Middleware and Management Support for Programmable QoS-Network Architectures

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

Abstract

This paper focuses on management and middleware support for incremental updating of code and policies on various levels of granularity in time and space and on code deployment in vastly large distributed systems such as a networking infrastructure. In particular, we describe a middleware for programmable Differentiated Service (DiffServ) architectures. DiffServ architectures are envisaged by the standardization body of the IETF to be open to an evolutionary process and to facilitate adjustments to local conditions. The middleware we suggest for enabling openness and programmability of future DiffServ QoS architectures uses XML in the definition of specific policy languages, provides XML extensions to satisfy new needs and the management of deployed policies on different active nodes. Based on our middleware, router packet forwarding policies may be flexibly defined and, in an extreme case, even dynamically updated at run-time on the active nodes as a function of the network/router status. Based on a two-level language approach, the middleware decouples policy management, which is handled through XML interpretation, from packet forwarding that is implemented in more efficient languages. The two-level language approach allows a compilation of highly efficient executable code from an intermediate XML virtual machine level representation.

1 Introduction

The main and most successful characteristics of the Internet are its scalability, flexibility, efficiency and relative simplicity. The Internet provides immediate *connectivity* to users and their applications as one of its most prominent qualities. Those favorable attributes have been achieved by rigorously avoiding any application-specific state, and even the mere notion of the existence of applications, inside the network, thereby following the so-called end-to-end argument. Instead, the Internet has been designed to forward packets as fast as possible towards their destination address in a very robust networking environment that is primarily resilient to network node failures. The Internet is considered as a rather dumb network, being mostly ignorant of any potentially requested application qualities. While application qualities are achieved by exploiting programmable end systems in that framework, traditional telecommunication circuit switched networks have deployed state and intelligence inside the network to yield application qualities.

More recently, an additional need arose to support multimedia-type communication applications with real-time requirements on the Internet. For that purpose, Quality-of-Service (QoS) architectures, notably ATM, Integrated Service (IntServ) and Differentiated Services (DiffServ) [1], have been designed and partly deployed. However, it soon became clear, that there may not exist a one-size-fits-all solution to the pending problem of QoS support. Rather, whatever state would be deployed in support of some type of applications, it should be incrementally changeable or easily removable if need arises. While network flexibility has traditionally been achieved by simplicity and statelessness, a promising alternative may be given with the advent of Programmable or Active Networks. While all QoS architectures imply a more or less severe violation of the end-to-end argument, Programmable and Active Networks may not only alleviate new service deployment but also help to limit the scope in time and space of introducing application-specific state inside the network.

DiffServ is currently being standardized while assuring future flexibility and freedom, both in terms of end-to-end services to be eventually deployed and in terms of the implementation of possible per-hop-behaviours chosen by a network provider within the confines of its domain. Such a proposal calls for a vastly flexible approach in network architecting so that an evolutionary process can be explicitly supported. Supporting flexibility of DiffServ architectures may be inherently most crucial for a successful and long-lasting market penetration of DiffServ itself. Since DiffServ may likely emerge as the major QoS architecture of the future Internet, we envision programmable DiffServ architectures to be of utmost significance and use in supporting flexibility, for programmability being the most radical solution to the flexibility challenge.

In this paper we introduce a framework that uses mobile code technology to implement a middleware for programmable networks and to flexibly and incrementally encode differentiated services on routers. The proposed architecture has two layers. The first layer uses mobile code technologies to define and evolve router configurations. The second and lower layer efficiently implements an abstract routing machine. The interpreter for configurations uses and modifies state and behaviour of the abstract routing machine in order to adapt the routing behaviour. This two-tiered approach has the advantage that router configurations can be defined at high-levels of abstraction while packet forwarding in the abstract routing machine can be efficiently implemented.

We propose to use the extensible Markup Language (XML) [2] to define the configuration of virtual programmable networks. XML standardizes the way context-free grammars are defined in document type definitions (DTDs) or Schemas. Off-the-shelf tools can be used to edit documents (i.e. instances of a DTD/Schema) and to validate that they conform to these grammars. In order to define an operational semantics of an XML language, an incremental compiler needs to be built that translates constructs of the XML language into an abstract machine. The construction of such compilers is considerably simplified again by the availability of XML parsers and implementations of the Document Object Model. XML parsers, such as Apache's Xerces, read a Schema and a document and validate that the document conforms to the grammar expressed in the Schema. They return a parse tree that can be accessed and further modified using a standardized interface that is defined in the Document Object Model. What remains to be done to execute a policy written in XML is to traverse the parse tree and translate each visited node into primitives available from some appropriate abstract machine.

