Event-Driven Network Programming

Jedidiah McClurg [1] Hossein Hojjat [2] Nate Foster [2] Pavol Černý [1]

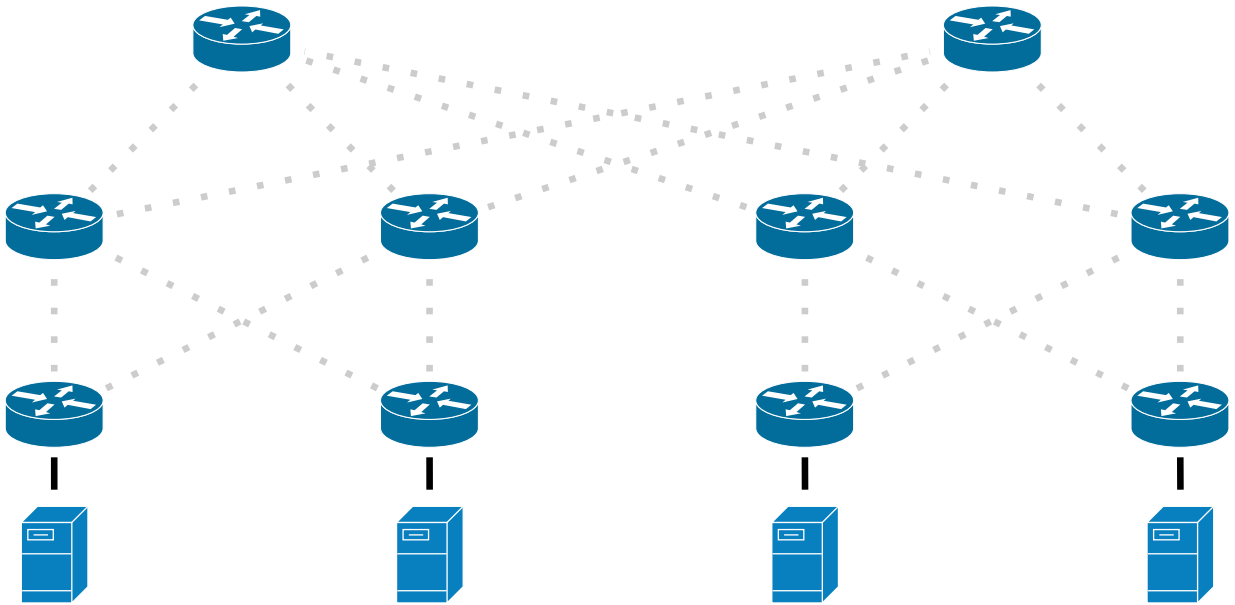



[1] University of Colorado Boulder

[2] Cornell University

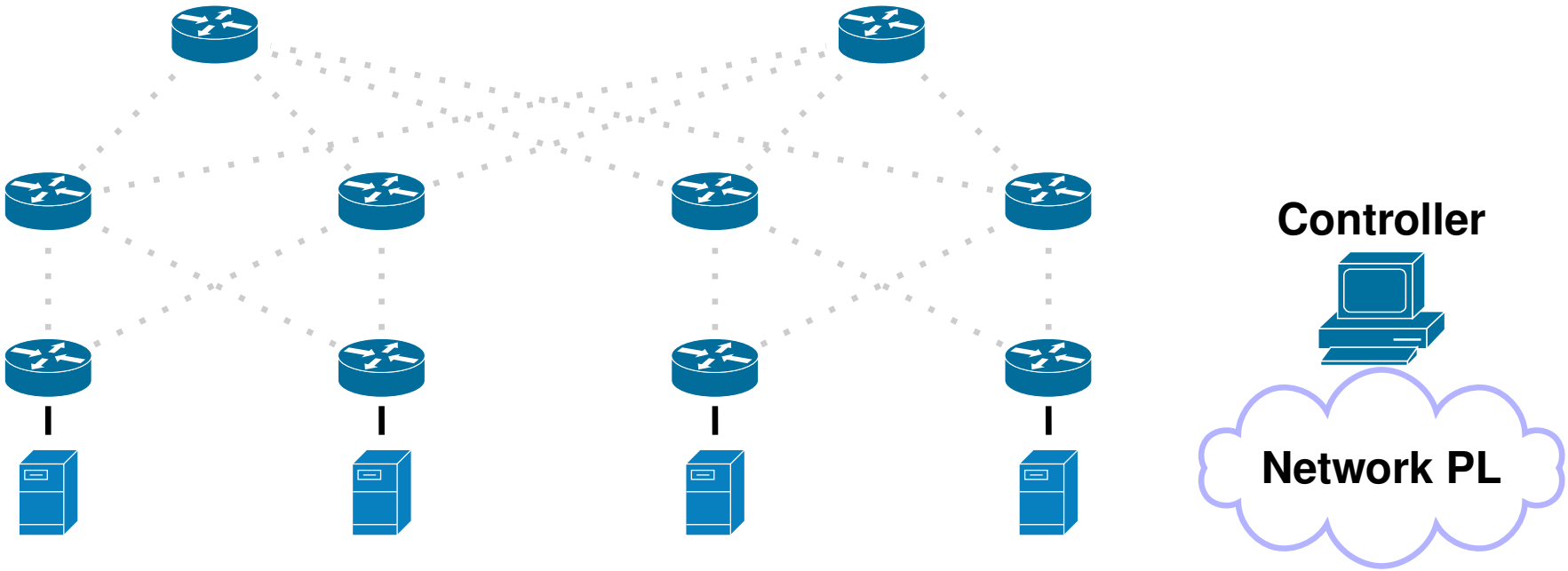
June 16th, 2016
(PLDI 2016)

Network Programming



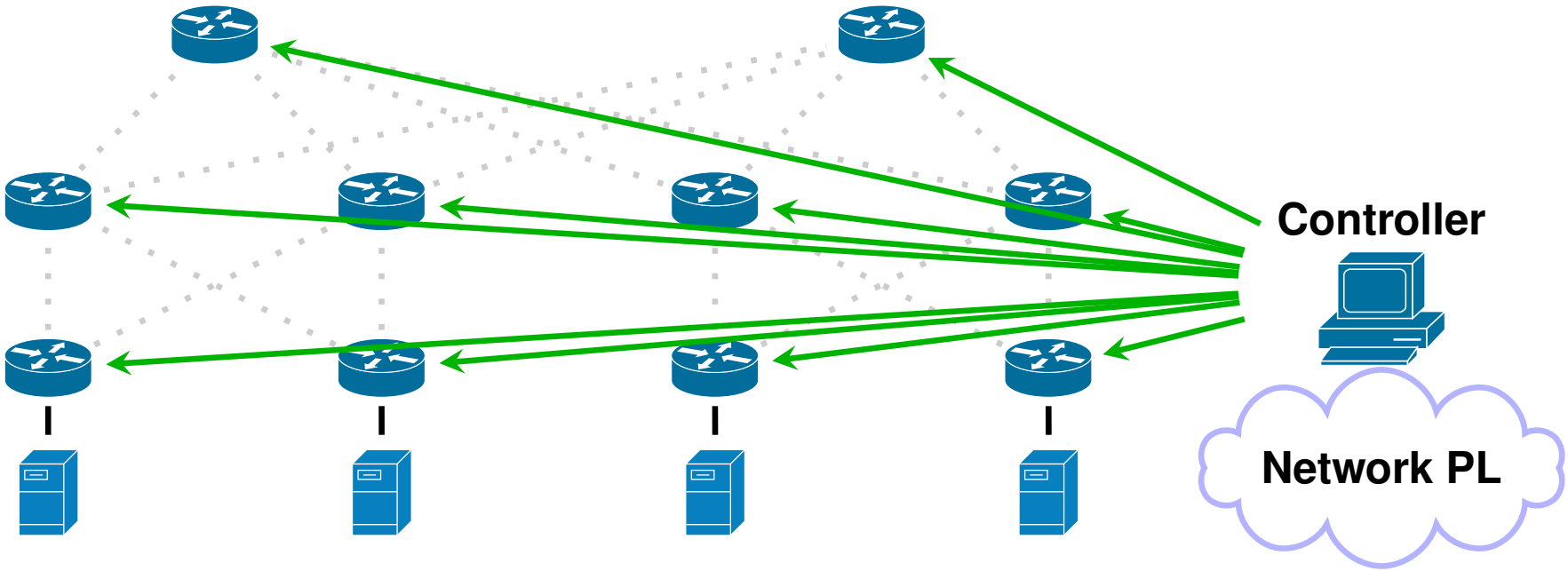
- *Network configurations*—static network forwarding behavior

Network Programming



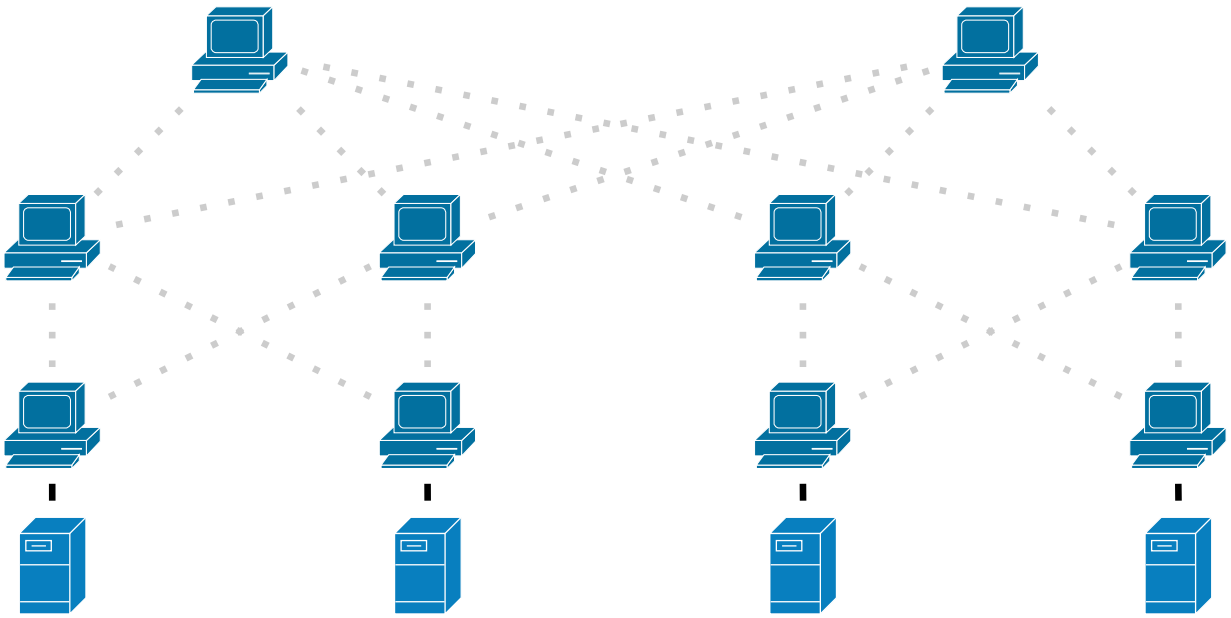
- *Network configurations*—static network forwarding behavior
- There are many languages for this (NetKAT [POPL'14], etc.)

Network Programming



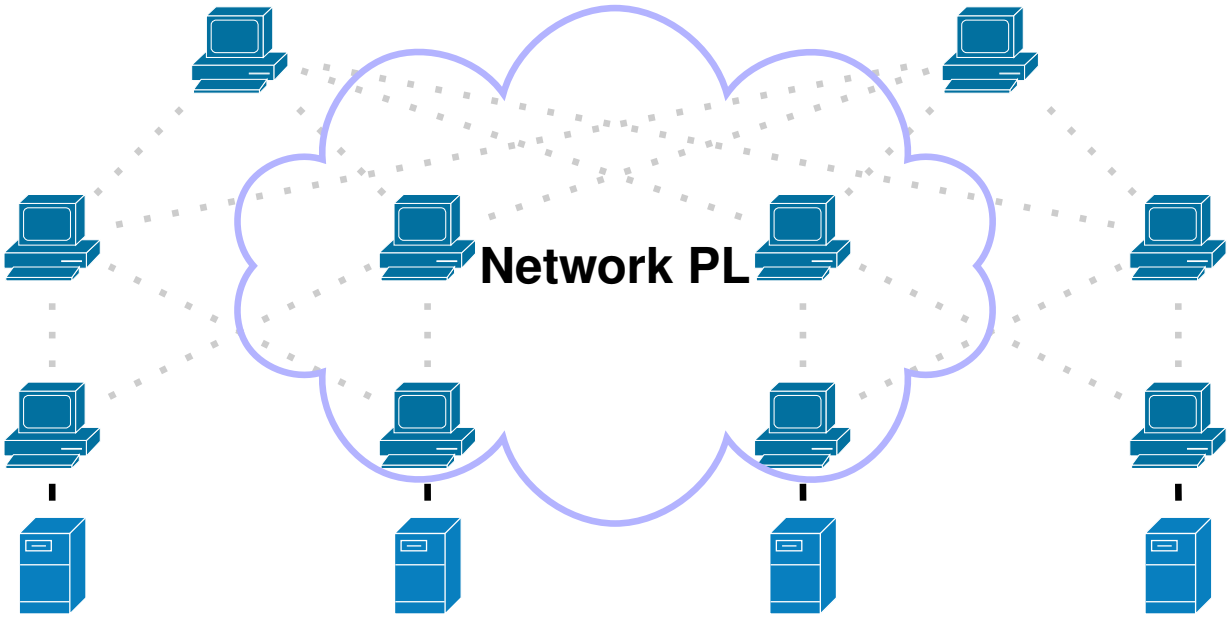
- *Network configurations*—static network forwarding behavior
- There are many languages for this (NetKAT [POPL'14], etc.)

Network Programming



- *Network configurations*—static network forwarding behavior
- There are many languages for this (NetKAT [POPL'14], etc.)
- SDN devices are becoming more programmable:
  mutable state, responding to events, etc.

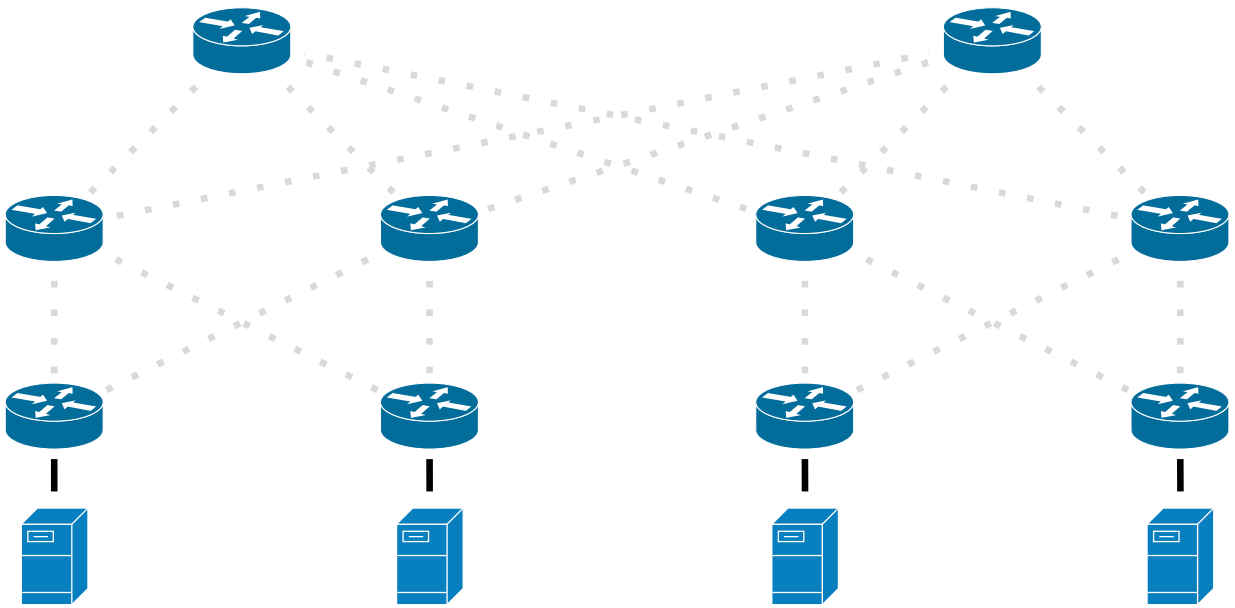
Network Programming



- *Network configurations*—static network forwarding behavior
- There are many languages for this (NetKAT [POPL'14], etc.)
- SDN devices are becoming more programmable: mutable state, responding to events, etc.
- Writing programs for these *dynamic* networks is difficult: handling shared state in a distributed system, etc.

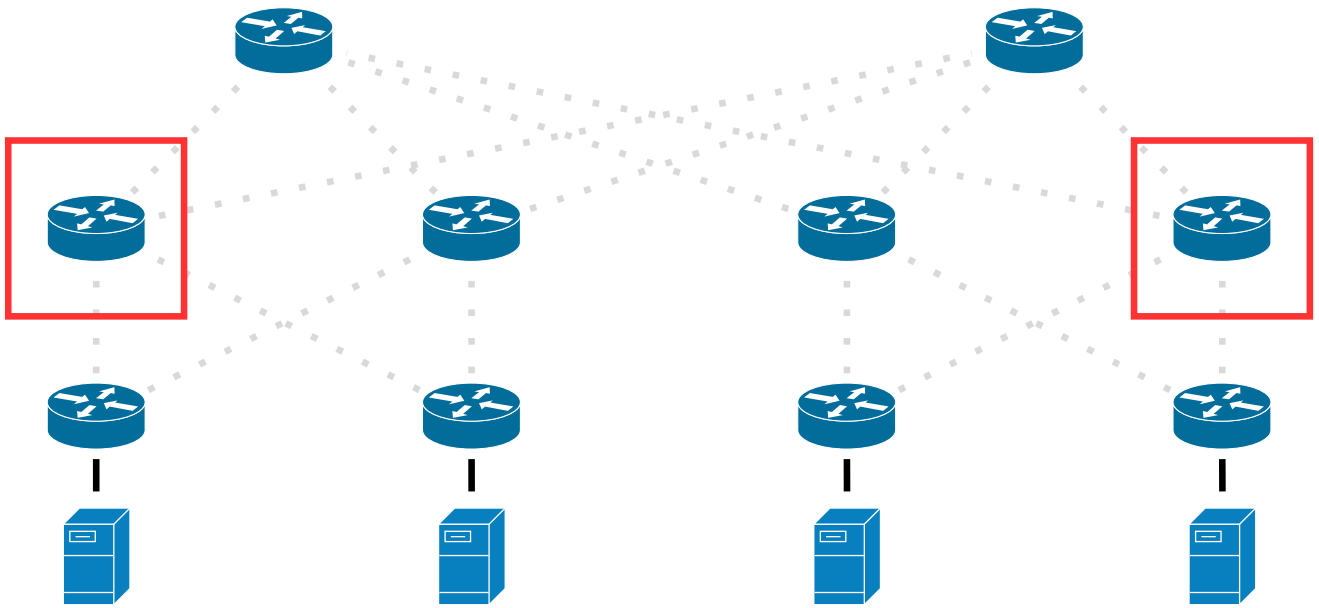
Consistency

- There are programming approaches for SDN which provide consistency in the form of an *atomic* construct



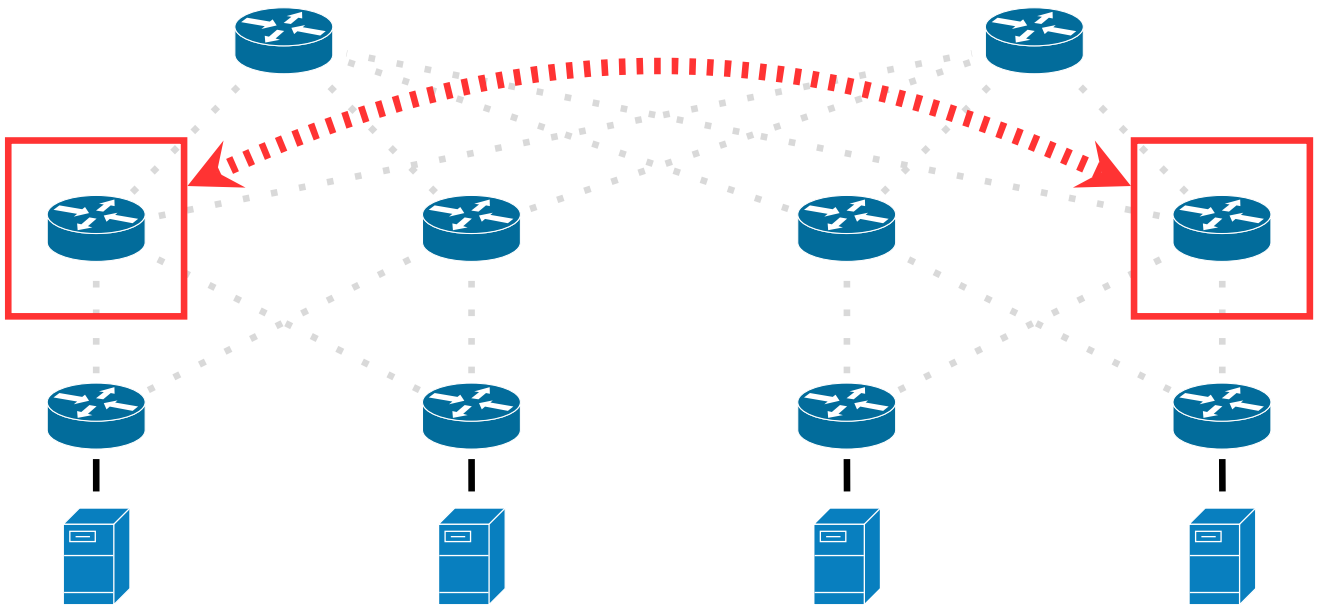
Consistency

- There are programming approaches for SDN which provide consistency in the form of an *atomic* construct
- Executing distributed updates atomically can be expensive



