FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates

Stefano Vissicchio (UCLouvain)

stefano.vissicchio@uclouvain.be

INFOCOM

13th April 2016

Joint work with Luca Cittadini (Roma Tre University)

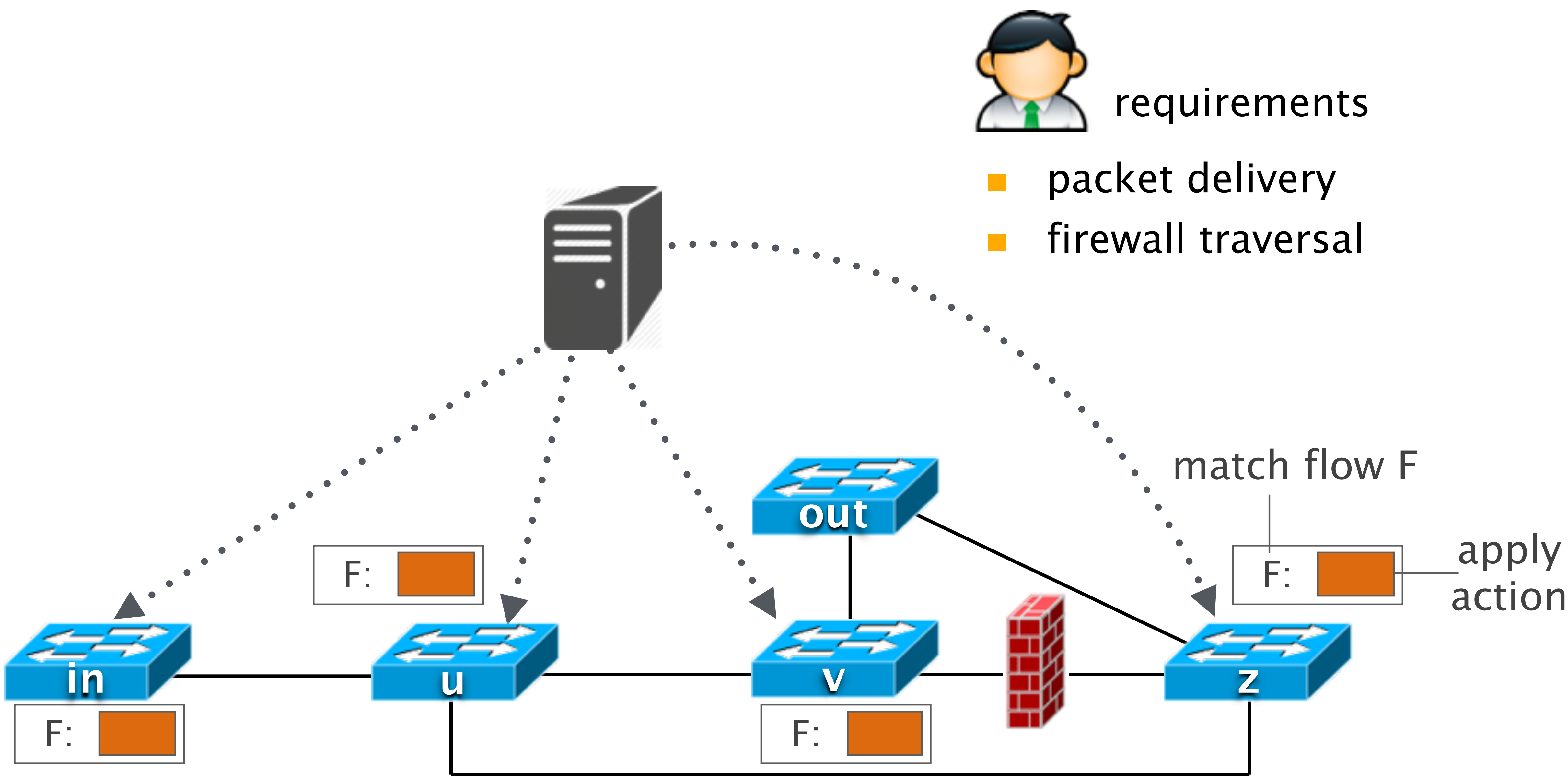

SDN controller

Operators' requirements are an input for SDN controllers

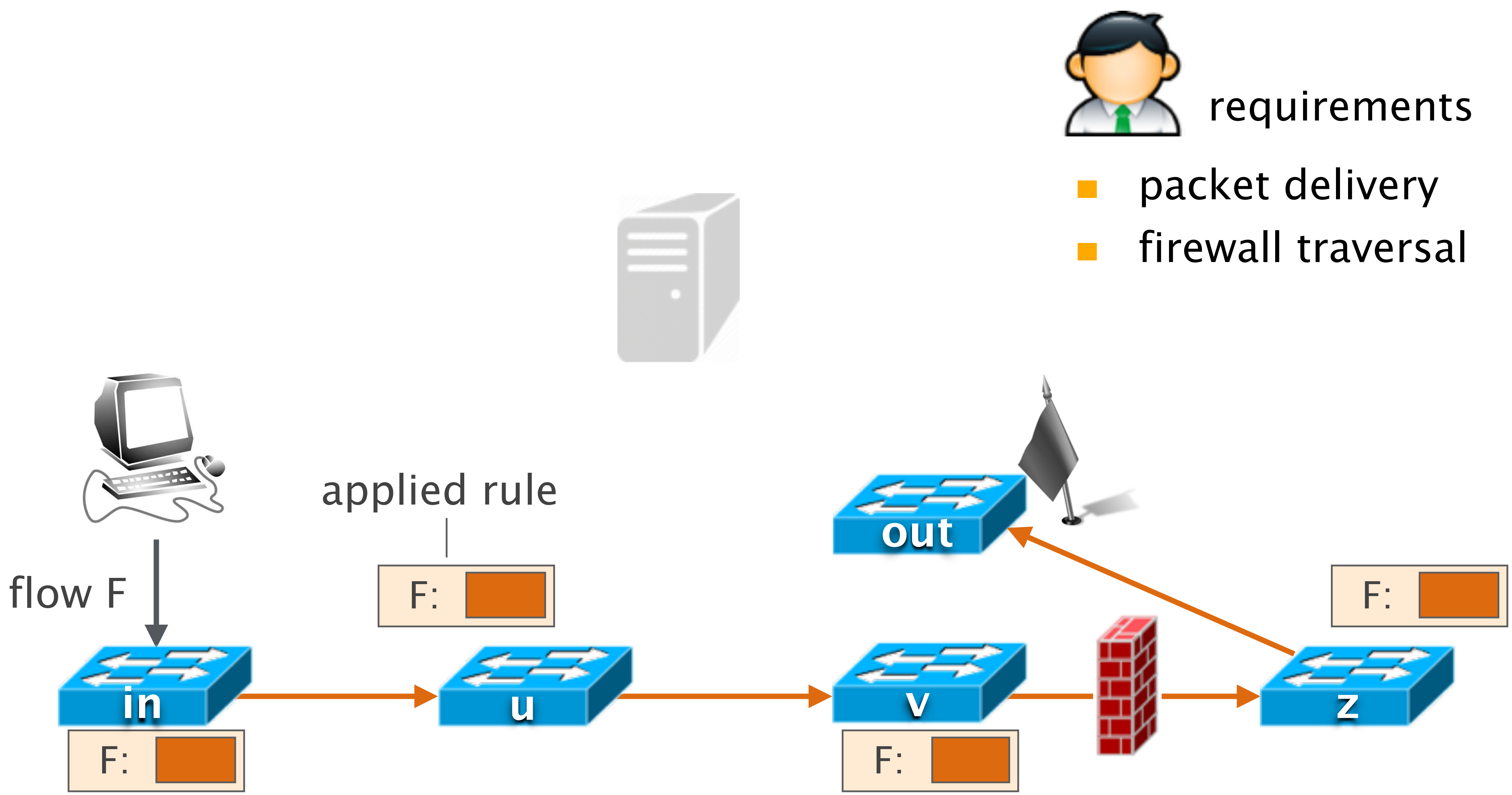
requirements

- packet delivery
- firewall traversal

To satisfy input requirements,
the controller programs rules on the switches



By applying such rules,
switches can process incoming packets

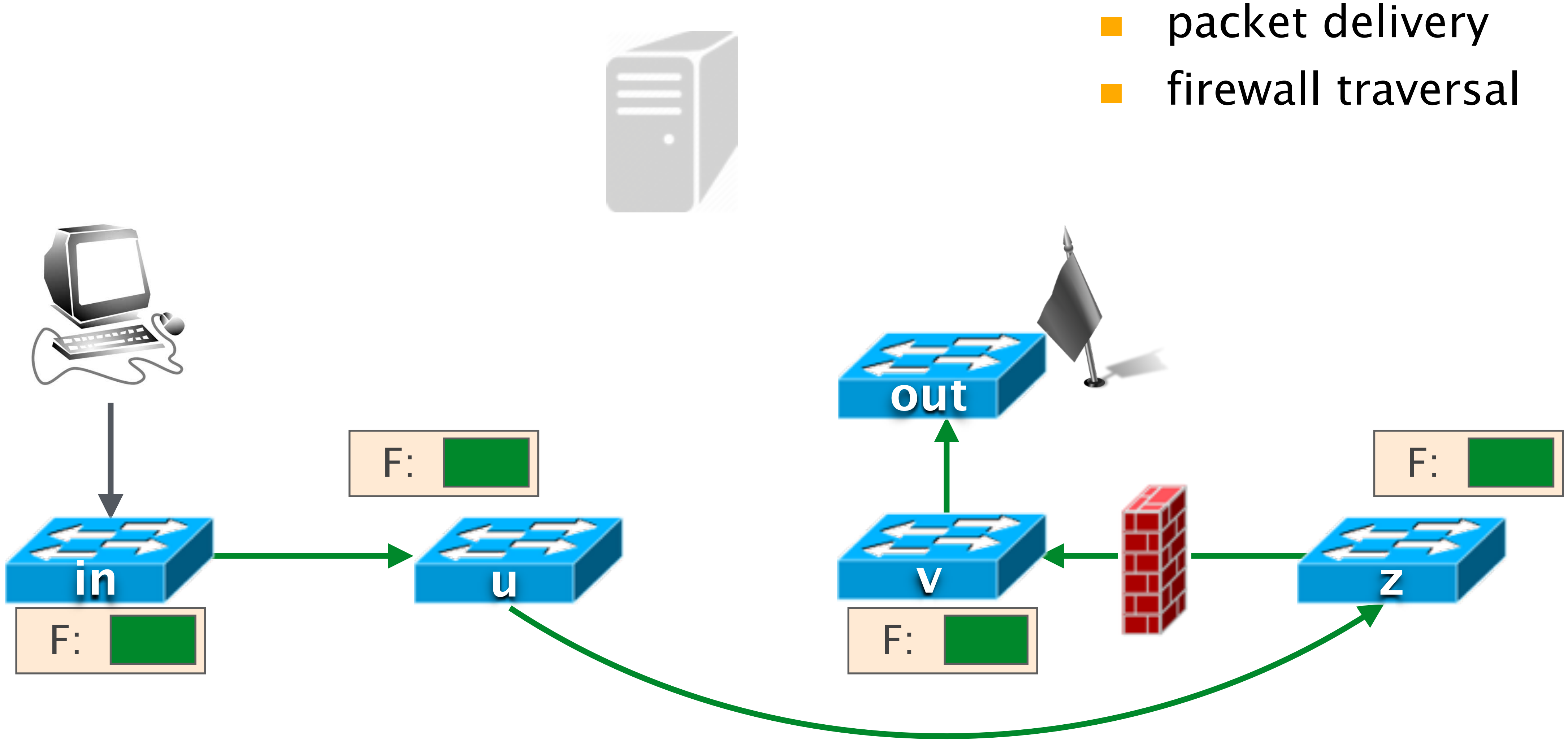


Switch rules have to be (frequently) updated

e.g., for traffic surges, maintenance, new policies

requirements

- packet delivery
- firewall traversal



How to update rules on switches safely, robustly and efficiently?

How to update rules on switches
safely, robustly and efficiently?

requirements are not
violated during the update

How to update rules on switches

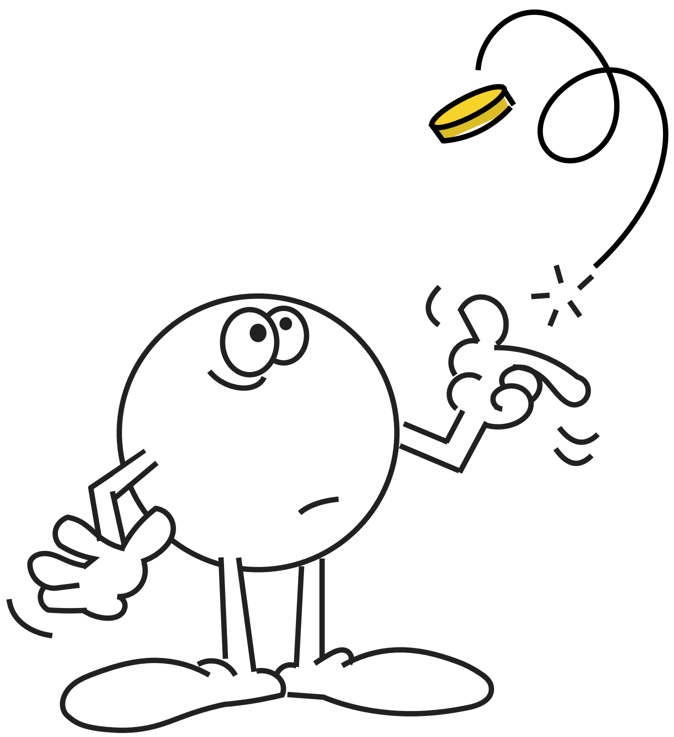
safely, robustly and efficiently?

independently of uncontrollable factors
(messages lost, switch installation time, etc.)

How to update rules on switches safely, robustly and efficiently?

quickly and with
low resource utilization

FLIP the (Flow) Table:
Fast LIghtweight Policy-preserving SDN Updates



- Limitations of prior works
- Additional degrees of freedom
- Our approach

FLIP the (Flow) Table:
Fast LIghtweight Policy-preserving SDN Updates

- Limitations of prior works
- Additional degrees of freedom
- Our approach

How to update rules on switches safely, robustly and efficiently?

Previous techniques cannot do it!

Previous techniques belong to two main families

- ordered rule replacements [McClurg15]
  replace initial with final rules
  in a carefully-computed order

- two-phase commit [Reitblatt12,Jin14]
  add final rules to initial ones
  apply rules consistently, with packet ***tags***

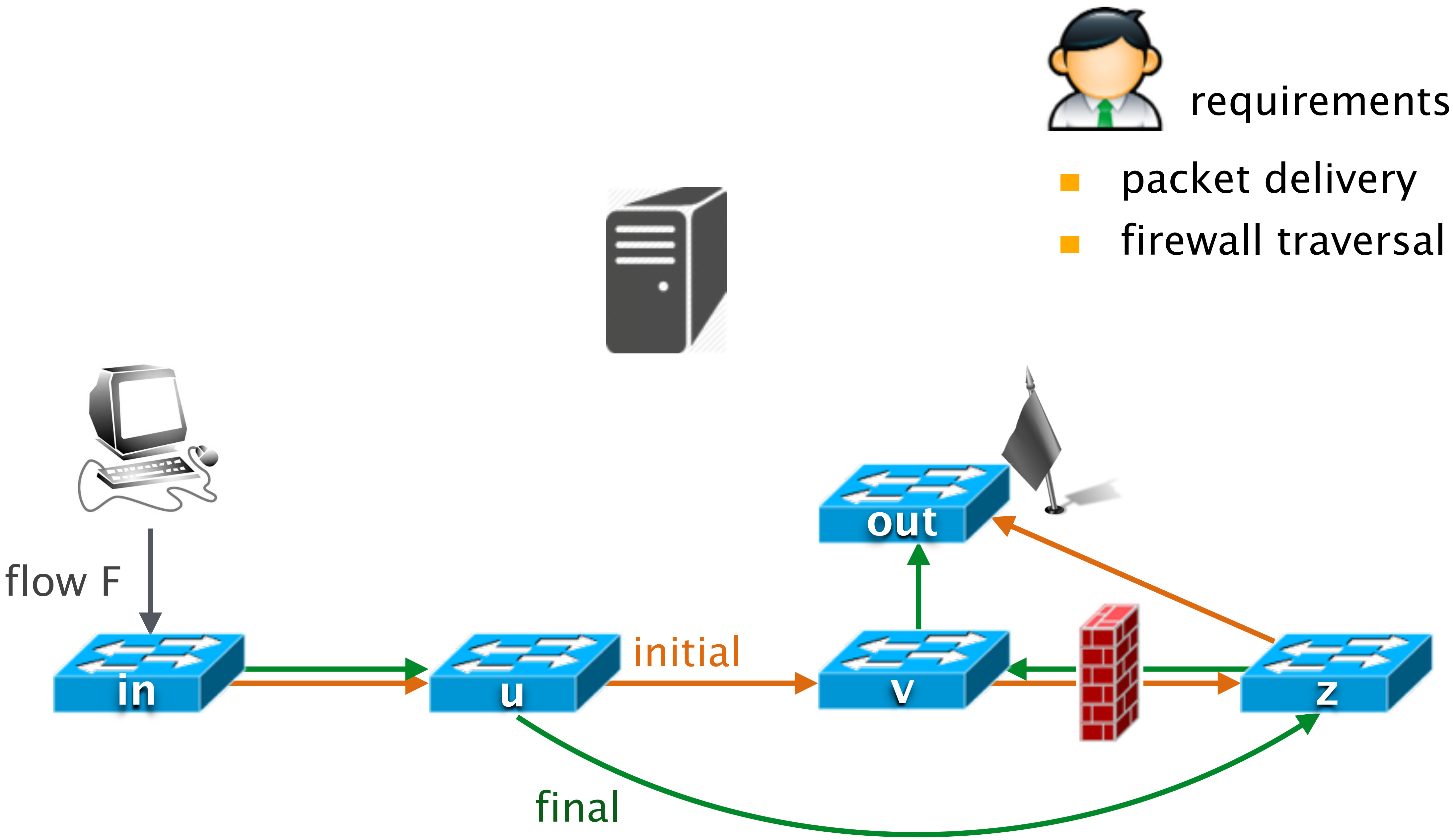
Previous techniques belong to two main families

