Dynamically Extensible Policy Server and Agent

Yasusi Kanada
Hitachi Ltd., Systems Development Laboratory

Background

■ The function of network node will be dynamically extensible.
  ◆ Software can be added/replaced by, e.g.,
    ▪ Active packets
    ▪ Java code injection
  ◆ Hardware can be added/replaced by, e.g.,
    ▪ Board addition/replacement
  ◆ Both software and hardware functions can be added/replaced on-the-fly (while the node is running).

■ Thus, policies should be dynamically extensible.
  ◆ New classes of policies should be able to be added dynamically
    ▪ if the network is controlled/managed by policies, and
    ▪ if the network function may be added dynamically.
Problem

- Conventional policy-based systems do not allow dynamic extension.
  - E.g., in COPS-PR, policies are stored in statically-specified PIBs.
    - New classes of policies require new PIB specification.
    - If standard-based, vendors must wait for PIB standardization.
    - No dynamic extension, even if non-standard PIB is used.

Solution

- The policy-extension-by-policy (PXP) method has been developed.
  - A new policy class is defined by predefined PD/PE policies in the PXP method.
    - A PD (policy definition) policy contains device-independent definitions of user-defined policy classes, and
    - A PE (policy embedding) policy contains device-dependent methods for translation of user-defined policies into device configurations.
    - PD/PE policies are meta policies.
An Architecture for the PXP Method

- **Software components**
  - User/Application interface
  - Policy manager (Policy server, PDP)
  - Policy database
  - Policy agents

- **Policy agents may be embedded in network nodes.**

Policy Deployment Process of the PXP Method

- Administrator
  - PD policy
  - PE policies

Then, policy manager can handle new classes of user-defined policies.

- Network node
  - Network node
  - Network node
  - Network node

Then, policy agent can translate new classes of user-defined policies.
Basic Policy Information-model

- A policy is a sequence of policy rules: \( P = \{ r_1, r_2, \ldots, r_n \} \).
- A policy rule consists of
  - A list of conditions: \( c_1, c_2, \ldots, c_m \)
  - A list of actions: \( a_1, a_2, \ldots, a_l \)
- A condition / an action consists of a variable and a value: \( \text{variable} = \text{value} \).
- A policy (instance) belongs to a policy class.
  - Class examples: Diffserv-edge, Access-control.
  - All the rules in a policy must have the same type of functionality.
- Example (a Diffserv-edge policy rule)
  - if (source_address = 192.168.1.1, protocol = 'tcp')
    \{ DSCP = 46; \}

Prototype Development

- Three policy classes were predefined.
  - PolicyToTelnet (an amalgame of PD & PE policies)
    - Most important
  - PolicyVariableDefinition (a PD policy)
  - PolicyValueTranslation (an amalgame of PD & PE policies)
- A PolicyToTelnet policy rule defines
  - a user-defined policy class, and
  - the method of translating a policy of this class into CLI commands.
### PE Policy

- Two essential elements of PE policy rules are
  - Command template
  - Template fillers

- A command is generated from the pattern by filling the *unfinished* portions by using template fillers.

#### Example

- Command template: `access-list %s permit %s %s %s`
- Fillers: 
  - `N + 1`, `protocol || 'ip'`, `source_address || 'any'`, `destination_address || 'any'`

- Command generation
  - Variable values: 
    - `N = 2`, `protocol = 'tcp'`, `source_address = '192.168.1.1'`, and `destination_adress = ''`
  - `access-list 3 permit tcp 192.168.1.1 any`

### PolicyToTelnet Policy and Policy Deployment

#### Condition part:

```plaintext
if (name = policy_class_name) {

```

#### Action part 1/3: Policy variable declarations

```plaintext
  condition_variables =
    {variable_name options, ...},
  action_variables =
    {variable_name options, ...},
```

#### Action part 2/3: Policy prologue/epilogue translators

```plaintext
  policy_initialization =
    {work variable = initial value, ...},
  policy_pre_deploy_commands =
    [[template, filler, filler, ...]], ...
  policy_post_deploy_commands =
    [[template, filler, filler, ...], ...],
  policy_pre_undeploy_commands =
    [[template, filler, filler, ...], ...],
  policy_post_undeploy_commands =
    [[template, filler, filler, ...], ...],
```

#### Action part 3/3: Policy rule translators

```plaintext
  rule_initialization =
    {work variable = policy bytecode program, ...},
  rule_deploy_commands =
    [[template, filler, filler, ...], ...],
  rule_undeploy_commands =
    [[template, filler, filler, ...], ... }
```

---

**Generation order**

- Policy initialization
  - Prologue
  - Rule initialization
    - Commands
  - Rule initialization
    - Commands
  - ... 
  - Rule initialization
    - Commands
  - Epilogue

---

Policy 2002  2002-6-5  Yasusi Kanada  (C) Hitachi Ltd.  9
Policy 2002  2002-6-5  Yasusi Kanada  (C) Hitachi Ltd.  10
**PolicyToTelnet Policy and Policy Undeployment**

```plaintext
# Condition part:
if (name = policy_class_name) {

# Action part 1/3: Policy variable declarations
condition_variables =
{variable_name options, ...},
action_variables =
{variable_name options, ...},

# Action part 2/3: Policy prologue/epilogue translators
policy_initialization =

\[\text{condition_variables = }\] if (name = policy_class_name) {

\[\text{action_variables = }\] if (name = policy_class_name) {

\[\text{policy_initialization = }\] if (name = policy_class_name) {

Policy initialization
Prologue

Rule initialization
Commands

Rule initialization
Commands

Rule initialization
Commands

Rule initialization
Commands

Epilogue


Conclusion

- By using the PXP method,
  - Policies with new functionality can be added/replaced by using preexisting interfaces such as CLI, MIBs, PIBs, APIs, hardware tables.
  - Policy classes can be defined by users or applications much easier.