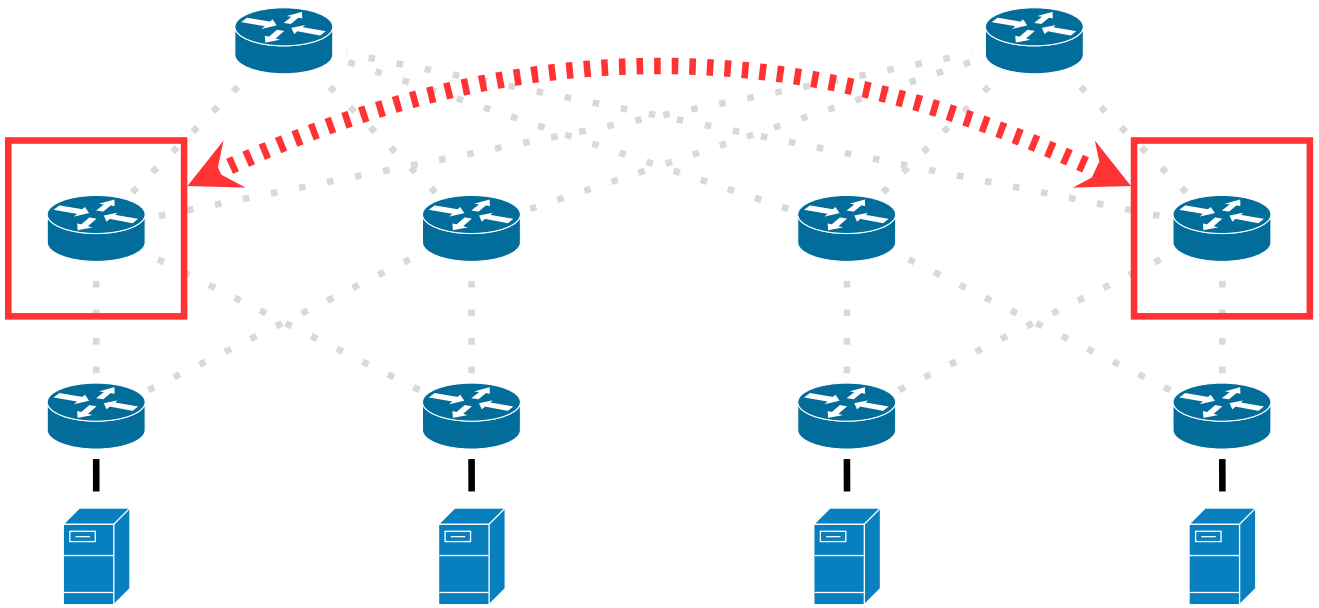
Consistency

- There are programming approaches for SDN which provide consistency in the form of an *atomic* construct
- Executing distributed updates atomically can be expensive



Consistency

- There are programming approaches for SDN which provide consistency in the form of an *atomic* construct
- Executing distributed updates atomically can be expensive



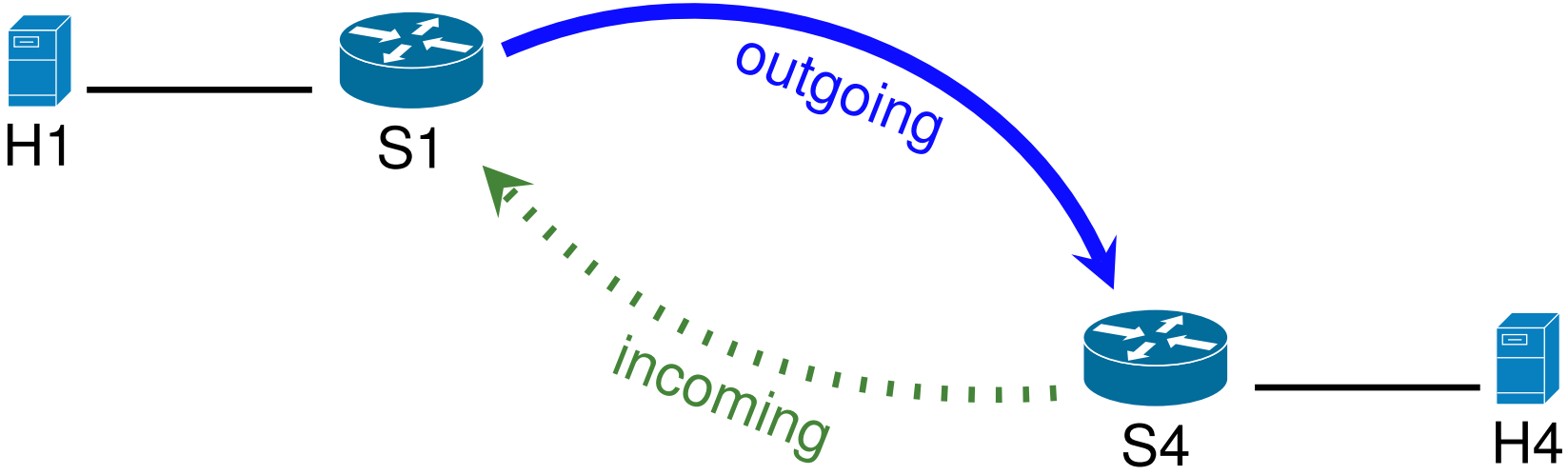
- We want to provide a useful form of consistency while ensuring that *all programs are efficiently implementable*

When Should we Respond to Events?

- In dynamic programs, a *packet event* ***e*** can trigger an update

When Should we Respond to Events?

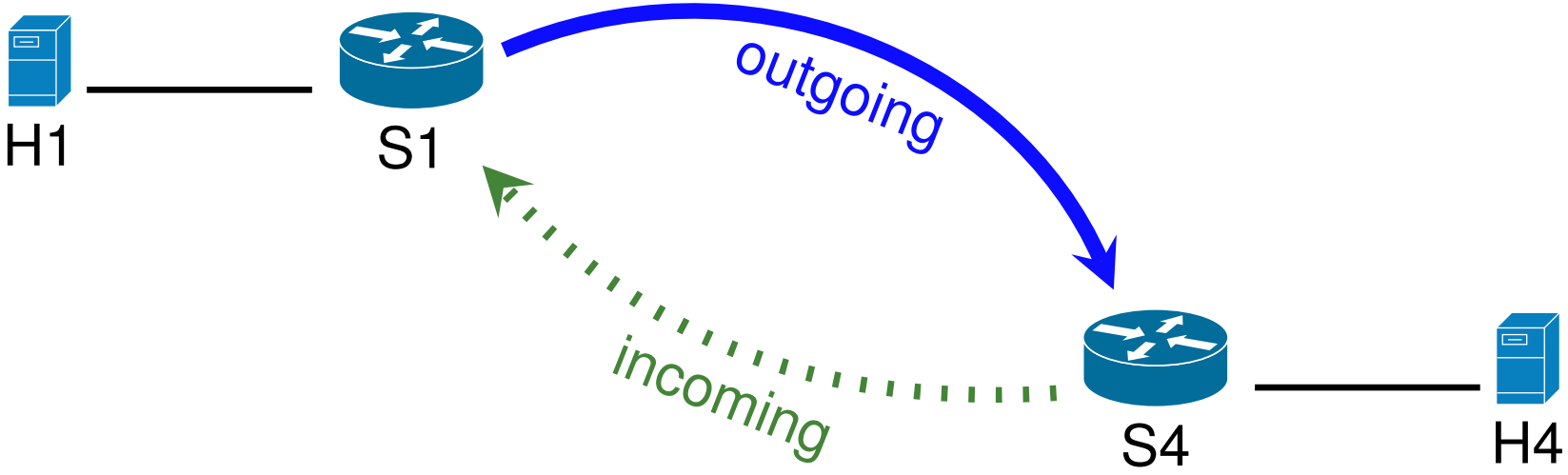
- In dynamic programs, a *packet event* ***e*** can trigger an update



Stateful Firewall

When Should we Respond to Events?

- In dynamic programs, a *packet event* ***e*** can trigger an update

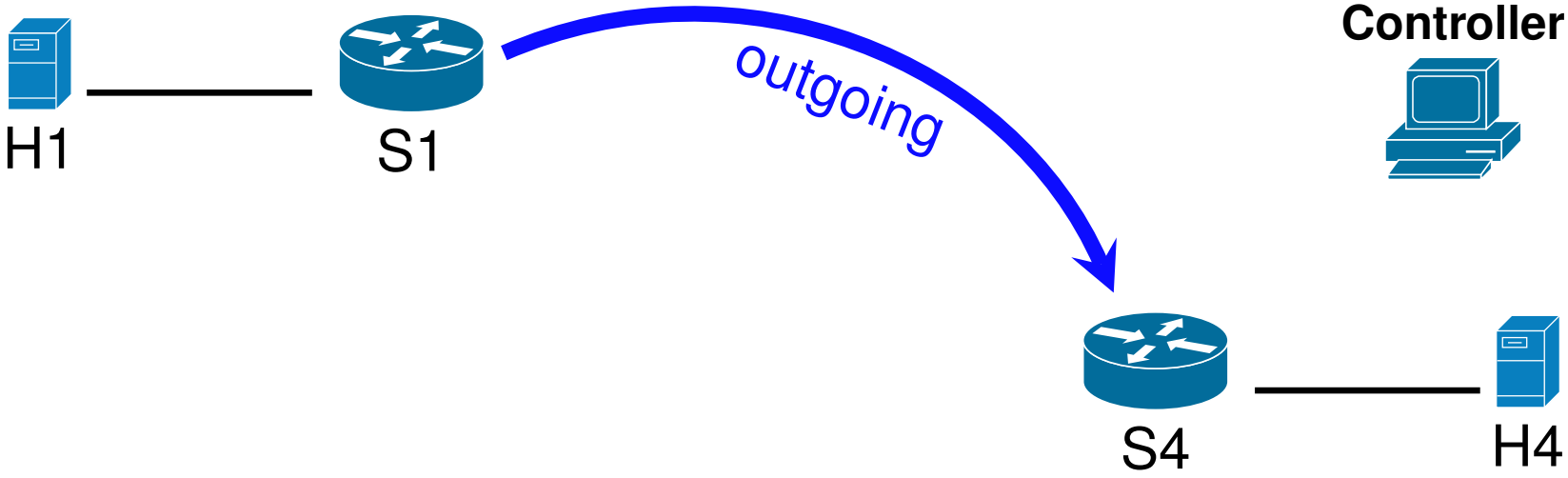


Stateful Firewall

- Consistency is a *multi-packet* property

When Should we Respond to Events?

- In dynamic programs, a *packet event* ***e*** can trigger an update

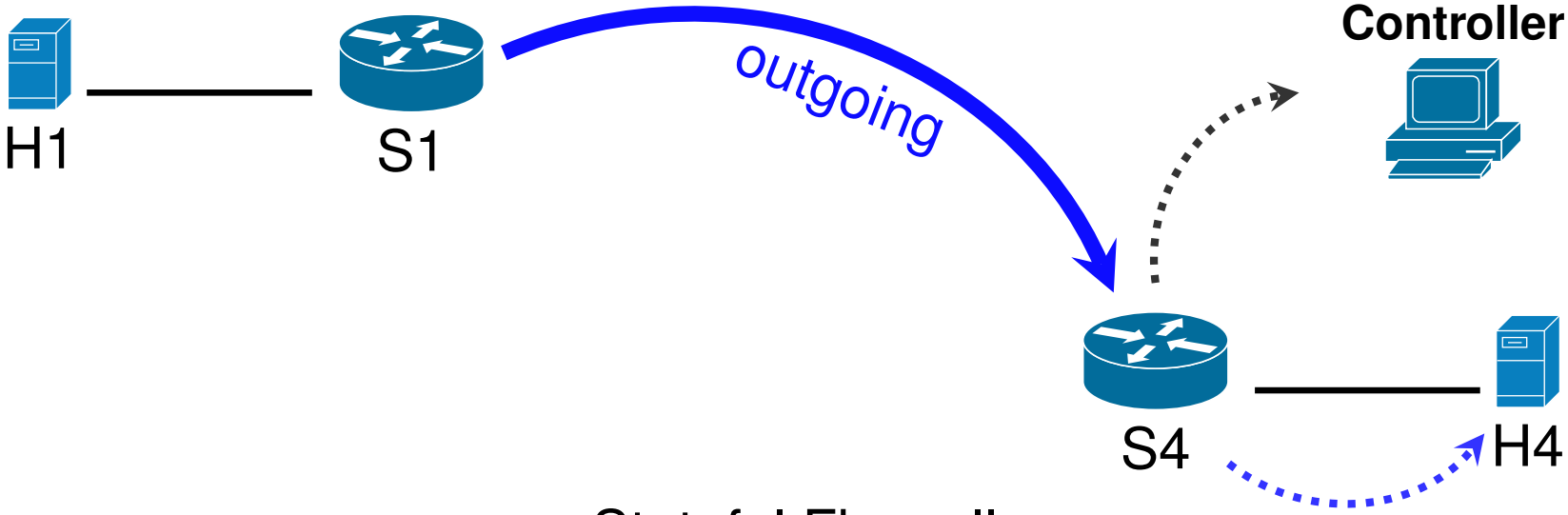


Stateful Firewall

- Consistency is a *multi-packet* property
- "Uncoordinated" updates cause problems in this example

When Should we Respond to Events?

- In dynamic programs, a *packet event* ***e*** can trigger an update

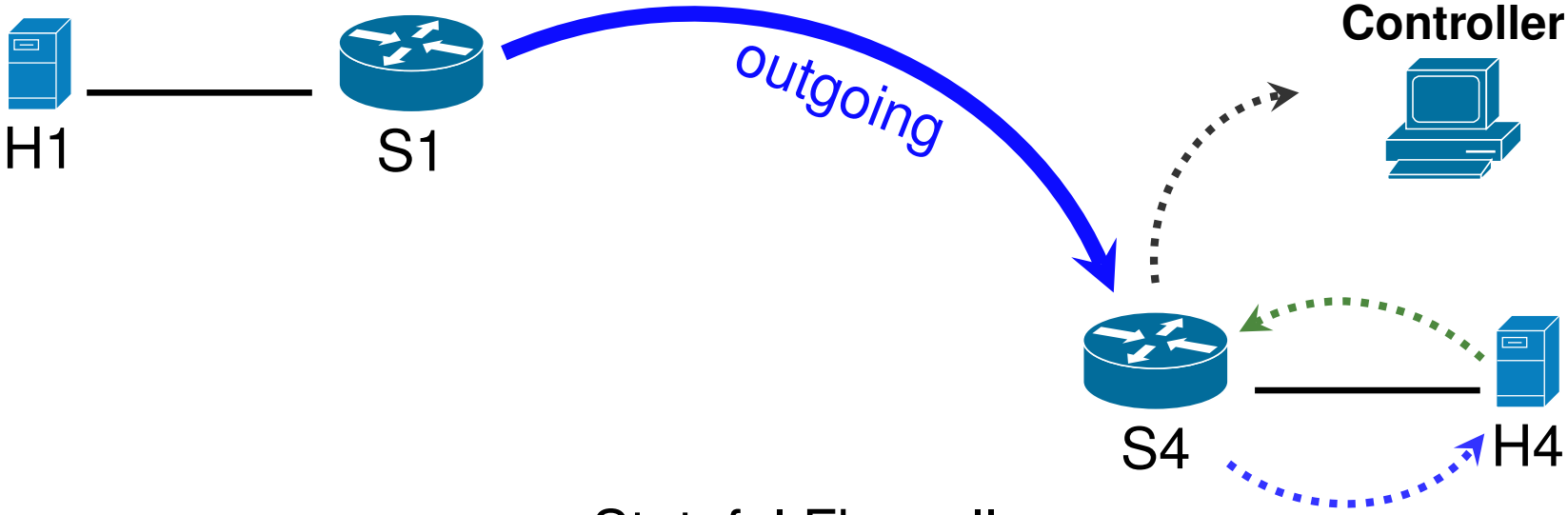


Stateful Firewall

- Consistency is a *multi-packet* property
- "Uncoordinated" updates cause problems in this example

When Should we Respond to Events?

- In dynamic programs, a *packet event* ***e*** can trigger an update

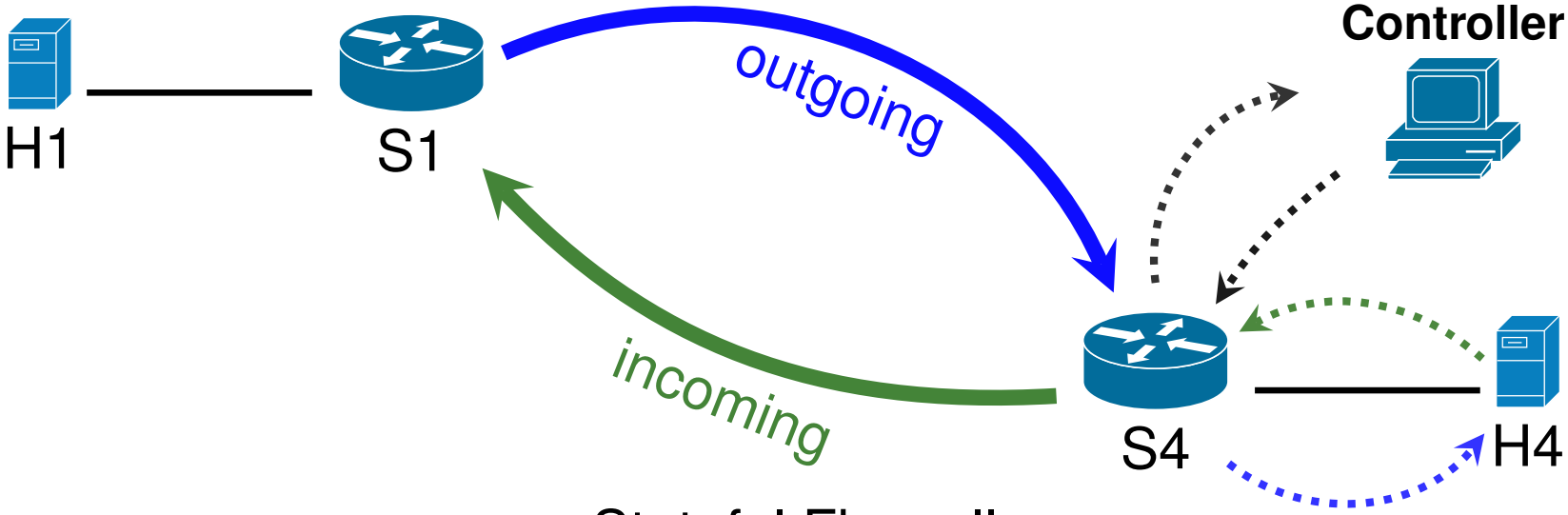


Stateful Firewall

- Consistency is a *multi-packet* property
- "Uncoordinated" updates cause problems in this example

When Should we Respond to Events?

