A Representation of Network Node QoS Control Policies Using Rule-based Building Blocks

Yasusi Kanada
Central Research Laboratory, Hitachi Ltd.
Higashi-Koigakubo 1-280, Kokubunji, Tokyo 185, Japan
E-mail: kanada@crl.hitachi.co.jp

Abstract: *Network node functions, such as QoS or the security functions of routers, are becoming increasingly complex, so programs, not only configuration parameters, are required to control network nodes. In a policy-based network, a policy is defined at a policy server as a set of rules that deployed at network nodes where it must be translated into an executable program or parameters. Thus, a policy must be represented by a form in which the syntax and semantics are clearly defined, and which can be mechanically translated into an executable program. This is possible if the policy is written in an appropriate rule-based programming language. This paper describes such a language in which functions required for DiffServ can be specified for the interface between a policy server and network nodes. In this language, a policy rule can be composed using predefined primitive building blocks and control structures.*

1. Introduction

A policy server is used for controlling QoS functions in a DiffServ (differentiated services) [Ber 99] domain. The operator defines, modifies, or deletes end-to-end policies. The policy server decomposes the policies into node policy rules that classify packets and control the QoS conditions of packets, and configures routers or other network nodes in the domain by deploying or undeploying the policy rules. There are several alternatives for the interface between a policy server and network nodes. SNMP or COPS [Boy 00] can be used as a protocol between the server and the nodes. CLIs (command language interfaces) designed for human operators are also often used for a policy server. APIs can be specified for both the policy-server side and the router side. If CORBA is used to define the APIs, IIOP [OMG 98] will be used as the protocol. This paper focuses on the representation of node policies that are conveyed by these interfaces.

All these interfaces work well when the policy is fairly simple. However, policies will soon become very complicated because both the services provided by a provider and the providers themselves will be differentiated and coexist in the Internet. If we have to specify complicated policies, a problem arises. Because the *grammar* (i.e., syntax and semantics) of these interfaces is limited, the expressive power of the interfaces is far from sufficient. Thus, there are large syntactic and semantic gaps between a policy server and network nodes. A particular problem is that a complicated system should be built by using a type of building blocks, but the above interfaces do not support any building-block architecture.

The syntax of SNMP is limited because, basically, only one value at a time can be handled, i.e., get or set, by SNMP. Although bulk data can be read in SNMPv2, this is not done in a structured way: no properly means, i.e., control structures, are given. The syntax of an API is also limited because the only possible syntax is a function call. This is better than a single value syntax, but again no control structure is given.

The semantics of SNMP, COPS, or an API are also limited because no relationships between nor constraints on the values passed through these interfaces can be described. For example, sometimes two or more values must be set simultaneously, but this is not possible through SNMP. Another example is that sometimes two or more functions must be called concurrently, and this is not possible through a usual API.

Lack of grammar may be compensated by defining a protocol usage, such as COPS for provisioning [Rei 00]. However, even if the usage defines the grammar, it may be easily violated because no method is given for proving its correctness. Thus, for the sake of interoperability, the grammar must be specified explicitly, and there must be means to test the correctness. Even if a low-level protocol, such as SNMP or COPS, is standardized, it does not guarantee interoperability at higher levels.

I believe the alternative interface should be a rule-based programming *language* because a language is defined by a grammar. Because the problem is the limitation of the grammar, the grammar should be defined in the form of a language.

We also need *programming* because policy-based control is a matter of programming. Network nodes have been configured using only parameters (data) because the nodes have been simple. However, we now need programs for configuration because the node functions to be configured are much more complex in today's network nodes, e.g., QoS-ready routers. Policy rules contain something that affects the behavior of the node. Thus, they are not just data, but are programs.

The language is *rule-based* because policies are, and should be, rule-based. A policy usually consists of policy rules which are if-then rules.

In the policy information models defined by the IETF Policy Framework WG, policy rules are defined using the Common Information Model (CIM) [DMT 99]. They are defined declaratively and regarded as data (i.e., configuration parameters). However, as explained above, policies must actually be translated into executable programs in network nodes. Operations for handling policies, such as addition, modification, or deletion, can be defined in these models because the information models are object-oriented. However, these are meta-level operations, and no method for handling object-level operations (i.e., such as classification, policing, or shaping) is supplied in these models. In addition, dataflow-based computation, which is the natural representation of packet flow processing, cannot be described using the UML [OMG 99] that CIM is based on.

Defining a program declaratively and translating it into an operational program is difficult if the policy is complicated. However, such translation is possible if the program representation is properly selected. In the field of artificial intelligence, knowledge representations were extensively studied in the 1970s and 1980s, and rule-based programming languages, especially logic programming languages (such as Prolog) were developed. These languages are declarative and operational simultaneously. We can apply the results of such research.

