SIGCOMM 2018
BUDAPEST

Trident

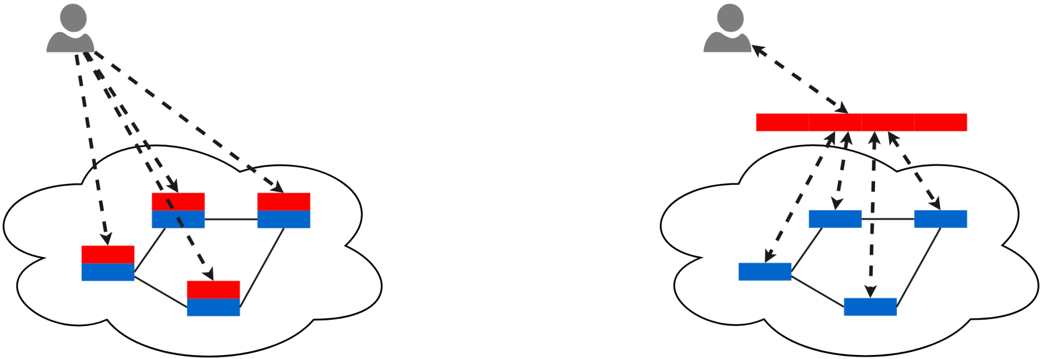
Toward a Unified SDN Programming Framework with Automatic Updates

Kai Gao[1] *Taishi Nojima*[2] *Y. Richard Yang*[2, 3]

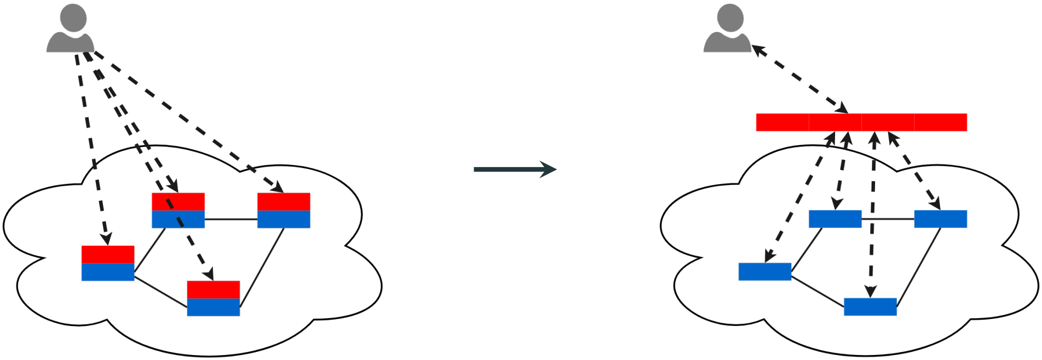
August 23, 2018

[1]Tsinghua University [2]Yale University [3]Tongji University

Software-Defined Networking

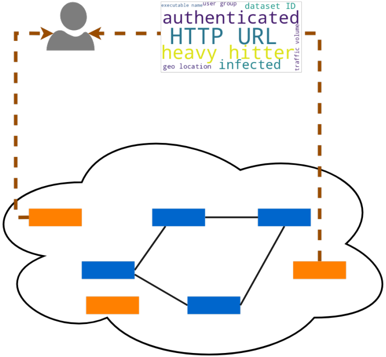


Software-Defined Networking

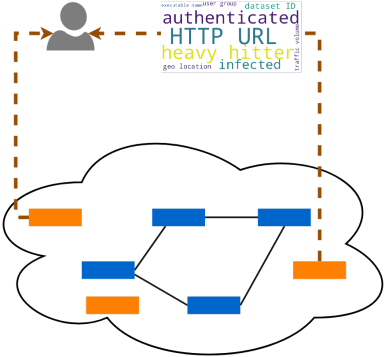


SDN simplifies network management with logically centralized network control.

Network Functions



Network Functions

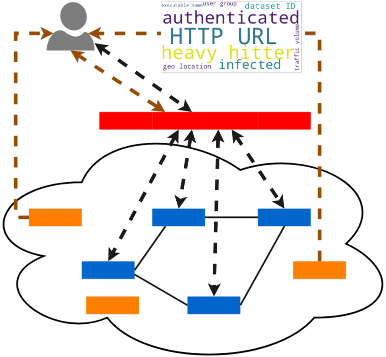


Network functions provide L7 information by extracting “state”.

Putting Them Together

Integrating the information extracted by network functions into SDN programming enables **adaptive**, **cross-layer** network control.

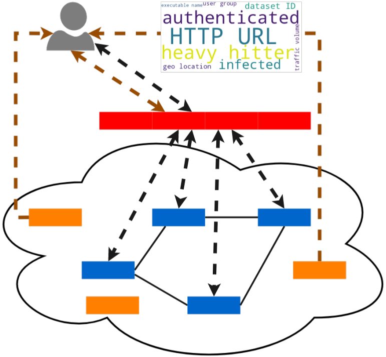
- adaptive: react dynamically to traffic
- cross-layer: control traffic based on L2-L7 information



Putting Them Together

Integrating the information extracted by network functions into SDN programming enables **adaptive**, **cross-layer** network control.