- In dynamic programs, a *packet event* ***e*** can trigger an update

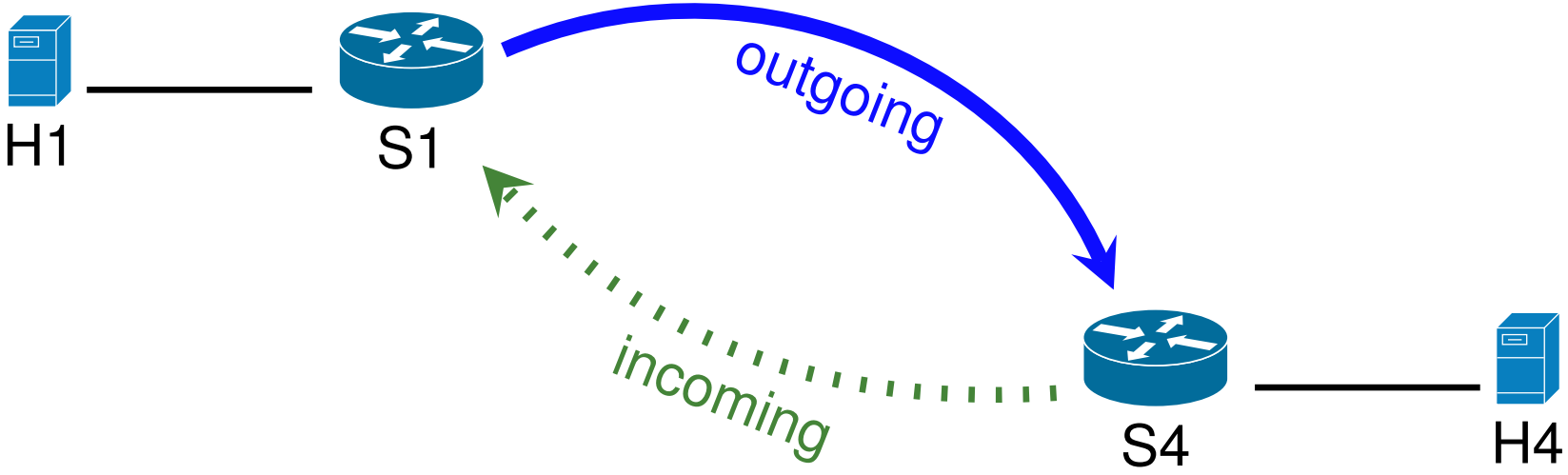


Stateful Firewall

- Consistency is a *multi-packet* property
- "Uncoordinated" updates cause problems in this example

When Should we Respond to Events?

- In dynamic programs, a *packet event* ***e*** can trigger an update

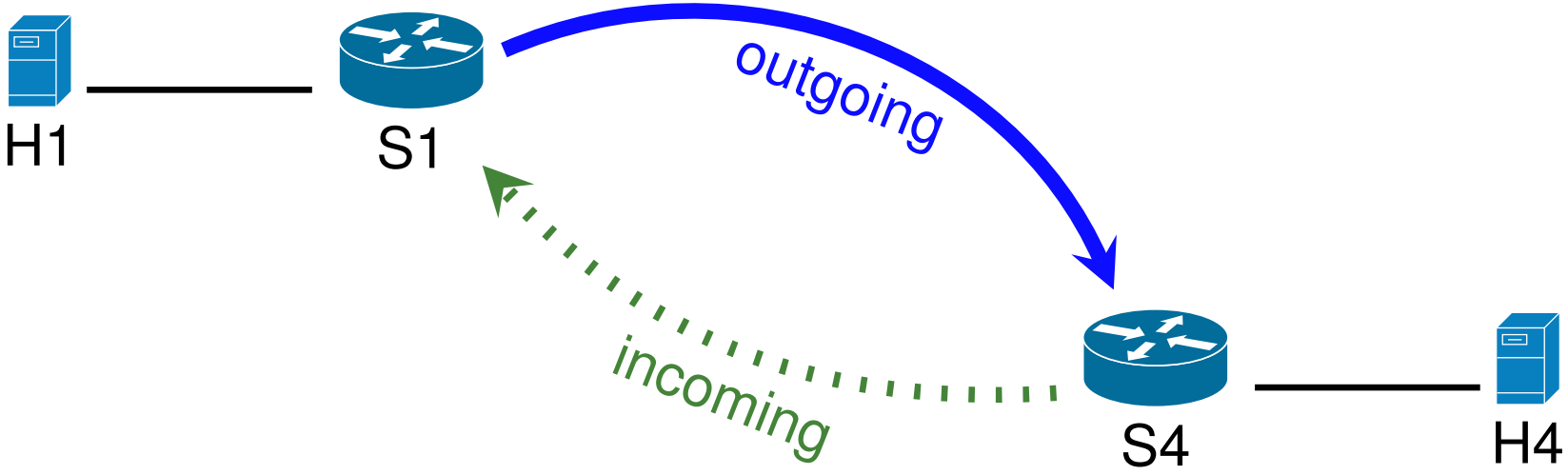


Stateful Firewall

- Consistency is a *multi-packet* property
- "Uncoordinated" updates cause problems in this example
- We need stronger guarantees about *when* configurations change with respect to events

When Should we Respond to Events?

- In dynamic programs, a *packet event* ***e*** can trigger an update



Stateful Firewall

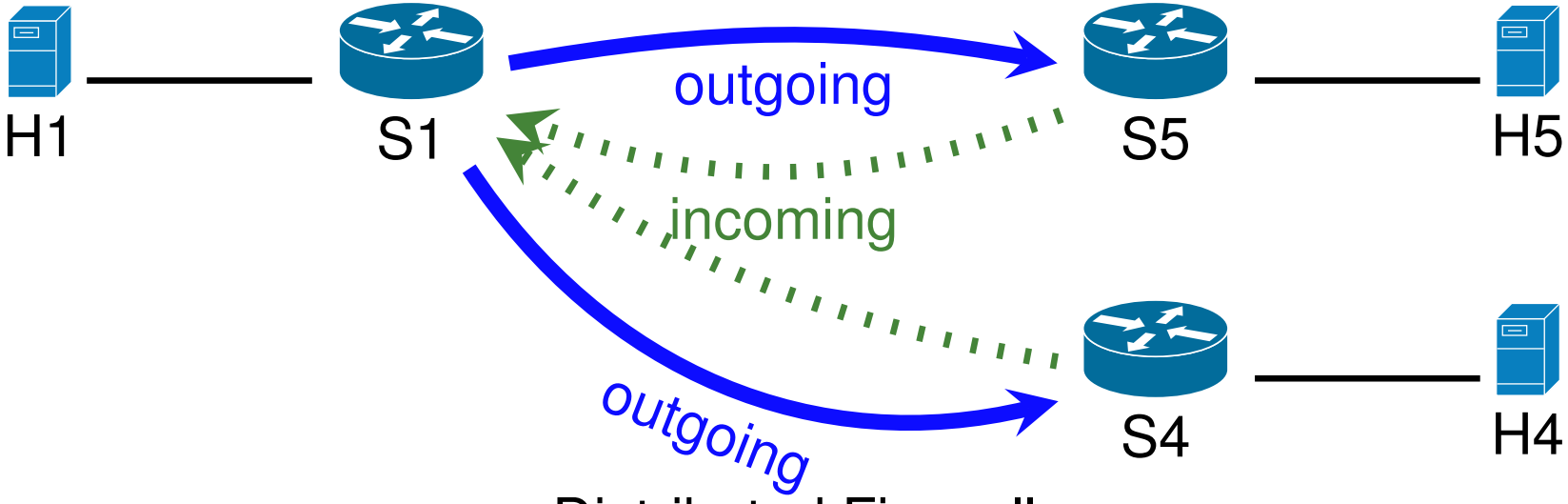
- Consistency is a *multi-packet* property
- "Uncoordinated" updates cause problems in this example
- We need stronger guarantees about *when* configurations change with respect to events
- *Don't respond to an event too late (and don't respond too early)!*

How Can we Handle Conflicting Events?

- Consider a firewall that authorizes **only the first-contacted** external host

How Can we Handle Conflicting Events?

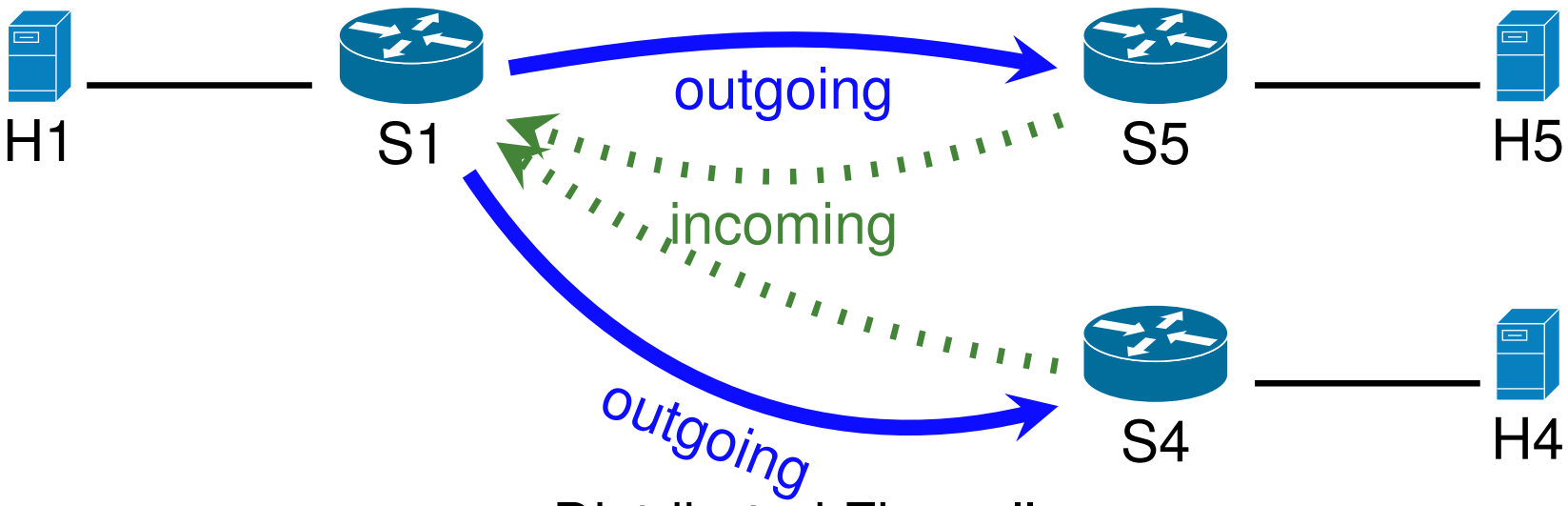
- Consider a firewall that authorizes **only the first-contacted** external host



Distributed Firewall

How Can we Handle Conflicting Events?

- Consider a firewall that authorizes **only the first-contacted** external host

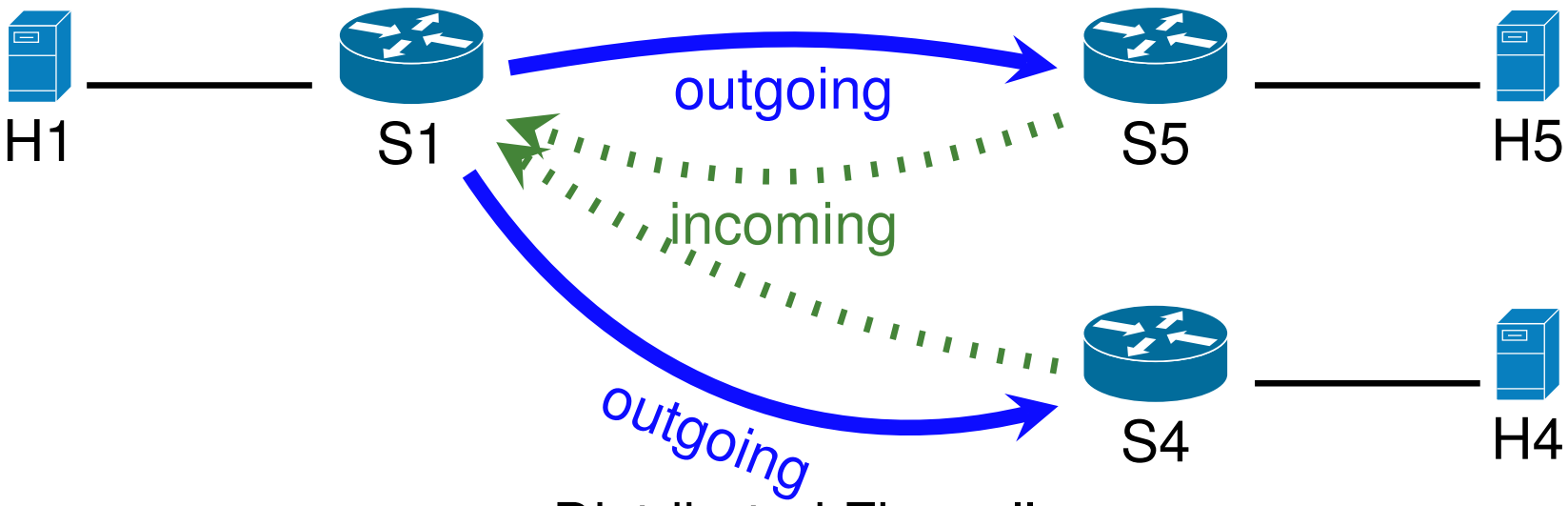


Distributed Firewall

- Switch *S4* and *S5* must somehow resolve conflicting views of which was "first"

How Can we Handle Conflicting Events?

- Consider a firewall that authorizes **only the first-contacted** external host



Distributed Firewall

- Switch *S4* and *S5* must somehow resolve conflicting views of which was "first"
- *This program cannot be implemented efficiently!*

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C')

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C')
   - If all switches traversed by a packet have heard about event e, the packet is processed in C'

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C')
   - If all switches traversed by a packet have heard about event e, the packet is processed in C'
   - If all switches traversed by a packet have *not* heard about e, the packet is processed in C

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C')
   - If all switches traversed by a packet have heard about event e, the packet is processed in C'
   - If all switches traversed by a packet have *not* heard about e, the packet is processed in C
   - We formalize what it means to have "heard about" an event with a *happens-before relation* $\prec$ *on packets*

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C')
   - If all switches traversed by a packet have heard about event e, the packet is processed in C'
   - If all switches traversed by a packet have *not* heard about e, the packet is processed in C
   - We formalize what it means to have "heard about" an event with a *happens-before relation* $\prec$ *on packets*
2. **Event Structure Consistency**

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C')
   - If all switches traversed by a packet have heard about event e, the packet is processed in C'
   - If all switches traversed by a packet have *not* heard about e, the packet is processed in C
   - We formalize what it means to have "heard about" an event with a *happens-before relation* $\prec$ *on packets*
2. **Event Structure Consistency**
   (specify how events interact with other events)

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C′)
   - If all switches traversed by a packet have heard about event *e*, the packet is processed in C'
   - If all switches traversed by a packet have *not* heard about *e*, the packet is processed in C
   - We formalize what it means to have "heard about" an event with a *happens-before relation* $\prec$ *on packets*
2. **Event Structure Consistency**
   (specify how events interact with other events)
   - We use restricted transition systems called *event structures*

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C′)
   - If all switches traversed by a packet have heard about event *e*, the packet is processed in C'
   - If all switches traversed by a packet have *not* heard about *e*, the packet is processed in C
   - We formalize what it means to have "heard about" an event with a *happens-before relation* $\prec$ *on packets*
2. **Event Structure Consistency**
   (specify how events interact with other events)
   - We use restricted transition systems called *event structures*
   - Conceptually, we require each transition in an event structure to be an event-driven consistent update

Ingredients of our Approach

1. **Event-Driven Consistent Update** $C \xrightarrow{e} C'$
   (determine whether a packet is processed in C or C′)
   - If all switches traversed by a packet have heard about event *e*, the packet is processed in C'
   - If all switches traversed by a packet have *not* heard about *e*, the packet is processed in C
   - We formalize what it means to have "heard about" an event with a *happens-before relation* $\prec$ *on packets*
2. **Event Structure Consistency**
   (specify how events interact with other events)
   - We use restricted transition systems called *event structures*
   - Conceptually, we require each transition in an event structure to be an event-driven consistent update
   - We provide a *locality condition* that guarantees efficient implementability

Programming with Events

- NetKAT language can specify a single network configuration

Programming with Events

- NetKAT language can specify a single network configuration
- We extend NetKAT to allow specifying *multiple* network configurations, and the event-driven updates between them

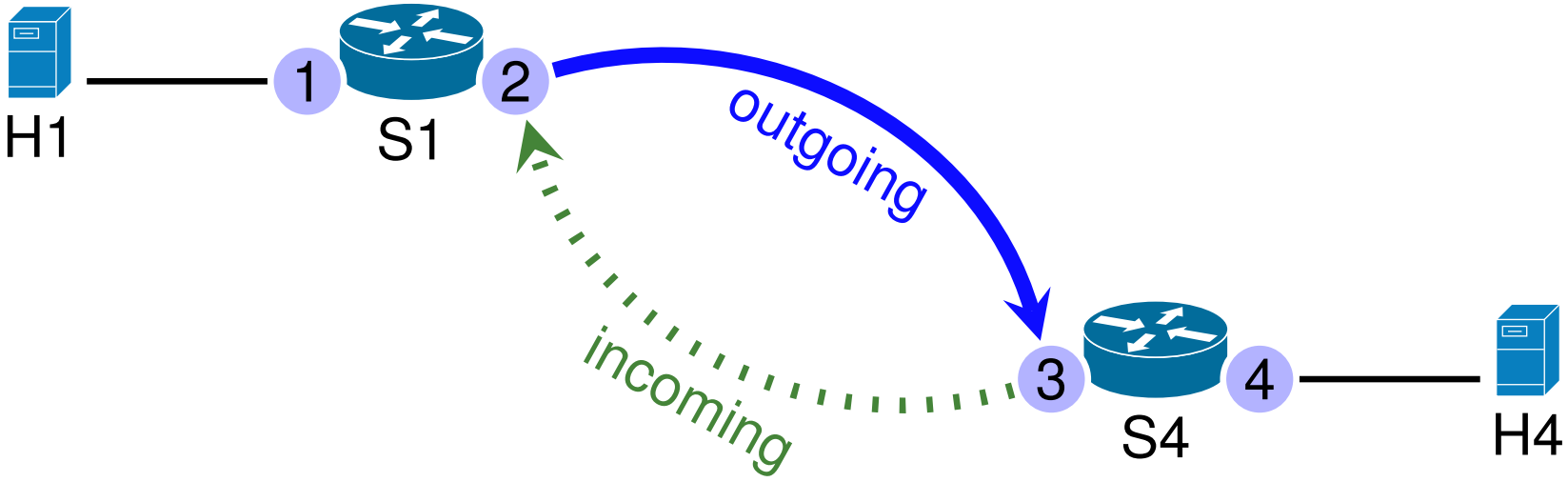
Programming with Events

- NetKAT language can specify a single network configuration
- We extend NetKAT to allow specifying *multiple* network configurations, and the event-driven updates between them

$$
\begin{array}{rcll}
f & \in & \mathit{Field} & \textit{(header-field name)} \\
n & \in & \mathbb{N} & \textit{(numeric value)} \\
a, b & ::= & \textbf{true} \mid \textbf{false} \mid f = n \mid \textbf{state}(n) = n \mid a \vee b \mid a \wedge b \mid \neg a & \textit{(test)} \\
p, q & ::= & a \mid f \leftarrow n \mid p + q \mid p \,;\, q \mid p* \mid (n{:}n) \rightarrowtriangle (n{:}n) & \textit{(command)} \\
 & \mid & \textbf{state}(n) \leftarrow n &
\end{array}
$$

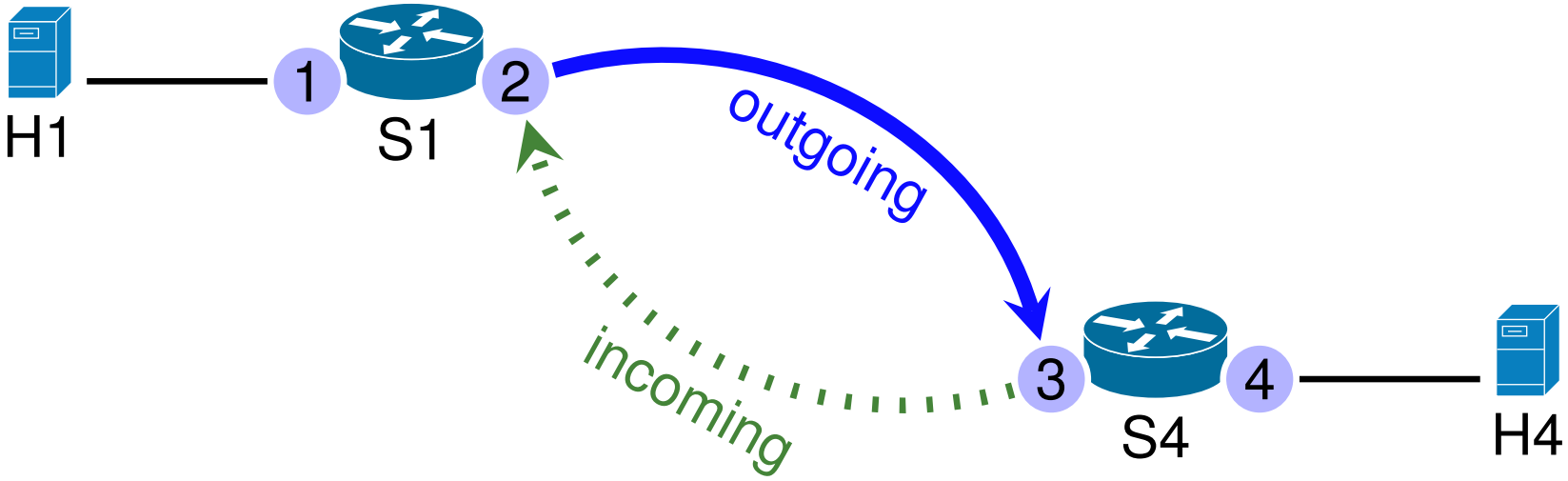
Programming Example

- Recall the Stateful Firewall Example:



