Procera: A Language for High-Level Reactive Network Control

Andreas Voellmy   Hyojoon Kim
Nick Feamster

Yale University, Georgia Tech, University of Maryland

August 13, 2012

Static & Dynamic Network Policies

Network operators want to implement their high-level network policies.

Static policies; constrain flow based on flow tuple and state.

- Superusers can access the network.
- Critical flows should have minimum bandwidth guaranteed.
- Guests can access the network daily between 9am and 5pm.

Dynamic policies; involve describing state change:

- Only authenticated devices can access the network and device authentications expire after 60 minutes.
- If a user's 5 day average exceeds the limit, turn off their access, permanently.

Two Approaches Available Today

General-purpose programming:

- Very expressive.
- Many details to program
- Easy to mix up code implementing high-level concepts with low-level code.

Specialized policy language, e.g. Flow Management Language (FML)

- Easy to use.
- Limited to static policies.

Today's Approaches: FML in Detail

E.g. define and allow superusers:

```
allow(Us,Hs,As,Ut,Ht,At,Prot,Req) <- superuser(Us).
superuser(todd).
superuser(michelle).
```

FML policy is *static*: it determines a function from states to flow constraints, but cannot specify what the states are or how they should change.

Procera: High-Level Reactive Network Control

Declarative language that allows users to define what the states are and how the system state changes in response to events.

Key elements:

1. Primitive events
2. Constructs for programming dynamic state; these maintain state *incrementally* in reaction to events.
3. Composition operators
4. Constructs that collect incremental changes into values such as sets, bags, and dictionaries.
5. Policy function expressed as a function of state and flow tuple and outputting flow constraints.

1. Primitive Events

The collection of primitive event streams is customizable.

For a sample application we have:

- *authEvents*: authentication events consist of (device, user) pairs.
- *usageEvents*: usage events consist of (device, usage) pairs.
- *capSetEvents*: cap settings consist of (device, usage) pairs.
- *adminResetEvents*: admin resets consist of device ids.

2. Incremental State Programming

60 second sliding window:

since 60

Input and Output is incremental, e.g.:

- Input
  - at time 0: insert a
- Output:
  - at time 0: insert a

2. Incremental State Programming

60 second sliding window:

since 60

Input and Output is incremental, e.g.:

- Input
  - at time 0: insert a
  - at time 30: insert b
- Output:
  - at time 0: insert a
  - at time 30: insert b

2. Incremental State Programming

60 second sliding window:

since 60

Input and Output is incremental, e.g.:

- Input
  - at time 0: insert a
  - at time 30: insert b
  - at time 50: insert c
- Output:
  - at time 0: insert a
  - at time 30: insert b
  - at time 50: insert c

2. Incremental State Programming

60 second sliding window:

since 60

Input and Output is incremental, e.g.:

- Input
  - at time 0: insert a
  - at time 30: insert b
  - at time 50: insert c
- Output:
  - at time 0: insert a
  - at time 30: insert b
  - at time 50: insert c
  - at time 60: delete a

2. Incremental State Programming

60 second sliding window:

since 60

Input and Output is incremental, e.g.:

- Input
  - at time 0: insert a
  - at time 30: insert b
  - at time 50: insert c
  - at time 70: insert d
- Output:
  - at time 0: insert a
  - at time 30: insert b
  - at time 50: insert c
  - at time 60: delete a
  - at time 70: insert d

2. Incremental State Programming

60 second sliding window:

since 60

Input and Output is incremental, e.g.:

- Input
  - at time 0: insert a
  - at time 30: insert b
  - at time 50: insert c
  - at time 70: insert d
- Output:
  - at time 0: insert a
  - at time 30: insert b
  - at time 50: insert c
  - at time 60: delete a
  - at time 70: insert d
  - at time 90: delete b

2. Incremental State Programming, Continued

Further incremental state operators:

- Reset on Clock: *resetWindow clockFun*
- Limit by count: *limitBy attr count*
- Filtering: *select pred*
- Projecting: *project f*
- Grouping: *groupWith op*
- Joining: *join*, *joinOn attr1 attr2*

3. Composition Operators

Operations can be composed, e.g.

$$since\ (days\ 5) \ggg limitBy\ attr\ 10$$

4. Accumulate Incremental State

Operators to collect incremental state signal into a data structure:

- *collectSequence*
- *collectBag*
- *collectSet*
- *collectTable*

5. Policy Functions

Policy function implemented as a function that references the state and outputs a *constraint*, e.g.:

> *policy overSet pkt* =
> **if** *member* (*etherSrc pkt*) *overSet*
> **then** *Deny*
> **else** *Allow*

Putting it all Together

Deny any devices whose five day usage exceeds 1000.

$$
\begin{aligned}
&\mathit{usageEvents} \\
&\quad \ggg \mathit{insertEach} \\
&\quad \ggg \mathit{since}\ (\mathit{days}\ 5) \\
&\quad \ggg \mathit{groupWith}\ \mathit{sum} \\
&\quad \ggg \mathit{select}\ (\lambda(\mathit{dev}, \mathit{usage}) \rightarrow \mathit{usage} > 1000) \\
&\quad \ggg \mathit{project}\ \mathit{fst} \\
&\quad \ggg \mathit{collectSet} \\
&\quad \ggg \mathit{pure}\ \mathit{policy}
\end{aligned}
$$

Implementation

Language designed to support efficient evaluation:

- Eliminate need to poll policy by accurately tracking the maximum amount of time until the state changes.
- Update state incrementally based on events and policy definition.
- Old events are deleted automatically when no state refers to the event anymore.

Next Steps

- Implement network controller; must be in implemented in host language that Procera is embedded in.
- Provide richer constraints, e.g. allow and encrypt, allow but avoid switch A, etc.
- Address fault tolerance: automated support for persisting controller state.
- Optimize incremental change algorithms.

Conclusions

- Procera is a language for writing dynamic network policies.
- Keeps flow constraints (as in FML) but adds ability to specify state and state changes.
- Implementation takes care of details of tracking policy change correctly and efficiently.

Questions?

andreas.voellmy@yale.edu