As described in [6], our approach also assists in distributing the same functional module, policy, or policy fragment to many different network nodes that form a network, a virtual network or a segment of a network. The approach to code mobility exploits the ability to address DOM nodes in a syntax tree using XPath expressions and the ability to modify parse trees using operations defined in the DOM. It achieves the ability to perform incremental changes to a, possibly large, number of replicated entities. We exploit this ability in order to facilitate the deployment, evolution and management of a set of co-operating routers or virtual routers.

2 Architecture Overview

The aim of this work is to define a middleware for *router management* and *policy update*. The approach is designed to support an open evolution of network architectures such as DiffServ on the middleware level. We envisage to support the introduction of new functional software modules on demand into an existing networking infrastructure, an updating and replacement of earlier deployed code and a dynamic modification of existing policies. Seemingly the latter aspect may have attracted the least attention in the Active Networking research. Although by no means restricted to the particular case, we focus our main attention in this paper on the discussion of supporting policy updates and modifications on a potentially fine-grain level of detail based on our new middleware concept.

The approach introduces a scripting-language based model for the definition of router policies. Policies specified in this language can be updated, and other policies may be added or deleted to/from the set of router policies. The grammar of the policy definition language can be modified as well in order to allow the definition of new policy constructs. This grammar can be defined by the designer and be bound to particular kind of policies that need to be expressed in the specific application context. Specific policy grammars can be used to define policy sets to configure router behaviour. The policies defined will then drive the router in the process of packet forwarding, dropping, routing and so on.

The approach we use is based on an XML based system called XMILE [7]. The essential properties of XMILE are described in detail in the next section. In brief, however, the advantages of the approach are related to the ability of dynamically modifying code at a very fine-grain level. We may use it, for example, to update policy definitions on routers, even at run-time, at the level of single policy parameter or program statements. In Section 3 we give an example of this. The need to dynamically modify the behaviour of an active network node at run-time has already been underlined by the research in the area. However, we think the ability to define application specific policy languages, and to dynamically update policies on a large number of routers or virtual routers at a very fine-grain level in an automatic way is an added value of our work.

A policy definition set can be dynamically modified as a response to some changing environmental conditions. Once a policy is modified, or added to a router policy set, router states and behaviours need also to be modified consistently. The approach we propose has the advantage of decoupling the high level policy update from the low level router behaviour update. Once the system is in place, the policy designer and manager is able to change the network node policies just using the high level policy language defined.

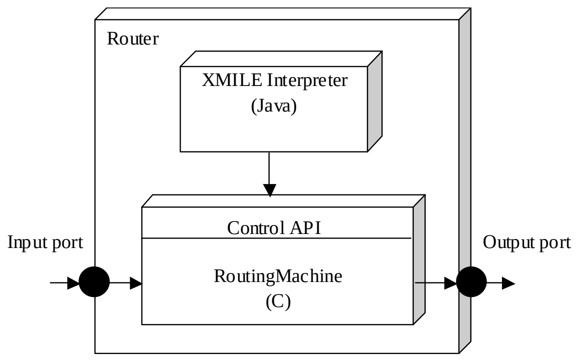

Figure 1: Programmable Router Architecture

Figure 1 shows the architecture that we propose for our approach. The high level XML policy language is independent from the implementation language (e.g. C) used for the network node forwarding behaviour. We thus achieve portability and flexibility for policy definition but at the same time do not compromise packet forwarding efficiency. Due to the ease of updates to router policies at the scripting language level remote management of policies for a set of network nodes can be achieved.

The lower layer of the router (i.e., the routing machine) may have some feedback mechanism to notify critical states to the configuration layer. The upper layer then will be able to adapt and improve the router behaviour. Although such an extreme form of flexibility may have to be handled with caution, we nevertheless consider it a valuable and novel feature for network management.

3 XML Router Configuration Management

XML [2] provides a flexible approach to describe data and document structures. We now show how XML can be used to describe routing *policies*. We will use this idea in order to have high level router programmability and run-time update in the way we described earlier.