Programming Example

- Recall the Stateful Firewall Example:

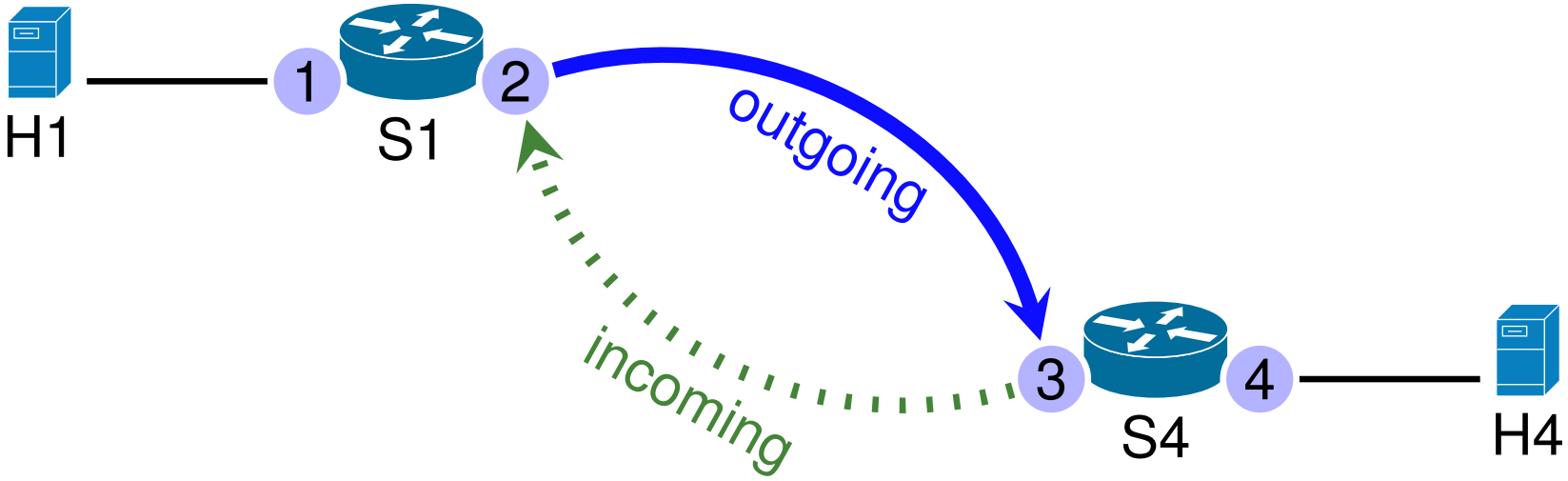


- We can write the program in Stateful NetKAT

```
dst=H4 ∧ pt=1; pt←2; (
    state=[0]; (S1:2)→(S4:3); state←[1]
  + state=[1]; (S1:2)→(S4:3)
); pt←4
+ dst=H1 ∧ pt=4 ∧ state=[1];
  pt←3; (S4:3)→(S1:2); pt←1
```

Programming Example

- Recall the Stateful Firewall Example:

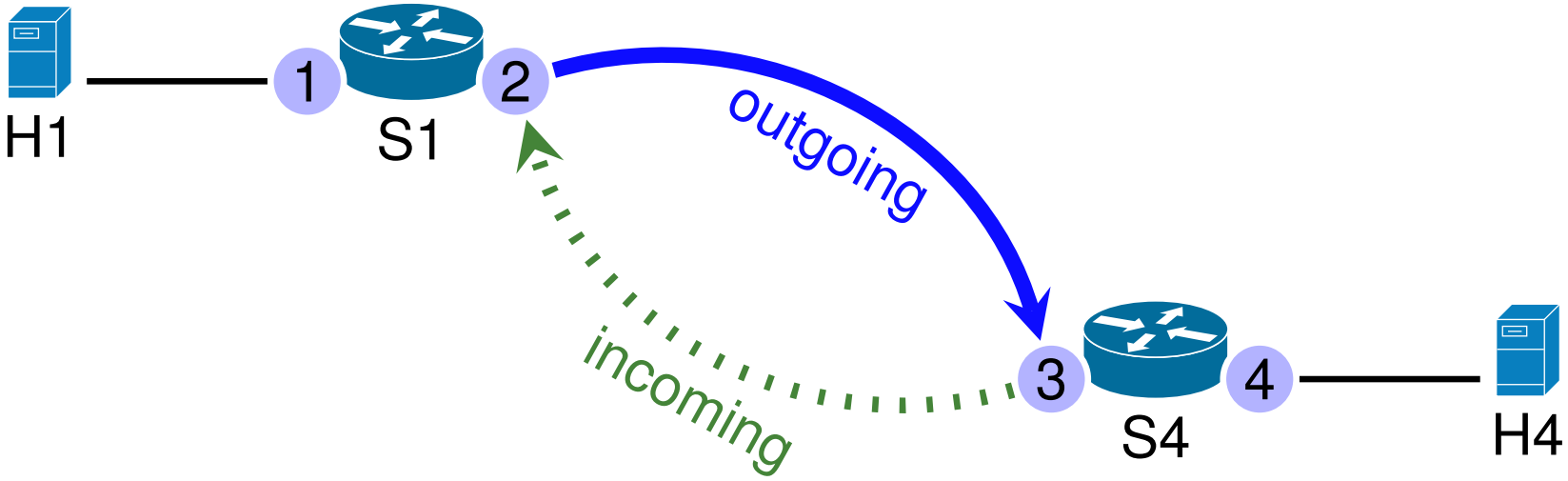


- We can write the program in Stateful NetKAT

```
dst=H4 ∧ pt=1; pt←2; (
    state=[0]; (S1:2)→(S4:3); state←[1]
  + state=[1]; (S1:2)→(S4:3)
); pt←4
+ dst=H1 ∧ pt=4 ∧ state=[1];
  pt←3; (S4:3)→(S1:2); pt←1
```

Programming Example

- Recall the Stateful Firewall Example:

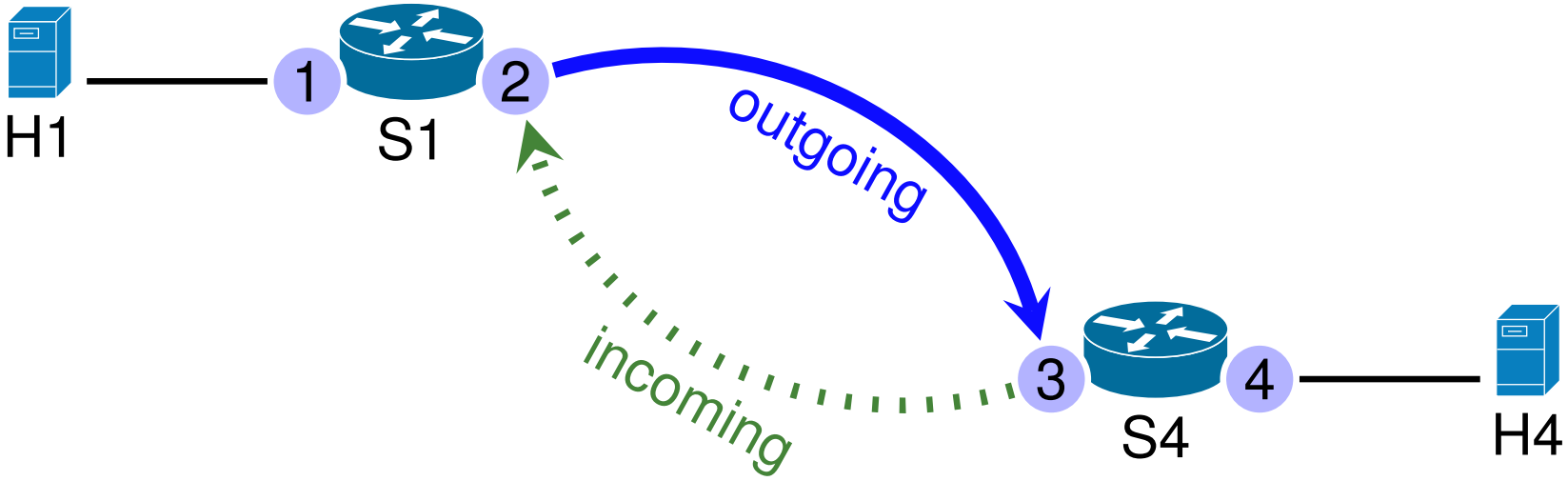


- We can write the program in Stateful NetKAT

```
dst=H4 ∧ pt=1; pt←2; (
    state=[0]; (S1:2)→(S4:3); state←[1]
  + state=[1]; (S1:2)→(S4:3)
); pt←4
+ dst=H1 ∧ pt=4 ∧ state=[1];
  pt←3; (S4:3)→(S1:2); pt←1
```

Programming Example

- Recall the Stateful Firewall Example:

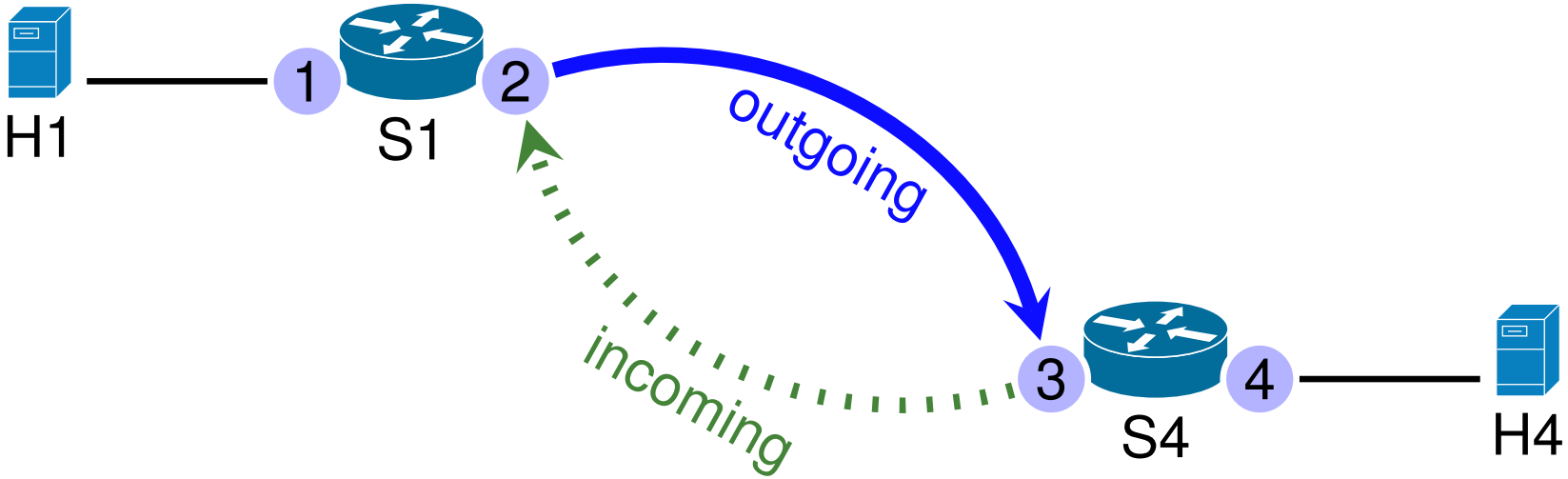


- We can write the program in Stateful NetKAT

```
dst=H4 ∧ pt=1; pt←2; (
    state=[0]; (S1:2)→(S4:3); state←[1]
  + state=[1]; (S1:2)→(S4:3)
); pt←4
+ dst=H1 ∧ pt=4 ∧ state=[1];
  pt←3; (S4:3)→(S1:2); pt←1
```

Programming Example

- Recall the Stateful Firewall Example:

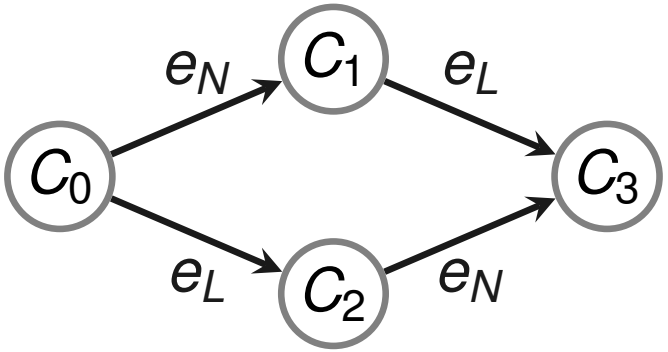


- We can write the program in Stateful NetKAT

```
dst=H4 ∧ pt=1; pt←2; (
    state=[0]; (S1:2)→(S4:3); state←[1]
  + state=[1]; (S1:2)→(S4:3)
); pt←4
+ dst=H1 ∧ pt=4 ∧ state=[1];
  pt←3; (S4:3)→(S1:2); pt←1
```

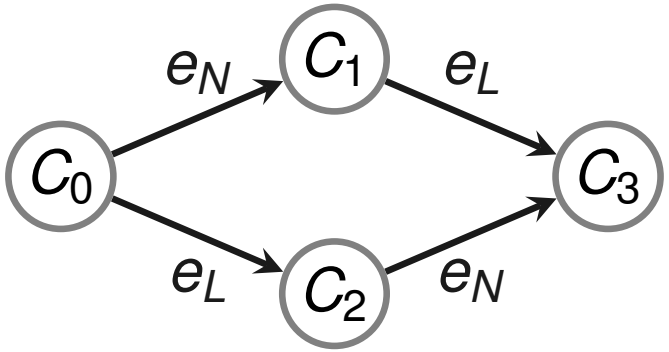
Network Event Structures

- We can define an event-driven program's semantics using a simple transition system:



Network Event Structures

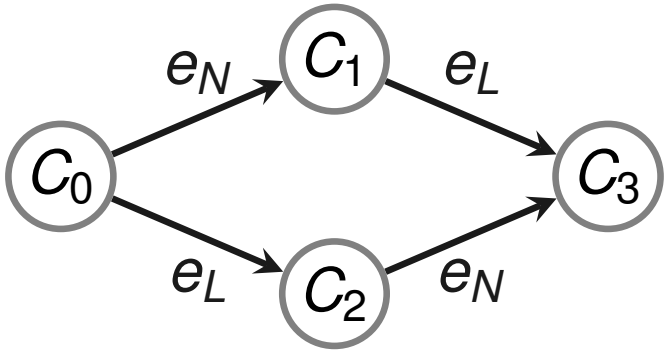
- We can define an event-driven program's semantics using a simple transition system:



  - Each node is a network configuration C

Network Event Structures

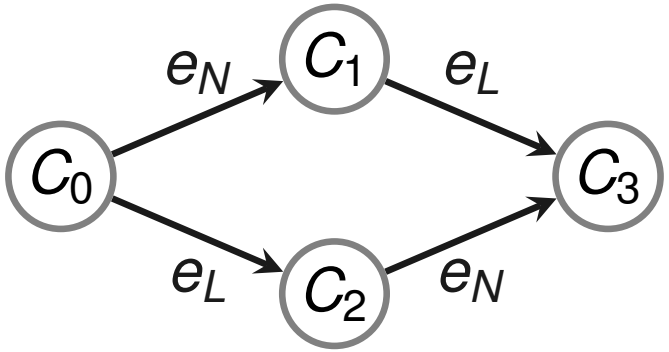
- We can define an event-driven program's semantics using a simple transition system:



  - Each node is a network configuration C
  - Each edge denotes an event-driven consistent update $C \xrightarrow{e} C'$

Network Event Structures

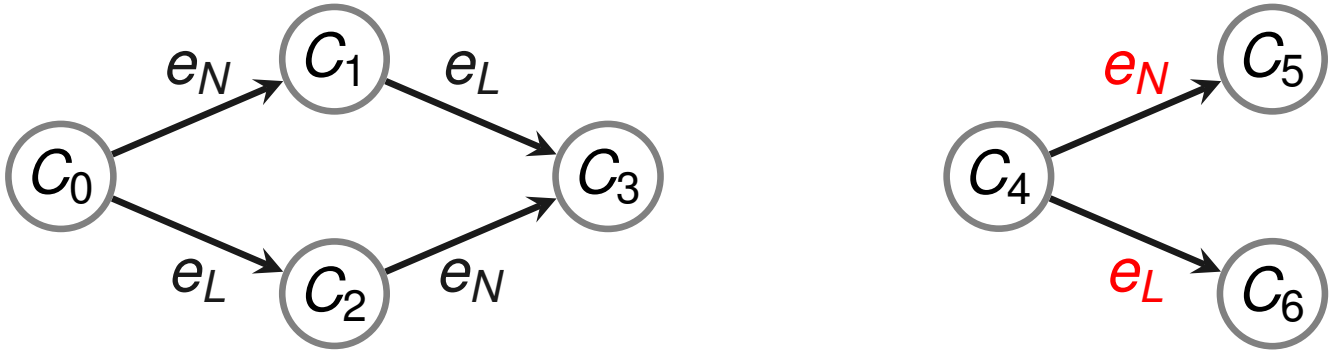
- We can define an event-driven program's semantics using a simple transition system:



  - Each node is a network configuration C
  - Each edge denotes an event-driven consistent update $C \xrightarrow{e} C'$
- What if e_N is detected in New York and e_L is detected in London?

Network Event Structures

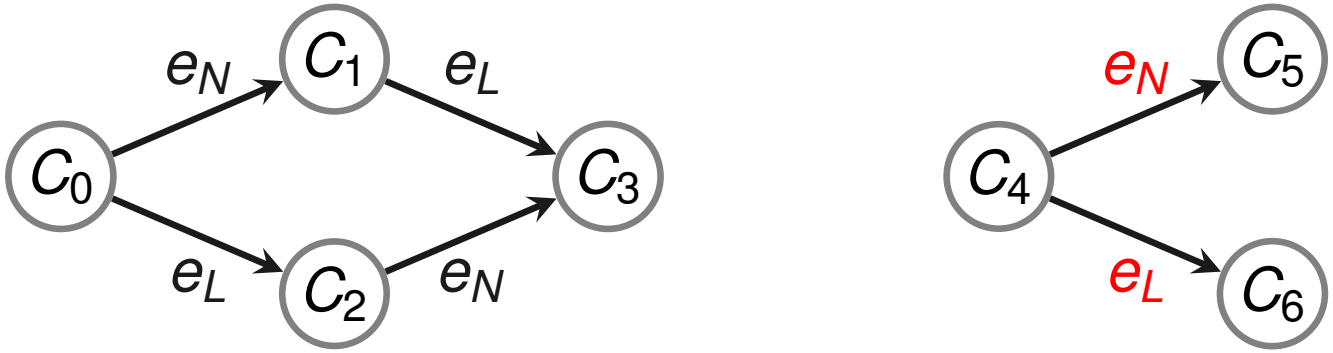
- We can define an event-driven program's semantics using a simple transition system:



  - Each node is a network configuration C
  - Each edge denotes an event-driven consistent update $C \xrightarrow{e} C'$
- What if e_N is detected in New York and e_L is detected in London?

Network Event Structures

- We can define an event-driven program's semantics using a simple transition system:



  - Each node is a network configuration C
  - Each edge denotes an event-driven consistent update $C \xrightarrow{e} C'$
- What if e_N is detected in New York and e_L is detected in London?
- Problem: different switches could have different views of which events have occurred (conflicts!)

