XMILE: An XML based Approach for Programmable Networks

Cecilia Mascolo, Wolfgang Emmerich, Hermann De Meer
Dept. of Computer Science
University College London
Gower Street, London WC1E 6BT
{c.mascolo | w.emmerich | h.demeer}@cs.ucl.ac.uk

Abstract

In this paper we describe an XML-based platform for dynamic active node policy updates. XML supports the definition of specific policy languages, their extension to satisfy new needs and the management of deployed policies on different active nodes. We show an example of the management of router packet forwarding policies where the XML policies that drive the packet routing are updated at run-time on the active nodes depending on the network status. The platform decouples policy management, which is handled through XML interpretation, from packet forwarding that, for performance reasons has to be implemented in more efficient languages.

1 Introduction

The tremendous success of the Internet as the dominating data communication network is largely due to 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) [Bernet et al. (2000)], 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, 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-behaviors 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 therefore 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.

The contribution of this paper is a presentation of an architecture that uses mobile code technology to implement programmable network routers. We use this architecture to implement DiffServ on programmable routers. The proposed architecture has two layers. The first layer uses mobile code technologies to define and evolve router configurations. The second layer provides a low-level machine that efficiently implements an abstract routing machine. The interpreter for configurations uses 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.

The structure of the paper is as follows: We first 2 discuss literature that is related to our work. In Section 3, we outline our approach and the application of it to active network context. In Section 5 we describe the XMILE platform and we introduce an example of use for programming of routers. Section 4 describes the architecture the programmable network nodes on which XMILE is deployed. Section 6 evaluates the approach and discusses related work and Section 7 concludes and lists possible future work.

2 Related Work

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 [Campbell et al. (1999b)]. 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. Most work on network programmability has been limited to enable programmability of single network nodes, such as router plug-ins [Decasper et al. (2000)]. While the Genesis project does focus on the management of programmable virtual networks as a whole, language support in particular has not been incorporated [Campbell et al. (1999a)]. Also related is the NetScript project at Columbia University, providing middleware support for programmable routers and switches [da Silva and Yemini (1999)]. NetScript programs, however, are packaged as mobile agents to be dispatched to network nodes for dynamic execution. Such an approach is limited in granularity and scope as discussed further below in this section. PLAN and SafetyNet [Kakkar et al. (1999); Wakeman et al. (2000)], on the other hand, are strongly typed functional and object-oriented programming languages, respectively, for providing security and safety in strong Active Networking, where a single packet may contain code that could be dynamically executed to customize individual network nodes. A seamless upgrading of network services or a partial non-disruptive removal of existing services appears as a rather compelling task without appropriate middleware support.

In this paper, we propose to use the eXtensible Markup Language (XML) [Bray et al. (1998)] to define the configuration of virtual programmable networks. XML standardizes the way context-free grammars are defined in document type definitions (DTDs) or Schemas [Fallside (2000)]. 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 interpreter needs to be built that translates constructs of the XML language into an abstract machine. The construction of such interpreters 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 is conform 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 (DOM)[Apparao et al. (1998)]. 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 in described in [Emmerich et al. (2000)] and [Mascolo et al. (2000)], our approach (XMILE) also assists in distributing the same policy or policy fragment to many different interpreters. The XMILE 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 policy copies. We exploit this ability in order to facilitate the deployment, evolution and management of the configurations of routers, which exist on many different network routers. The focus of our approach is on the reconfiguration of a high number of routers which all together contribute to the routing of packets on the network. We are aware of the developed approaches both in policy management and distribution [Sloman and Lupu (1999)], and in single router reconfiguration [Kohler et al. (2000)]. Our aim is to merge the aims of those existing work, using XML based technologies for the policies definition and update, and targeting the reconfiguration not only of single routers but of groups of routers. Some work has been carried out also in the context of programmable network languages. The advantage of using XML as meta language for the definition of programming network languages with respect to approach such as in [Wakeman et al. (2000); Kakkar et al. (1999)], is the ability to extend and evolve the language easily, offering an additional degree of flexibility. In NetScript