- ordered rule replacements [McClurg15]
  replace initial with final rules
  in a carefully-computed order

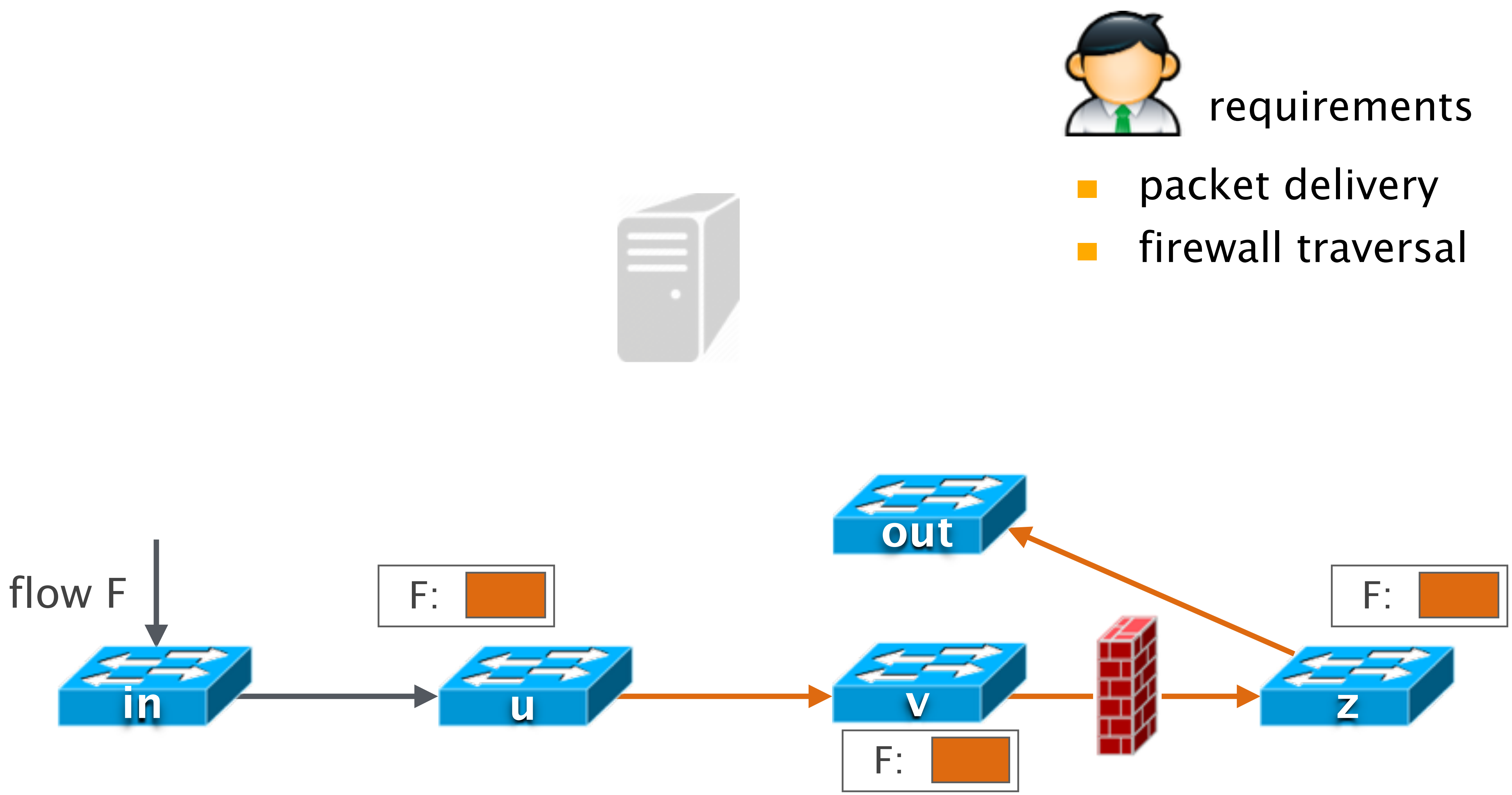
not always applicable

- two-phase commit [Reitblatt12,Jin14]
  add final rules to initial ones
  apply rules consistently, with packet ***tags***

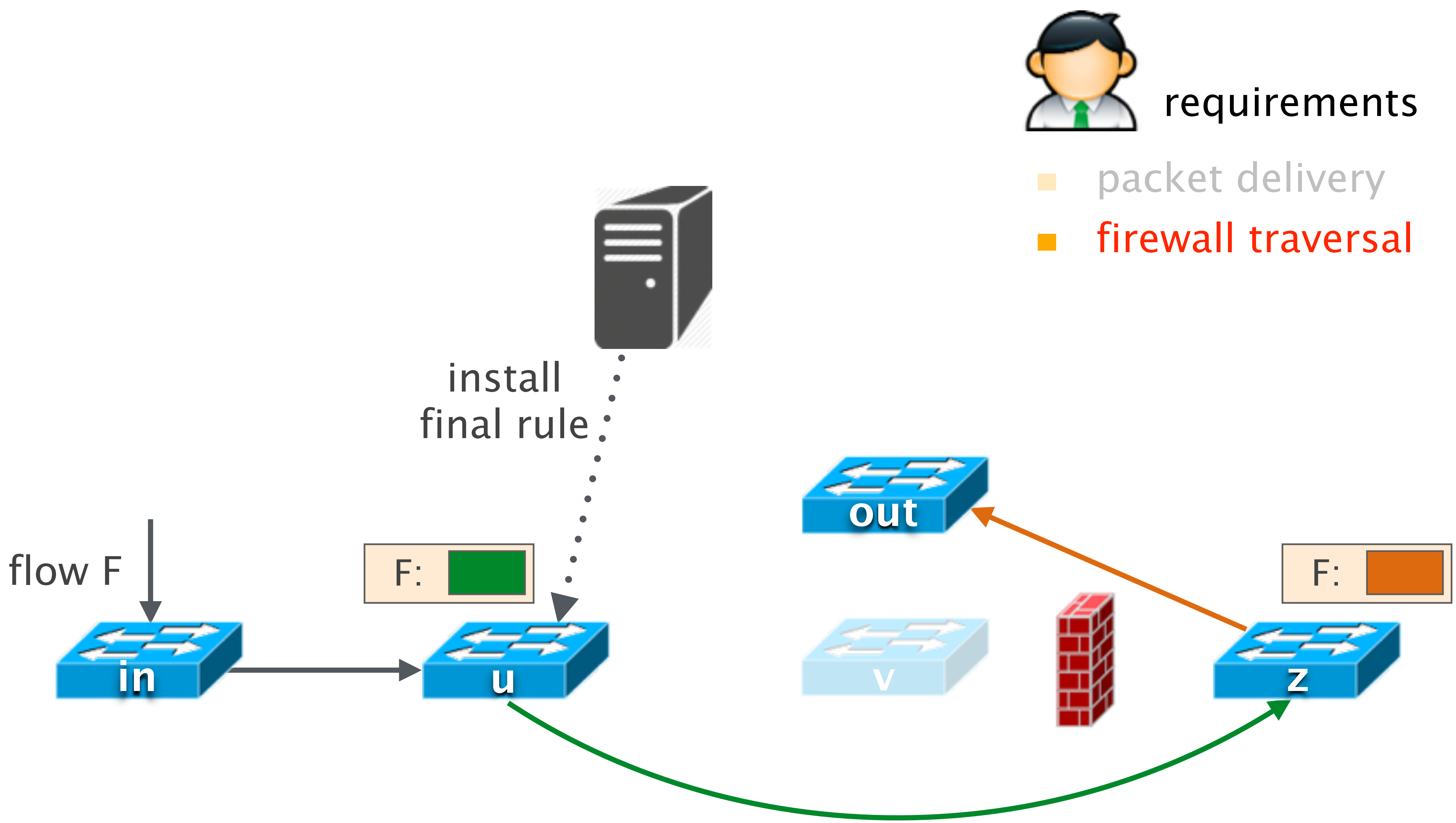
Ordering rule replacements is not possible in our example



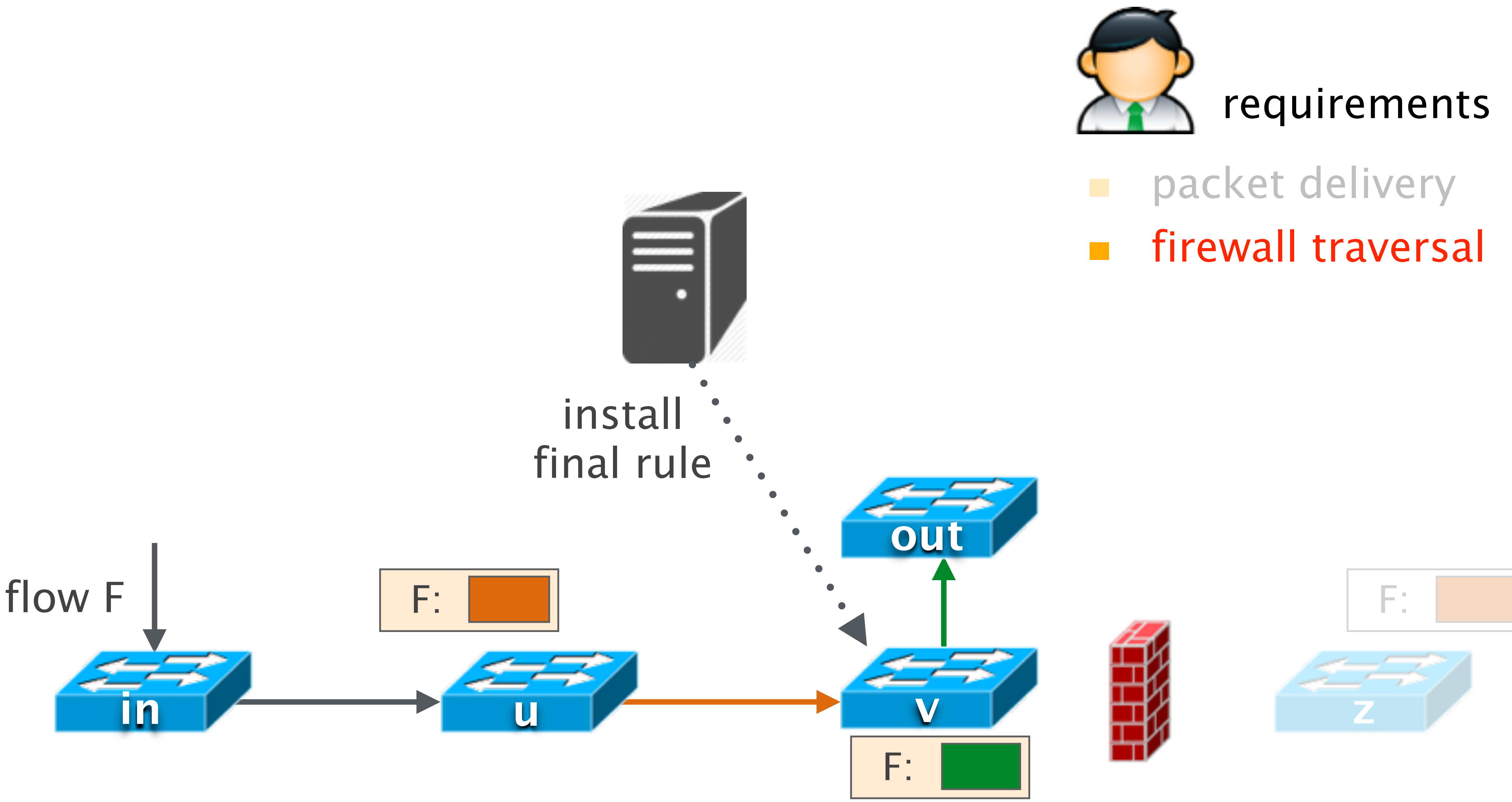
Final rules cannot be installed on any switch among u, v and z



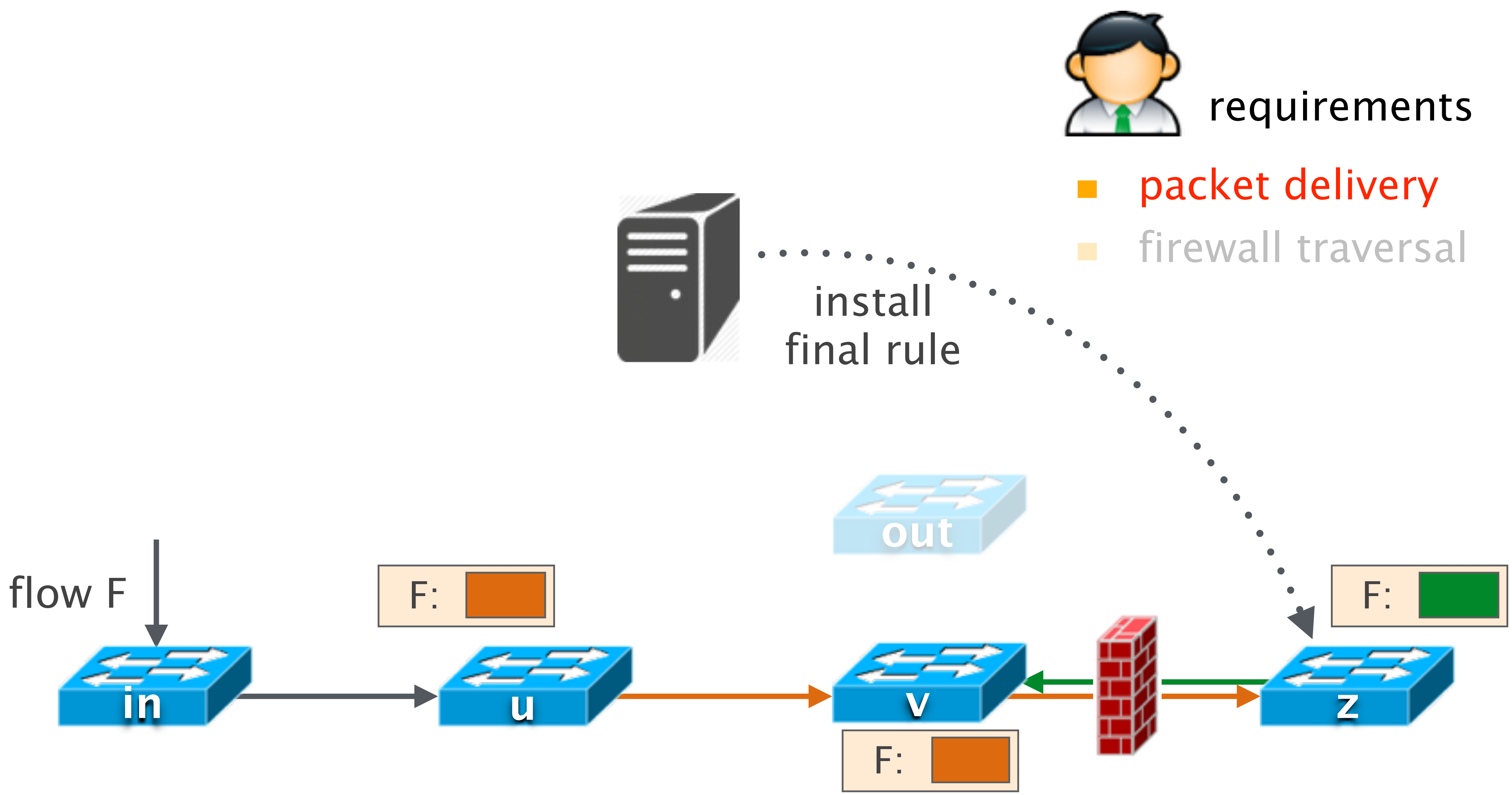
Case 1) Installing final rule at u would violate our security policy