Network Event Structures

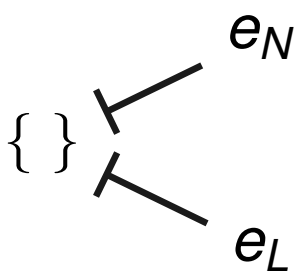
- Solution: impose additional structure on the transition system

Network Event Structures

- Solution: impose additional structure on the transition system
- *Network Event Structure (NES)*—constrain how an event *e* can be related to other events:

Network Event Structures

- Solution: impose additional structure on the transition system
- *Network Event Structure (NES)*—constrain how an event e can be related to other events:
  1. Causal dependency: e' can happen after $\{e_1, e_2, \cdots, e_k\}$



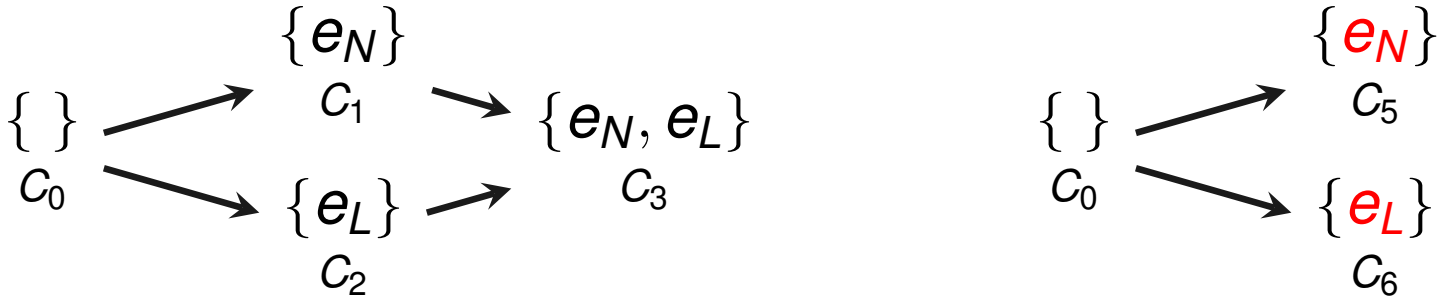
Network Event Structures

- Solution: impose additional structure on the transition system
- *Network Event Structure (NES)*—constrain how an event e can be related to other events:
  1. Causal dependency: e' can happen after $\{e_1, e_2, \cdots, e_k\}$
  2. Incompatibility: e and e' cannot happen in the same execution

Network Event Structures

- Solution: impose additional structure on the transition system
- *Network Event Structure (NES)*—constrain how an event e can be related to other events:
  1. Causal dependency: e' can happen after $\{e_1, e_2, \cdots, e_k\}$
  2. Incompatibility: e and e' cannot happen in the same execution

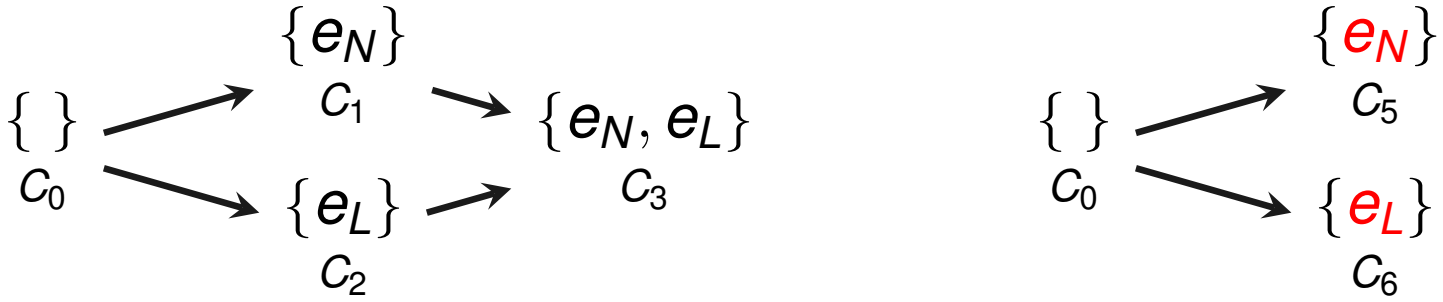
- Note: the *event-sets* allowed by an NES form a transition system



Network Event Structures

- Solution: impose additional structure on the transition system
- *Network Event Structure (NES)*—constrain how an event e can be related to other events:
  1. Causal dependency: e' can happen after $\{e_1, e_2, \cdots, e_k\}$
  2. Incompatibility: e and e' cannot happen in the same execution

- Note: the *event-sets* allowed by an NES form a transition system



- **Locality condition:** we require incompatible events to occur at the same switch

Importance of Locality

- Programs that don't satisfy locality condition require either

Importance of Locality

- Programs that don't satisfy locality condition require either
  1. buffering to enable "instantaneous" action at a distance, or

Importance of Locality

- Programs that don't satisfy locality condition require either
  1. buffering to enable "instantaneous" action at a distance, or
  2. changing state in a way that could result in future conflict

Importance of Locality

- Programs that don't satisfy locality condition require either
  1. buffering to enable "instantaneous" action at a distance, or
  2. changing state in a way that could result in future conflict
- Programs that **do** satisfy the locality condition can be implemented efficiently, and without conflicts

Importance of Locality

- Programs that don't satisfy locality condition require either
  1. buffering to enable "instantaneous" action at a distance, or
  2. changing state in a way that could result in future conflict
- Programs that **do** satisfy the locality condition can be implemented efficiently, and without conflicts

Lemma

In general, can't correctly implement an NES that does not satisfy the locality condition, unless availability can be sacrificed

Compilation

- The compiler produces an NES from a Stateful NetKAT program:

Compilation

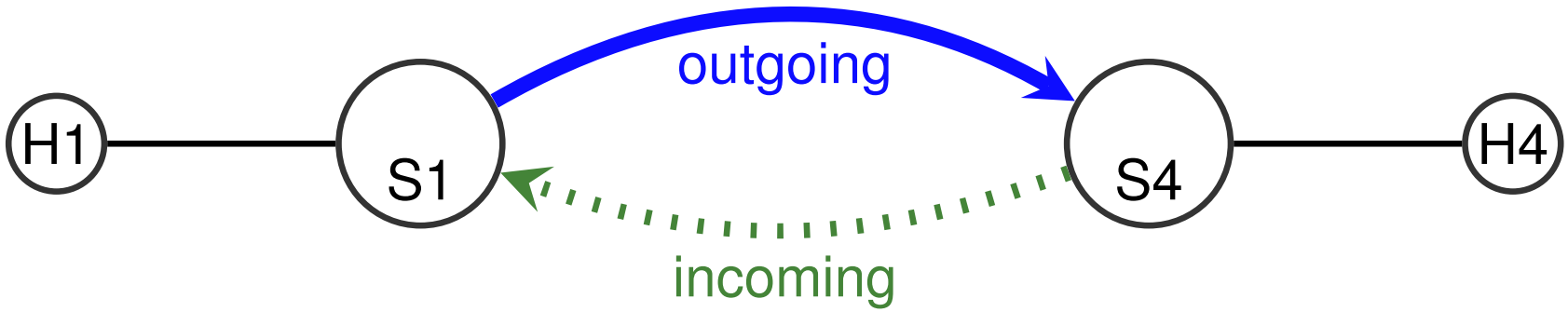
- The compiler produces an NES from a Stateful NetKAT program:
- The **state**$(n) = n$ tests let us produce the static configurations C

Compilation

- The compiler produces an NES from a Stateful NetKAT program:
- The **state**$(n) = n$ tests let us produce the static configurations C
- The **state**$(n) \leftarrow n$ assignments let us produce the events e

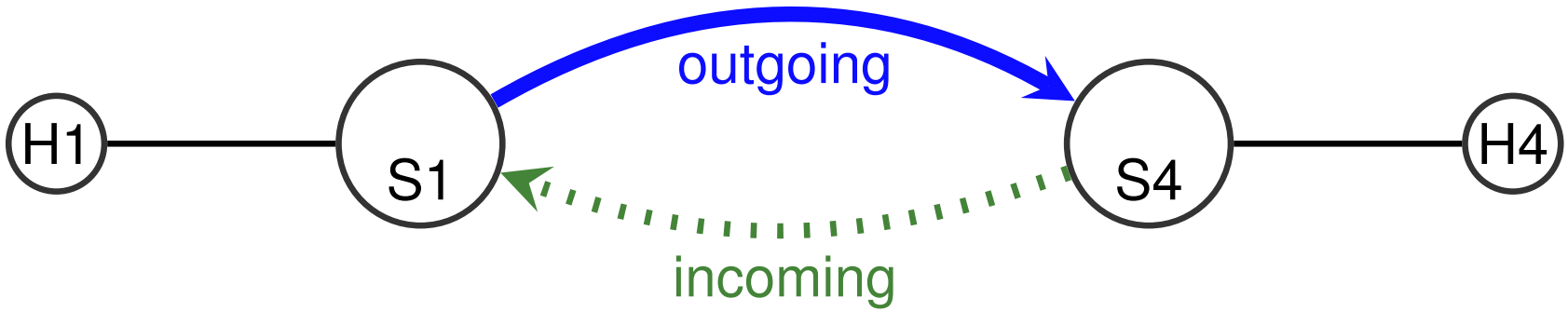
Correctly Implementing Event-Driven Behavior

- Let's "run" the Stateful Firewall example:



Correctly Implementing Event-Driven Behavior

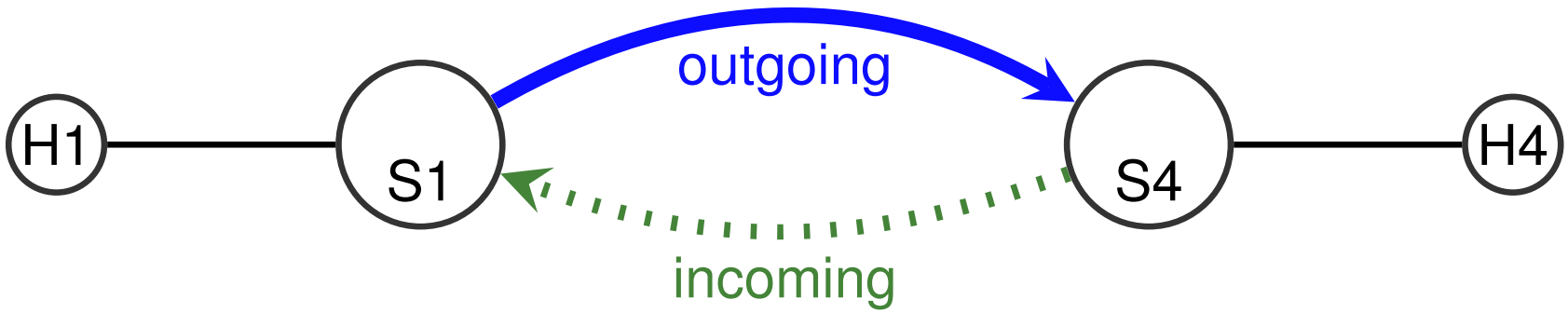
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*

Correctly Implementing Event-Driven Behavior

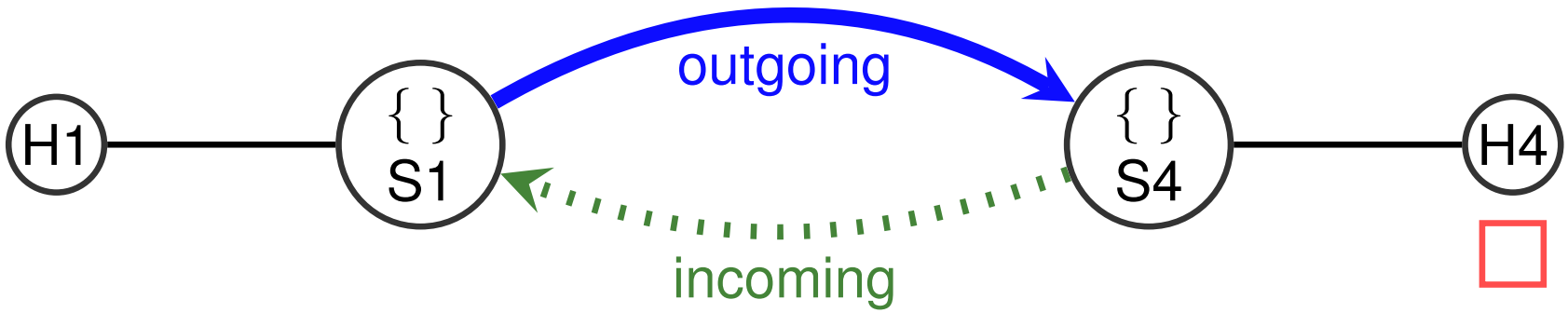
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

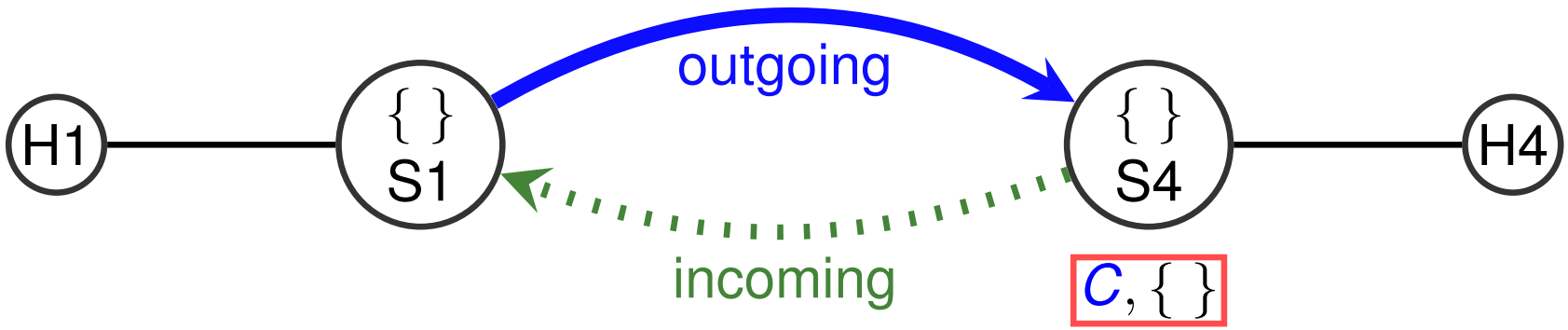
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

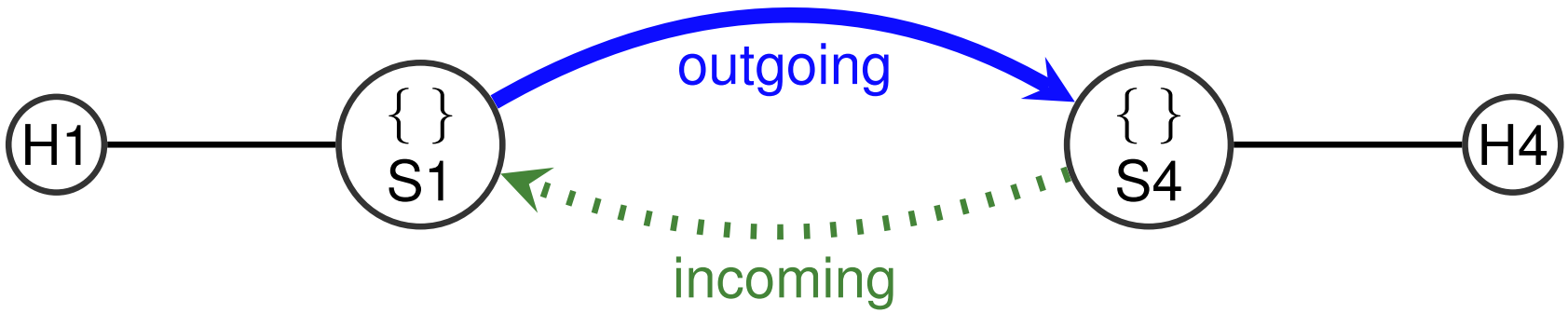
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

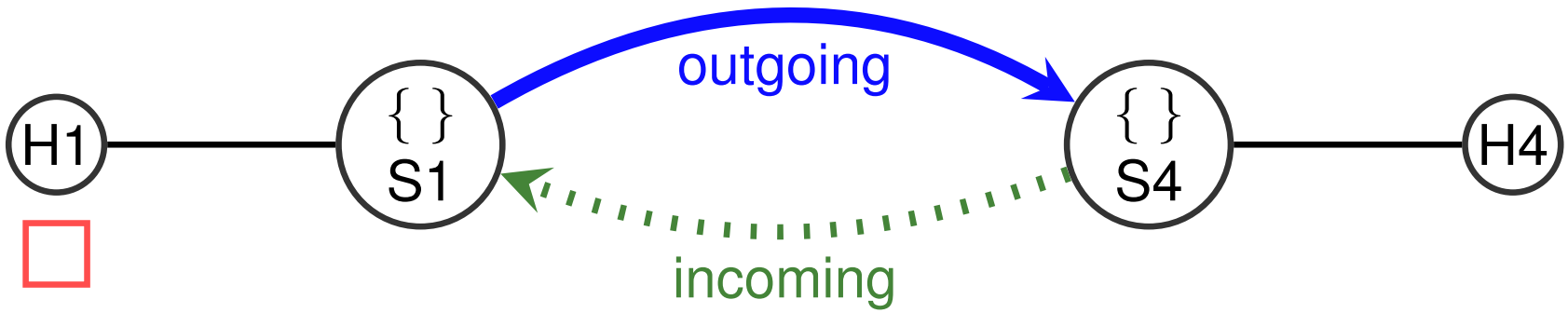
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

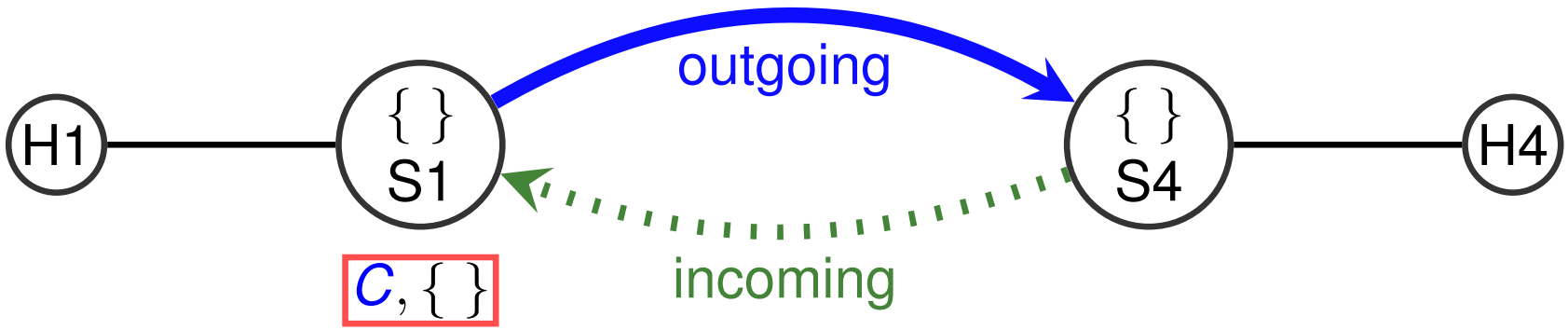
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

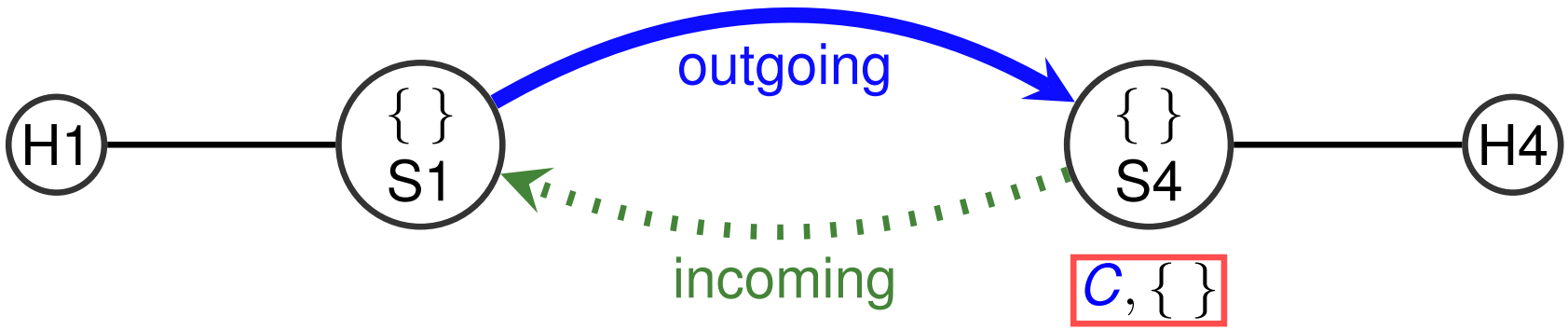
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

