Promile - A Management Architecture for Programmable Modular Routers

M. Rio, N. Pezzi, L. Zanolin, H. De Meer, W. Emmerich and C. Mascolo
Dept. of Computer Science
University College London
Gower Street, London WC1E 6BT
{ m.rio, n.pezzi, l.zanolin, h.demeer, w.emmerich, c.mascolo }@cs.ucl.ac.uk

1 Introduction

In recent years the field of Active or Programmable networks has received much attention from the networking community. The goal is to achieve flexible programmability in routers and switches. This will be particularly useful in the context of differentiated services [1] where different functionalities may or may not be present.

This document describes a novel active router architecture that provides policy management and it is completely updatable and configurable at run-time providing the possibility of changing the packet flow inside the router without significant traffic disruption. We use XML to describe the behaviour of the router.

2 Architecture Overview

The architecture seen in figure 1 represents an active router with two levels of functionality. On the lower level resides the OS kernel and on the upper level the XML based engine (described in [6]) , running in user space, which will manage and configure the lower level.

The use of XML for high level management allows flexibility defining router behaviour since XPath [2] allows the insertion or modification of the rules that manage the router. Using XML Schema [4] the behaviour grammar can be defined, checked and modified at runtime providing extra flexibility, security and easy upgrading. The existence of several off-the-shelf XML tools and related technologies is another advantage of following this approach.

Our router is built to run on different platforms with different hardware architectures. In order to accomplish this we use Java to develop the XML based engine. Assuming that all the platforms have a Java Virtual Machine the code will be portable without need for a new implementation. The use of Java also allows dynamic downloading of configuration code of the low level part of the architecture.

The XML based engine communicates with the lower level through a set of primitives provided to the user space in a special library. The manager inserts and deletes modules in the kernel and connects them according to the graph defined by the administrator. After a module is inserted in the kernel and properly connected to the graph, it can be configured in real time by the XML engine depending on the rules defined by the administrator.

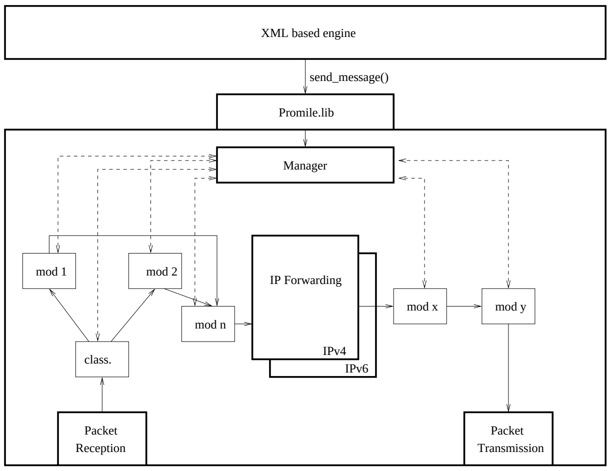

Figure 1: Architecture

The modules will typically be the usual components in a router and/or more specifically in a differentiated services capable router. Examples include classifiers, markers, droppers, shapers and schedulers.

After a packet is received it is passed to the first module (in the example a classifier) which then passes to the next module depending on the conditions expressed by the rules.

At some point the packet is passed to the forwarding engine (more than one may exist in the router) which decides which output port the packet is intended to. At each output port the packet may find different modules (a shaper or a TCP marker for example) and finally a scheduler. The system allows different ports to run different schedulers and this is configured using our graph mechanism. A different scheduler is implemented by a different module which is inserted and connected in the graph the same way as the other modules.

A similar functionality can be installed in the different input ports. Different modules can be installed only in the path of packets arriving on a specific input port (a dropper or a marker for example).

3 Module Management

Each module is defined by a set of input and output gates. The internals of the module are not important for the rest of the system. The module manager will only have to know which connections will need to be defined/established. For administration purposes each module should include metadata explaining the semantic of each gate.

When a module is loaded into memory it registers with the module manager, registering its gates and functions to update parameters.

The first thing the module manager does is to insert a module. It inserts it in memory and keeps the references to the gates in memory (as pointers to functions). When receiving a graph command it changes the pointer accordingly in order to implement the definition of the graph. Packets are implemented as structures being passed through these functions. This doesn't need to do major changes in the linux kernel since it already uses these data structures (skbuff) all over the packet forwarding process.

Upon insertion of a module it returns to user space a module reference number so that the module can be uniquely identified in future messages.

4 Graph and Rules definition

Graphs are defined by the administrator using XML as seen in the example below

```
<\?xml version="1.0" encoding="UTF-8"\?\>
<message>
 <action value="reset">
  <module name="mark">
   <outputgate number="1">
    <inputgate module="shape" number="2"/>
   </outputgate>
   <rules>
    <rule InIP="123.125.126.129"
            InPort="1269" mark="first"/>
    <rule OutIP="123.125.126.129"
            OutPort="1269" mark="second"/>
   </rules>
  </module>
 </action>
</message>
```

Here we see the configuration of the module **mark** which implements a differentiated services marker. First the gate connections are defined, linking the modules together (in this case with a shaper) building the graph.