In this paper, a building-block architecture based on a rule-based language called *SNAP* (Structured Network programming by And-Parallel language) is proposed. In this architecture, primitive building blocks are defined as rules that can be combined using several control structures. New large building blocks can also be defined using the primitive building blocks and the control structures. The syntax and semantics of the language are rich enough for our purpose, but not excessively complex. The building-block architecture is described in Section 2. The syntax and semantics of SNAP, including the definition of primitive building blocks, is explained in Section 3. Language and building block issues are discussed in Section 4.

2. A Building-Block Architecture

A model of network nodes and network domains based on building blocks are described in this section.

Network nodes usually consists of network interfaces and switches or routing processors between them. In routers with flexible functions, such as functions for DiffServ, the interfaces are programmable (i.e., they can be flexibly configured). In this model, the interface contains programmable building blocks, such as filters, policers, or schedulers, which are described in Section 3.2. The parameters in building blocks can be set, and a

program, i.e., a combination of functions, can be created by connecting the building blocks. Thus, a program can be represented by a directed acyclic graph (DAG). The order of functions and the program structure are restricted because inbound/outbound units are not general-purpose and there are orders and structures that cannot be realized by the units because router functions are vendor-dependent. This building block architecture enables various router functions to be described by using a limited number of primitive elements.

A network domain can be modeled as follows. The function of each network node is represented by connected building blocks, and they are connected each other. This connection corresponds to lines in the network (but maybe corresponds to *virtual* lines). So the network is also modeled by a DAG.

The network contains edge and core routers, and the edge routers have *edge interfaces* and both edge and core routers have *core interfaces*. In DiffServ, IP packets are classified at ingress (inbound) edge interfaces, and are marked in their DS field [Nic 98]. The value in the DS field is called the DSCP, and indicates the service class that the packet belongs to. At core interfaces, the QoS conditions of the packets are controlled according to the DSCP.

IP packets are classified by using a *classifier*. A classifier uses a set of filtering conditions, and each condition corresponds to an action. Each set of a condition and a corresponding action can be regarded as an if-then rule:

```
if (condition) action;
```

This rule works for each packet. The behavior of an interface can be specified using a set of if-then rules.

Classifiers used at ingress edge interfaces are called *MF* (multi-field) *classifiers*. An MF classifier checks mainly the following five items: the source and destination IP address, the IP protocol, and the source and destination IP port. The action taken as the result of classification is usually marking, which means assigning a DSCP to the DS field of the packets. A example policy rule for an MF classifier is:

```
if (SrcIP == 192.168.0.1 && SrcPort == 80)
    DSCP = 46;  // 46 means EF (expedited forwarding)     (a)
```

A different type of classifiers are used at core and egress edge interfaces. They are called *BA* (basic aggregate) *classifiers*. A BA classifier checks the DSCP. A resulting action may be to assign a priority to the queue used for the packets.

3. SNAP: A Stream-oriented Language for the Building-Block Approach

A programming language, SNAP, for the building-block approach is defined, and building blocks for DiffServ are shown in this section. A comparison of this approach with another rule-based one will be described by Kanada [Kan 00b].

3.1 Outline of SNAP

In this language, the building blocks are rules (predicates) used to input packet streams (usually only one) and output a packet stream. So a simple building block, bb, can be written as:

```
bb(Si, So).
```

Si is the input packet stream and So is the output packet stream. SNAP is similar to a parallel logic programming language (a committed-choice and-parallel language), such as GHC [Ued 85], Parlog [Cla 86], or Concurrent Prolog [Sha 86]. SNAP is also similar to a dataflow language [Fin 96]. The grammar must be defined using formal methods. However, they are defined in an informal way in this paper.

In general, each building block is described as:

class_name[*parameters*](*si*1, *si*2, …, *sin*, *so*).

Here, *class_name* is the class of rules, the arguments, *si*1, *si*2, …, *sin* are the input packet streams, and *so* is the output packet stream. Also, *parameters* are the parameters that define a specific rule, i.e., *class_name*[*parameters*] means a rule instance.

For example, rule *a* can be expressed in SNAP as follows.

```
filter_mark(Si, So) :–
    filter[SrcIP = 192.168.0.1, SrcPort = 80](Si, S1),
    mark[DSCP = 46](S1, So).
```

In this policy rule, filter_mark is the rule name (i.e., the name of the building block). All the names that begin with a capital letter are variables. Variables can be assigned only once in SNAP. However, they are not logical variables, as are used in logic programming languages. The first term, filter, inputs Si and passes only packets whose source IP address is 192.168.0.1 and whose source port is 80 through the packet stream S1. The second term, mark, inputs S1, marks all the packets in S1, and outputs them into the output stream So.