- adaptive: react dynamically to traffic
- cross-layer: control traffic based on L2-L7 information



Unified SDN Programming

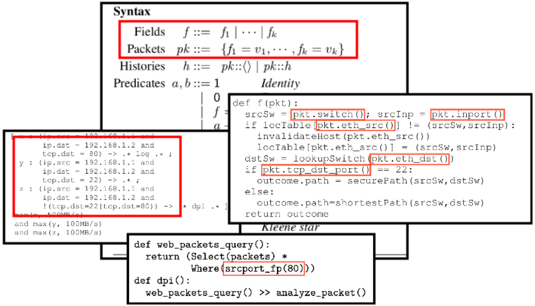
What are the design challenges of a unified SDN programming framework?

&

Why are existing SDN programming frameworks not sufficient?

C1: Integrating Network Function State into SDN Programming

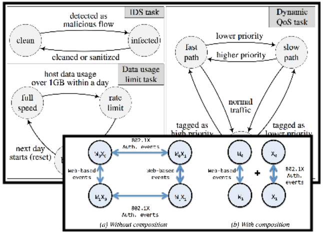
- State-of-the-art SDN programming languages support L2-L4 programming naturally **as all L2-L4 information is contained in every single packet.**



Examples from NetKAT (Anderson et al.), Frenetic (Foster et al.), Maple (Voellmy et al.) and Merlin (Soulé et al.)

C1: Integrating Network Function State into SDN Programming

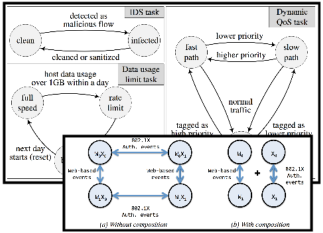
- State-of-the-art SDN programming languages support L2-L4 programming naturally as all L2-L4 information is contained in every single packet.
- Network function states are L7 which are **NOT** contained in packet header fields. They can be **unknown** and **constantly updated by finite state machines**.



Examples from Kinetic (Kim et al.) and Resonance (Kim et al.)

C1: Integrating Network Function State into SDN Programming

- State-of-the-art SDN programming languages support L2-L4 programming naturally as all L2-L4 information is contained in every single packet.
- Network function states are L7 which are **NOT** contained in packet header fields. They can be **unknown** and **constantly updated by finite state machines**.



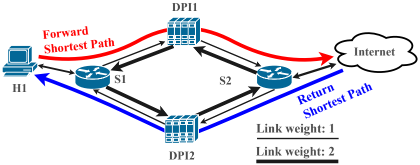
Examples from Kinetic (Kim et al.) and Resonance (Kim et al.)

We need a simple abstraction to encode L7 information in SDN programming.

C2: Constructing Consistent, Correlated Routes

- Route constructions **may be required to be correlated**: routes cannot be calculated independently.

Requirement Case 1:
The return path be the inverse of the forward path (i.e., symmetry).

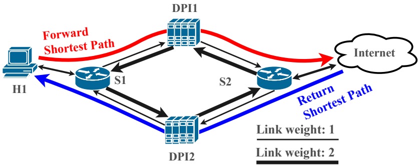


If the forward and return paths are computed independently using shortest path, the requirement will not be satisfied.

C2: Constructing Consistent, Correlated Routes

- Route constructions **may be required to be correlated**: routes cannot be calculated independently.

Requirement Case 1:
The return path be the inverse of the forward path (i.e., symmetry).



If the forward and return paths are computed independently using shortest path, the requirement will not be satisfied.

We need to systematically construct consistent, correlated routes.

High-level Programming Abstractions in Trident

Packet Selector $\xrightarrow{\textit{Binding}}$ Route Specification

C1: Encode L7 Information

C2: Systematically Construct
Consistent Correlated Routes

To address the aforementioned challenges

High-level Programming Abstractions in Trident

Packet Selector $\xrightarrow{\textit{Binding}}$ Route Specification

Stream Attributes &
3-Way/Fallback Branch

Route Sets &
Algebraic Operations

To address the aforementioned challenges, Trident introduces

- **stream attribute**, to encode a network function state as if it is a header field so that programmers can select packets based on network function states,
- **route algebra**, a simple yet flexible abstraction to systematically construct consistent, correlated routes.

High-level Programming Abstractions in Trident

Packet Selector $\xrightarrow{\textit{Binding}}$ Route Specification

Stream Attributes &
3-Way/Fallback Branch

Route Sets &
Algebraic Operations

C3: Handling Dynamicity

Network function states are dynamic

- When the state of a finite state machine for a network function changes, the corresponding route should be updated to be consistent.
- Handling dynamicity manually is complex and error-prone.

High-level Programming Abstractions in Trident

Packet Selector $\xrightarrow{\textit{Binding}}$ Route Specification

Stream Attributes &
3-Way/Fallback Branch

Route Sets &
Algebraic Operations

C3: Handling Dynamicity

Network function states are dynamic

- When the state of a finite state machine for a network function changes, the corresponding route should be updated to be consistent.
- Handling dynamicity manually is complex and error-prone.

We need to automatically handle consistent updates!