Case 2) Installing final rule at v would violate our security policy



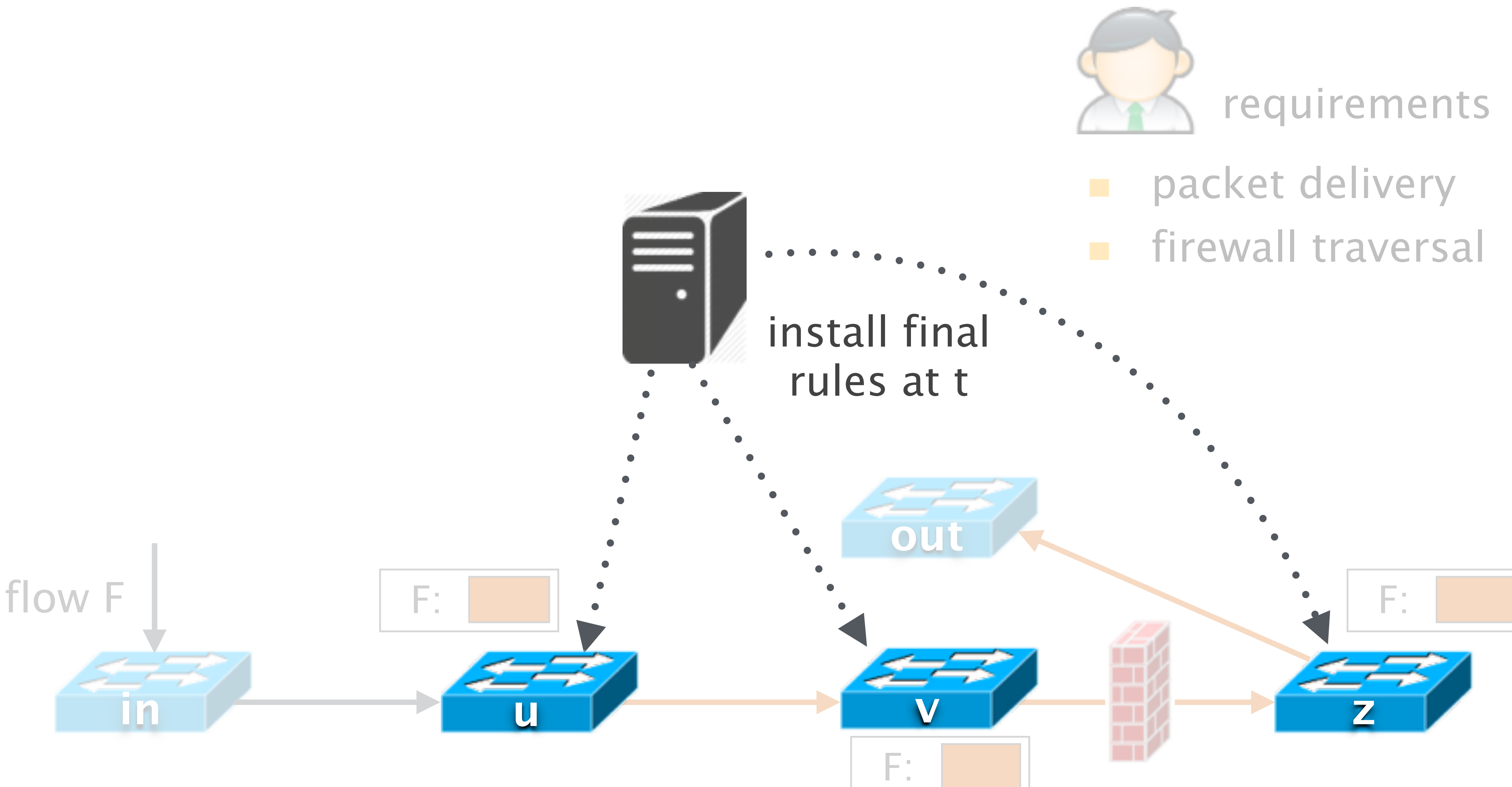
Case 3) Installing final rule at z would prevent packet delivery



Simultaneous rule replacements are not robust

e.g., like in time-based approaches [Mizrahi16]

In our example, we could instruct u, v and z to replace their rules at the same time t



However, this can lead to *transient* problems at t

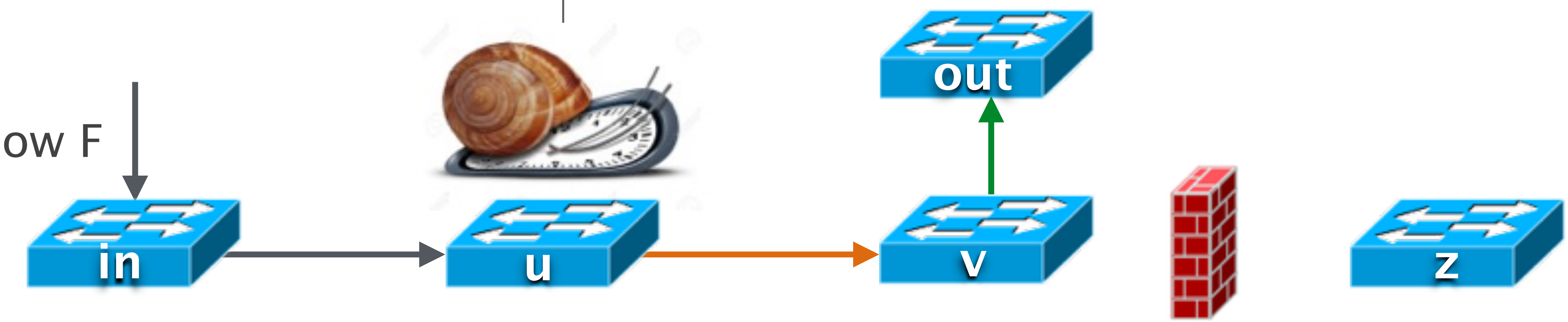
e.g., because of per-switch installation time

requirements

- packet delivery
- firewall traversal

up to several
seconds [Jin14]

flow F



Also, we can trigger *permanent* problems at t

e.g., if a switch does not apply a command

requirements

- packet delivery
- firewall traversal

Previous techniques belong to two main families

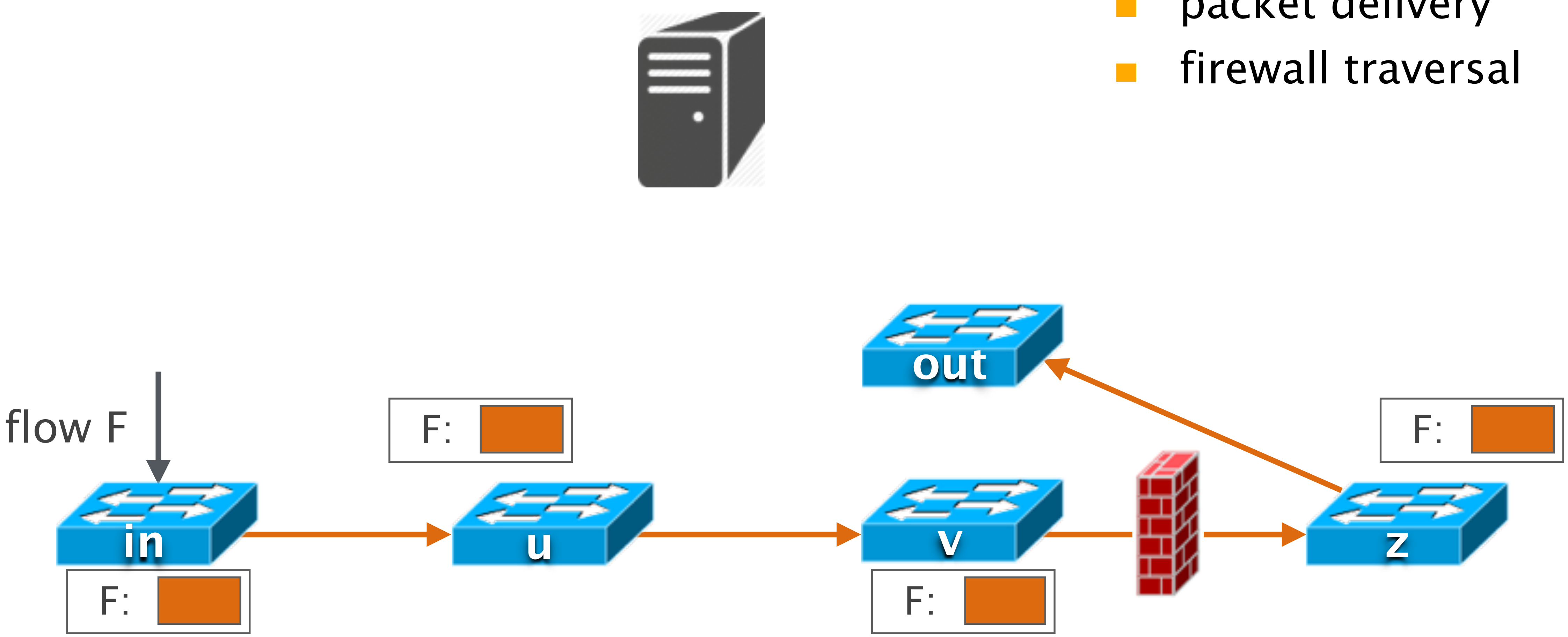
- ordered rule replacements [McClurg15]
  replace initial with final rules
  in a carefully-computed order

- two-phase commit [Reitblatt12,Jin14]
  add final rules to initial ones
  apply rules consistently, with packet ***tags***

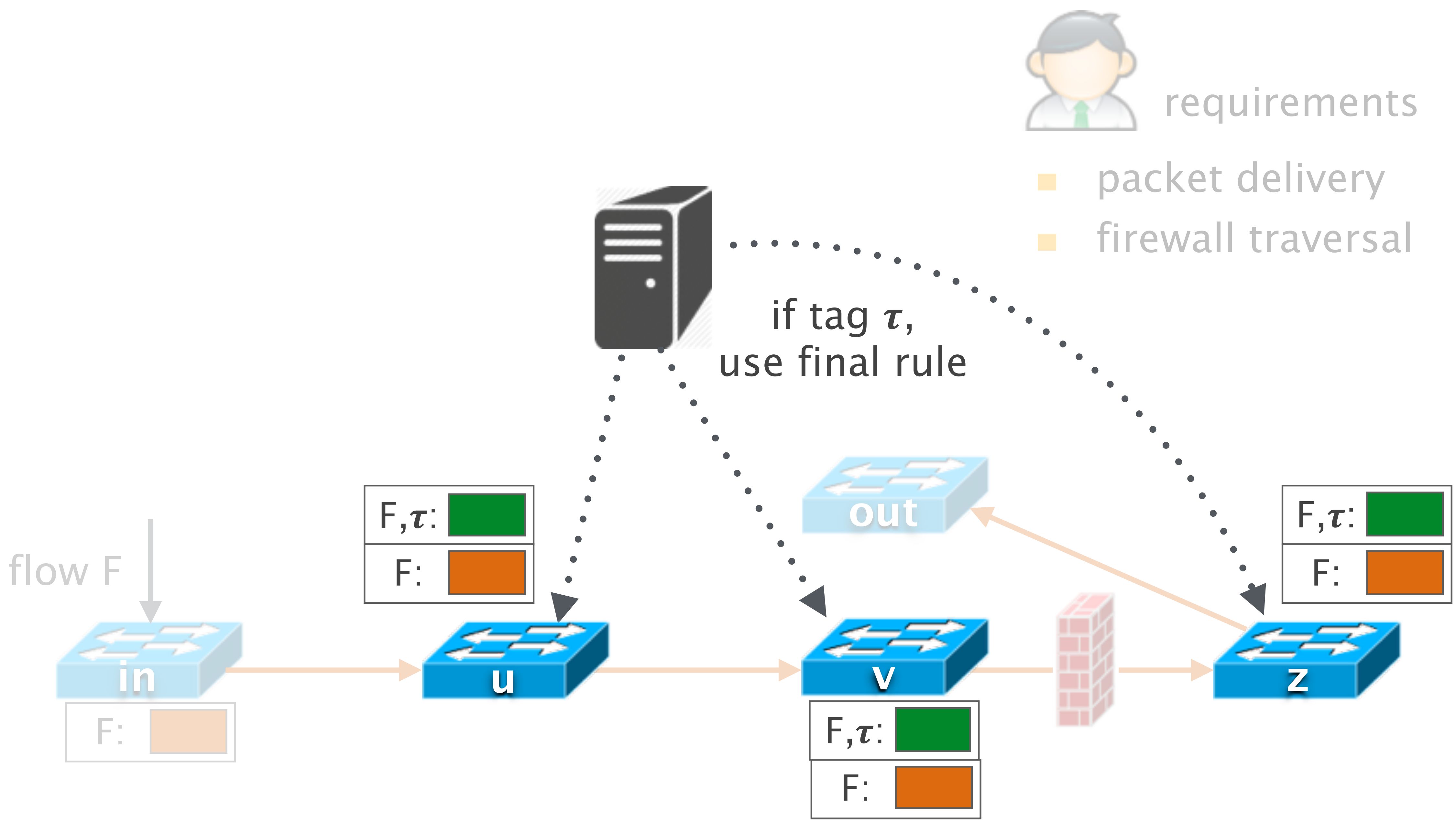
inefficient

Two-phase commit techniques are not efficient in our example

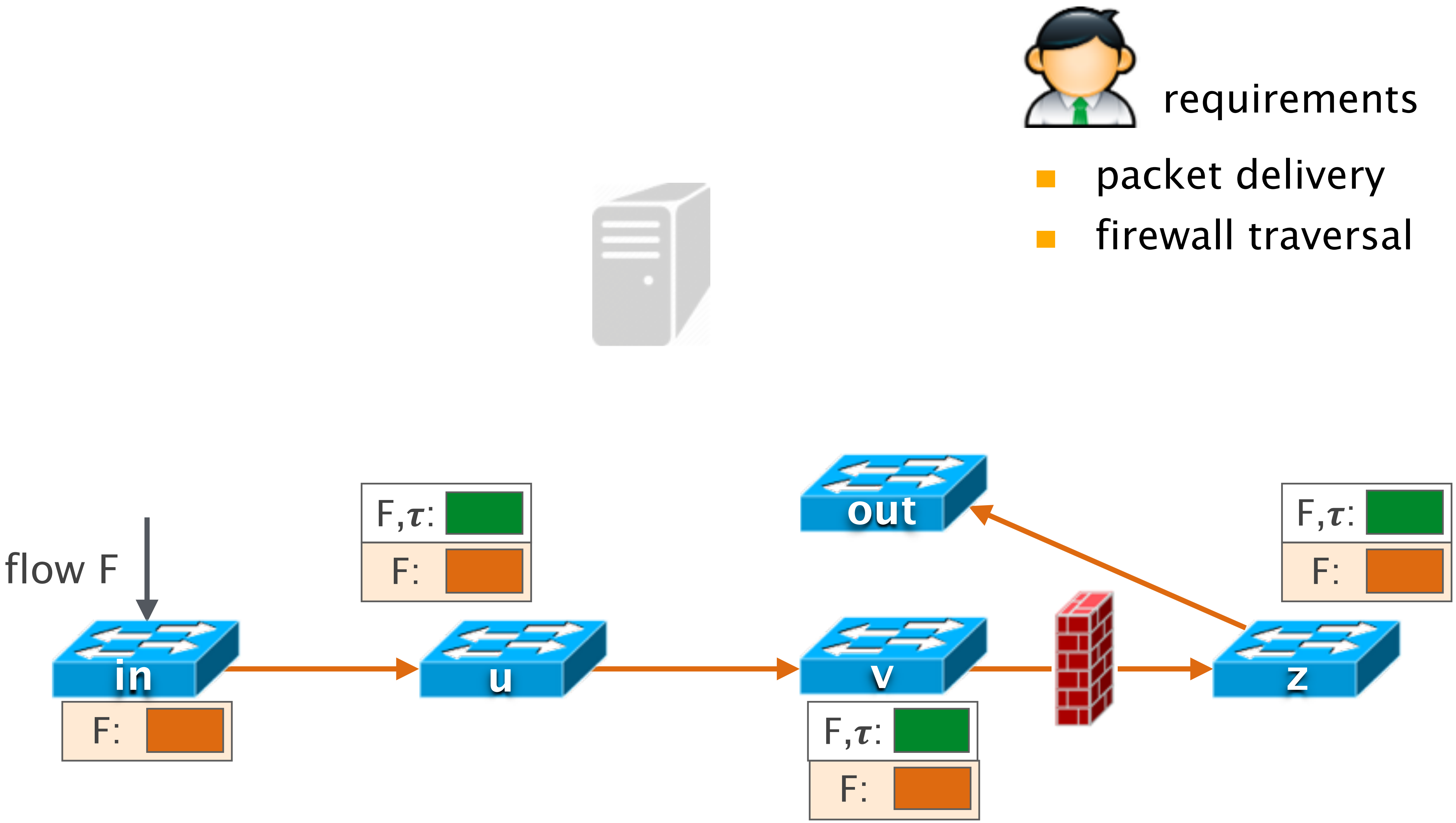
- packet delivery
- firewall traversal



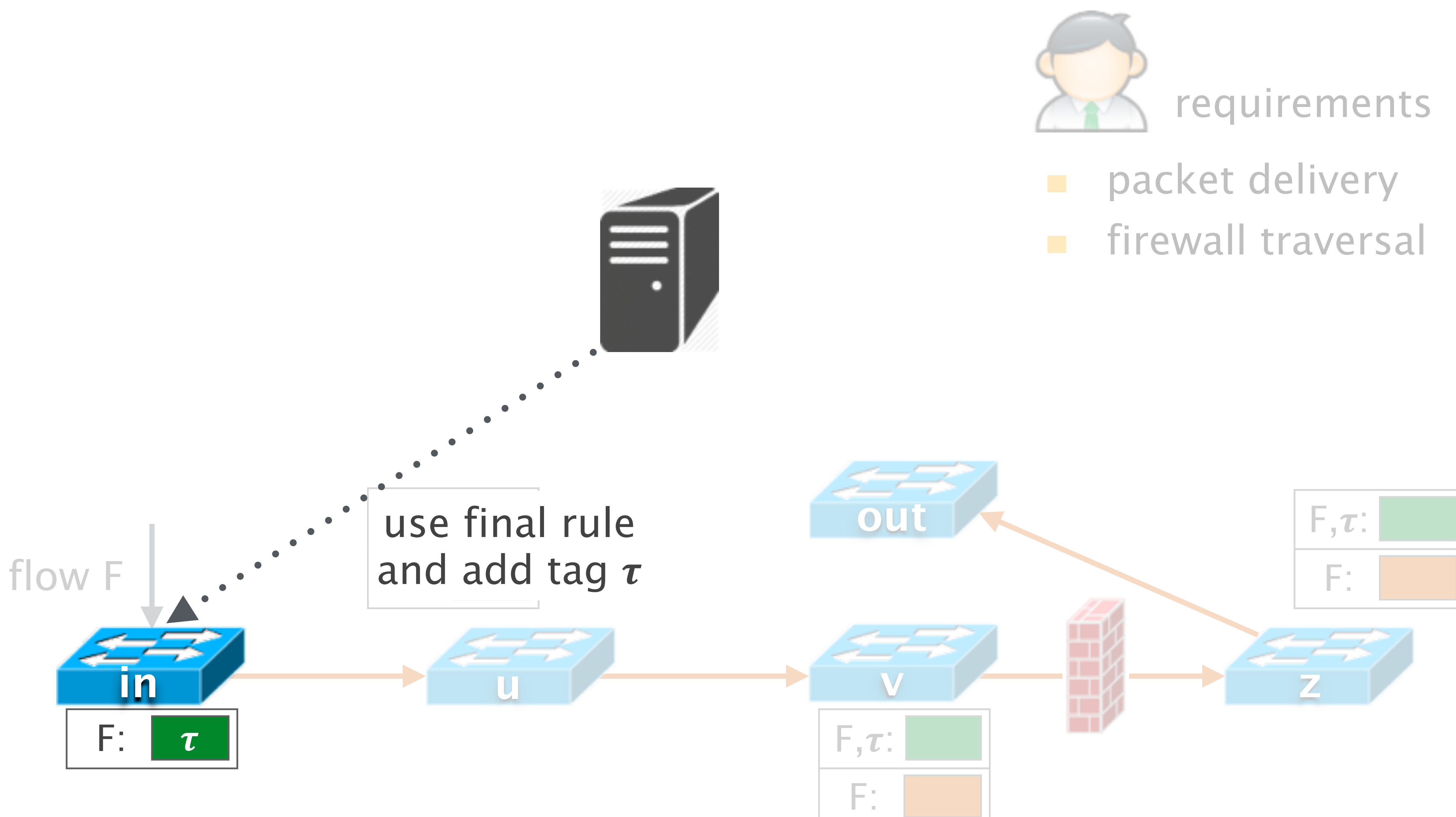
Indeed, they are based on maintaining both initial and final rules on internal switches