A conditional expression is written using a case structure:

```
or( c1 | a1 ;  c2 | a2 ;  … ;  cn | an )
```

Here, *c*1, *c*2, …, *cn* are building blocks that filter the packets, and they work as conditions. (They are called *guards* in GHC.) Building blocks *a*1, *a*2, …, *an* are actions. If stream *si* is inputted to the case structure, *si* is inputted to all the conditions. If *ci* (*i* = 1, …, *n*) outputs stream *si*, *si* is inputted to *ai*. The same packet must not be included in any two of the streams *s*1, *s*2, …, *sn*, i.e., the conditions must be exclusive.

A policy may contain meters and multiple actions that depend on the metering results. For example, the following policy, expressed in a C-like informal language, contains a meter ("average_rate <= 1Mbps") and three sequences of actions ("dscp = 46; …", "discard", and "dscp = 0; …").

```
if (SrcIP == 192.168.1.*) {
    if (average_rate <= 1Mbps) {
        dscp = 46;  queue_priority = high;
    } else {          // If the packet is out of the bandwidth limit
        discard;      //    The packet is discarded.
    };
} else {
    dscp = 0;  queue_priority = low;
};
```

In our method, the above policy can be expressed as:

```
ef_ingress(Si, So) :–   // Building block ef_ingress inputs
                        // stream Si and outputs stream So.
    or(filter[SrcIP = 192.168.1.*](Si, C1) |
          or(meter[Average_rate_max = 1Mbps](C1, P1) |
                mark[DSCP = 46](P1, M1)
           ;  otherwise(C1, P2) |  discard(P2) )
     ; otherwise(Si, Sother) |
          mark[DSCP = 0](Sother, Sother1) ),
    schedule[Algorithm = priority](M1, Sother1, So).
```

This policy can be applied by using the following expression:

```
ef_ingress(S1, S2).
```

The structure of the above rule is illustrated in **Figure 1**. In the above program, ef_ingress is the policy name. Only two alternatives are specified in the outer case structure. However, there are usually more, maybe tens of thousands of, filters as alternatives. The stream C1 is inputted to the inner case structure. The policing rule passes some of the packets. The output stream satisfies the metering condition. For example, the band-

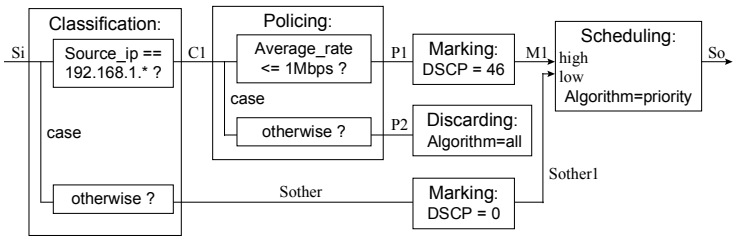

Figure 1. Connection between building-block rules

width of the output stream P1 in the above metering rule is limited to 1 Mbps. The packets in P1 are marked and put into a queue. The stream So is the output stream from the queue. The condition "otherwise" passes the packets that do not pass through any other condition. So, a packet in C1 belongs to P1 or P2, but not both. The building blocks from this example are explained in the next subsection.

3.2 Building blocks for DiffServ

Six types of primitive building blocks are defined for DiffServ: filtering, metering, marking, discarding, scheduling, and merging rules. Previous versions of these building blocks were described by Kanada [Kan 99][Kan 00a], and the detail will be described by Kanada [Kan 00b]. The order and repetition of rule applications are shown in **Figure 2**. (Case structures and merging rules are omitted here.) Metering rules can be repeated because a flow can be policed with two or more conditions, and two or more out-profile traffic streams can be handled differently. Scheduling rules can be repeated because a hierarchical scheduler is sometimes required. Use of a case structure enables a stream to fork at filtering and policing rules, and a use of a merging rule enables multiple streams to join.

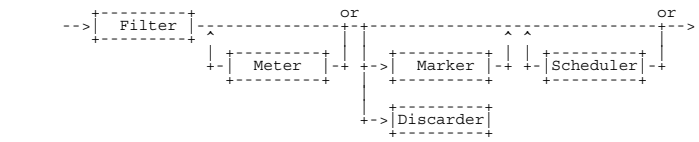

Figure 2. Possible connection between building-block rules

4. Discussions

The policy representation proposed in this paper is a rule-based dataflow representation. There are two other types of policy representations: recursive representations and non-rule-based dataflow representations. We compare these three here.

First, recursive representations and the representation in SNAP are compared. The information models developed in the IETF Policy WG are recursive representations. For example, a policy condition (the "if" part of a policy) may recursively contain another policy condition, so an arbitrarily nested condition can be described. This is powerful, but an overly complicated condition that is not understandable and cannot be implemented can be easily described. In the representation in SNAP, a complicated condition must be decomposed into two or more building blocks, which are probably more understandable and more easily implemented. In addition, the data structures in recursive representations are trees, and shared structures (i.e. DAGs) cannot be expressed using policy representations. DAGs can be expressed in SNAP. Thus, optimized conditions, such as the results of common expression elimination (i.e., the elimination of the same subconditions), can be expressed in SNAP.

