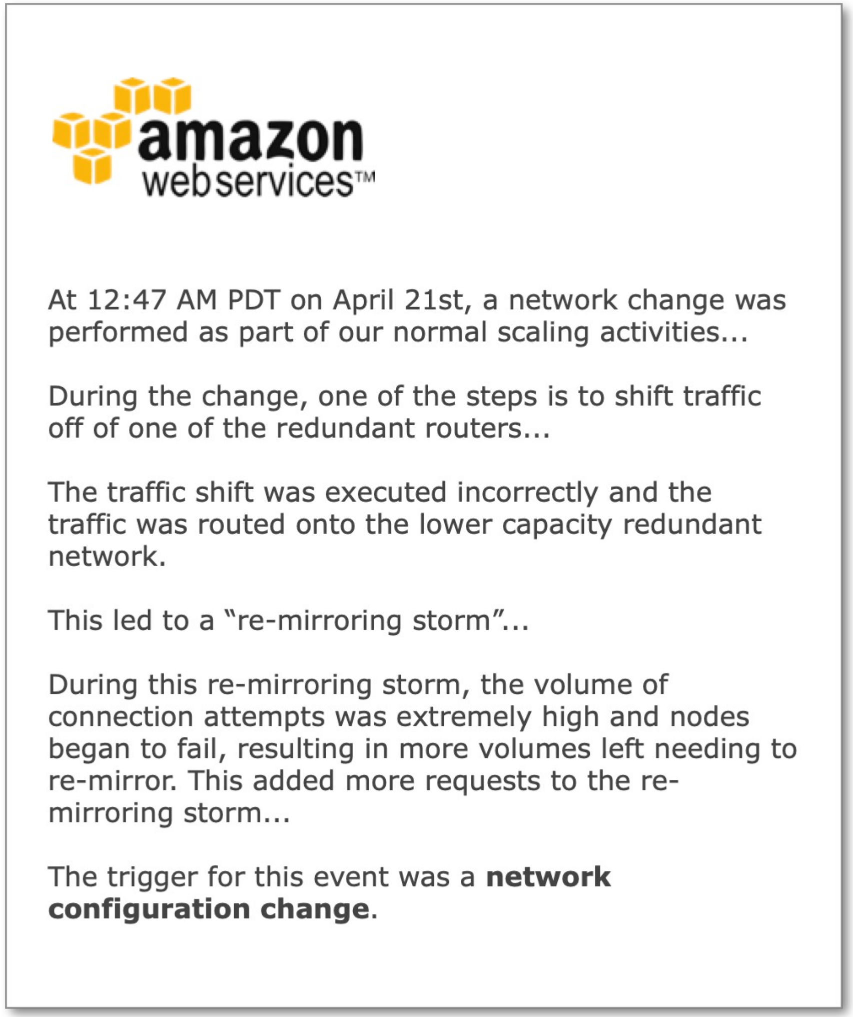
- VPN via Firewalls #1-2
- Public via Firewall #3

Network Updates in Practice

Network updates are a frequent source of faults including:

- Broken connections
- Access control violations
- Degraded quality of service
- Transient forwarding loops

Common heuristics, like “make before break,” do not handle every situation that arises in practice



amazon webservices™

At 12:47 AM PDT on April 21st, a network change was performed as part of our normal scaling activities...

During the change, one of the steps is to shift traffic off of one of the redundant routers...

The traffic shift was executed incorrectly and the traffic was routed onto the lower capacity redundant network.

This led to a “re-mirroring storm”...

During this re-mirroring storm, the volume of connection attempts was extremely high and nodes began to fail, resulting in more volumes left needing to re-mirror. This added more requests to the re-mirroring storm...

The trigger for this event was a **network configuration change**.