XML Schemas define the grammar for XML documents. The structure of all the elements that can be put in an XML file is defined in a schema. Each element in an XML Schema corresponds to a production of a grammar. The complex type of an element defines the right-hand side of the production. Contents can be declared as enumerations of further elements, element sequences (i.e., `<xsd:sequence>`) or element alternatives (i.e., `<xsd:choices>`). These give the same expressive power to Schemas as BNFs have for context free grammars. Elements of XML Schemas can be attributed. These attributes can be used to store the value of identifiers, constants or static semantic information, such as symbol tables and static types. Thus, XML Schemas can be used to define the abstract syntax of programming or policy languages. We refer to documents that are instances of such Schemas as XML *programs*. XML programs can be executed; interpreters and compilers can be constructed using XML technologies. By sending XML programs or fragments of them from a management application to a (virtual) router we achieve a very fine granularity for the unit of code mobility.

As shown in Figure 1 the proposed architecture has two layers. The XMILE engine (more details in [7]) that we use at the upper level also relies on Java class loading for the fetching of the behavioural classes needed to implement the policies written in XML. The interpreter executes a Java class for each XML tag encountered. The Java classes then take care of the interaction and management of the lower level router. More details on the underlying routing machine structure will be given in Section 4.

Concerns that have to be programmed in routers, such as a configuration of virtual private networks or security policies, are defined in a high-level mark-up language, which is implemented using XML technologies. Editing support for these languages is available off-the-shelf in the form of XML Editors, such as XML Spy or Microsoft's XML Pad. These editors are akin to the syntax-directed editors contained in programming environments and enforce well-formedness; they also check whether programs and policies are valid.

Once an XML policy is written, it can be distributed to different routers to be executed by an XML interpreter for the mark-up language. During this execution, the interpreter accesses the low-level API of the packet forwarding module of the router. Using this API, the interpreter modifies local information that influences how the packet forwarding is performed.

We use an example in the DiffServ scenario to demonstrate this idea. Consider a simple set of DiffServ policies for forwarding and processing of packets on a set of related network nodes. In Figure 2 we can see an example, using XML, of the definition of policies for a DiffServ router. It contains rules for marking packets, dropping packets, shape flows and allocate bandwidth. Figure 2 defines the operations that the lower level routing machine should execute. The interpreter embedded in the engine executes the policy each time an update occurs. Every tag of the XML policy (e.g., Drop in Figure 2) corresponds to a Java class which is executed by the interpreter (if the class is not on site the interpreter fetches it with Java class loading). This class identifies one or several functions of the packet forwarding engine that execute the policy. Then, a pointer to this function is inserted into the list of functions to be used by the forwarding process, and the parameters are passed for the call. The Mark tag is interpreted by the XML engine and adds a specific marking function into the routing machine at the lower level (see Figure 2) with the parameters specified in the XML tag (i.e. the source address of the packet and the source port). The Drop tag works similarly. Shaping is done here on traffic aggregates belonging to a specific application (in the example identified by port 2028) and a token bucket shaping policy is used on the aggregate (specified by a token rate r and a bucket depth b).

In the example, IP source address and TCP port are checked and taken into consideration to establish which class the packets belong to. For the core routers we may specify the bandwidth allocation to each class. This allocation parameters affect the packet scheduling process inside the router since they configure the scheduling module (see section 4).

The program in Figure 2 is written in an XML based language whose syntax is defined by a Schema, an excerpt of which is shown in Figure 3. The root of the grammar in Figure 3 is the definition of the `policies` element, which contains the different tags for policies definition.

```
<?xml version="1.0"?>
<policies>
     <Mark whenSourceAddress="128.16.5.102" whenDestPort="2028"
          DSCP="001010">
     <Mark whenSourceAddress="128.16.5.102" DSCP="001100">
     <Mark whenSourceAddress="128.16.6.206" DSCP="000000">
     <Drop whenDSCP="100110" avgLen>"70">
     <Drop whenDSCP="001010" avgLen>"95">
     <Shape whenSourcePort="2028" r="250Kbs" b="1000b">
     <Allocate whenDSCP="000000" outport="1" bandwidth="10%">
     <Allocate whenDSCP="001010" outport="1" bandwidth="30%">
</policies>
```

Figure 2: XML code for router policies definition.

```
 <xsd:element name="policies">
  <xsd:complexType>
   <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element ref="Mark"/>
    <xsd:element ref="Drop"/>
    <xsd:element ref="Shape"/>
     ...
   </xsd:choice>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="Mark">
  <xsd:complexType content="elementOnly">
   <xsd:attribute name="whenSourceAddress" use="required"/>
   <xsd:attribute name="DSCP" use="required"/>
   <xsd:attribute name="whenDestPort" use="optional"/>
  </xsd:complexType>
 ...
</xsd:schema>
```

Figure 3: XML schema for policy definitions.