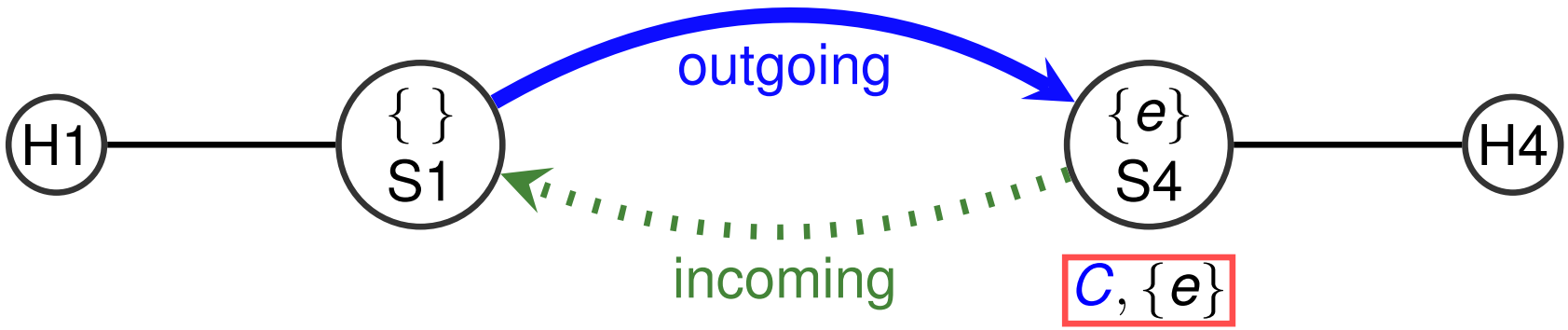
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

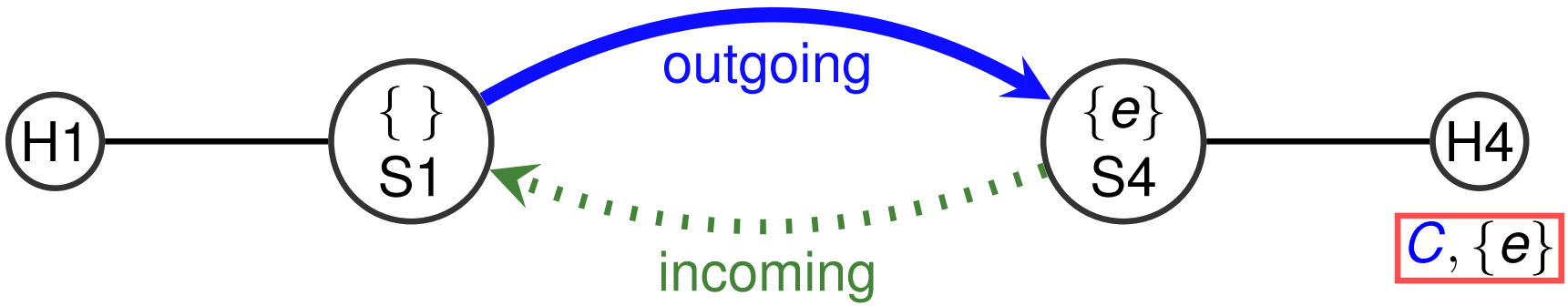
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

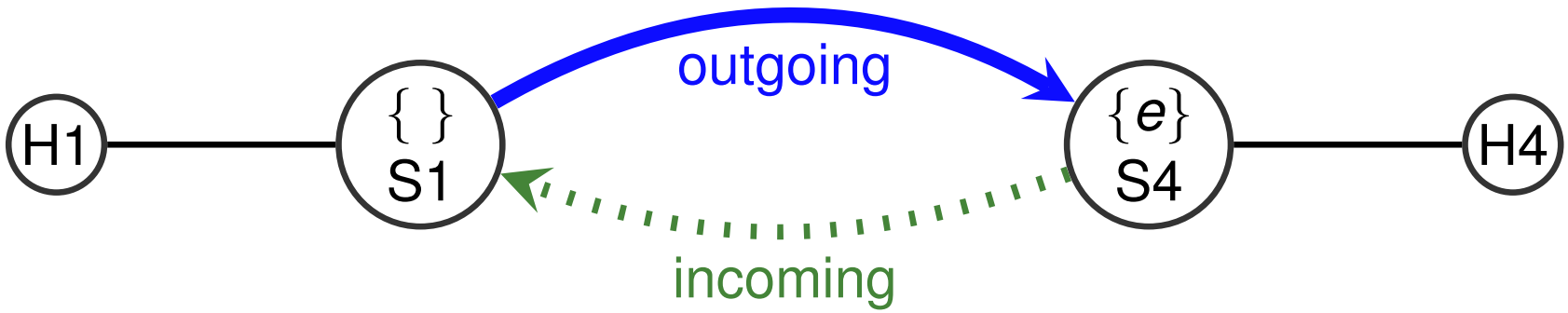
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

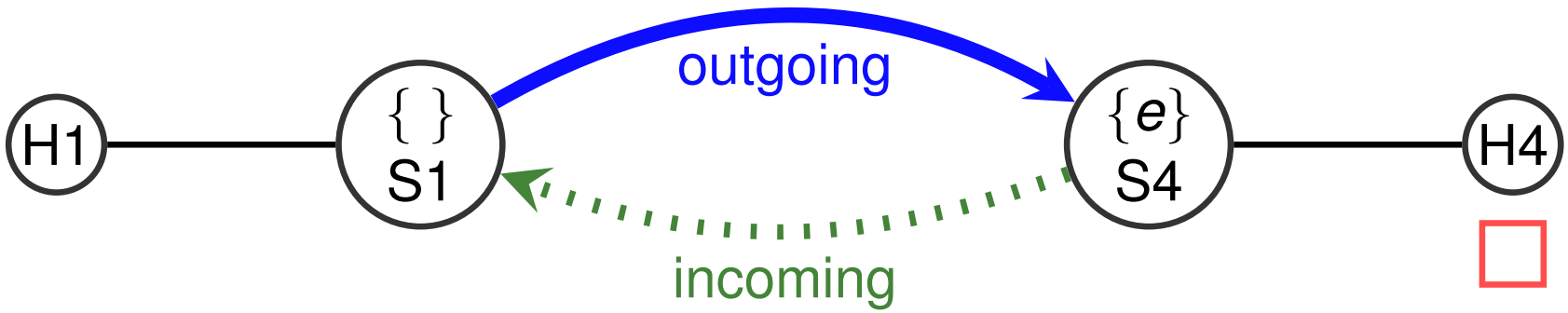
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

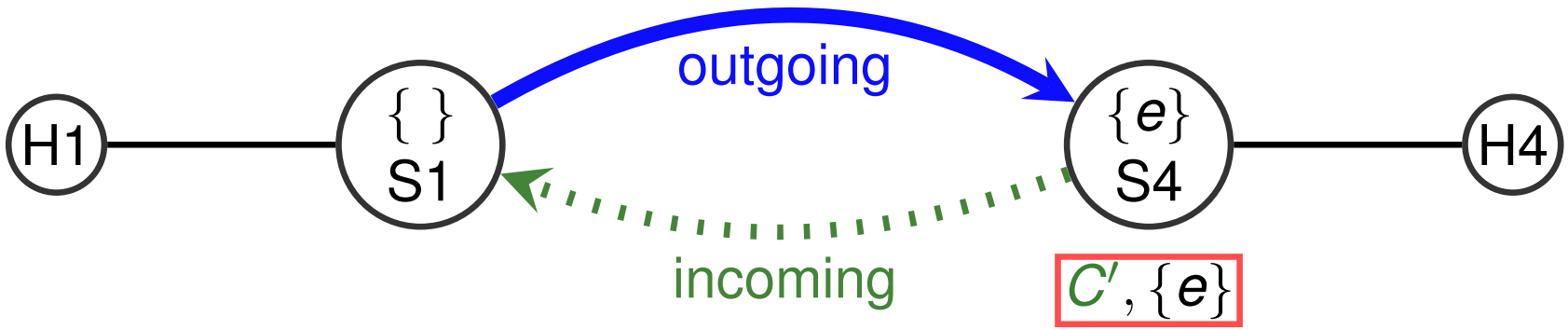
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

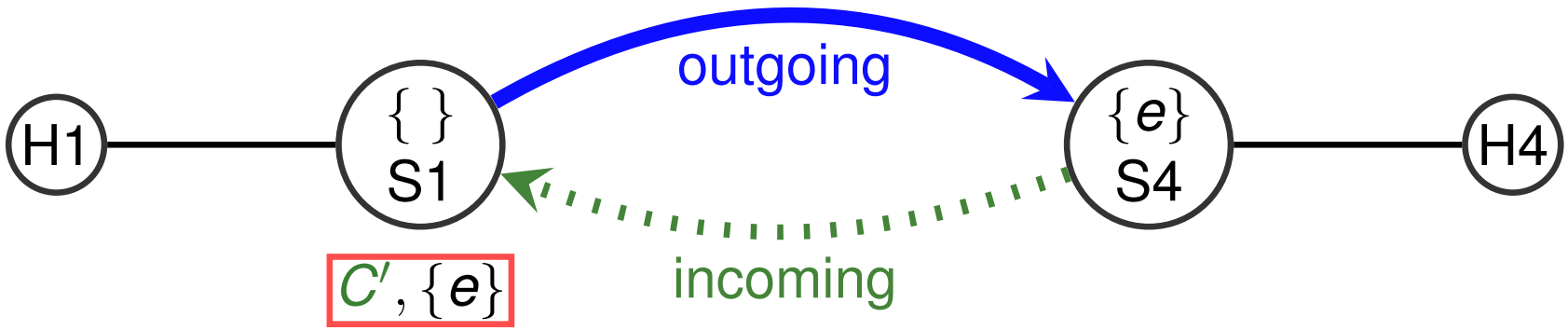
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

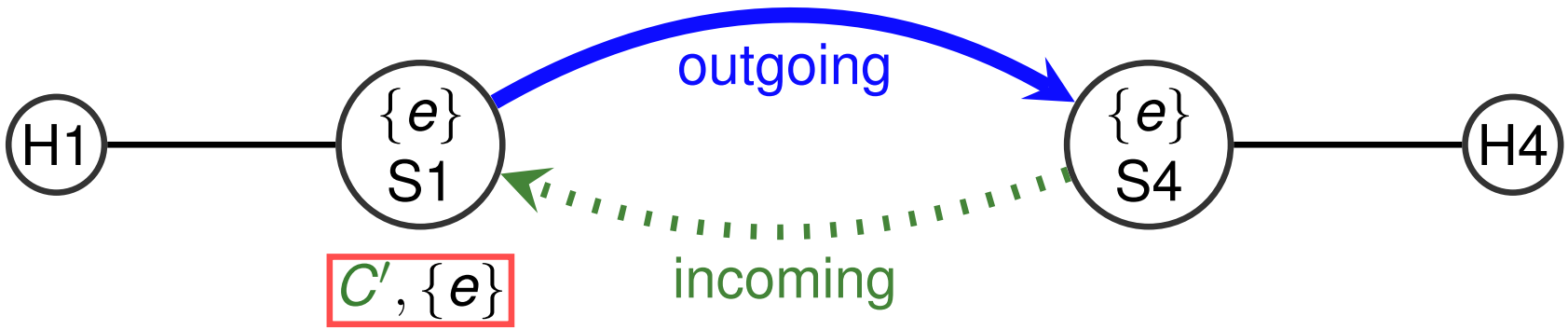
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

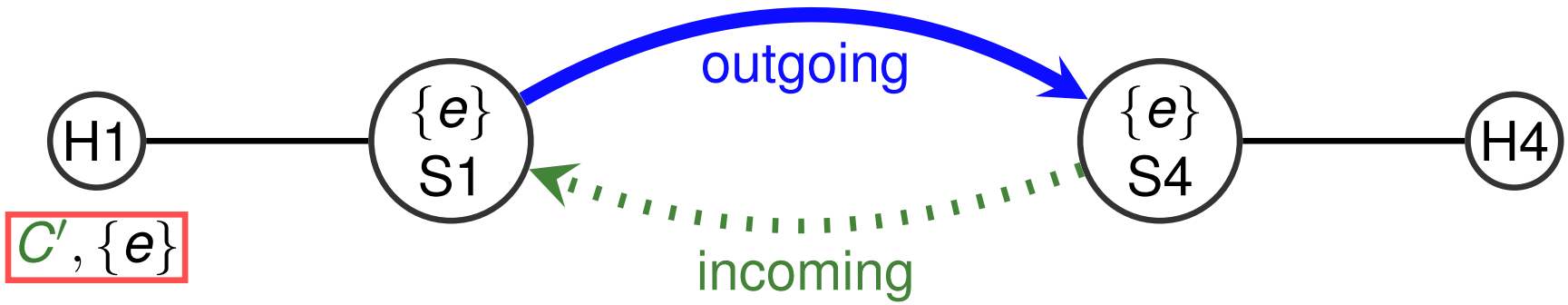
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

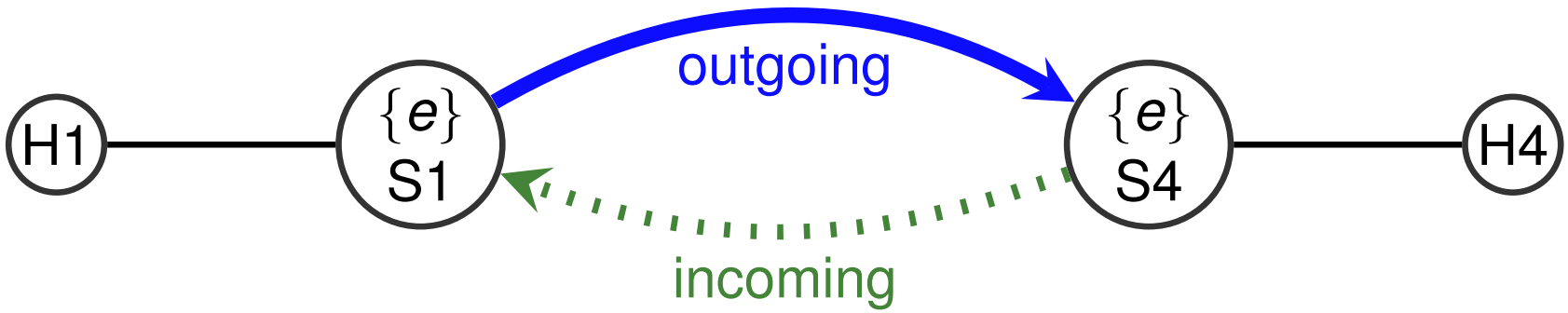
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

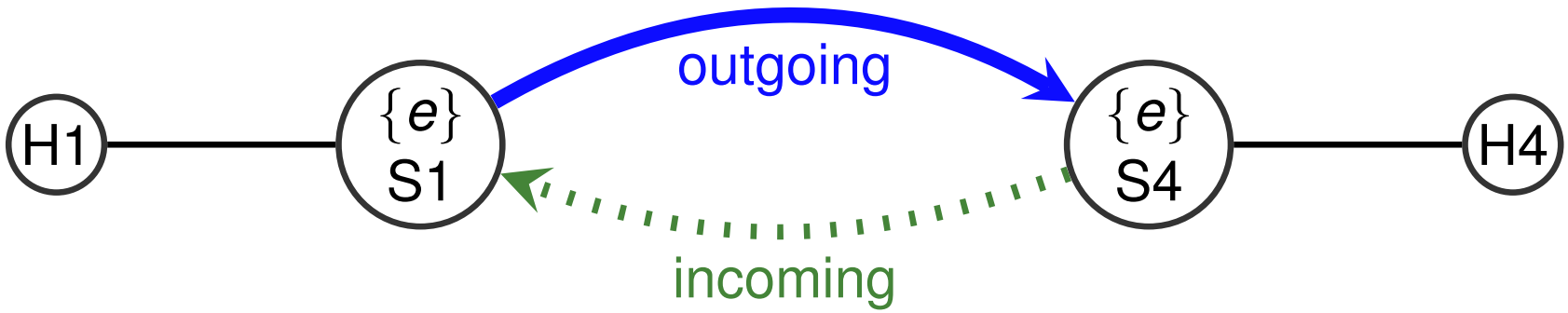
- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Correctly Implementing Event-Driven Behavior

- Let's "run" the Stateful Firewall example:



- NES is simply $\{\ \} \vdash e$, where e is arrival of outgoing packet at *S4*
- There are two event-sets:
  1. $\{\ \}$ maps to configuration C (only outgoing)
  2. $\{e\}$ maps to configuration C' (both outgoing and incoming)

Theorem (Correctness)

Implementing an event-driven program in this way guarantees:

- *event-triggered consistent update*
- *event structure consistency*

Implementation and Experimental Setup

- We built the Stateful NetKAT compiler, which produces NESs

Implementation and Experimental Setup

- We built the Stateful NetKAT compiler, which produces NESs
- We customized the OpenFlow implementation with needed functionality (stateful switches, etc.)

Implementation and Experimental Setup

- We built the Stateful NetKAT compiler, which produces NESs
- We customized the OpenFlow implementation with needed functionality (stateful switches, etc.)
- We wrote scripts to deploy and run our NESs in Mininet

Experimental Results—Performance

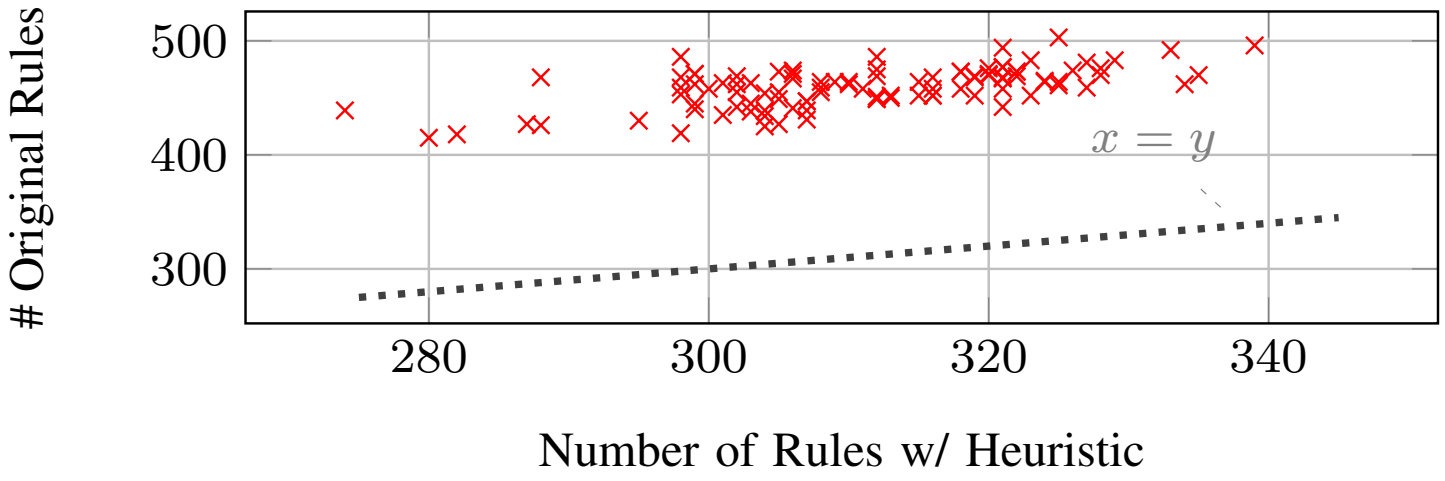
- Recall that in the running example, we needed to know *all* configurations at each switch

Experimental Results—Performance

- Recall that in the running example, we needed to know *all* configurations at each switch
  - We use a rule-reduction heuristic