So that switches keep applying initial rules…



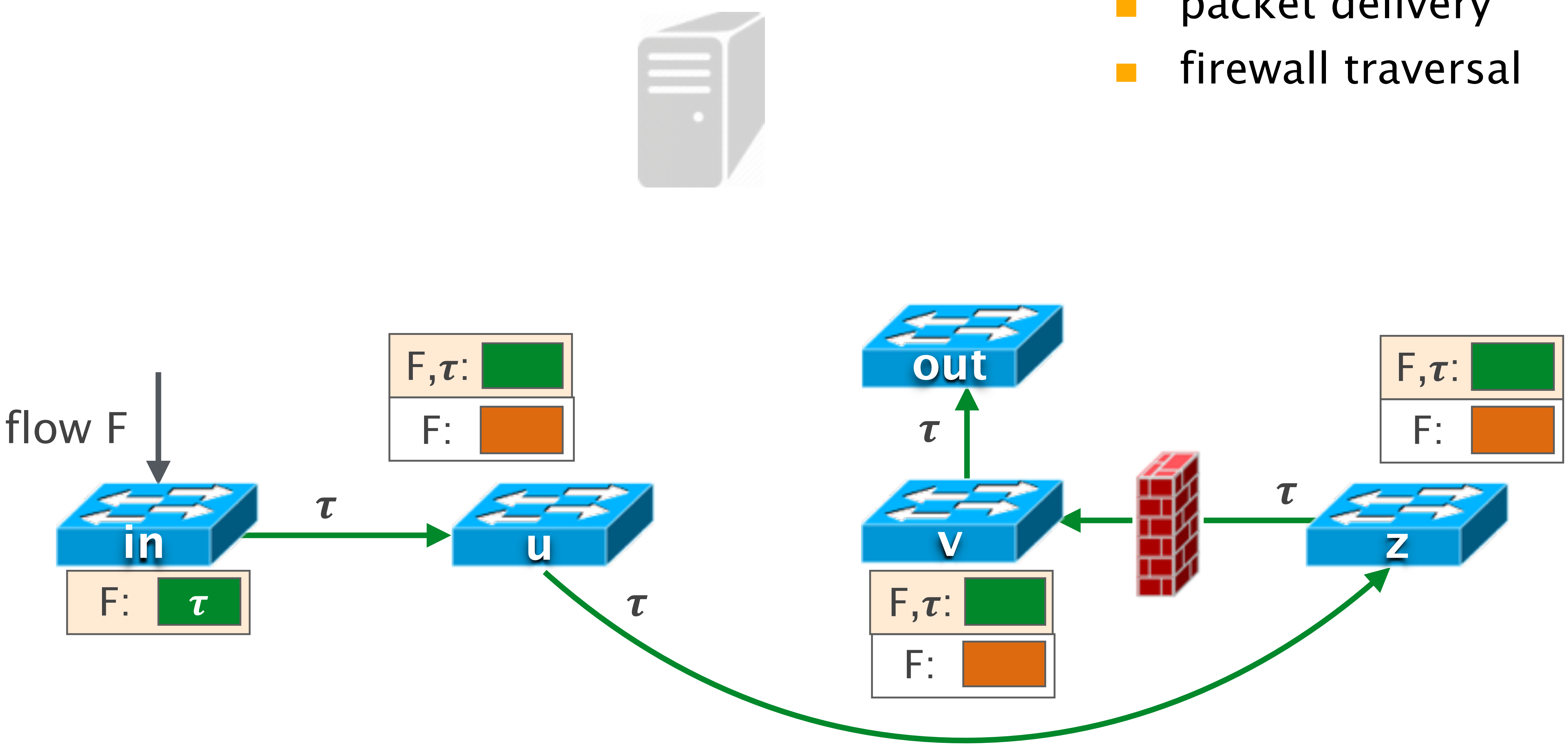
… as long as packets are not tagged at the ingress



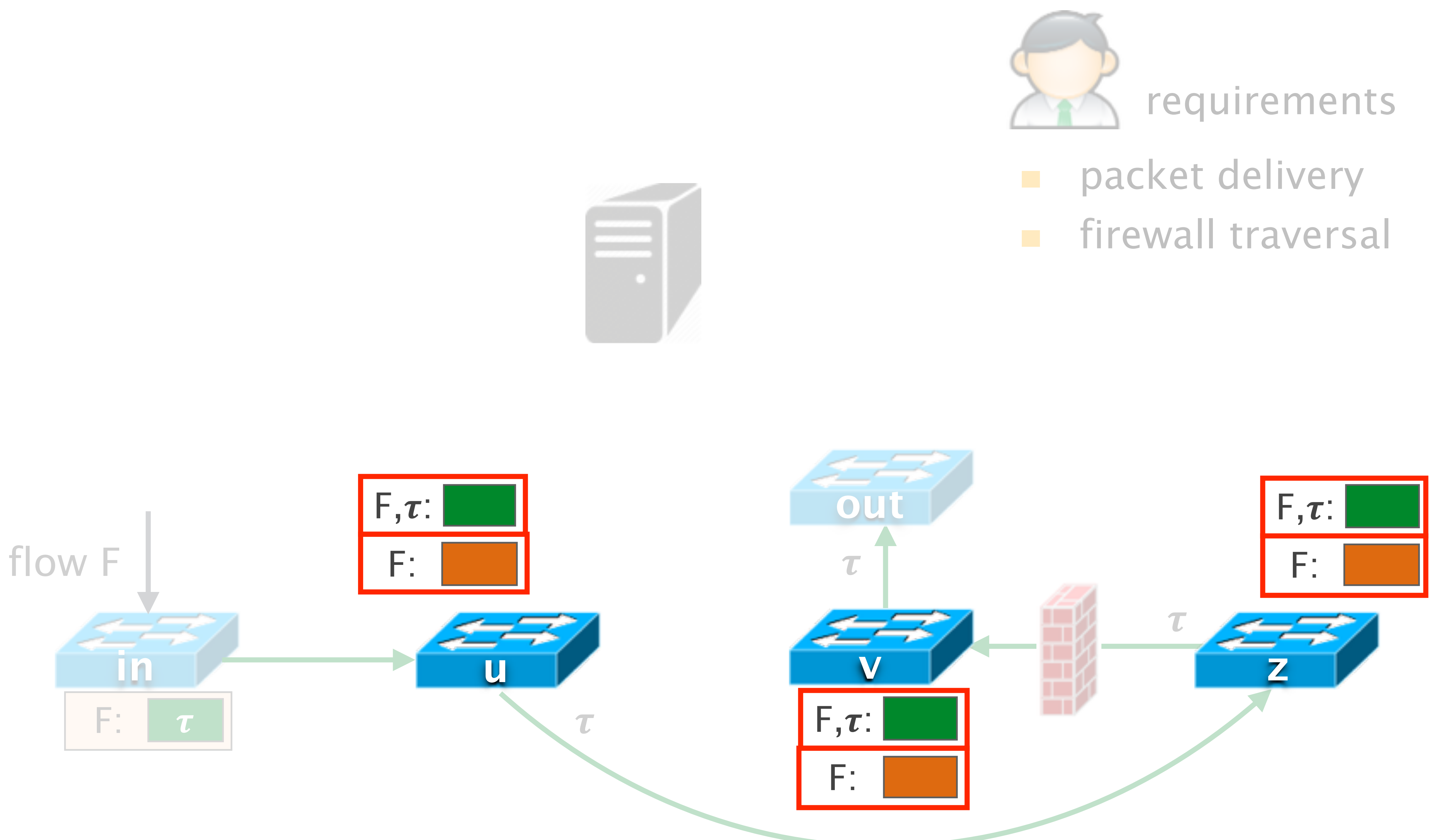
When packets are tagged at the ingress, all switches consistently use the final rules

requirements

- packet delivery
- firewall traversal



However, these techniques consumes precious and expensive memory (TCAM) entries



FLIP the (Flow) Table:
Fast LIghtweight Policy-preserving SDN Updates

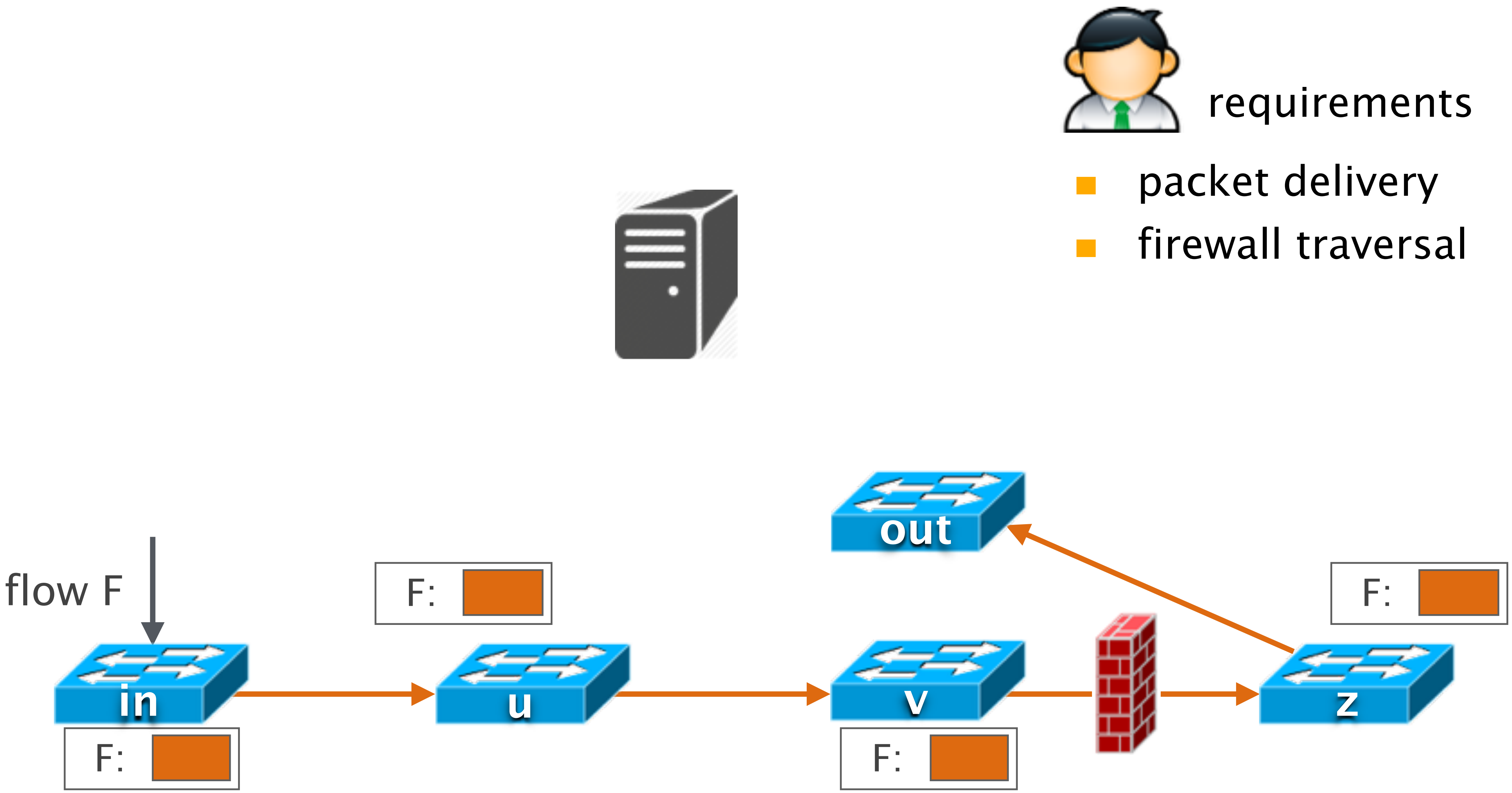
- Limitations of prior works
- Additional degrees of freedom
- Our approach

How to update rules on switches safely, robustly and efficiently?

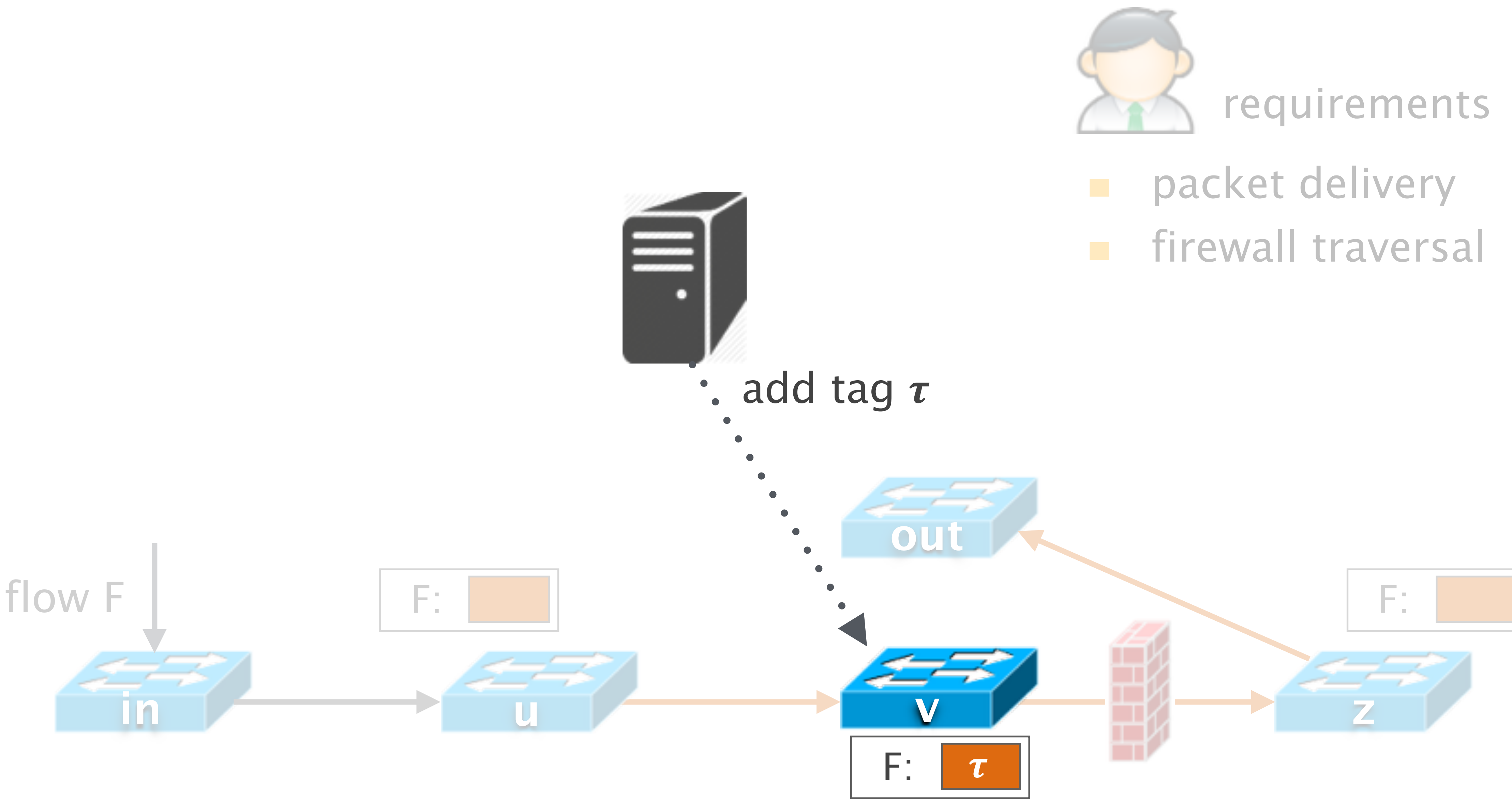
We can do it!