High-level Programming Abstractions in Trident

Packet Selector $\xrightarrow{\textit{Binding}}$ Route Specification

Stream Attributes &
3-Way/Fallback Branch

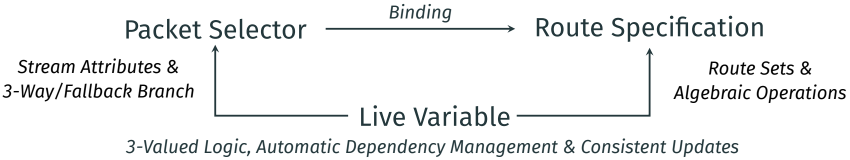
Route Sets &
Algebraic Operations

Live Variable

3-Valued Logic, Automatic Dependency Management & Consistent Updates

Trident introduces live variable abstraction

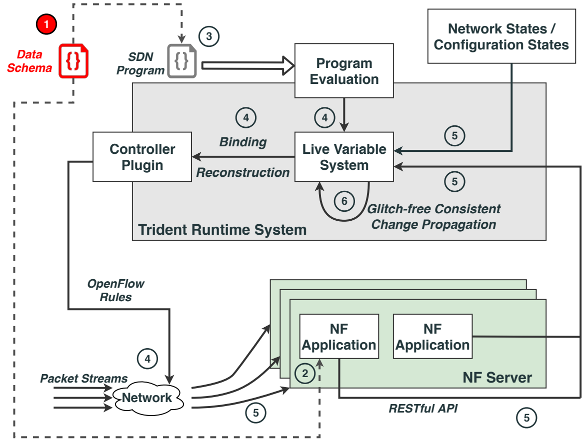
High-level Programming Abstractions in Trident



Trident introduces **live variable abstraction** to handle dynamicity of both stream attributes and route algebra.