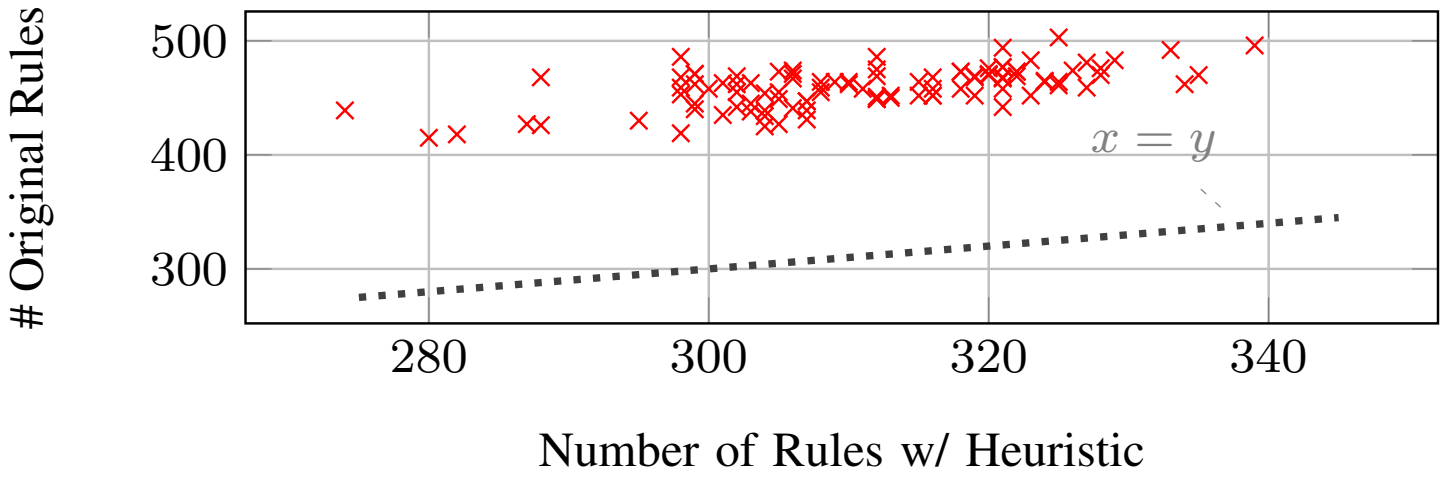
Experimental Results—Performance

- Recall that in the running example, we needed to know *all* configurations at each switch
  - We use a rule-reduction heuristic
  - Reduced number of total rules by about 32%



Experimental Results—Performance

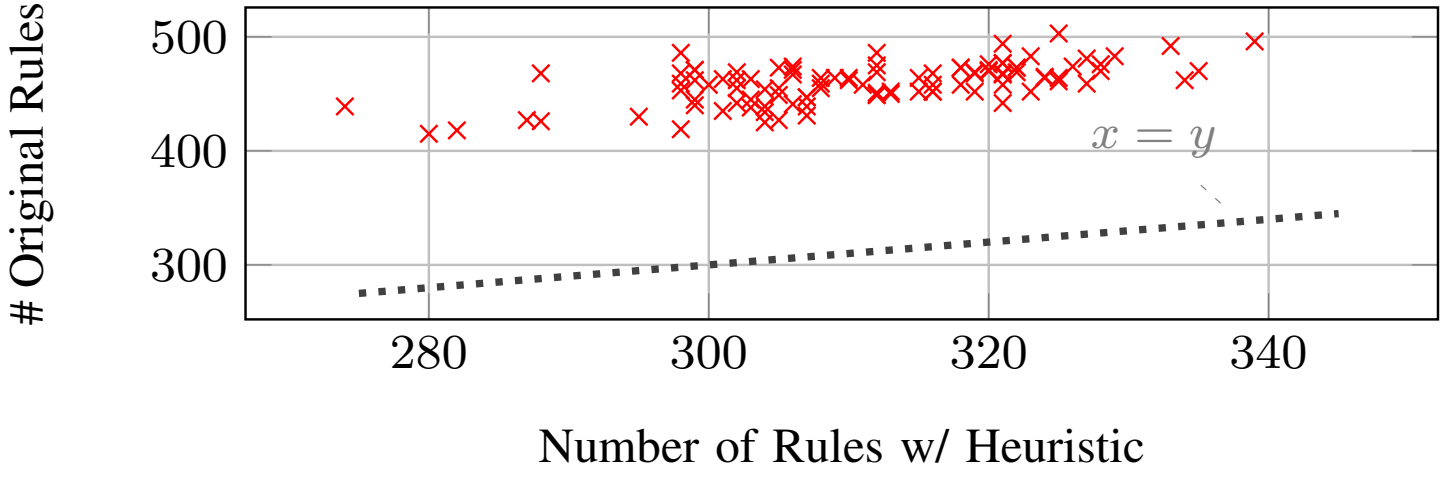
- Recall that in the running example, we needed to know *all* configurations at each switch
  - We use a rule-reduction heuristic
  - Reduced number of total rules by about 32%



- Performance of customized OpenFlow implementation

Experimental Results—Performance

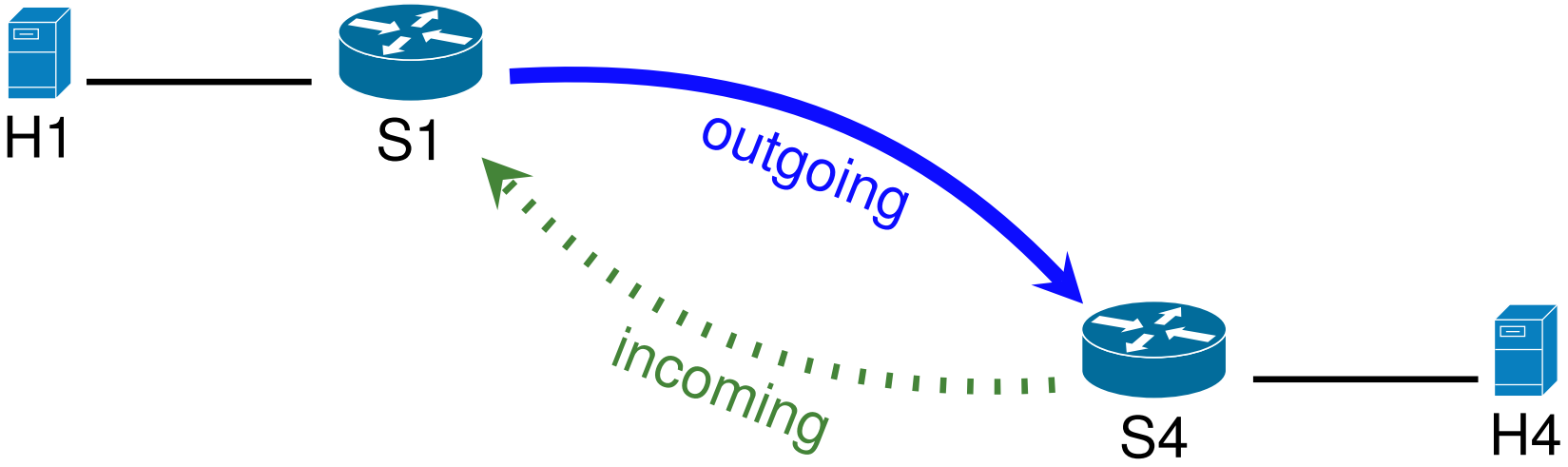
- Recall that in the running example, we needed to know *all* configurations at each switch
  - We use a rule-reduction heuristic
  - Reduced number of total rules by about 32%



- Performance of customized OpenFlow implementation
  - Within 6% of reference implementation

Case Studies—Correctness

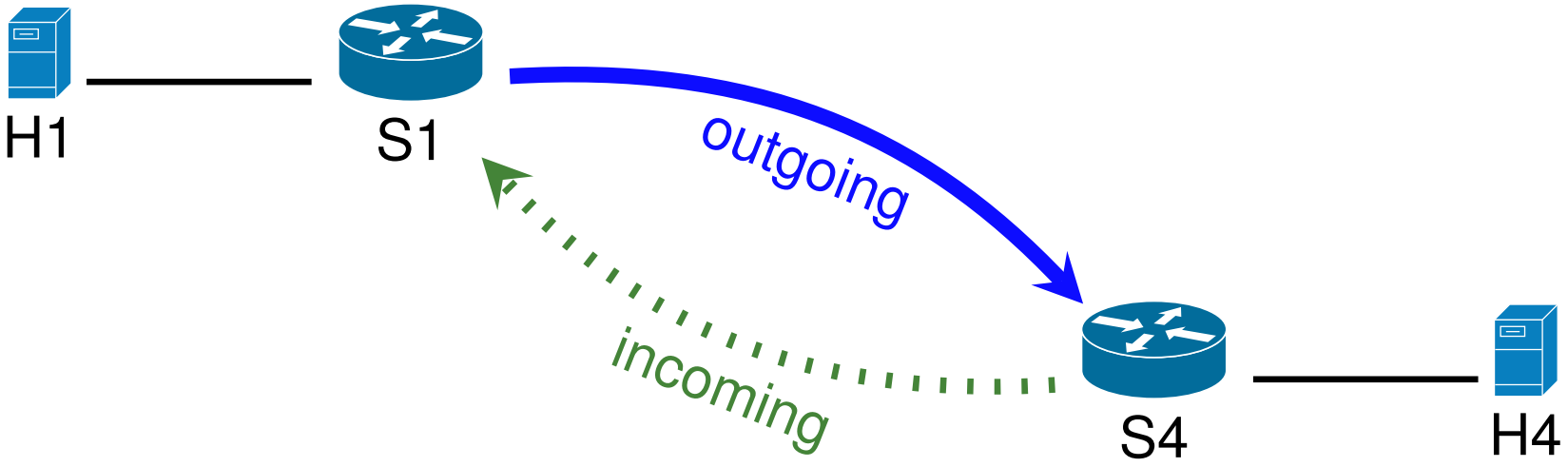
- Compared correct implementation vs. “uncoordinated” updates