The key intuition is to combine
rule replacement and additions

Let's take back our example
and start from the initial state



We can start tagging packets at v,
at the very beginning of the update

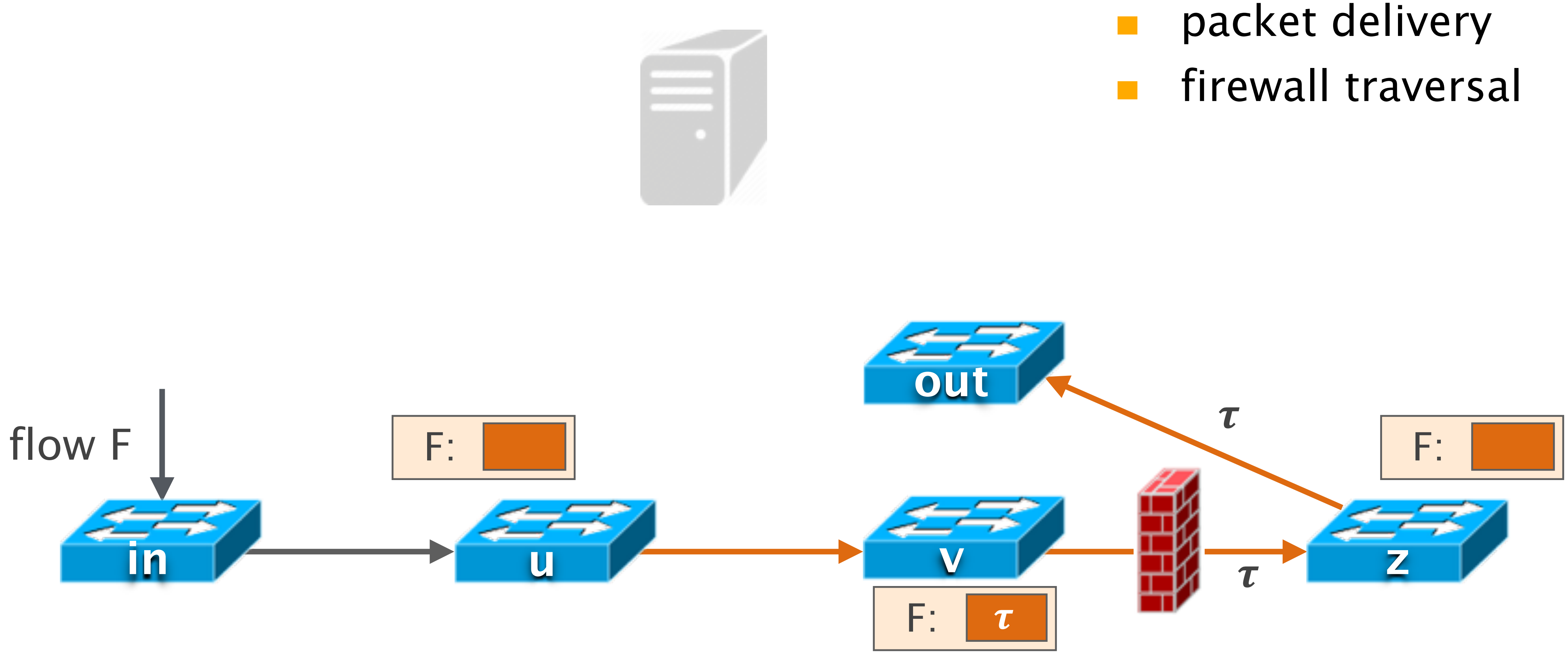


This does not change the applied rules

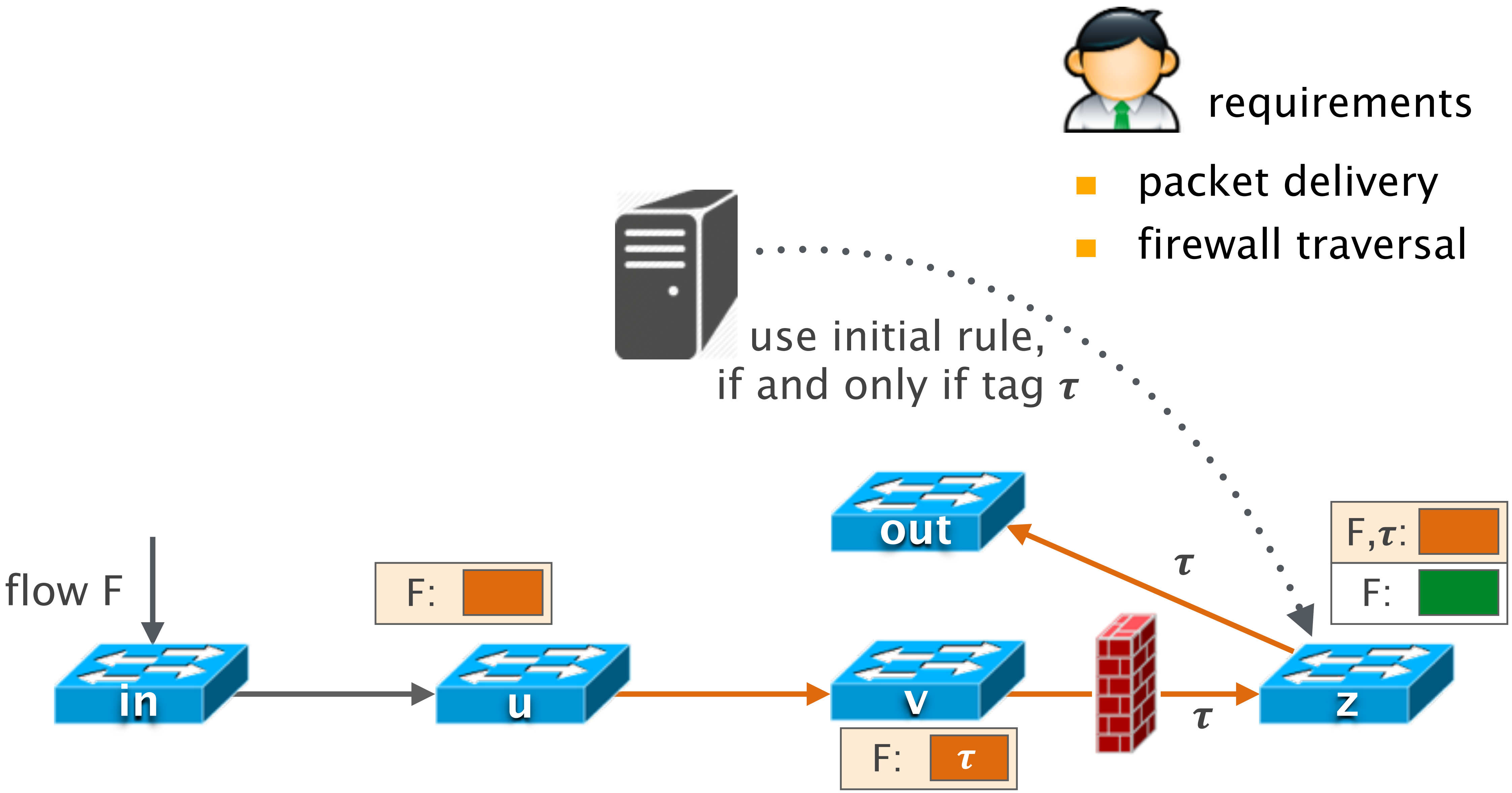
(since no switch matches the tag yet)

requirements

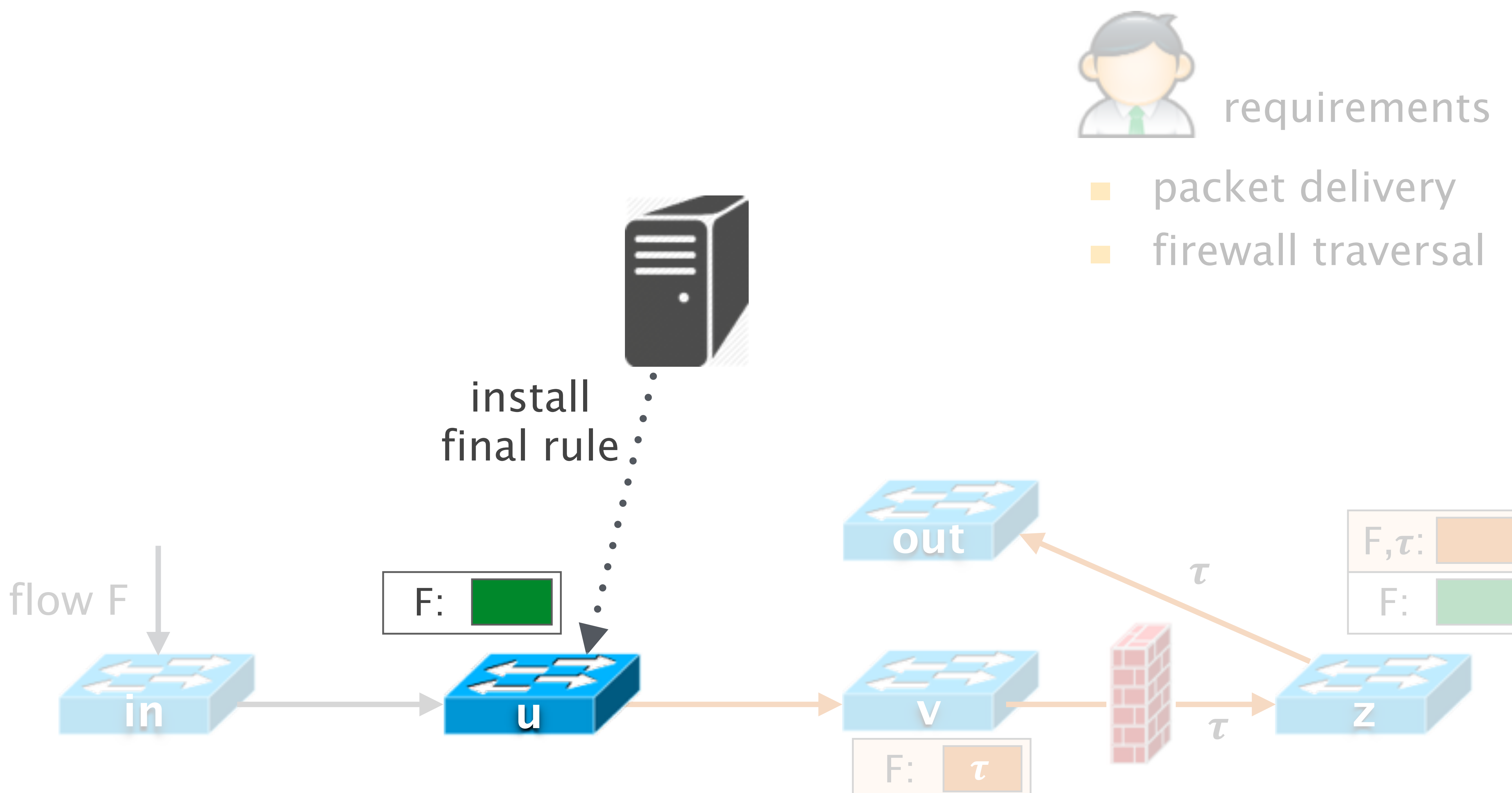
- packet delivery
- firewall traversal



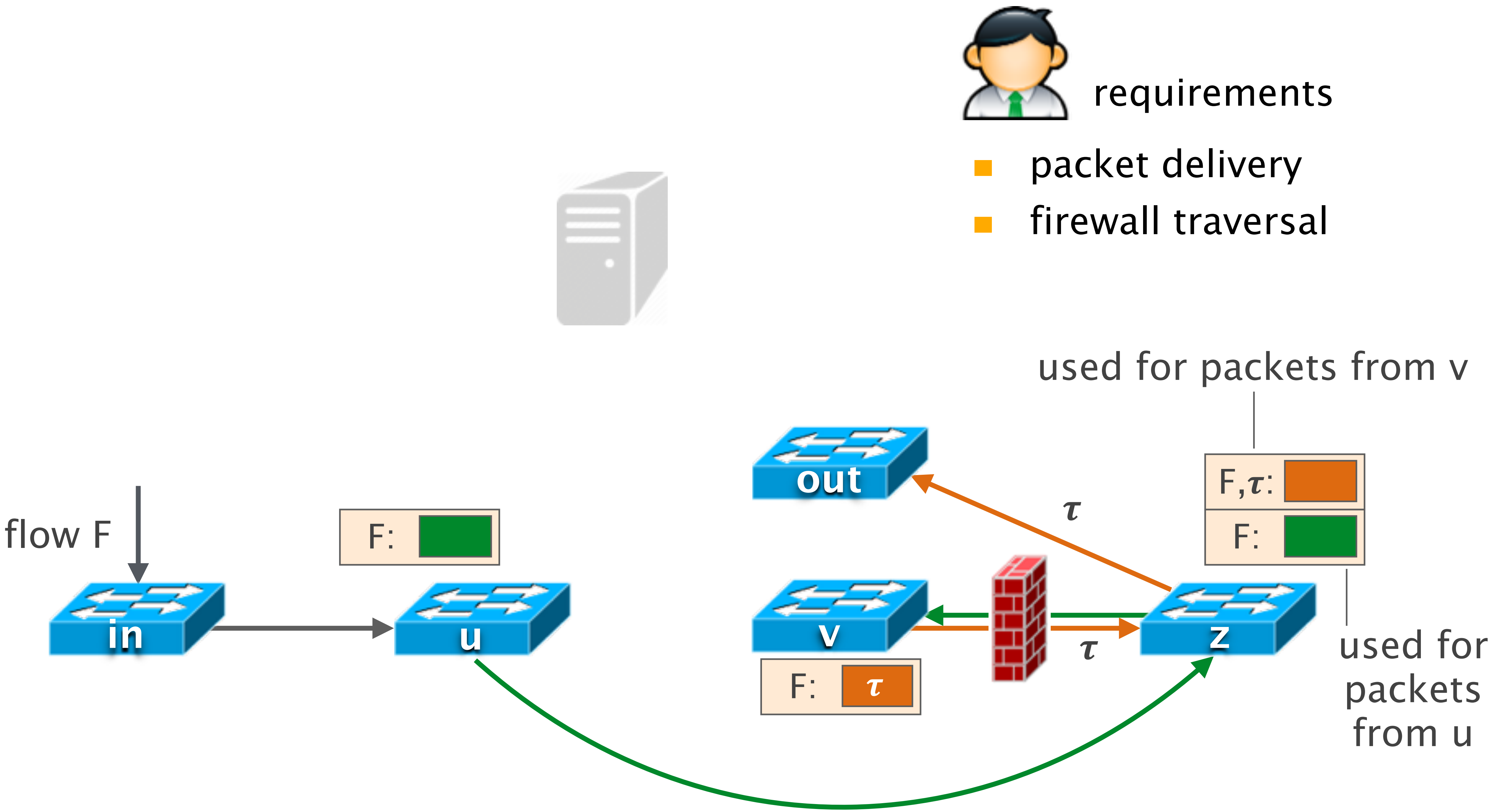
We can then match the tag at z, still without changing the forwarding