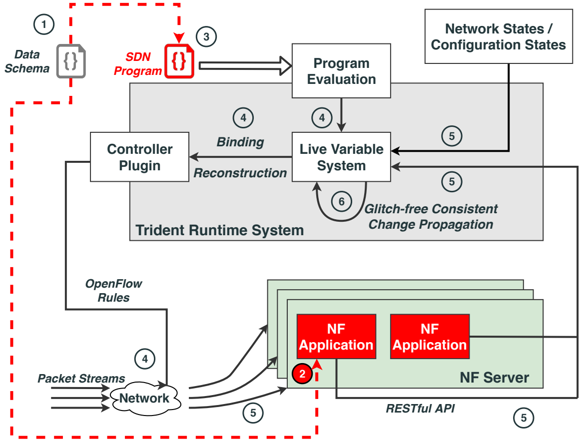
Workflow of Trident

1 Network operator & programmer specifies data schema for network function states.



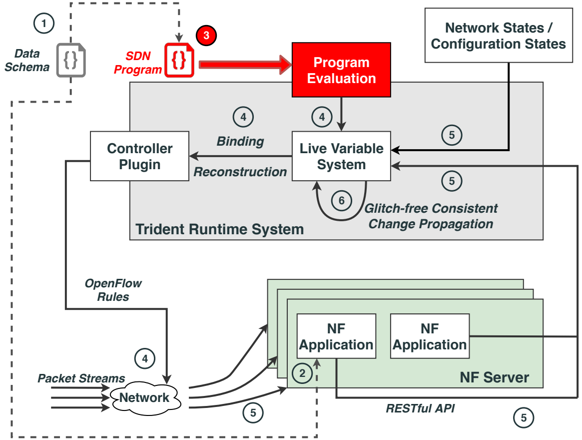
Workflow of Trident

2 Network functions implement the schema.



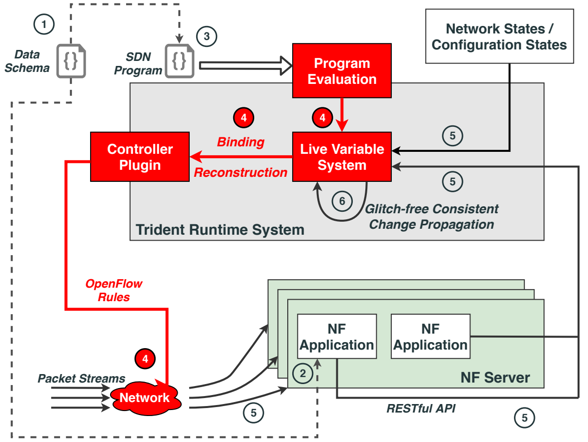
Workflow of Trident

3 Network operator submits the program to Trident.



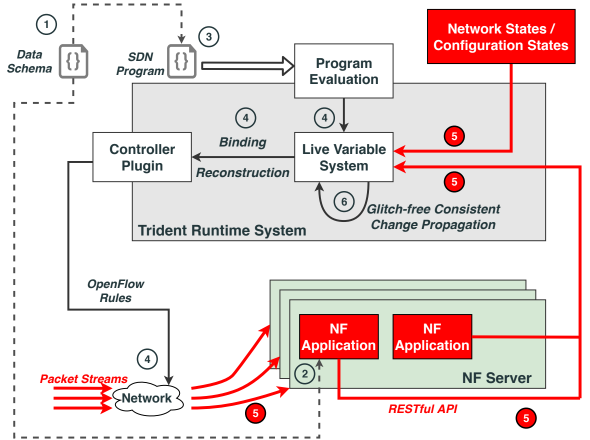
Workflow of Trident

4 Trident evaluates the program and calculates the corresponding routes.



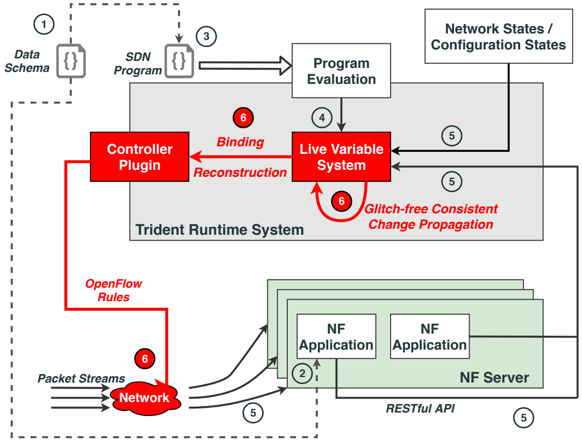
Workflow of Trident

5 A change comes: a network function updates its state, a network state changes (e.g., a link fails), or a configuration is changed (e.g., a change to an access control list).



Workflow of Trident

6 Trident automatically updates the routes for any change.



Stream Attribute: Detail

Observation: Different network function states are computed from different sets of packets.

Example

For example:

- HTTP URI: Computed from packets of the same TCP connection defined by TCP 5-tuple (e.g., `<10.0.0.2, 10.0.1.2, 1234, 80, tcp>`)
- Heavy hitter (source): Computed from packets with the same source IP address (e.g., `10.0.0.2`).

Stream Attribute: Detail

Define a stream attribute:

```
val http_uri = StreamAttribute[String]("HTTP_URI", TCP5TUPLE)
```

Stream Attribute: Detail

Define a stream attribute:

```
val http_uri = StreamAttribute[String]("HTTP_URI", TCP5TUPLE)
```

- The type information, e.g., `String`, `Int`

Stream Attribute: Detail

Define a stream attribute:

```
val http_uri = StreamAttribute[String]("HTTP_URI", TCP5TUPLE)
```

- The type information, e.g., `String`, `Int`
- A descriptive name, e.g., `HTTP_URI`, `authenticated`

Stream Attribute: Detail

Define a stream attribute:

```
val http_uri = StreamAttribute[String]("HTTP_URI", TCP5TUPLE)
```

- The type information, e.g., `String`, `Int`
- A descriptive name, e.g., `HTTP_URI`, `authenticated`
- The **stream type** (bit masks on packet header fields) specifying the set of packets to compute the network function state, e.g., `TCP5TUPLE`, `SRC_IPADDR`, `DST_IPADDR`

Stream Attribute: Detail

Use a stream attribute just like a packet header

```
pkt.http_uri, pkt.authenticated, pkt.heavy_hitter, ...
```

Stream Attribute: Detail

Use a stream attribute just like a packet header

```
pkt.http_uri, pkt.authenticated, pkt.heavy_hitter, ...
```

Stream attribute MAY have an **unknown** value.

Trident treats unknown values as valid and uses **Kleene's 3-valued logic** to select packets based on stream attribute.

Truth tables for $\wedge$ and $\vee$ in Kleene's 3-valued logic (T - True, F - False, U - Unknown)

$\wedge$	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

$\vee$	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Stream Attribute: Detail

Use a stream attribute just like a packet header

```
pkt.http_uri, pkt.authenticated, pkt.heavy_hitter, ...
```

Stream attribute MAY have an **unknown** value.

Trident treats unknown values as valid and uses **Kleene's 3-valued logic** to select packets based on stream attribute.

```
// 3-way branch
if ((pkt.authenticated) && (pkt.http_uri === "www.xyz.com")) {
  // true branch
} else {
  // else branch
} unknown {
  // unknown branch
}
```

```
// fallback branch
iff ((pkt.authenticated) && (pkt.http_uri === "www.xyz.com")) {
  // true branch
} else {
  // else and unknown branch
}
```

Route Algebra: Details

Objective: Use well-structured, declarative expressions to specify the construction of consistent, correlated routes (motivated by prior studies such as **waypoint-based routing**[1] and **relational algebra**[2]).

- The basic unit of route algebra is **route set**.
- Each route set has a **network function indicator** to specify the symmetry requirements of network functions.