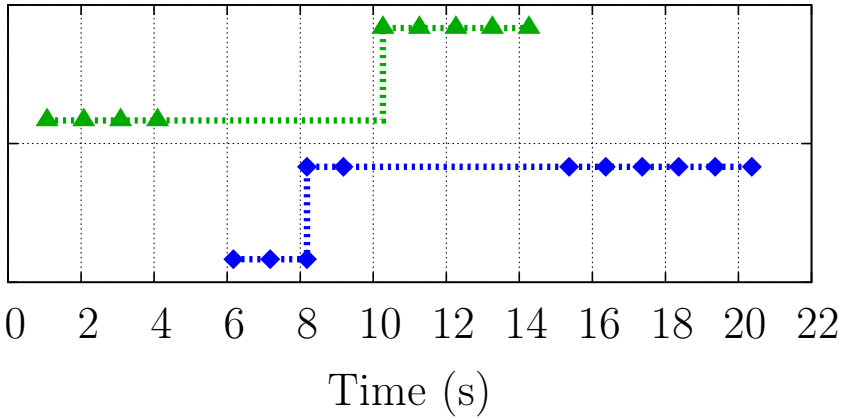
Stateful Firewall—Topology

Case Studies—Correctness

- Compared correct implementation vs. “uncoordinated” updates



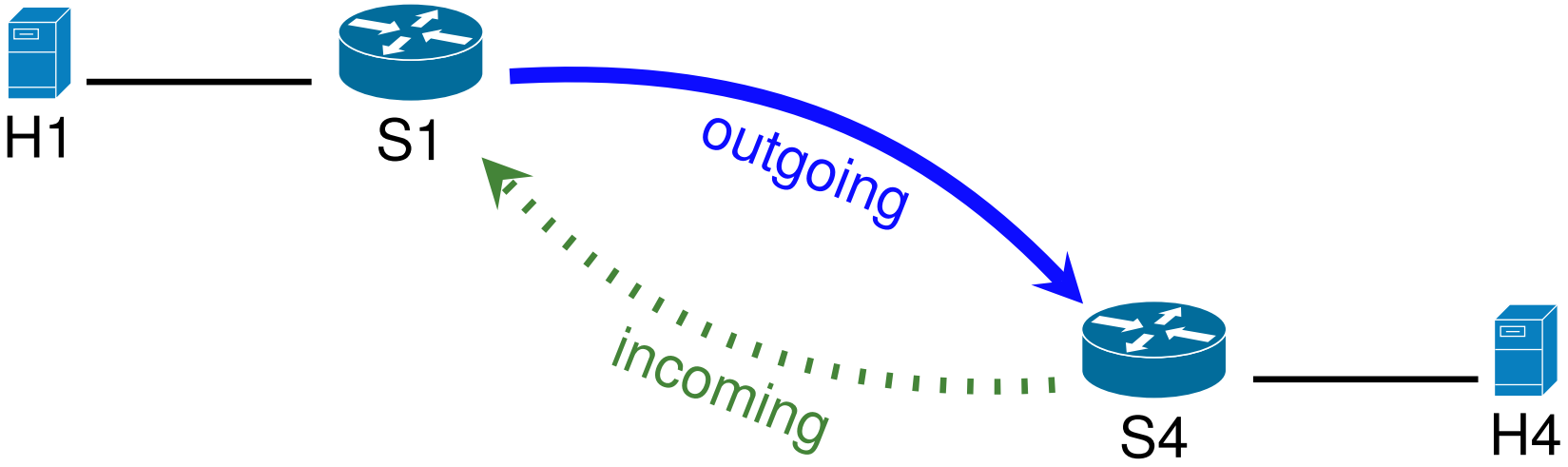
Stateful Firewall—Topology



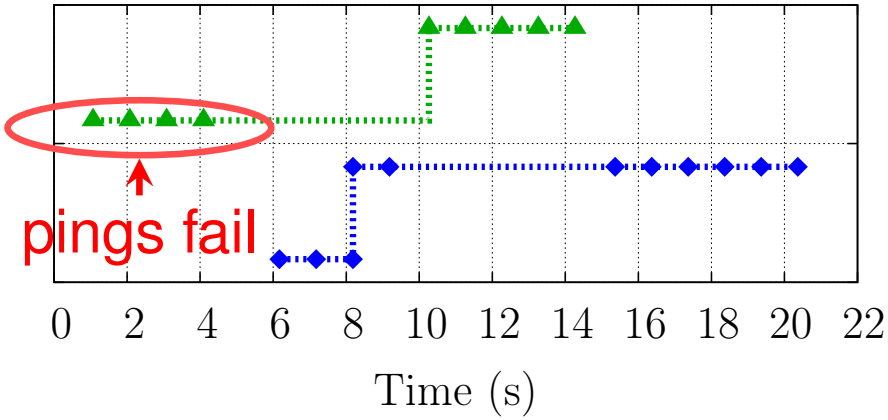
Incorrect behavior

Case Studies—Correctness

- Compared correct implementation vs. “uncoordinated” updates



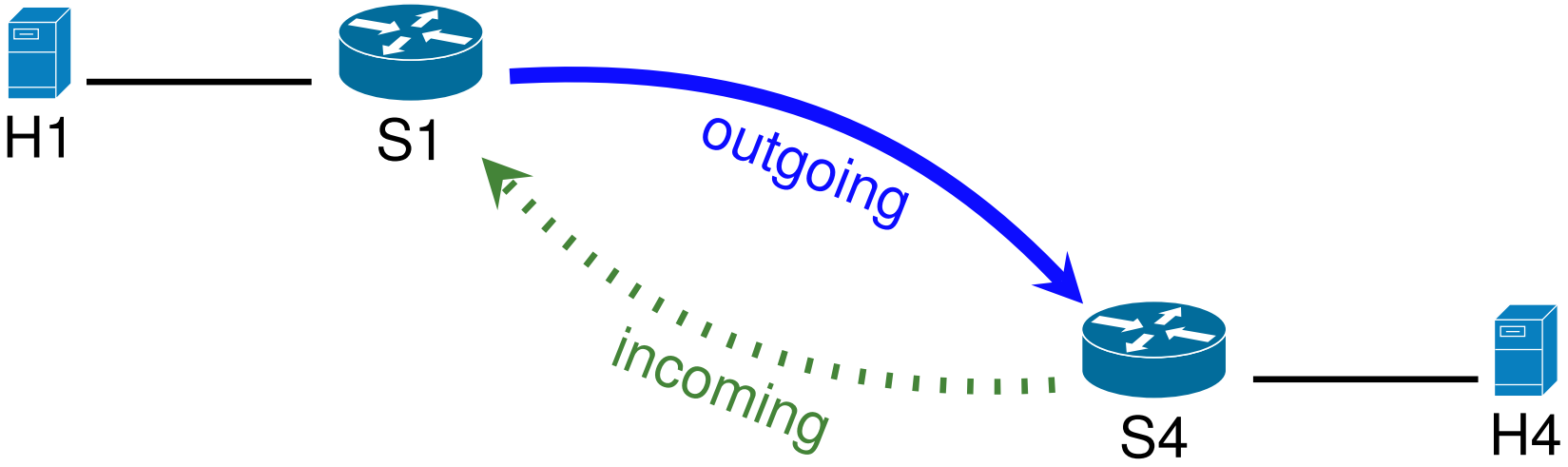
Stateful Firewall—Topology



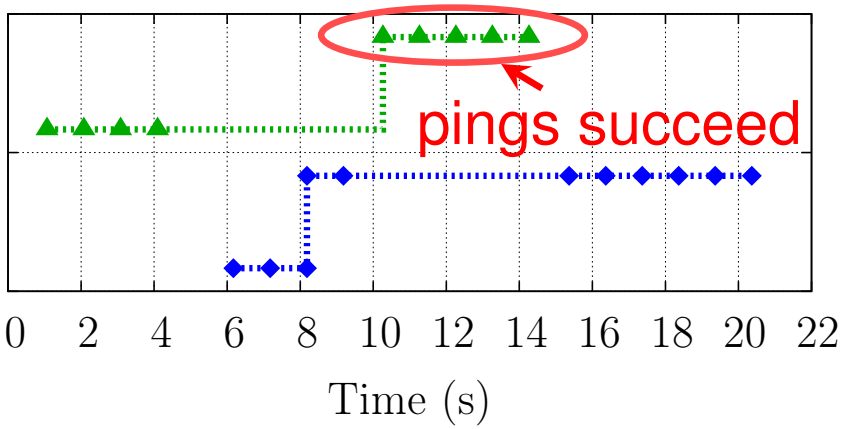
Incorrect behavior

Case Studies—Correctness

- Compared correct implementation vs. “uncoordinated” updates



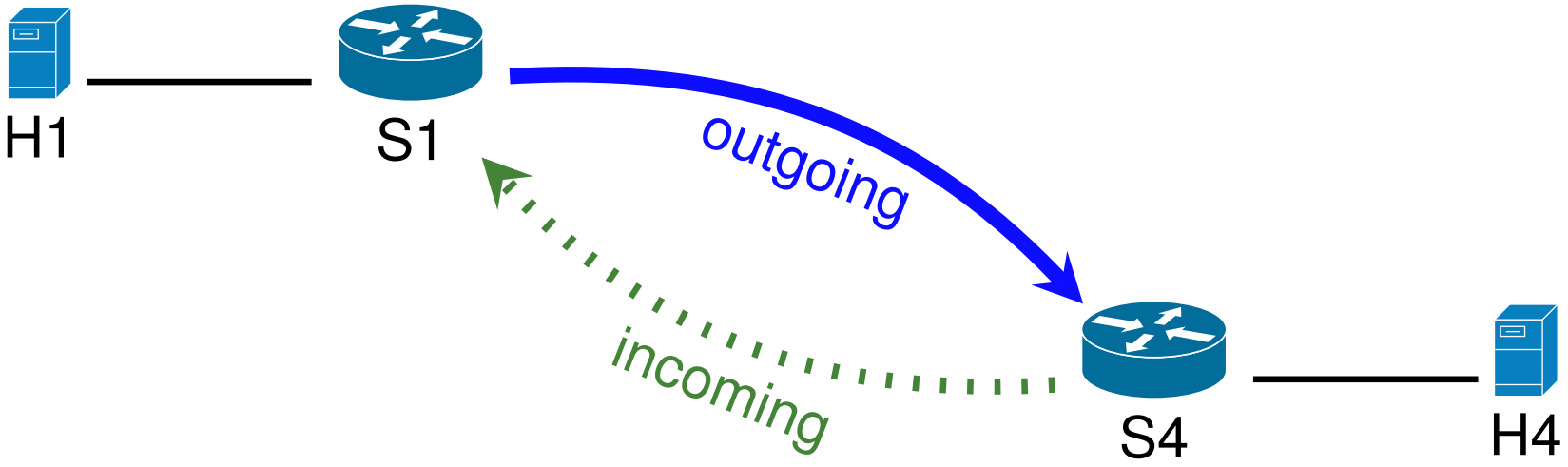
Stateful Firewall—Topology



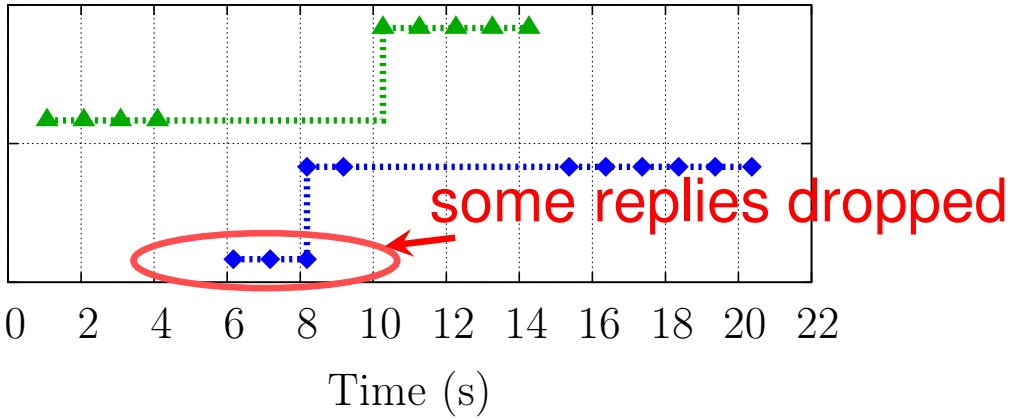
Incorrect behavior

Case Studies—Correctness

- Compared correct implementation vs. “uncoordinated” updates



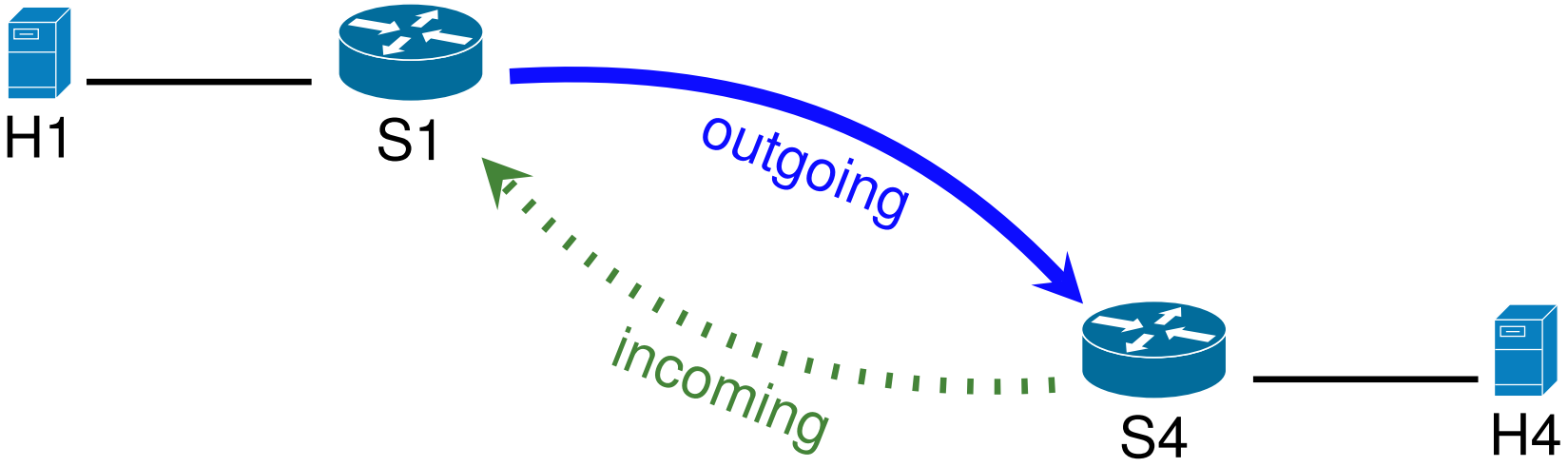
Stateful Firewall—Topology



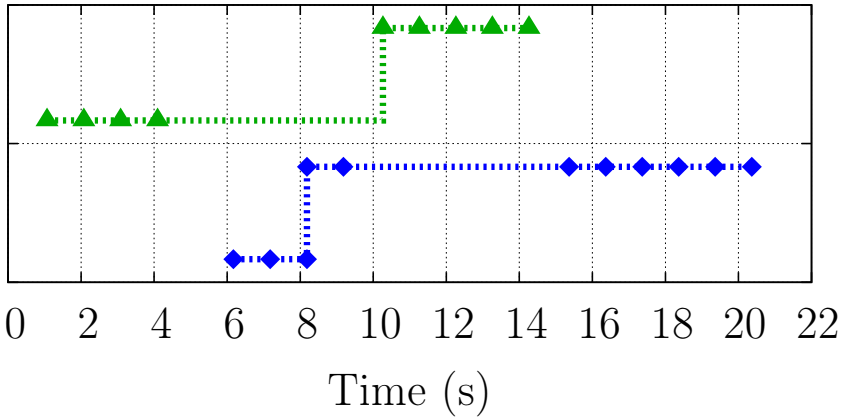
Incorrect behavior

Case Studies—Correctness

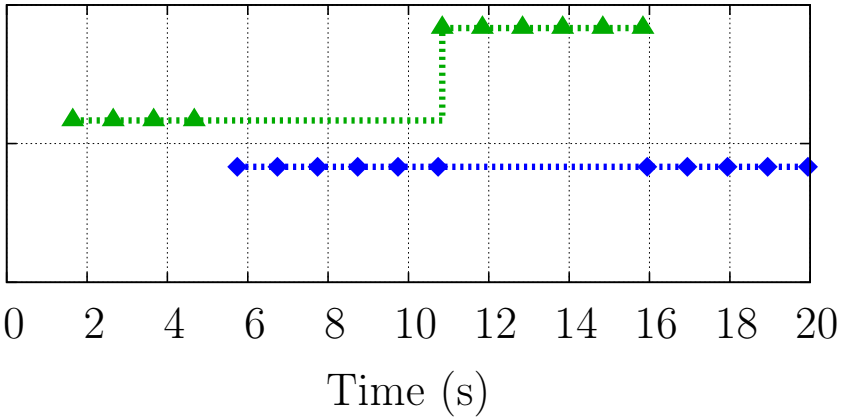
- Compared correct implementation vs. “uncoordinated” updates



Stateful Firewall—Topology



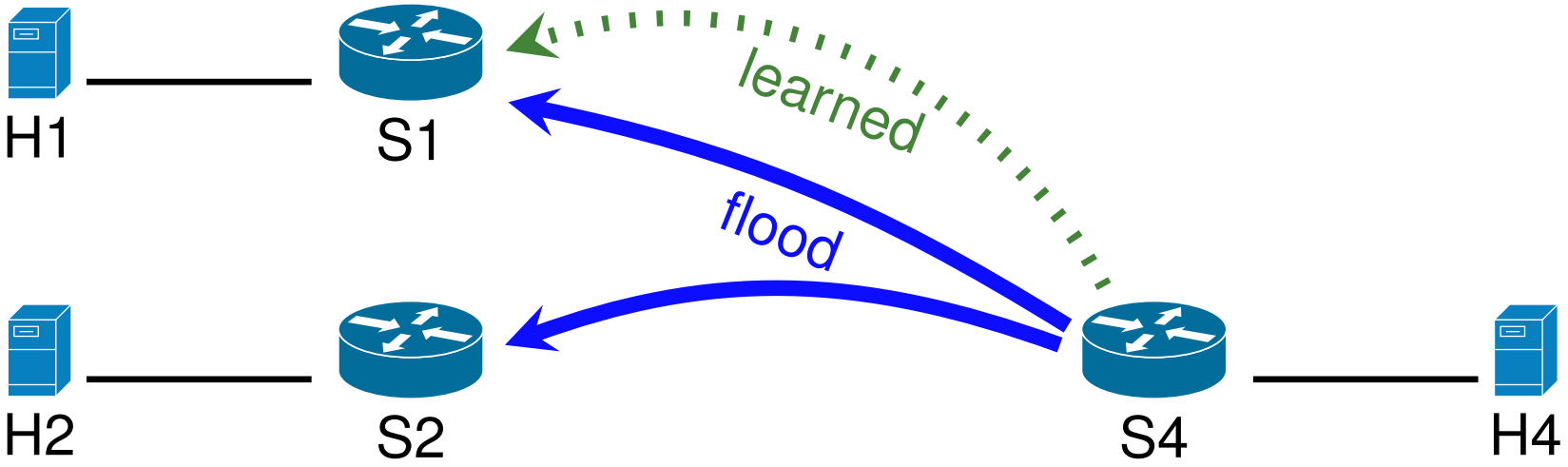
Incorrect behavior



Correct behavior

Case Studies—Correctness

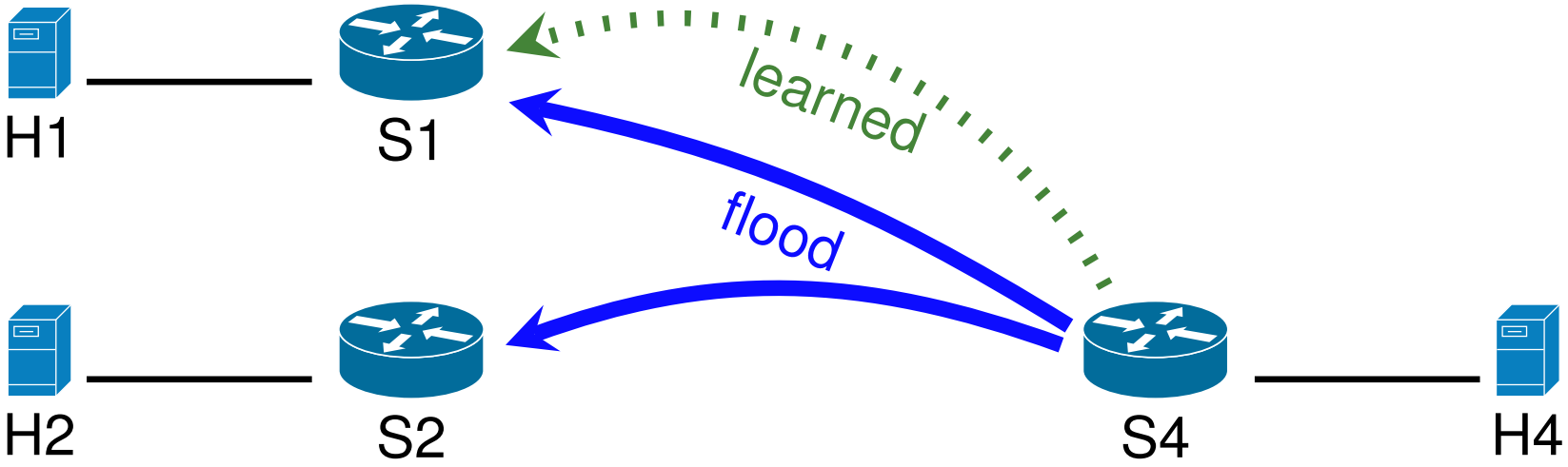
- We also built several other real-world examples:



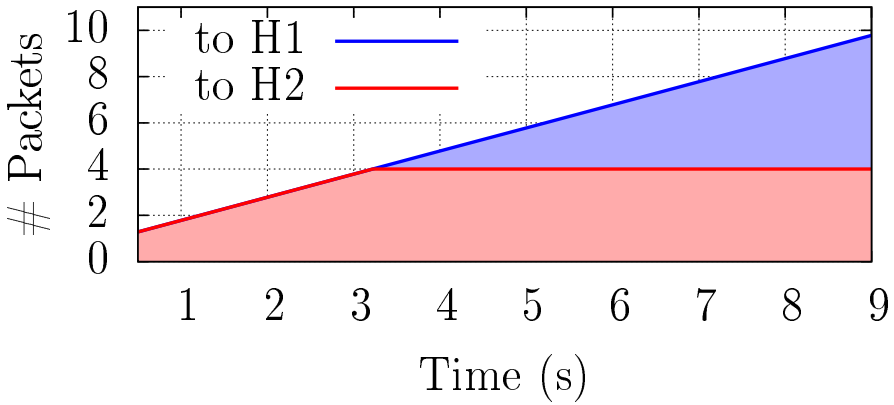
Learning Switch—Topology

Case Studies—Correctness

- We also built several other real-world examples:



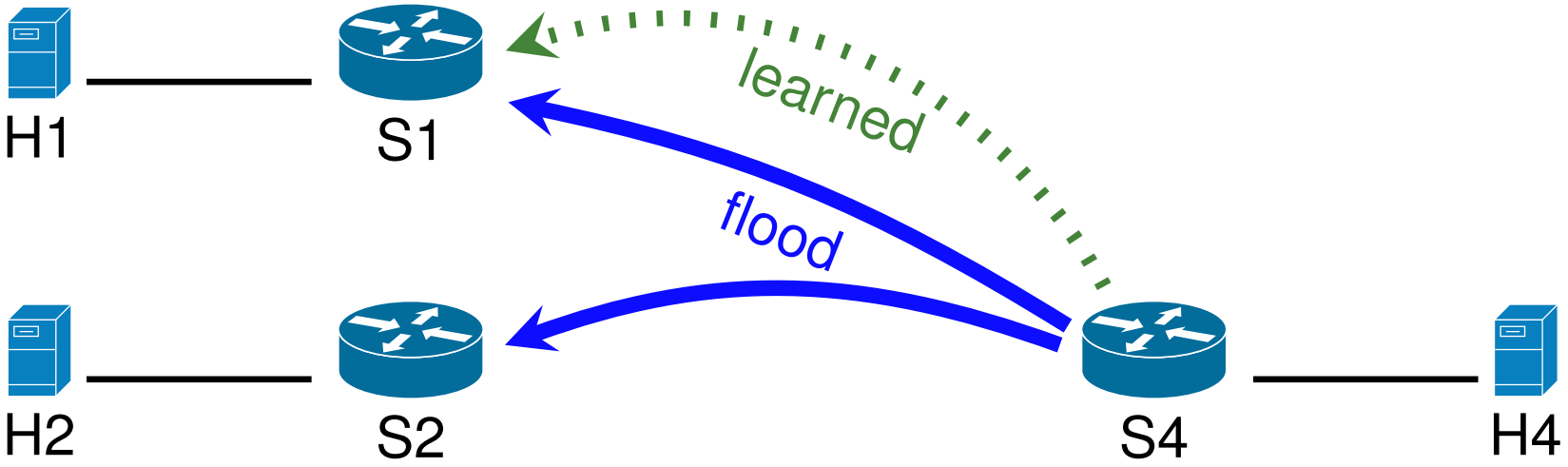
Learning Switch—Topology



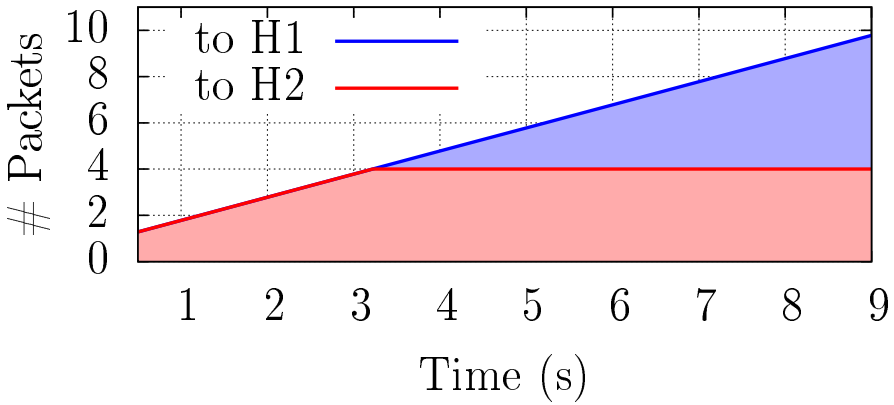
Incorrect behavior

Case Studies—Correctness

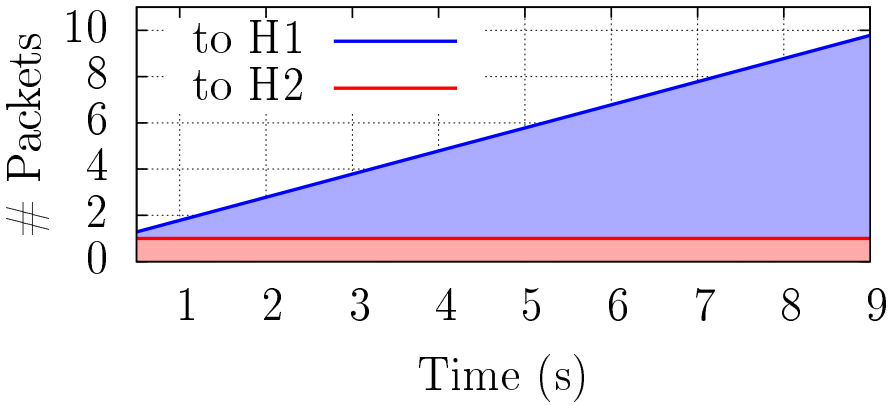
- We also built several other real-world examples:



Learning Switch—Topology



Incorrect behavior



Correct behavior

Related Work

- Network Updates, Verification, and Synthesis
  - Consistent Updates [SIGCOMM'12], Dionysus [SIGCOMM'14], CCG [NSDI'15], Update Synthesis [PLDI'15], Checking Beliefs [NSDI'15], NetEgg [CoNEXT'15]
- High-Level Network Functionality
  - Pyretic [NSDI'13], SDX [SIGCOMM'14]
- Network Programming Languages
  - Frenetic [POPL'12], NetKAT [POPL'14]
  - Maple [SIGCOMM'13], FlowLog [NSDI'14], Kinetic [NSDI'15]
  - SNAP [SIGCOMM'16], Domino [SIGCOMM'16]

Related Work

- Network Updates, Verification, and Synthesis
  - Consistent Updates [SIGCOMM'12], Dionysus [SIGCOMM'14], CCG [NSDI'15], Update Synthesis [PLDI'15], Checking Beliefs [NSDI'15], NetEgg [CoNEXT'15]
- High-Level Network Functionality
  - Pyretic [NSDI'13], SDX [SIGCOMM'14]
- Network Programming Languages
  - Frenetic [POPL'12], NetKAT [POPL'14]
  - Maple [SIGCOMM'13], FlowLog [NSDI'14], Kinetic [NSDI'15]
  - SNAP [SIGCOMM'16], Domino [SIGCOMM'16]

Related Work

- Network Updates, Verification, and Synthesis
  - Consistent Updates [SIGCOMM'12], Dionysus [SIGCOMM'14], CCG [NSDI'15], Update Synthesis [PLDI'15], Checking Beliefs [NSDI'15], NetEgg [CoNEXT'15]
- High-Level Network Functionality
  - Pyretic [NSDI'13], SDX [SIGCOMM'14]
- Network Programming Languages
  - Frenetic [POPL'12], NetKAT [POPL'14]
  - Maple [SIGCOMM'13], FlowLog [NSDI'14], Kinetic [NSDI'15]
  - SNAP [SIGCOMM'16], Domino [SIGCOMM'16]

Summary

1. We provide a new **consistency model** for networks *(allows defining correct-by-construction event-driven behavior, without sacrificing availability)*

Summary

1. We provide a new **consistency model** for networks *(allows defining correct-by-construction event-driven behavior, without sacrificing availability)*
   - **Event-Driven Consistent Update** (when to respond to events)

Summary

1. We provide a new **consistency model** for networks *(allows defining correct-by-construction event-driven behavior, without sacrificing availability)*
   - **Event-Driven Consistent Update** (when to respond to events)
   - **Event Structure Consistency** (how to handle conflicts)

Summary

1. We provide a new **consistency model** for networks *(allows defining correct-by-construction event-driven behavior, without sacrificing availability)*
   - **Event-Driven Consistent Update** (when to respond to events)
   - **Event Structure Consistency** (how to handle conflicts)
2. We provide tools built around this model:

Summary

1. We provide a new **consistency model** for networks *(allows defining correct-by-construction event-driven behavior, without sacrificing availability)*
   - **Event-Driven Consistent Update** (when to respond to events)
   - **Event Structure Consistency** (how to handle conflicts)
2. We provide tools built around this model:
   - **PL** *(allows developers to write high-level network programs)*

Summary

1. We provide a new **consistency model** for networks *(allows defining correct-by-construction event-driven behavior, without sacrificing availability)*
   - **Event-Driven Consistent Update** (when to respond to events)
   - **Event Structure Consistency** (how to handle conflicts)
2. We provide tools built around this model:
   - **PL** *(allows developers to write high-level network programs)*
   - **compiler** *(produces correct & efficient SDN implementations)*

Summary

1. We provide a new **consistency model** for networks *(allows defining correct-by-construction event-driven behavior, without sacrificing availability)*
   - **Event-Driven Consistent Update** (when to respond to events)
   - **Event Structure Consistency** (how to handle conflicts)
2. We provide tools built around this model:
   - **PL** *(allows developers to write high-level network programs)*
   - **compiler** *(produces correct & efficient SDN implementations)*
3. We demonstrate usability via real-world examples

Summary

1. We provide a new **consistency model** for networks *(allows defining correct-by-construction event-driven behavior, without sacrificing availability)*
   - **Event-Driven Consistent Update** (when to respond to events)
   - **Event Structure Consistency** (how to handle conflicts)
2. We provide tools built around this model:
   - **PL** *(allows developers to write high-level network programs)*
   - **compiler** *(produces correct & efficient SDN implementations)*
3. We demonstrate usability via real-world examples

Thanks!