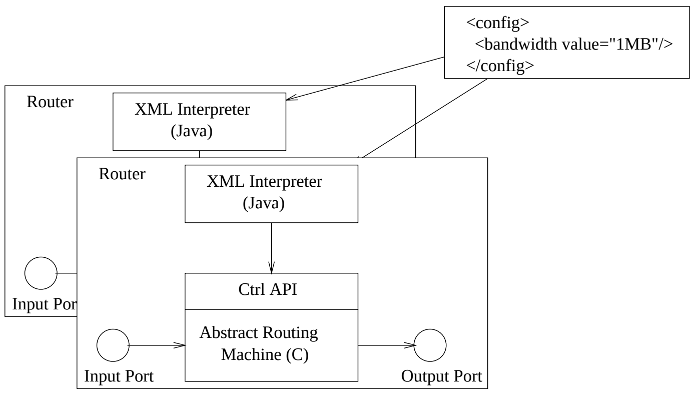

Figure 1: Programmable Router Architecture

[da Silva and Yemini (1999)] components can be composed at runtime into the routers in order to add features and functionalities. Our approach takes a more fine-grain perspective on compositionality. The XML based approach we are exploiting allows us to manipulate single lines of code and single values, as well as to modify complete components.

3 Description of the Approach

The aim of this work is to define a platform for router policy management and update. 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 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 type of policies. The grammar for policy definition 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 behavior. 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. The advantages of XMILE are described in detail in the next section, however, in brief, they are related to the ability of dynamically modifying code at a very fine-grain level. We use the approach to update the policy definitions on the router at run-time at the level of single policy parameter or statement. In Section 5 we give an example of this. The ability 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 update network node policies at run-time at a very fine-grain level is an added value of our work.

The policy definition set is dynamically modified at run time, when some environmental conditions emerge. Once a policy is modified, or added to the router policy set, the router state and behaviour 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. If new router behaviour is needed because a new policy is added to the system our middleware takes care of fetching and linking the new module to the router code.

Figure 1 shows the architecture that we propose for our approach. The high level XML policy language is independent from the implementation language used for the network node forwarding behaviour. This mean that the platform may be flexibly ported on different architectures and systems.

4 Architecture

Due to the ease of updates to router policies at the scripting language level, and the run-time ability to react to the environment, remote management of policies for a set of network nodes can be achieved (Figure 2). The network manager can observe the network with some performance tool and distribute policy changes on a set of routers which run the XMILE platform. A mechanism for automated distribution of changes on the different router can be installed as well in order to further ease the process.

5 The Xmile Platform

XML provides a flexible approach to describe data and document structures. We now show how XML can be

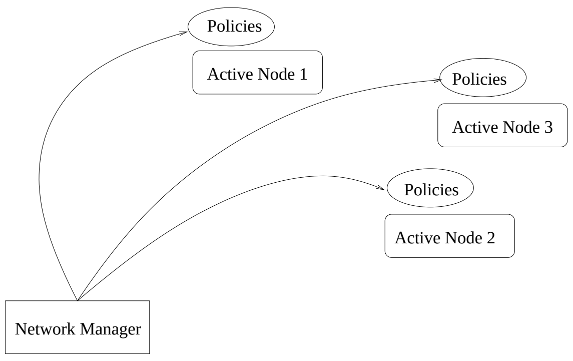

Figure 2: Router policy distribution.

used to describe code and explain how XML can be used to define *policies*.

XML Schemas [Fallside (2000)] define the grammar for XML documents. The structure of all the element that can be put in an XML file are defined in a schema. XML Schemas (or DTDs, Data Type Definition which are the predecessors of Schemas) are very similar to attribute grammars [Knuth (1968)]. 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 *policies*. XML policies can be interpreted and interpreters can be constructed using XML technologies. By sending XML policies or fragments of them from host to host we achieve a very fine granularity for the unit of code mobility. In the context of active networking the code we are interested in updating and maintaining is the set of policies for the router behaviour update.

We use a simple example to demonstrate this idea. Consider a simple set of policies for the routing of packets on a network node. The initial set of policies allow the router to drop or shape packets when the packet forwarding rate is above certain levels. Figure 3 shows the XML policy defining the policies for a specific router. The policy is written in an XML policy language whose syntax is defined by a Schema, an excerpt of which is shown in Figure 4. The policy starts with the definition of the policy set. The example we are using here is very simple. However new and more complicated policies could be defined. The policies defined in Figure 3 instruct the router to drop any kind of packets if the forwarding rate is higher than a certain value (i.e., the tag $Drop$) and to shape packets delaying the packets destined to the Janet network in case the rate is too high.

```
<?xml version="1.0"?>
<policies>
     <Drop whenRate"100">
     <Shape whenRate="50" delay="20"
            destination="janet.ac.uk"/>
</policies>
```

Figure 3: XML code for router policies definition.

```
 <xsd:element name="policies">
  <xsd:complexType>
   <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element ref="Drop"/>
    <xsd:element ref="Shape"/>
     ...
   </xsd:choice>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="Drop">
  <xsd:complexType content="elementOnly">
   <xsd:attribute name="whenRate" use="required"/>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="Shape">
  <xsd:complexType>
   <xsd:attribute name="whenRate" use="required"/>
   <xsd:attribute name="delay" use="required"/>
   <xsd:attribute name="packetsFrom" use="required"/>
  </xsd:complexType>
 </xsd:element>
 ...
</xsd:schema>
```

Figure 4: XML schema for the policies.