[1] NetKAT (Anderson et al., POPL'14), Merlin (Soulé et al., CoNEXT'14), Propane (Beckett et al., SIGCOMM'16/PLDI'17), Genesis (Subramanian et al., POPL'17)
[2] EF Codd. "RELATIONAL COMPLETENESS OF DATA BASE SUBLANGUAGES". In: *Computer* (1972)

Route Algebra: Details

Union ($\cup$)/Intersection ($\cap$)/Difference ($\setminus$)

Given two route set Δ_1 and Δ_2, return the union/intersection/difference of Δ_1 and Δ_2:

$$\Delta_1 \cup \Delta_2 = \{r \mid r \in \Delta_1 \vee r \in \Delta_2\},$$
$$\Delta_1 \cap \Delta_2 = \{r \mid r \in \Delta_1 \wedge r \in \Delta_2\},$$
$$\Delta_1 \setminus \Delta_2 = \{r \mid r \in \Delta_1 \wedge r \notin \Delta_2\}.$$

Union ($\cup_\sim$)/Intersection ($\cap_\sim$)/Difference ($\setminus_\sim$) by Equivalence

Given two route set Δ_1 and Δ_2, return the union/intersection/difference of Δ_1 and Δ_2 using $\in_\sim$ instead of $\in$:

$$\Delta_1 \cup_\sim \Delta_2 = \{r \in \Delta_1 \cup \Delta_2 \mid r \in_\sim \Delta_1 \vee r \in_\sim \Delta_2\},$$
$$\Delta_1 \cap_\sim \Delta_2 = \{r \in \Delta_1 \cup \Delta_2 \mid r \in_\sim \Delta_1 \wedge r \in_\sim \Delta_2\},$$
$$\Delta_1 \setminus_\sim \Delta_2 = \{r \in \Delta_1 \cup \Delta_2 \mid r \in_\sim \Delta_1 \wedge r \notin_\sim \Delta_2\}.$$

Concatenation ($+$)

Given two route sets Δ_1 and Δ_2, return a new route set by concatenating all route pairs (r_1, r_2) in $\Delta_1 \times \Delta_2$ and removing the invalid ones:

$$\Delta_1 + \Delta_2 = \{r_1 + r_2 \mid r_1 \in \Delta_1, r_2 \in \Delta_2, dst_{r_1} = src_{r_2}\}.$$

Inversion ($\asymp$)

Given a route set Δ, return the inverse of $r \in \Delta$:

$$\asymp \Delta = \{\asymp r \mid r \in \Delta\}.$$

Preference ($\triangleright$)

Given two route sets Δ_1 and Δ_2, return the *preferred* route. (If there is an equivalent route in Δ_1, do not use the ones in Δ_2):

$$\Delta_1 \triangleright \Delta_2 = \{r \mid r \in \Delta_1 \vee (r \in \Delta_2 \wedge \nexists r' \in \Delta_1, r \sim r')\}.$$

Selection (σ)

Given a route set Δ and an evaluation function $f : R^* \mapsto \{0, 1\}$, return all routes in Δ that are evaluated as 1:

$$\sigma_f(\Delta) = \{r \in \Delta \mid f(r) = 1\}.$$

Optimal selection ($\diamond$)

Given one route set Δ and a routing cost function $d : R^* \mapsto \mathbb{R}$, return *any* route with the minimum value:

$$\diamond_d(\Delta) = \underset{r \in \Delta}{\arg\min}\, d(r).$$

Arbitrary selection ($*$)

Given one route set Δ, return a route set containing exactly one route r in Δ:

Please see the paper for detailed specification.

Route Algebra: Example

- Route for a flow from a host to a gateway with link capacity preference (prefers 100 Gbps over 10 Gbps):

$$* \left(\sigma_{cap=100Gbps}(H : - : GW) \triangleright \sigma_{cap=10Gbps}(H : - : GW)\right)$$

Step 1: Compute the primary route set with high link capacity.

Route Algebra: Example

- Route for a flow from a host to a gateway with link capacity preference (prefers 100 Gbps over 10 Gbps):
  - $(\sigma_{cap=100Gbps}(H : - : GW) \triangleright \sigma_{cap=10Gbps}(H : - : GW))$

Step 2: Compute the backup route set with low link capacity.

Route Algebra: Example

- Route for a flow from a host to a gateway with link capacity preference (prefers 100 Gbps over 10 Gbps):

$$* \left(\sigma_{cap=100Gbps}(H : - : GW) \triangleright \sigma_{cap=10Gbps}(H : - : GW)\right)$$

Step 3: Combine them together with the **preference** operator.

Route Algebra: Example

- Route for a flow from a host to a gateway with link capacity preference (prefers 100 Gbps over 10 Gbps):

$$* \left(\sigma_{cap=100Gbps}(H : - : GW) \triangleright \sigma_{cap=10Gbps}(H : - : GW)\right)$$

Step 4: Select only one route for unicast.

Live Variable: Details

Objective: Make dependency tracking and updates **transparent to programmers** (motivated by functional reactive programming[3]).

- Live variable is a **traceable data type** which stores the value and the computation process (i.e., dependencies and computation methods).
- Stream attribute and route algebra are just **higher abstractions** of live variables.
- The update of live variables satisfies the **glitch-free consistency**.

[3] Fran (Elliott and Hudak, IFIP'97), Dream (Margara and Salvaneschi, DEBS'14) and REScala (Drechsler et al., OOPSLA'14)

Live Variable: Details

Objective: Make dependency tracking and updates **transparent to programmers** (motivated by functional reactive programming[3]).

- Live variable is a **traceable data type** which stores the value and the computation process (i.e., dependencies and computation methods).
- Stream attribute and route algebra are just **higher abstractions** of live variables.
- The update of live variables satisfies the **glitch-free consistency**.

Glitch-freedom[4]: Intermediate consequences of a data change must not be observed.