Tagging at v and matching at z unlock rule replacement at u

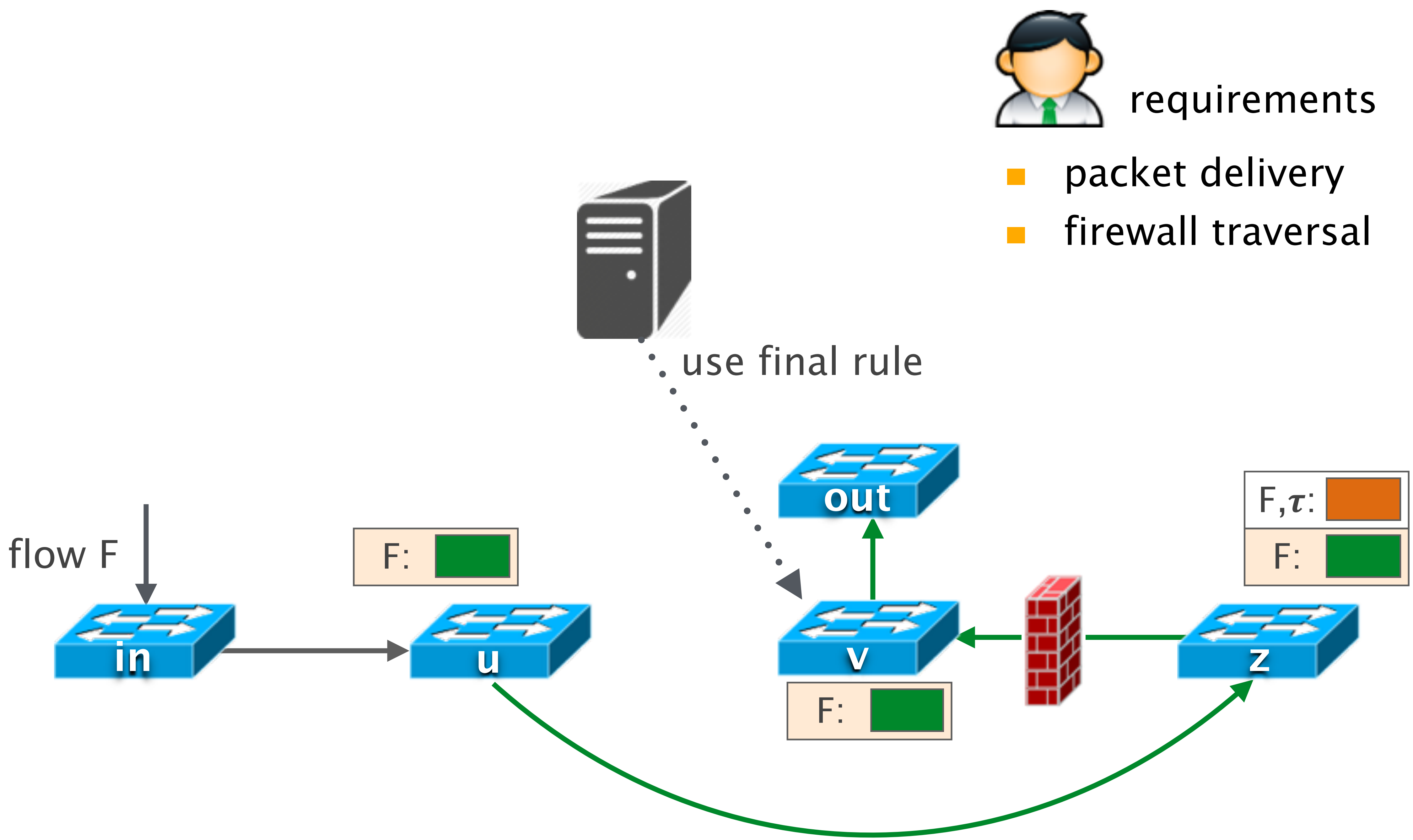


Indeed, the resulting forwarding loop is traversed only once by packets



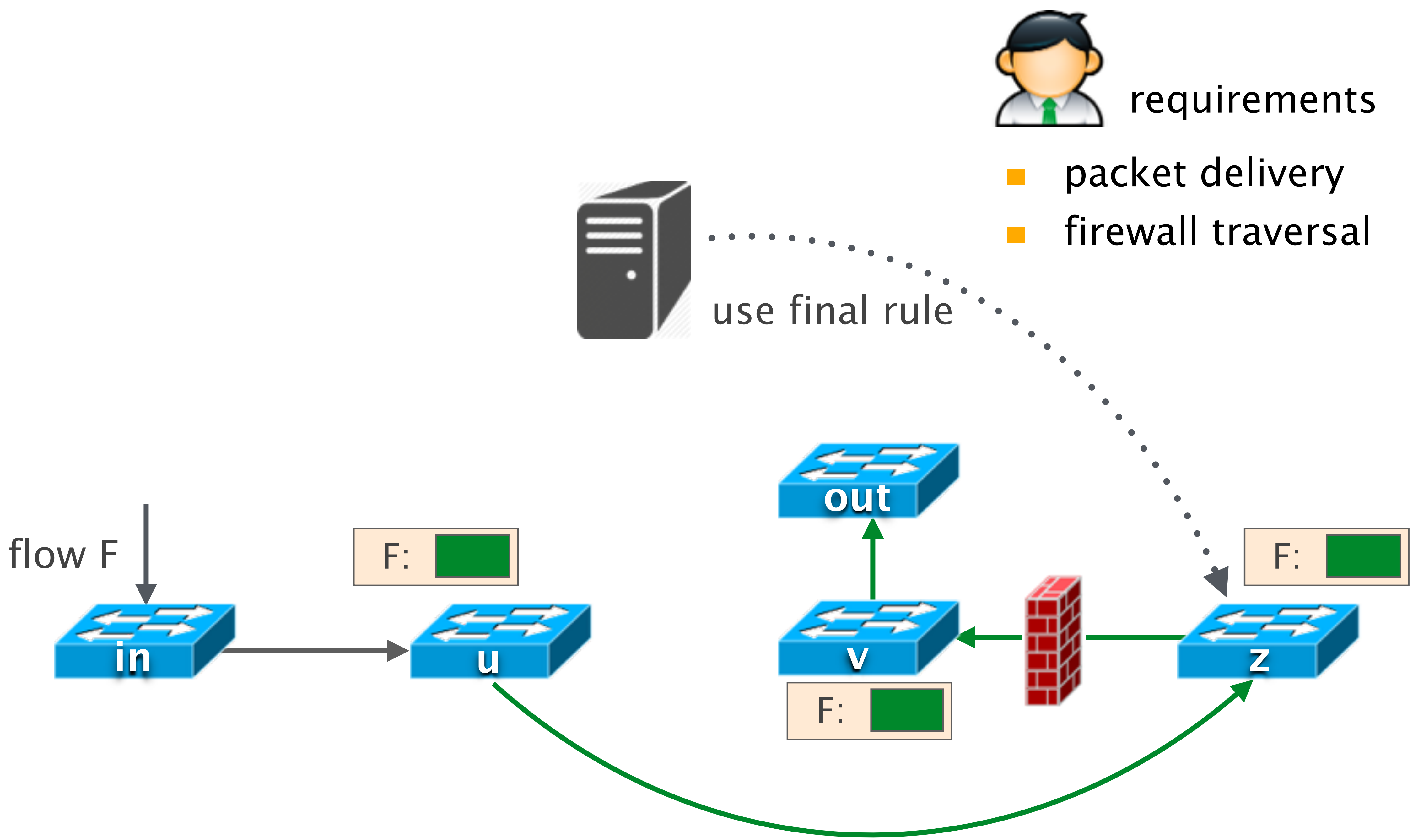
We can then instruct v to apply its final rule

(even in parallel with u)



and complete the update

by cleaning z's configuration



Using both rule replacements and additions is more powerful than restricting to any of them

- solves our update problem
  contrary to ordered rule replacement
- ensures robustness
  rollbacking before affecting safety
- uses additional rules only on z
  33% with respect to two-phase commit

Using both rule replacements and additions makes the update problem more challenging

- larger search space

  we must consider combinations of rule replacements and additions

- tricky interactions in intermediate states

  e.g., we must distinguish loops that prevent packet delivery from the good ones

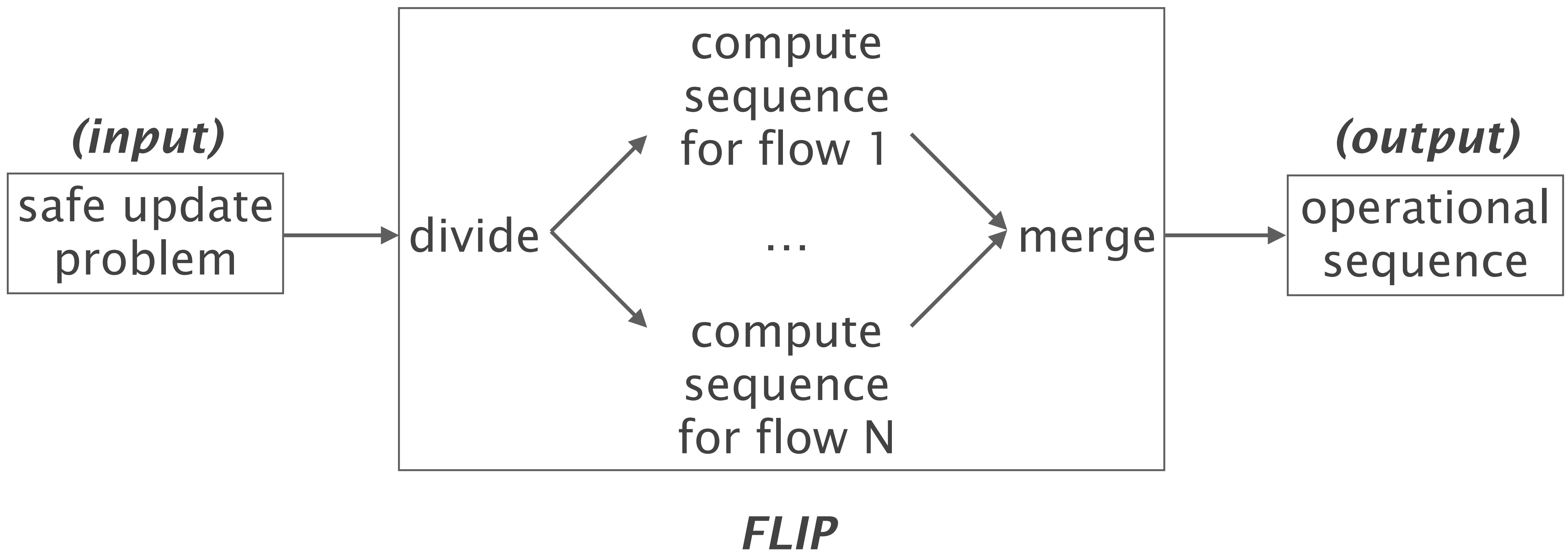
FLIP the (Flow) Table:
Fast LIghtweight Policy-preserving SDN Updates

- Limitations of prior works
- Additional degrees of freedom
- Our approach

We propose a framework to systematically combine rule replacements and additions

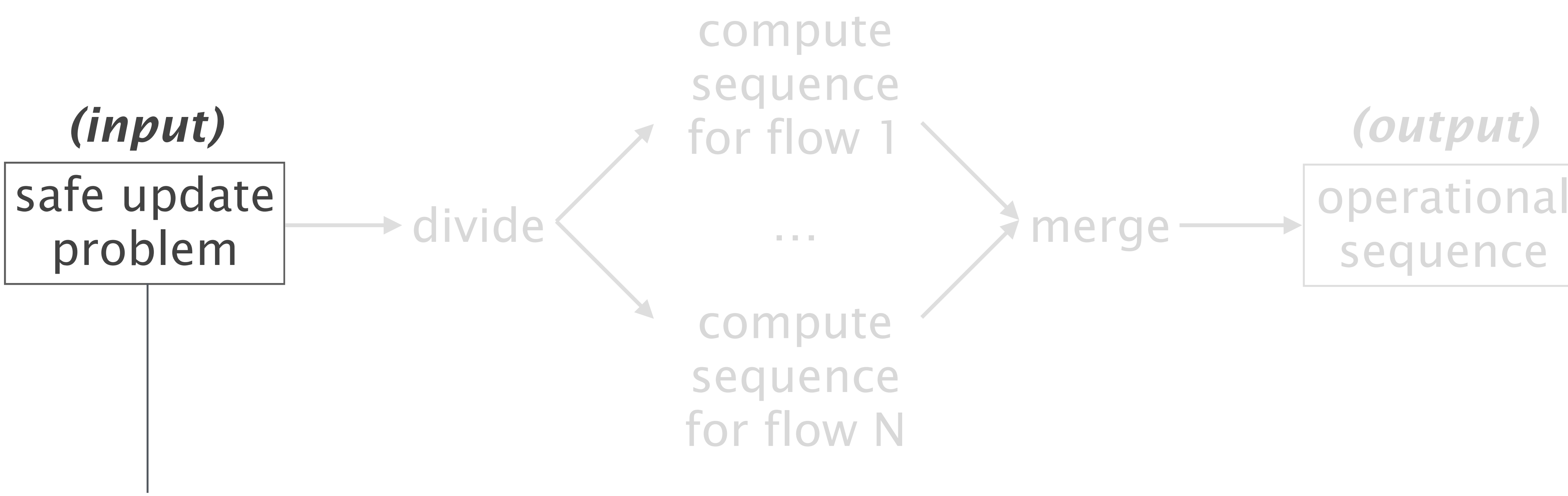
- **formalization & modeling**
  of problem, search space, and solutions
- **FLIP algorithm**
  to compute safe operational sequences
- **evaluation**
  including a comparison with the state of the art

We released a prototype implementation of our approach



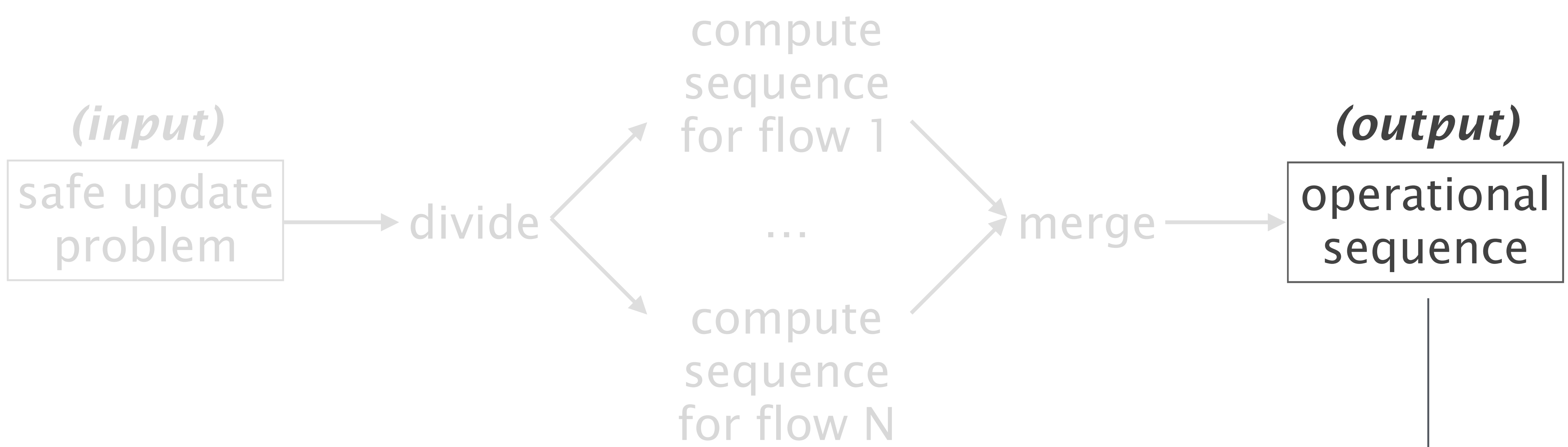
code available at http://inl.info.ucl.ac.be/softwares/flip

In our formalization, we allow complex policies…



- initial and final rules
- forwarding correctness
- policies: a flow must traverse path P1 or path P2 or … Pn

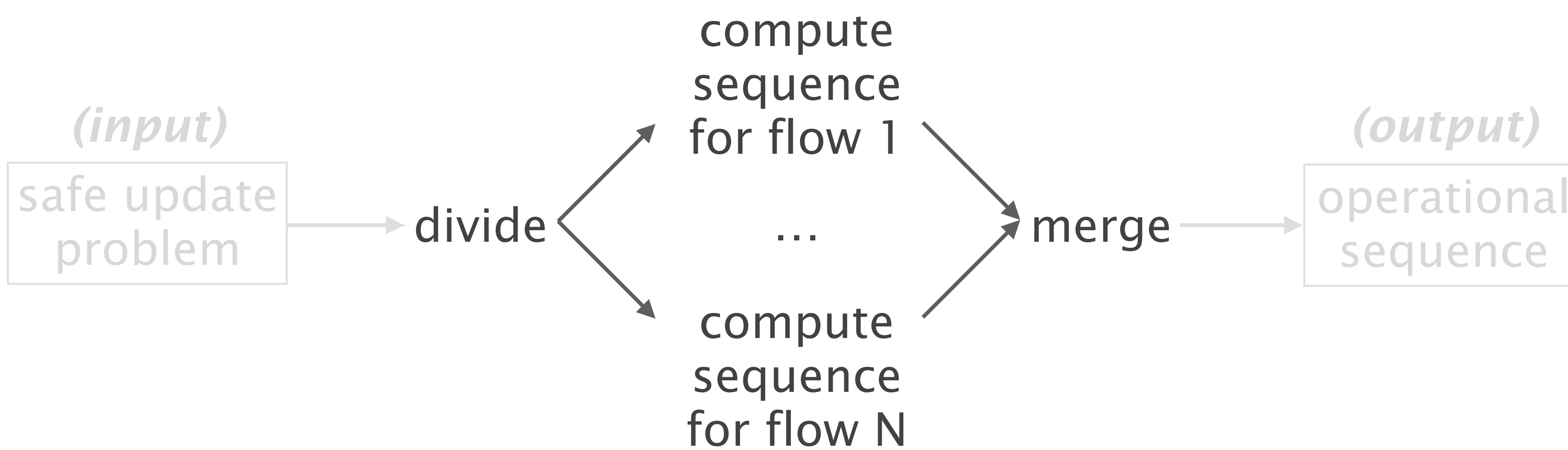
… and combinations of rule replacements and additions



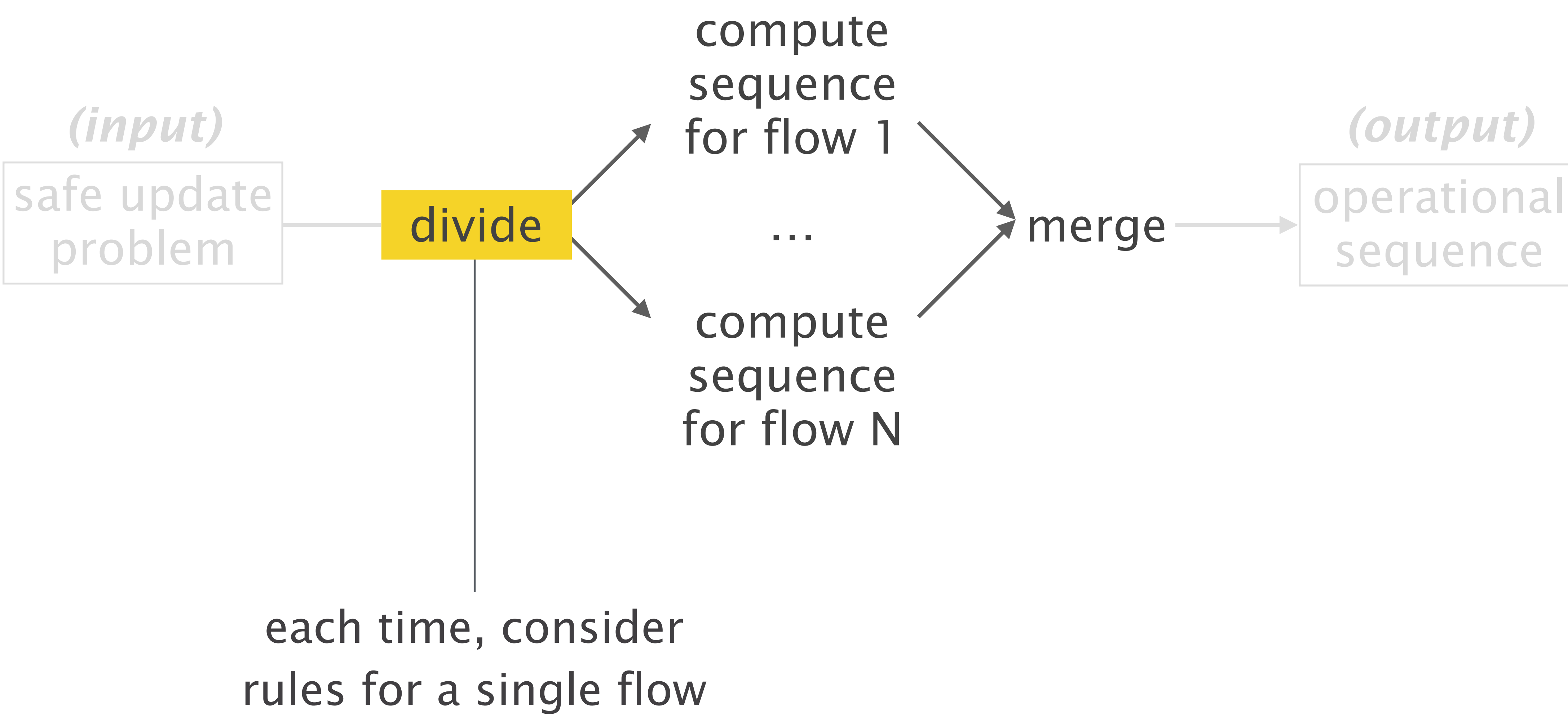
Each **step** includes replacements and additions safe to apply