If a policy action (the "then" part of a policy) can contain a policy rule recursively, which is not the case in the IETF information models, an arbitrarily complicated action can be composed from primitive actions and conditions. However, because the action part will work as a procedural program, this type of "policy rule" will not actually be a rule.

Second, non-rule-based dataflow representations and the representation in SNAP are compared. The conceptual model of Diffserv-ready routers [Ber 00], and the Diffserv MIB [Bak 99] and Diffserv PIB [Fin 00] that are based on the model, are packet-flow (dataflow) based, not rule-based. Thus, the correspondence between a rule-based policy and a traffic control block (TCB) built using the building blocks is not clear. This correspondence is clear in the representation in SNAP.

In addition, in representations such as the DiffServ MIB, a classifier consists of a *sequence* of building blocks that correspond to filtering conditions. So the evaluation order of the "policy rules" is strictly specified. This strict ordering of semantics encourages the user to write non-exclusive filtering conditions that make the program less modular, less understandable, and more difficult to parallelize.

5. Conclusion

A rule-based language, called SNAP, for the interface between a policy server and network nodes has been described. SNAP enables policies and new building blocks to be composed using existing building blocks and case structures. This architecture enables the description of policies that are interoperable and that simultaneously use various (including vendor-dependent) functions of the network nodes. Future work will include the detailed specification and implementation of SNAP on a policy server and the routers of several vendors.

Acknowledgments

I thank Takeshi Aimoto and Takeki Yazaki from Hitachi Japan for their discussion concerning lower-level router modeling, and thank Spyros Denazis from Hitachi Europe and Satoshi Yoshizawa from Hitachi America for their intensive discussion regarding the higher-level router modeling.

References

[Bak 00] F. Baker, et al, "Management Information Base for the Differentiated Services Architecture", draft-ietf-diffserv-mib-02.txt, *Internet Draft*, March 2000.

[Ber 99] Y. Bernet, et al, "A Framework for Differentiated Services", draft-ietf-diffserv-framework-02.txt, *Internet Draft*, February 1999.

[Ber 00] Y. Bernet, et al, "A Conceptual Model for Diffserv Routers", draft-ietf-diffserv-model-02.txt, *Internet Draft*, March 2000.

[Boy 00] J. Boyle, et al, "The COPS (Common Open Policy Service) Protocol", RFC 2741, January 2000.

[Cla 86] K. Clark, and S. Gregory, "PARLOG: Parallel Programming in Logic", *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 1, pp. 1–49, 1986.

[DMT 99] "Common Information Model (CIM) Specification", Distributed Management Task Force, Inc., 1999.

[Fin 96] R. Finkel, *Advanced Programming Language Design*, Addison Wesley, ISBN: 0-8053-1191-2, 1996.

[Fin 00] M. Fine, et al, "Differentiated Services Quality of Service Policy Information Base", draft-ietf-diffserv-pib-00.txt, *Internet Draft*, March 2000.

[Kan 99] Y. Kanada, et al, "SNMP-based QoS Programming Interface MIB for Routers", draft-kanada-diffserv-qospifmib-00.txt, *Internet Draft*, October 1999.

[Kan 00a] Y. Kanada, "Rule-based Modular Representation of QoS Policies", *Networking Architecture Workshop*, pp. 106-113, IEICE (The Institute of Electronics, Information and Communication Engineers), February, 2000.

[Kan 00b] Y. Kanada, "Two Rule-based Building-block Architectures for Policy-based Network Control", *2nd International Working Conference on Active Networks* (*IWAN 2000*), submitted, 2000.

[Nic 98] K. Nichols, et al, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers." RFC 2474, December 1998.

[OMG 98] "The Common Object Request Broker: Architecture and Specification", Revision 2.2, *Object Management Group, Inc.*, February 1998.

[OMG 99] "OMG Unified Modeling Language Specification", Version 1.3, *Object Management Group, Inc.*, June 1999.

[Rei 00] F. Reichmeyer, et al, "COPS Usage for Policy Provisioning", draft-ietf-rap-pr-01.txt, *Internet Draft*, October 1999.

[Sha 86] E. Shapiro, "Concurrent Prolog: A Progress Report", *IEEE Computer*, August 1986, pp. 44–59, 1986.

[Ued 85] K. Ueda, "Guarded Horn Clauses", *Logic Programming Conference '85*, pp. 225–2236, 1985. Also in *ICOT Technical Report*, TR-103, Institute for New Generation Computer Technology, 1985, and in *New Generation Computing*, Vol. 5, pp. 29-44, 1987.