The root of the grammar in Figure 4 is the definition of the `policies` element, which contains the different tags for policies definition. The `Drop` element contains just an attribute defining the packet forwarding rate limit for beginning to drop packets. The policy can be defined in a more complex way, maybe also defining attributes for limiting the dropping of packets only from certain networks. The `Shape` tag defines three attributes, one for the rate above which beginning to shape, one for the shaping delay to be assigned to packets, and one for defining the network the packets to be shaped are destined.

The XMILE approach allows us to update the policies on a router by transferring fragments of new code from another server to the router and then dynamically patching the original code. Unlike Java programs, which are sent in a compiled form, XML policies are transferred in source form and then interpreted on the remote host. We refer to such code fragments as XML *policy increments*. Hence, we can specify complete policies 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 policy. For instance, let us assume that we want to update the router policies adding the ability of shaping packets destined to the network routerucl.ac.uk. For this we need to add a new policy. Figure 5 shows the policy increment that is shipped to the router.

```
<Shape whenRate="50" delay="30"
       destination="routerucl.ac.uk"/>
```

Figure 5: XML policy increment.

The policy increment that we need to update can be sent separately without the need to re-send the complete policy policy.

The XML policy structure makes the dynamic manipulation of the code a lot easier. An XML policy can be seen as a tree and the DOM API [Apparao et al. (1998)] provides operations for the navigation and modification (adding/deleting/changing) of branches of the policy tree. The addressing of the particular branch that needs to be modified is performed using the XPath language [Clark and DeRose (1999)]. Going back to our example, Figure 6 shows the XPath expression addressing the point where the new increment needs to be added in the policy.

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

Figure 6: 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. For example, while policies are updated the packet forwarding process is going on.

```
<xsd:element name="Remark">
 <xsd:complexType>
  <xsd:attribute name="fromNtw"
                  use="required"/>
  <xsd:attribute name="howtomark"
                  use="required"/>
 </xsd:complexType>
</xsd:element>
```

Figure 7: XML Schema update code.

As described earlier, 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 `Shape` 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 so that in a subsequent update we can also include a remarking tag to the policy, we could transmit the schema update shown in Figure 7 with a specific XPath expression to define the update point.

We now describe the architecture of the platform for dynamic active network nodes updates. The policy management engine and the packet forwarding engine are kept separate; the former engine is in fact based on an interpreter that accepts and processes XML files (Figure 8).

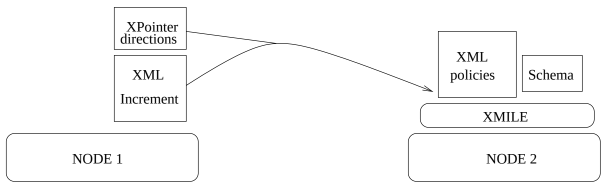

Figure 8: XML policy update architecture

Concerns that have to be programmed in a router, such as the configuration of virtual private networks or security policies are defined in a high-level markup 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.

A control API exploiting Java JNI to embody the XML policies and to control the router control state and the packet forwarding component. Note also that the architecture is independent of the implementation language chosen for the router packet forwarding process.

Once such an XML policy is written, it can be distributed to different routers to be executed by an XML interpreter for the markup 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.

Figure 1 shows the general architecture of our system. The XMILE engine (more details in [Emmerich et al. (2000); Mascolo et al. (2001)] that we use at this level also relies on Java class loading for the fetching of the behavioural classes needed to implement the policies written in XML. 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 3) corresponds to a Java class which is executed by the interpreter (if the class in not on site the interpreter fetches it with Java class loading). This class checks whether the corresponding behavioural function for the packet forwarding engine (for instance for the Dropping policy) is present in the Dy-

```
while(true){
  inPort.readP();
  Port outPort=processP(packet);
  if(outPort!=null){
    port1.writeP(packet);
}
```

Figure 9: Code for the packet forwarding process.

namic Linking Library; if not the function is remotely fetched. 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. For the time being this list is created dynamically every time the interpreter executes, however some optimization exploiting the information of where a fine-grain XML update is performed can be implemented. Some XML policies may require the insertion of more than one function into the function list but we do not discuss this issue in this paper.