- in any relative order
- before the next step

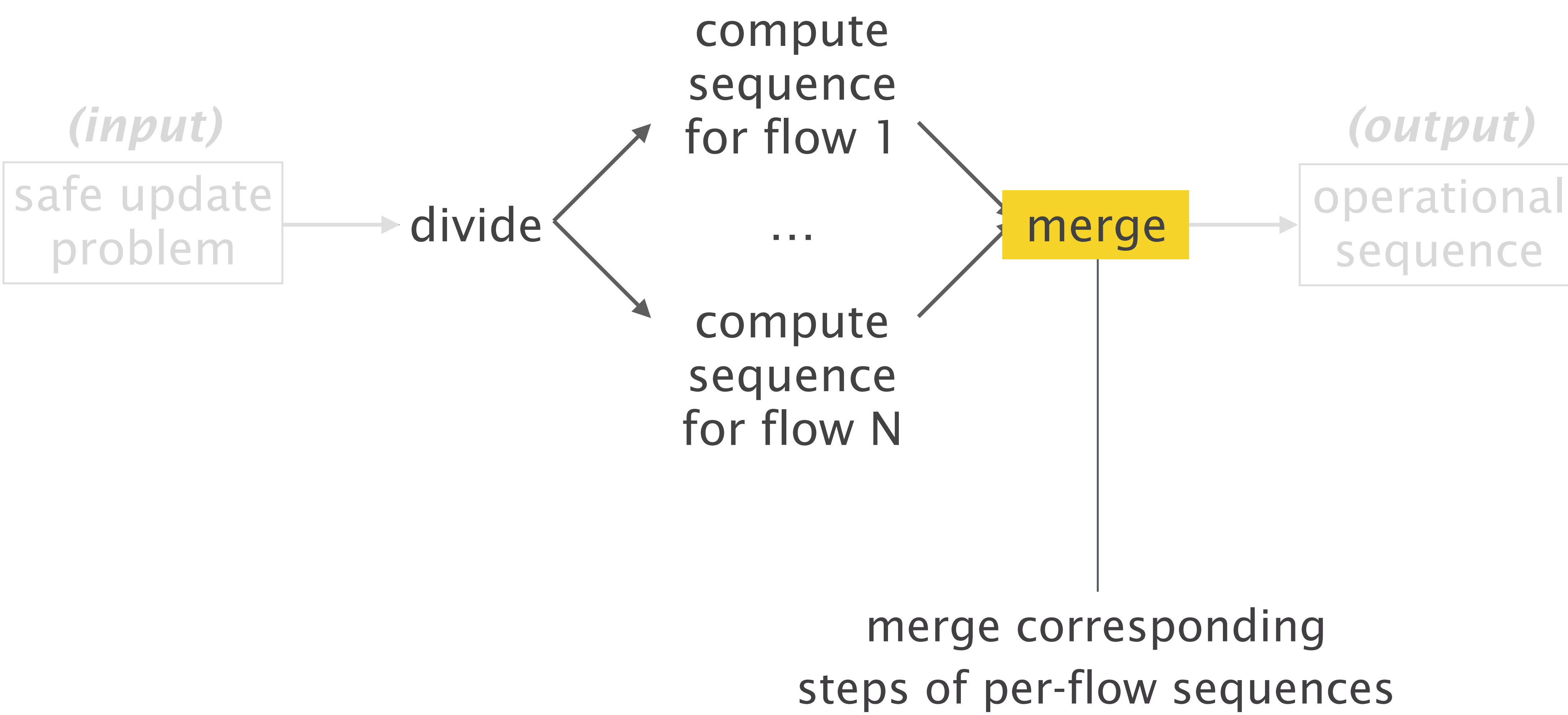
FLIP is based on a divide-and-conquer approach



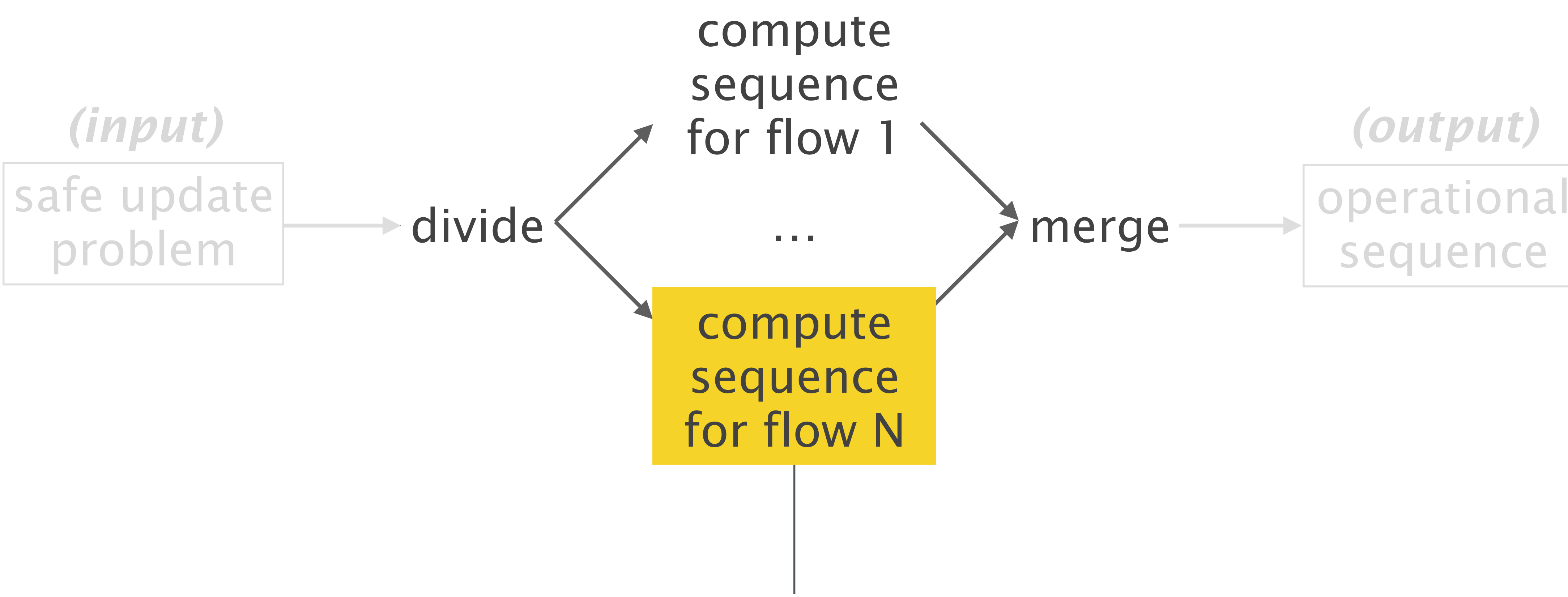
Breaking down the input problem is easy



Merging per-flow operational sequences is also easy

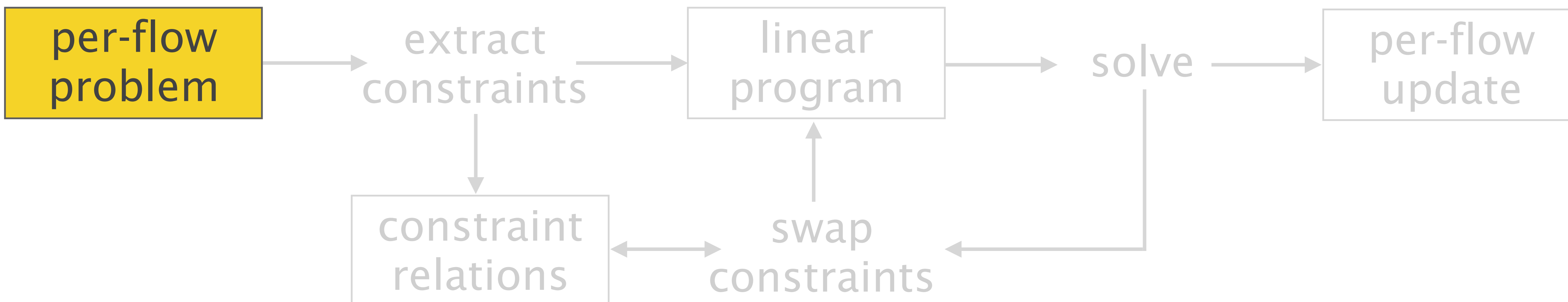


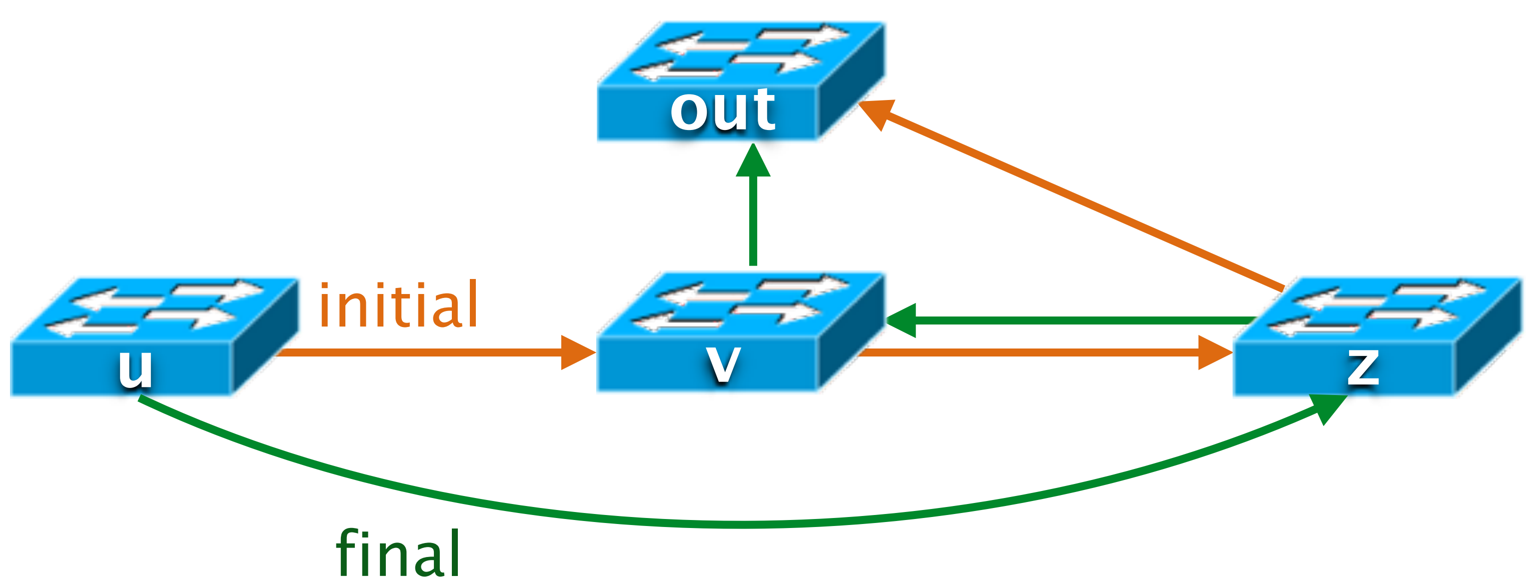
The heart of FLIP is computing per-flow sequences



- maps violations to constraints
- swaps alternative constraints
- **always** finds a satisfiable set of constraints

As an example, we now apply FLIP to our update problem scenario

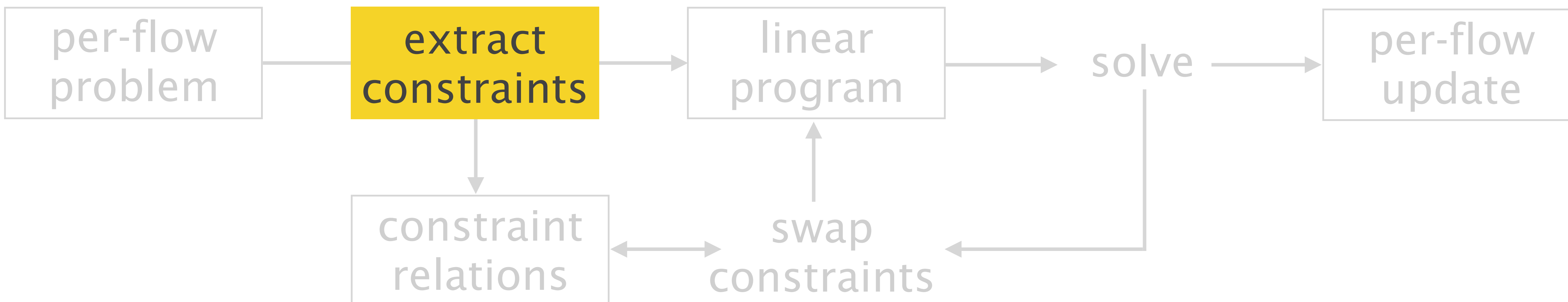




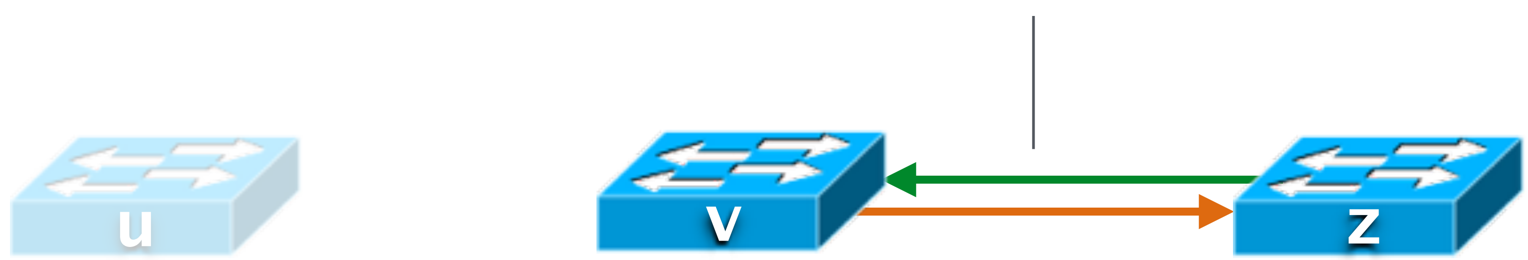
requirements

- packet delivery
- firewall traversal

For every possible requirement violation, FLIP extracts operation constraints



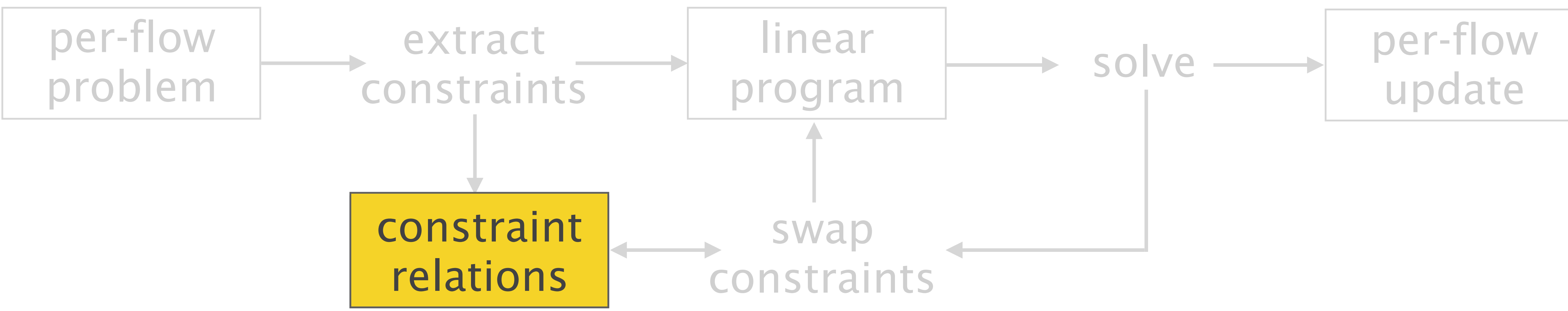
replace(v) < replace(z) OR
tag(v) & match(z) OR
tag(z) & match(v)



requirements

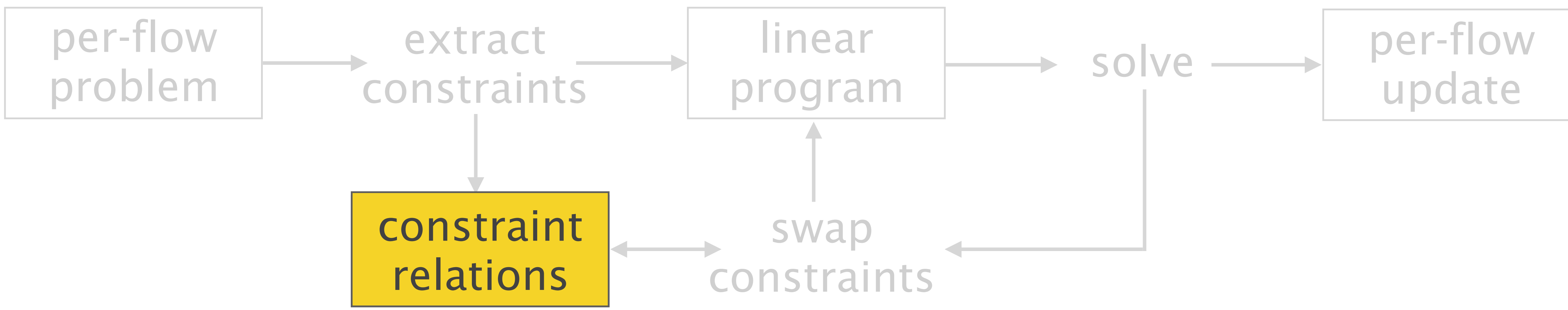
- packet delivery
- firewall traversal

The extracted constraints and their relations are stored in a table



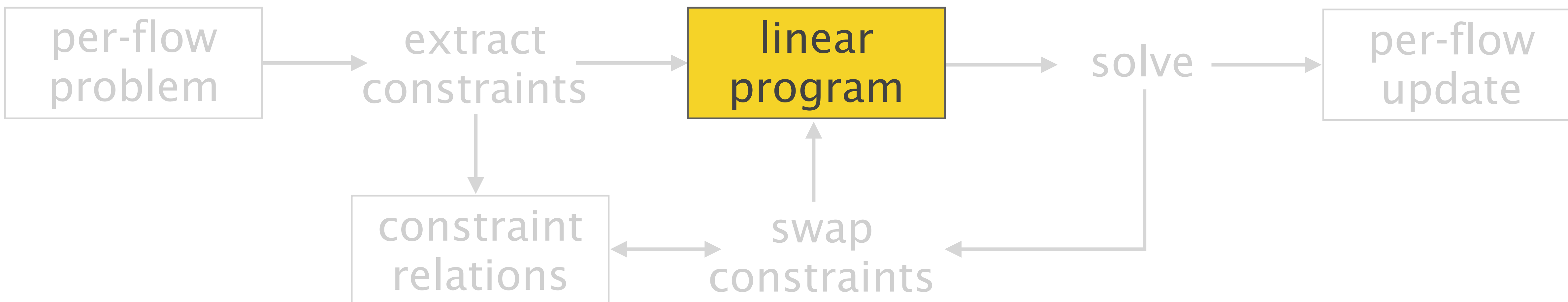
cause	active constraints	alternatives
loop v-z	replace(v) < replace(z)	match(z) match(v)