The XMILE approach [7] allows us to update the policies on routers by transferring fragments of new code from a server to (virtual) routers and then dynamically patching the original code. We refer to such code fragments as XML *policy increments*. Hence, we can specify complete programs as well as arbitrarily fine-grained increments in XML. The XML increments will be shipped together with some information on how to modify the remote XML program. For instance, let us assume that we want to update the XML code presented in Figure 2 adding a marking function for a specific class of packets (Figure 4). The policy increment that we need to update (Figure 4) can be sent separately without the need to re-send the complete program.

The XML program's structure makes the dynamic manipulation of the code a lot easier. An XML policy can be seen as a tree and the DOM API provides operations for the navigation and modification (adding/deleting/changing) of branches of the policy tree. The addressing of the particular branch that

```
<policies>
 <Mark whenSourceAddress="128.16.6.206" DSCP="001101">
</policies>
```

Figure 4: XML policy increment.

needs to be modified is performed using the XPath language. Going back to our example, Figure 5 shows the XPath expression addressing the point where the new increment needs to be added in the policy.

```
xpath="/policies"
```

Figure 5: XPath expression addressing insertion point.

XMILE was conceived with the requirement that code updates need to be performed on-the-fly. We exploit the DOM tree structure to determine when and how the updates and the code interpretation may be interleaved.

```
 <xsd:element name="Remark">
  <xsd:complexType content="elementOnly">
   <xsd:attribute name="whenFromNtw"
                  use="required"/>
   <xsd:attribute name="oldDSCP"
                  use="required"/>
   <xsd:attribute name="newDSCP"
                  use="required"/>
  </xsd:complexType>
 </xsd:element>
```

Figure 6: XML Schema update code.

XML Schemas are themselves XML files. This allows us to dynamically modify the grammar of an XML policy language. In the above example, we did not have to change the grammar and assumed that our XML policy language already has a `Mark` operator that we then used in the XML code increment. If we, however, want to modify the language, for example by adding a "remarking" operator (Figure 6) so that in a subsequent update we can also include a remarking tag to the policy, we could transmit a new schema with a specific XPath expression to define the update point in the schema.

4 Router Implementation

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.

The modules will typically be the usual components in a router like the forwarding engine (responsible for calculating the routes) and/or more specifically in a differentiated services capable router. Examples include classifiers, markers, droppers, shapers and schedulers.

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. First a module manager is inserted in the kernel which is then responsible for connecting the c modules and to pass the policies from the upper level to the corresponding modules.

5 Conclusions

The XMILE approach supports a dynamic update of code and policies at different levels. As explained in Section 2 the XML policy file can be updated adding, modifying and deleting single lines or parameters. Also, the grammar itself can be updated using XMILE. This would facilitate the use of different policies in the policy definition files used to control (virtual) routers. It is also possible to have dynamic update of the functions used by routers at the packet forwarding level. Moreover, the higher level component (i.e., the

XMILE engine) is able to react to environmental changes notified by the routing machine at the lower level and change the policies consistently, in order to affect router behaviours at run-time.

The flexibility of the XMILE approach also support the update of multiple routers from a common source: given the XML tree structure defined for every policy file on the remote routers, a global manager could deploy the same update strategies on all the routers, or could vary the strategy depending of the different local network states.

In general, much progress has been achieved to introduce programmability on the network level by the main proponents, the DARPA Active Networks research group and the IEEE PIN1520 standardization committee [3]. Much less attention, however, has been paid to middleware support and programming languages supporting network programmability. In particular, the shipping of mobile code to remote network nodes in a *manageable* fashion constitutes a major remaining challenge. The challenge is mainly due to the fact that network services generally emerge as distributed algorithms in excessively large systems.

For the kernel implementation, an important source of inspiration for our work was MIT's Click router [8]. 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 [4] 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.

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] Y. Bernet, S. Blake, D. Grossman, and A. Smith. A Conceptual Model for Diffserv Routers. http://www.ietf.org/internet-drafts/draft-ietf-diffserv-model-04.txt, 2000.

[2] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.

[3] A. Campbell, H. De Meer, M. Kounavis, K. Miki, J. Vicente, and D. Villela. A survey of programmable networks. *ACM Computer Communications Review*, April 1999.

[4] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Route plugins: A software architecture for next-generation routers. *IEEE/ACM transactions on Networking*, 8(1), July/August 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] C. Mascolo, L. Zanolin, and W. Emmerich. XMILE: an XML based Approach for Incremental Code Mobility and Update. 2001. Submitted for Journal Publication.

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