Consistent Updates

Intuitively, the problem with naïve upda
processes packets with a mixture of old

Key insight: view the network as a function, rather than a distributed collection of routing tables

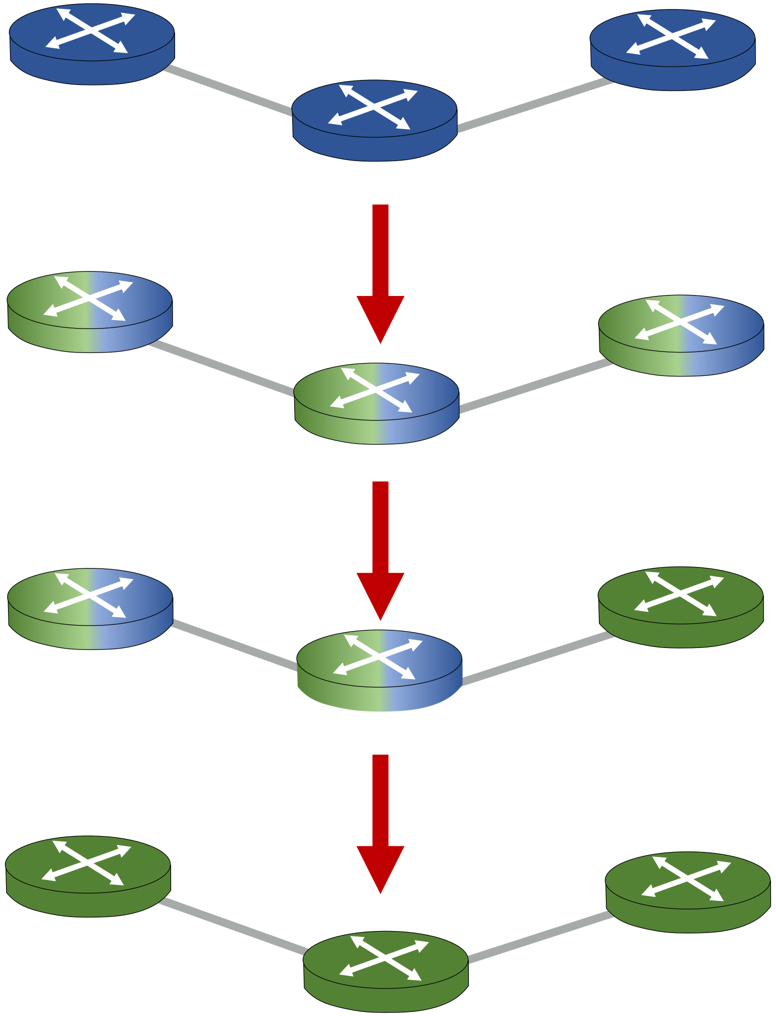
Definition [Per Packet Consistency]: an update from A to B is *per-packet consistent* if every packet is either entirely processed by A or by B (but not a mixture of the two!)

Theorem [Preservation]: a per-packet consistent update preserves every safety property

Two-Phase Updates

Algorithm

- Modify forwarding rules to check packet version
- Install new configuration in network core
- Install configuration at network edge to stamp packets with new version
- Wait for all in-flight packets to exit network
- Garbage collect old configurations