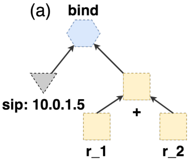
[4] Alessandro Margara and Guido Salvaneschi. "We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees". In: DEBS '14. New York, NY, USA: ACM, 2014.

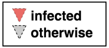
Tie Everything Together

Example Program: Block traffic for infected hosts and construct routes using concatenation

Events:

```
iff (pkt.is_endhost_infected) {
  drop(pkt)
} else {
  bind(pkt, r_1 + r_2)
}
```





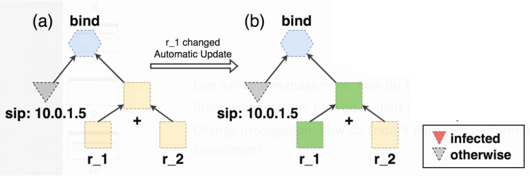
Tie Everything Together

Example Program: Block traffic for infected hosts and construct routes using concatenation

Events:

- When any network component on `r_1`/`r_2` changes, the new route (`r_1 + r_2`) is automatically recomputed

```
iff (pkt.is_endhost_infected) {
  drop(pkt)
} else {
  bind(pkt, r_1 + r_2)
}
```



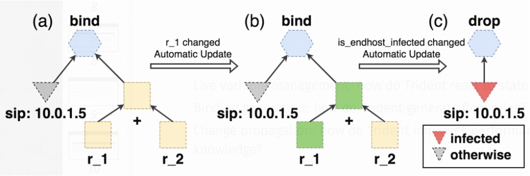
Tie Everything Together

Example Program: Block traffic for infected hosts and construct routes using concatenation

Events:

- When any network component on `r_1`/`r_2` changes, the new route (`r_1 + r_2`) is automatically recomputed
- **When the host status changes to `infected`, all packets are dropped**

```
iff (pkt.is_endhost_infected) {
  drop(pkt)
} else {
  bind(pkt, r_1 + r_2)
}
```



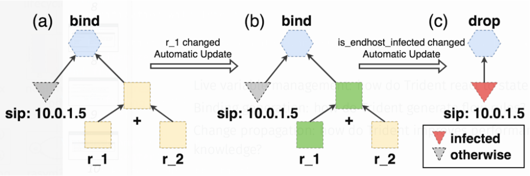
Tie Everything Together

Example Program: Block traffic for infected hosts and construct routes using concatenation

Events:

- When any network component on `r_1`/`r_2` changes, the new route (`r_1 + r_2`) is automatically recomputed
- When the host status changes to `infected`, all packets are dropped
- When the host status is cleared (e.g., through an admin interface or timeout), a route `r_1 + r_2` is automatically recomputed

```
iff (pkt.is_endhost_infected) {
  drop(pkt)
} else {
  bind(pkt, r_1 + r_2)
}
```



Implementation Highlights

Efficient update: *how does Trident achieve fast updates?*

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

Example:

```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```

- Assume a change to `G` that invalidates both `p` and `b`.
- Trident recomputes `p` and `b`.
- Trident returns `r` as soon as `p` is ready and not `Unknown`.

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

Example:

```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```

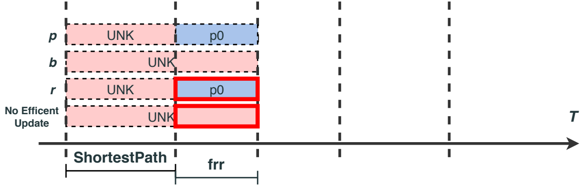
At time 0: no routes is ready

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

Example:

```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```



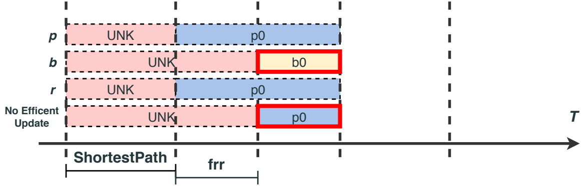
At time 1: the primary route *p0* is ready. With efficient update, Trident selects *p0* as *r*.

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

Example:

```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```



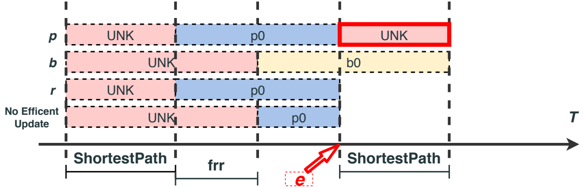
At time 2: the backup route *b*0 is also ready. The standard update will selects *p*0 as *r*.

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

Example:

```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```



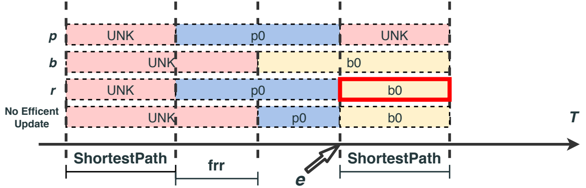
At time 3: a data change *e* happens and invalidates the primary route.

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

Example:

```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```



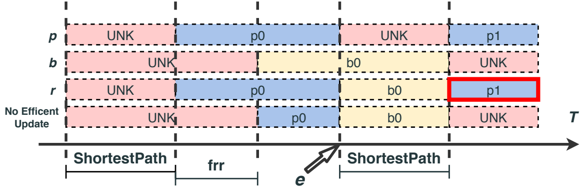
At time 3: now *p* is not ready, Trident selects *b0* as *r*.