The extracted constraints and their relations are stored in a table



cause	active constraints	alternatives
loop v-z	replace(v) < replace(z)	match(z) match(v)
firewall	replace(z) < replace(u)	match(z)
firewall	replace(u) < replace(v)	match(v)

The active constraints for rule replacements are translated into a linear program



cause	active constraints	alternatives
loop v-z	replace(v) < replace(z)	match(z) match(v)
firewall	replace(z) < replace(u)	match(z)
firewall	replace(u) < replace(v)	match(v)

min $u+v+z$

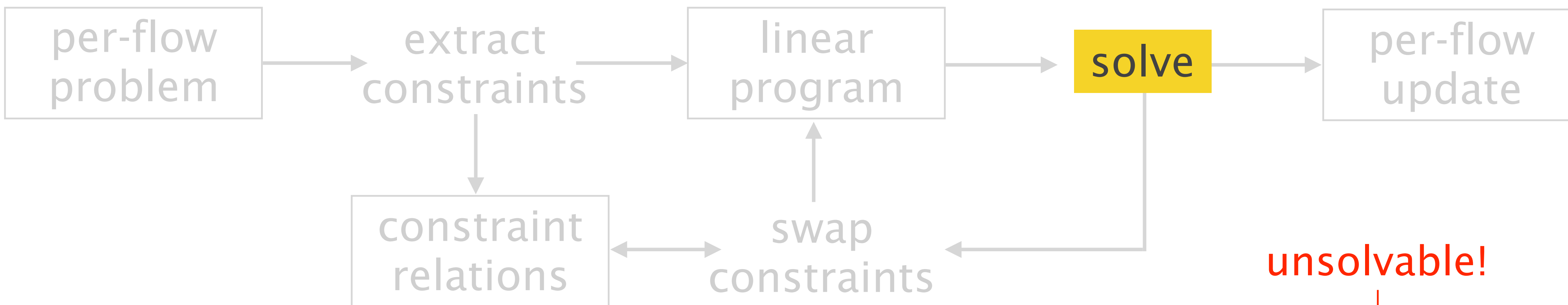
$v<z$

$u<v$

$z<u$

u,v,z integer

Then, FLIP tries to solve the generated linear program



cause	active constraints	alternatives
loop v-z	replace(v) < replace(z)	match(z) match(v)
firewall	replace(z) < replace(u)	match(z)
firewall	replace(u) < replace(v)	match(v)

unsolvable!

min u+v+z

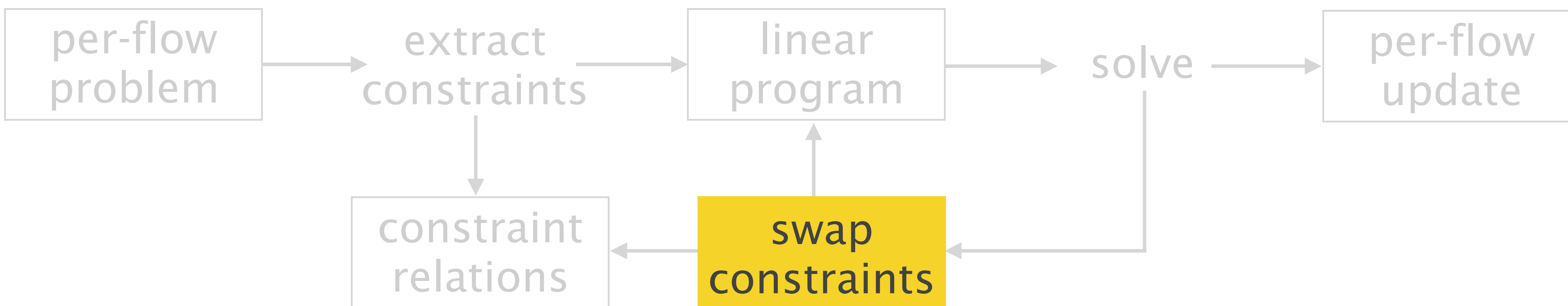
$v<z$

$u<v$

$z<u$

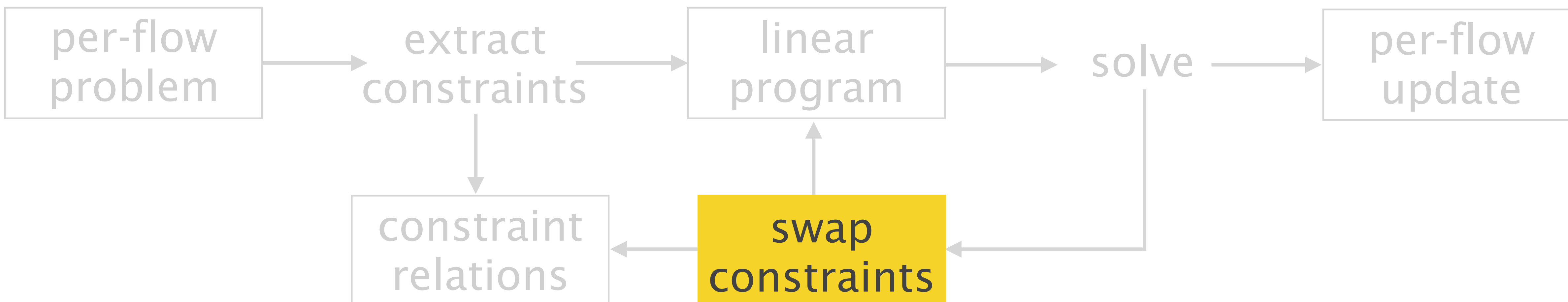
u,v,z integer

If the linear program is unsolvable, FLIP selects one constraints in a set of unsatisfiable active ones



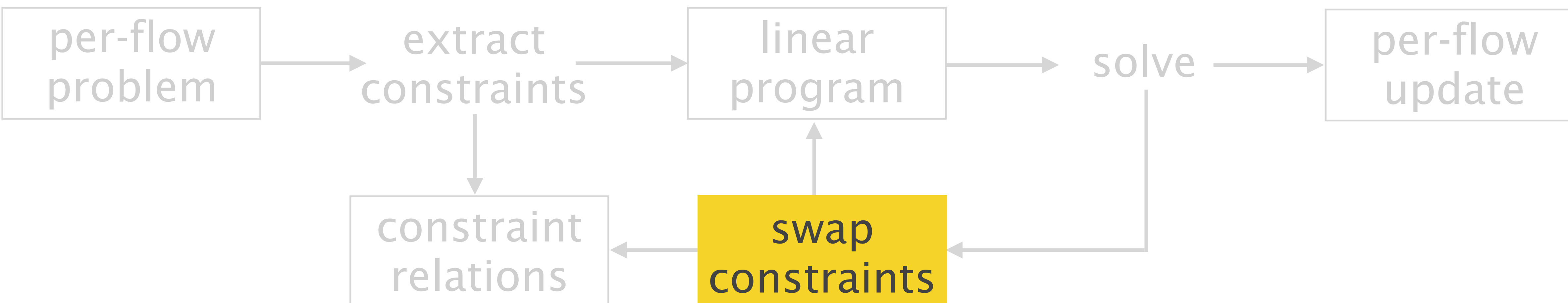
cause	active constraints	alternatives
loop v-z	replace(v) < replace(z)	match(z) match(v)
firewall	replace(z) < replace(u)	match(z)
firewall	replace(u) < replace(v)	match(v)

The selected constraint is swapped with one of its alternatives



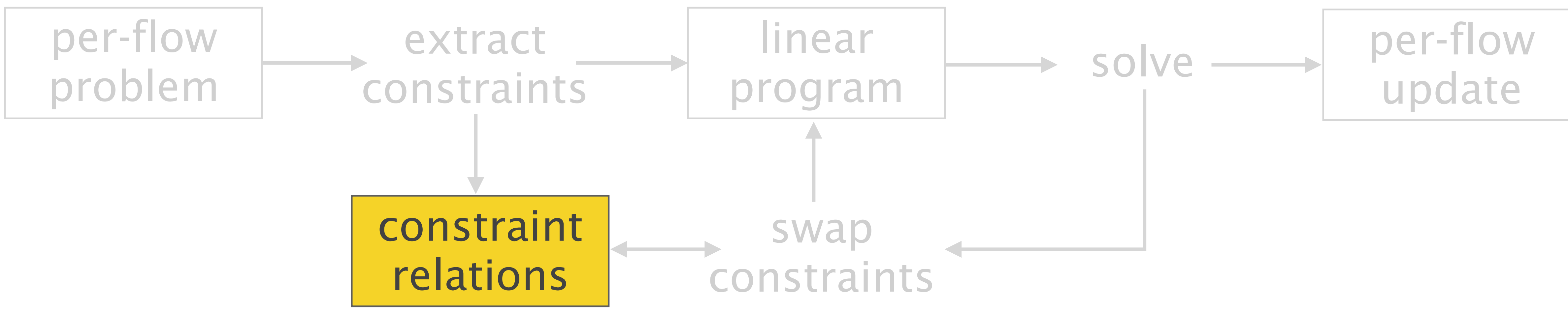
cause	active constraints	alternatives
loop v-z	~~replace(v) < replace(z)~~	**match(z)** match(v)
firewall	replace(z) < replace(u)	match(z)
firewall	replace(u) < replace(v)	match(v)

The effects of the swap are also propagated to other active constraints



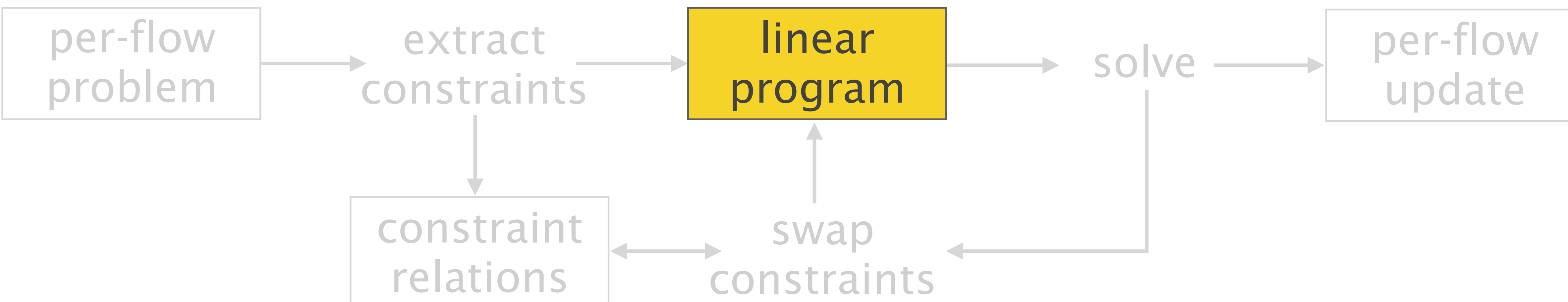
cause	active constraints	alternatives
loop v-z	~~replace(v) < replace(z)~~	**match(z)**
		match(v)
firewall	~~replace(z) < replace(u)~~	**match(z)**
firewall	replace(u) < replace(v)	match(v)

This phase leads to a new set of active constraints



cause	active constraints	alternatives
loop v-z	match(z)	
firewall	match(z)	
firewall	replace(u) < replace(v)	match(v)

In turn, the new set of active constraints is translated into a new linear program



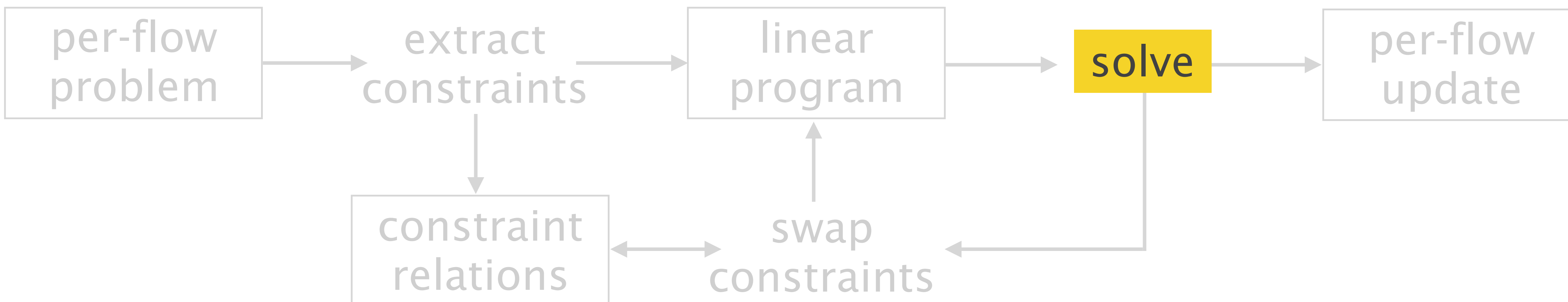
cause	active constraints	alternatives
loop v-z	match(z)	
firewall	match(z)	
firewall	replace(u) < replace(v)	match(v)

$\min u+v$

$u<v$

u,v integer

When the set of active constraints is satisfiable, a consistent sequence is generated

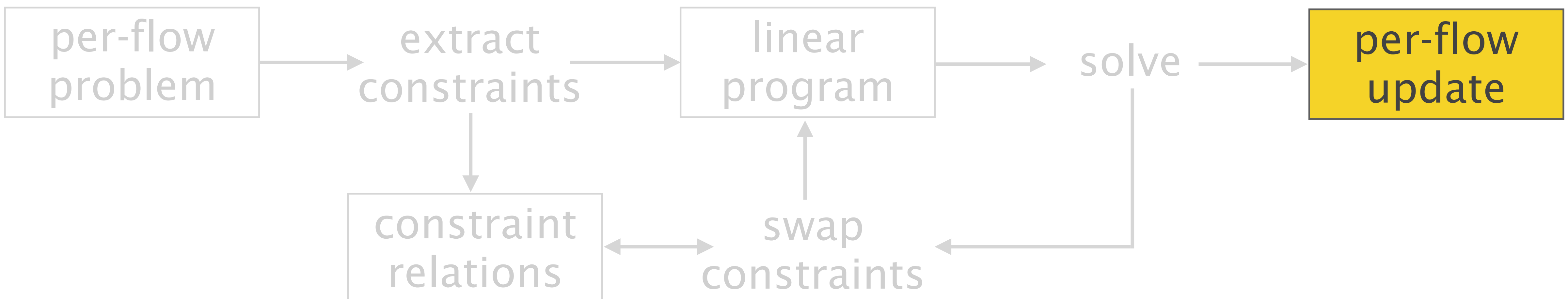


cause	active constraints	alternatives
loop v-z	match(z)	
firewall	match(z)	
firewall	replace(u) < replace(v)	match(v)

u=1

v=2

If the active constraints are satisfiable, a consistent sequence is generated



cause	active constraints	alternatives
loop v-z	match(z)	
firewall	match(z)	
firewall	replace(u) < replace(v)	match(v)

u=1

v=2

[**tag(v), match(z)**, **replace(u), replace(v)**, **replace(z)**]

FLIP also manages many algorithmic details and complications

- support for complex policies
  for middleboxing, NFV and performance

- dependency between constraints
  with propagation of constraint-swap effects

- assemble operations in one update step
  with a heuristic approach

We thoroughly evaluate FLIP with 50,000 simulations on Rocketfuel topologies

In each simulation, we randomly

- select one destination and several sources
  sources are 10% of the nodes
- significantly modify paths
  changing the weights of 80% of the links
- consider complex policies
  with sub-paths longer than 2

FLIP overcomes limitations of state-of-the-art techniques

- solves all update scenarios
  75% more than ordered-replacement techniques
- needs a few additional rules
  90% less than two-phase commit techniques
- quickly produces fast updates
  4-8 update steps computed in **sub-second** (95th perc.)

FLIP the (Flow) Table:
Fast LIghtweight Policy-preserving SDN Updates

- new model, framework and heuristics
- combine rule replacements and additions
- 75% more effective than replacement-only
- 90% more efficient than addition-only

code available at http://inl.info.ucl.ac.be/softwares/flip