The second part is module dependent and describes its specific rules. In this case we define how the packets should be marked (first or second class) according to their IP addresses and TCP ports.

When receiving data with information about gate connection in the graph the engine connects modules using the connect_modules() function.

connect_modules(input module,output gate, output module, input gate)

The graph manager is kept at user space in order to minimise the work and memory executed in kernel space. The only thing the module manager does is to connect the modules in memory which is something that cannot be done in user space.

5 Implementation

The system is implemented using the linux kernel 2.4.4. On a first stage we defined as a requirement that the kernel should not needed to be recompiled. The module manager is inserted as a normal module and the *promile.lib* library is implemented in user space using the **Netlink** functionality to communicate with the manager.

Modules are inserted in the kernel and linked to the packet flow using **Netfilter** (a linux kernel feature that allows code to be inserted in some points of the packet flow).

The XML engine is implemented in Java and communicates with *promile.lib* via JNI. The use of Java allows the portability of the XML engine to different platforms having the same high layer architecture in all router types. Moreover Java permits the download of more complex functions to update the rules at runtime. Performance at the higher level is not a relevant issue since policy decisions are not made often and some latency is acceptable.

The use of a "glue" library is needed because netlink requires complex data structures to be exchanged between kernel and user space.

6 An Example

Let's suppose that a particular provider wants to configure a router with the f following requirements:

- Some traffic should be dropped on entering the router
- Some traffic should be remarked on entering the router

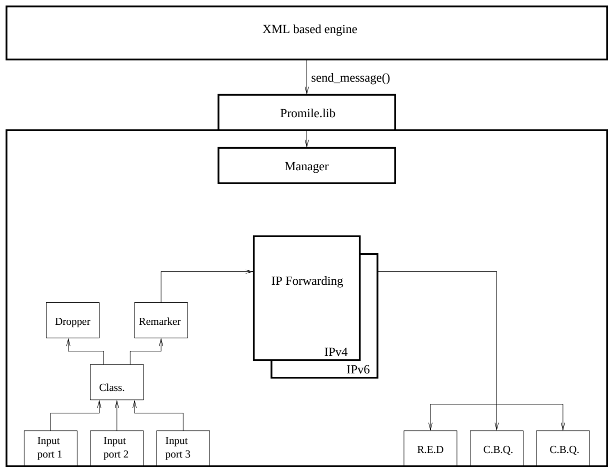

Figure 2: A simple example

- Traffic going to provider D should be scheduled using RED (the link is often congested). The others are scheduled using CBQ. RED and CBQ are two well-known packet schedulers.

Three modules are installed in the router: a classifier that selects which packets should be dropped, a dropper that just drops packets and a remarker that remarks packets according to rules specified in XML. The connections between modules are made according to the rules also specified in the XML message.

7 Related Work

An important source of inspiration for our work was MIT's Click router [7]. Click is also configured through a graph where nodes are units of router processing and edges, or connections, between two elements represent a possible path for packet transfer. Contrary to our approach Click compiles all the elements in one module that then is installed in the router. Promile approach is more dynamic.

Router Plugins [3] follows a similar modular approach to ours in that it can install and uninstall plugins at run-time. But plugins always return the packet to a PCU (Plugin Classifier Unit) which makes the implementation of a packet flow graph much more complex.

The Pronto router [5] which also uses linux gave us some ideas for the implementation.

8 Conclusions

The use of XML for high level management in our solution allows flexibility defining router behaviour since it is portable and it is a well-known markup language that is easy to create using existing application tools. Using XML Schema the behaviour grammar can be defined, checked and modified at run-time.

The choice of Java provides portability of the XML based engine and allows dynamic downloading of configuration code of the low level part of the architecture

Our architecture allows the insertion and removal at run time of modules inside the active router. It allows the modules to be connected to any place inside the kernel and it provides a uniform interface to parameterise and configure the modules at any time after they are inserted into the kernel. It uses the linux kernel without any need for recompilations. We believe that the final active router does not present significant efficiency overhead compared with a normal linux router.

References

[1] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475 - An Architecture for Differentiated Services. December 1998.

[2] J. Clark and S. DeRose. XML Path Language (XPath). Technical Report http://www.w3.org/TR/xpath, World Wide Web Consortium, November 1999.

[3] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next-Generation Routers. *IEEE/ACM transactions on Networking*, 8(1), July/August 2000.

[4] David C. Fallside. XML Schema. Technical Report http://www.w3.org/TR/xmlschema-0/, World Wide Web Consortium, April 2000.

[5] Gisli Hjalmtysson. The Pronto Platform. Technical report, AT&T Labs Research, 1999.

[6] C. Mascolo, W. Emmerich, and H. De Meer. XMILE: An XML based Approach for Programmable Networks. In *Symposium on Software Mobility and Adaptive Behaviour*. Aisb, March 2001.

[7] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. *Operating Systems Review*, 34(5):217–231, December 1999.