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

Example:

```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```



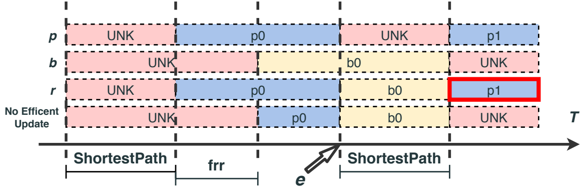
At time 4: *p* has a new value *p*1. With efficient update, Trident selects *p*1 as *r*.

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

Example:

```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```



With this simple code, Trident achieves lifecycle management for backup routes.

Efficient Update

Objective: Leverage semantics of operators to achieve fast updates.

General rule: For all route algebra operators, if the partial result has no unknown subsets, the output guarantees glitch-free consistency and we can apply efficient update.

Expr	*Known Subset*	*Unknown Subset*
$\Delta_1 \cup \Delta_2$	$\mathcal{K}_1 \cup \mathcal{K}_2$	$\mathcal{U}_1 \cup \mathcal{U}_2$
$\Delta_1 \cap \Delta_2$	$\mathcal{K}_1 \cap \mathcal{K}_2$	$(\mathcal{K}_1 \cap \mathcal{U}_2) \cup (\mathcal{U}_1 \cap \mathcal{K}_2) \cup (\mathcal{U}_1 \cap \mathcal{U}_2)$
$\Delta_1 \setminus \Delta_2$	$T_{\mathcal{U}_2=\emptyset}(\mathcal{K}_1 - \mathcal{K}_2)$	$(T_{\neg(\mathcal{U}_2=\emptyset)}(\mathcal{K}_1) \cup \mathcal{U}_1) - (\mathcal{K}_2 \cup \mathcal{U}_2)$
$\Delta_1 + \Delta_2$	$\mathcal{K}_1 + \mathcal{K}_2$	$(\mathcal{K}_1 + \mathcal{U}_2) \cup (\mathcal{U}_1 + \mathcal{K}_2) \cup (\mathcal{U}_1 + \mathcal{U}_2)$
$\asymp \Delta$	$\asymp \mathcal{K}$	$\asymp \mathcal{U}$
$\sigma_f(\Delta)$	$\sigma_f(\mathcal{K})$	$\sigma_f(\mathcal{U})$
$\diamond_d(\Delta)$	$T_{\mathcal{U}=\emptyset}(\diamond_d(\mathcal{K}))$	$\diamond_d(T_{\neg(\mathcal{U}=\emptyset)}(\diamond_d(\mathcal{K})) \cup \diamond_d(\mathcal{U}))$
$\Delta_1 \triangleright \Delta_2$	$\mathcal{K}_1 \cup T_{\mathcal{U}_1=\emptyset}(\mathcal{K}_2 - \mathcal{K}_1)$	$\mathcal{U}_1 \cup ((T_{\neg(\mathcal{U}_1=\emptyset)}(\mathcal{K}_2) \cup \mathcal{U}_2) \setminus (\mathcal{K}_1 \cup \mathcal{U}_1))$
$*\Delta$	$*\mathcal{K}$	$T_{\mathcal{K}=\emptyset}(*\mathcal{U})$

$T_\varepsilon(S)$ - the value is $S \cup \{\varepsilon\}$ if $\varepsilon = \mathit{true}$, and $\{\varepsilon\}$ otherwise.

Table 1: Known/Unknown Subsets of Route Algebra.

Evaluation

- How much effort does one need to integrate a network function into Trident?
- How useful is efficient update?

Evaluation Settings

- CPU: Intel Xeon CPU E5-2650 2.30GHz
- Memory: 64G
- OS: Fedora 26
- Network: Mininet 2.3.0d1

Modification on Network Functions

Two case studies:

- Bro: a deep packet inspection framework
- FreeRadius: an open-source Radius server

Name	*Attribute*	*Language*	*LoC (f)*	*LoC (a)*	*LoC (c)*
DPI	HTTP URL	Bro	40	2	2
FreeRadius	Auth status	DSL	0	12	0

LoC - Additional lines of code, **f** - LoC to implement the library in the given framework/language, **a** - LoC in a given NF, **c** - LoC for configuration.