Pros

- Handles arbitrary network updates
- Many operations can be parallelized

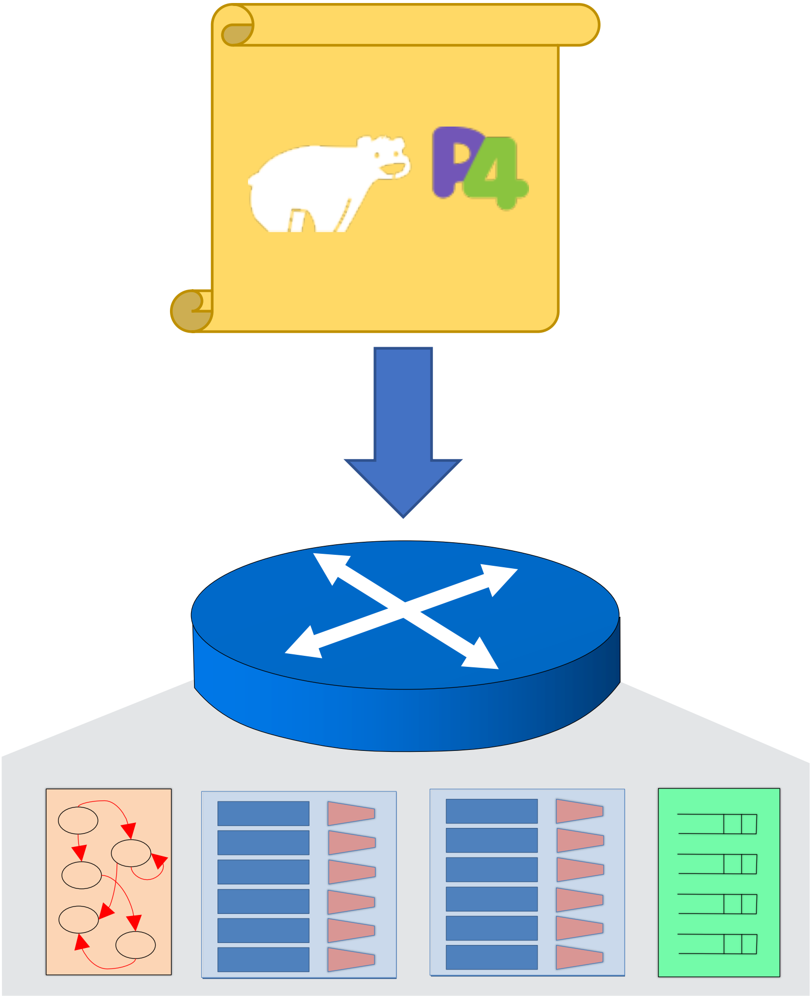
Cons

- Requires extra memory (2X in worst case)
- Packets must carry version tag

Looking Ahead

P4 Language

- P4: a new-ish DSL for specifying the behavior of programmable data planes
- Does not bake in *any* legacy protocols
- Instead, packet formats and pipeline are defined as imperative program
- Supports limited forms of state
- Programs terminate* and can be compiled to high-speed hardware



Petr4 Framework

Reference Interpreter

```
$ petr4
```

- Clean-slate implementation
- Architecture “plugins”
- Validated against open-source compiler

Core Calculus

$$\Gamma \vdash e : \tau$$

$$\langle \sigma, e \rangle \Downarrow v$$

- Formal semantics
- Termination theorem
- Language extension

Coq Mechanization

- Mechanized semantics
- Automata model of parsers
- Program equivalence

Emerging Opportunities

Deep programmability provides *many* opportunities to apply PL ideas to networking problems—come join the party!

Relevance and adoption:

- NetKAT-like policy languages used in *intent frameworks* for SDN controllers (Cisco, ONF, OpenDayLight, and others)
- *Network virtualization* is key technology behind VMware's NSX
- *Consistent updates* are used in Google Cloud
- *Network verification* teams at big companies (Amazon, Intel, VMware, Google) and startups (Intentionet, Forward Networks, Veriflow Systems)
- Galois developing a *cellular verification* framework based on NetKAT
- Growing community of academic and industrial users of Petr4

Some Open Problems...

Language Design: intent models, “chain-of-trust” networks

Compilation: heterogeneous architectures, P4, eBPF, WASM

Verification: compilers, hardware, timing channels, program logics, etc.

“The Edge”: cellular, access, Linux kernel, etc.

Cross-cutting issues:

- Stateful functions
- Failures
- Performance
- ML is coming...

Thank You

Collaborators

- Carolyn Anderson (Wesleyan)
- Arjun Guha (Northeastern)
- Dexter Kozen (Cornell)
- Nick McKeown (Intel)
- Mark Reitblatt (Facebook)
- Jennifer Rexford (Princeton)
- Cole Schlesinger (Akita)
- Steffen Smolka (Google)
- Alexandra Silva (Cornell)
- David Walker (Princeton)
- Spiros Eliopoulos (Jane Street)

`www.cs.cornell.edu/~jnfoster`