Efficient Update Micro Benchmark

We demonstrate the effect of efficient update by measuring the recovery time of

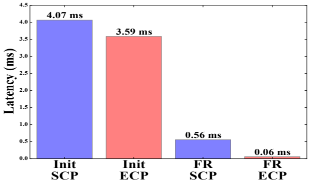
```
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```

for a single (src, dst) pair in 4 different topologies.

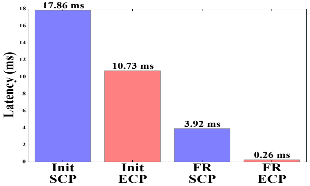
We compare the results of

- The initial computation (**Init**) v.s. fast rerouting stage (**FR**)
- The standard update method (**SCP**) v.s. the efficient update (**ECP**)

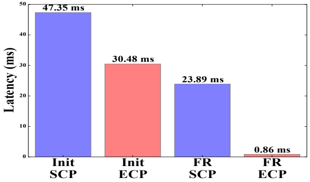
Efficient Update Micro Benchmark



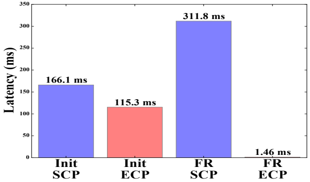
Sprint (11 nodes)



Noel (19 nodes)

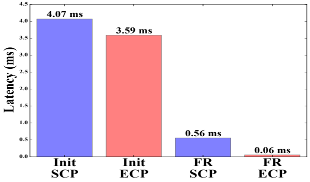


Agis (25 nodes)

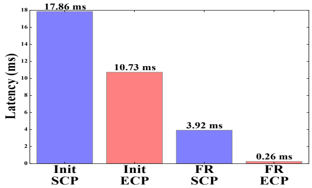


Geant (40 nodes)

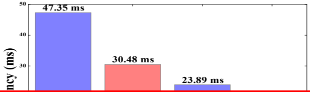
Efficient Update Micro Benchmark

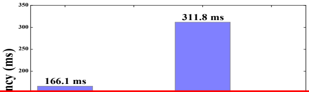


Sprint (11 nodes)



Noel (19 nodes)





Trident demonstrates an improvement of up to 39% in initial computation and an improvement of 1 to 2 magnitudes in the fast rerouting stage.

Agis (25 nodes)

Geant (40 nodes)

Summary

Trident is a unified SDN programming framework which

- uses **stream attribute** to naturally integrate network function state into logically centralized SDN programming;
- uses **route algebra**, a simple yet powerful abstraction to systematically construct consistent, correlated routes;
- uses **live variable** to achieve unified automatic data dependency management and glitch-free updates – achieving a more general **intent** networking framework.

Future directions:

- Extend from "write-only" network functions to generic network functions
- Verification with Trident

Thanks for your attention!

Q & A

Kai Gao	Taishi Nojima	Y. Richard Yang
godrickk@gmail.com	taishi.nojima@aya.yale.edu	yry@cs.yale.edu

Design Space

Basic Programming Model

SDN Programming Languages

Define behaviors as if processing **each packet** based on its attributes

- Simple programming paradigm
- Cannot handle layer-7 information naturally

Event-Driven SDN+NF Systems

Define behaviors by specifying **state** and **transitions**

- Fit well with how NF process packets
- Require *manual* efforts to identify transitions and can be complex when many NF states are involved

Can we inherit the simple paradigm of SDN programming but integrate network function states naturally?

Design Space

Route Specification

Fully Customizable Routing

The policy can execute a routing function over the graph to find the path

- Very flexible
- Complex
- May not be efficient

One-Big-Switch Abstraction

The policy only specifies the output logical port

- Simple
- Optimized by the system
- Not flexible
- Cannot express path requirements

Can we have a simple abstraction for route specification but still retain enough flexibility?

Design Space

Handling Dynamicity

Pre-defined Dependencies

The system predefines some dependencies with domain knowledge (such as failure model of network components)

- Automatic
- Simplify the job for programmers
- Hard-coded

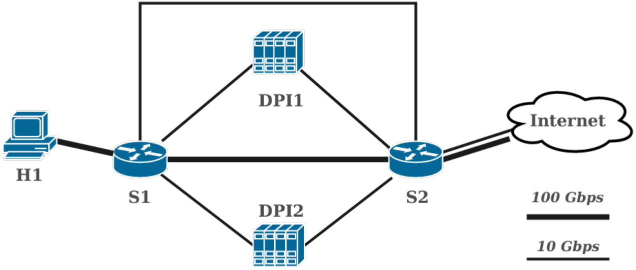
Manual Dependency Management

Programmers manually identify and handle all dependencies (for example, using the Observer pattern)

- Optimized
- Accurate
- Error-prone
- Difficult to manage and maintain

Can we achieve automatic dependency management with guaranteed correctness (i.e., data plane configuration is consistent with the control plane state)?

Motivating Example



- HTTP connections with a trusted URL **can** skip the DPI nodes
- HTTP connections with a trusted URL to a large-scale data transfer service **should** use high-bandwidth links whenever possible

Why cannot existing SDN programming languages work?

The ambiguity of “failed” predicates with network function states

- For `sip == 10.0.0.2`, the result is either true or false for all packets
- For `is_trusted(http_url)`, the result can be true, false and **unknown**

Why cannot existing SDN programming languages work?

The correlation of routes cannot be explicitly specified

- Some network functions require traffic of the same stream to be processed on the same node and must migrate to the same new instance simultaneously (e.g., TCP & HTTP state machine)
- Some routes depend on others (e.g., link protection)
- Some routes depend on certain conditions (e.g., link capacity requirement)

Why cannot existing SDN programming languages work?

The dynamicity in both packet selection and route computation is closely related

- A single NF state may affect multiple streams (for example, streams with the same heavy hitter status may have different HTTP URL values)
- Routes for multiple streams are computed together by a single routing algorithm