The packet forwarding process will approximately look like the pseudo-code we report in Figure 9. The router accepts packets from the input port. The processing function `processP` will sequentially execute the function of the list generated/updated by the interpreter for each packet. For example for the policy in Figure 3 the list will contain a dropping function and a shaping function. The packet is first passed into the dropping function which will check the actual forwarding rate of the router (in the router status component). If the rate is higher than 100 the packet will be dropped and the following function will be skipped. If not, the packet is then passed to the shaping function which will decide, based on the packet destination if the packet needs to be delayed. If yes the packet is put into a queue and reconsidered for forwarding when time is ready. The final function of the list is the actual forwarding function which returns the right output port for the packet. In Figure 9 the packet is then passed to the destined output port for forwarding.

When the policies are updated with the update defined in Figure 5 the interpreter re-runs to update the function list. When the policy grammar is updated, like in Figure 7, the interpreter that handles the grammar file also takes care of loading the DLL function corresponding to the new operator introduced in the grammar, so that the operator can be freely used in the XML policy file, and the corresponding function loaded in the function list of the router packet forwarding component. Figure 10 shows the two layers of our architecture in more detail.

6 Discussion and Evaluation

The XMILE approach supports the update of code and policies at run-time and at different levels. As explained in Section 3 the XML policy file can be updated adding, modifying and deleting single lines or parameters. The grammar itself can be updated using XMILE. This would facilitate the use of different policies in the policy definition files used to control the router. It is also possible to have dynamic update of the functions used by the router at the packet forwarding level.

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 router, or could vary the strategy depending of the different local network states. These features are novel with respect to approaches such as Kohler et al. (2000) where the authors focus on the reconfiguration of specific routers instead of global network management. The fact that network administrators are allowed to define, update and extend the domain specific languages based on XML for policies description introduced new flexibility which is not available in other approaches such as Sloman and Lupu (1999); Wakeman et al. (2000). In Smith and Hutchison (2000) XML is also used for active networking. In particular, the definition of a specific language for the packets payload. Our approach is more articulated but shares the view that XML can be used to define specific structures for policies or data. With XMILE we allow the dynamic update which is not contemplated by Smith and Hutchison (2000).

7 Conclusions and Future Work

XMILE is based on the interpretation of XML tags and Java class-loading. As previously addressed, the system we described has a two layer architecture, which exploits the flexibility of XML at the router language and policy definition level and the performance of C or other compiled languages for the packet forwarding process.

Security issues need to be addressed. We plan to take advantage of the Schema-XML relationship to validate XML policies against a properly defined grammar. We are also introducing typing to run more static checks, but we believe more work in the direction of security could be done.

We also plan to optimize the function list for the packet forwarding component. At the moment the list is regenerated every time the interpreter re-execute after a policy update. However, it is possible to exploit the structure of the XML policy file and the XPath expression for the update to know exactly how the function list need to be updated without regenerating it.

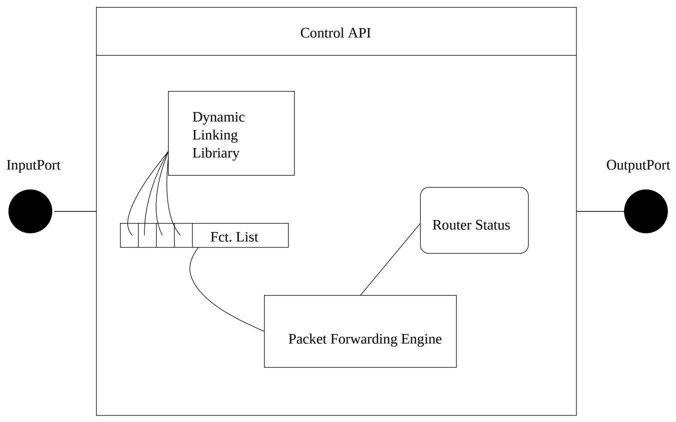

Figure 10: Detailed two layer architecture

Acknowledgements

We would like to thank Jon Crowcroft, Richard Gold, Anthony Finkelstein and Adam Greenhalgh for comments, discussion and suggestions on an earlier draft of this paper.

References

Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Hors, A. L., Nicol, G., Robie, J., Sutor, R., Wilson, C., and Wood, L. (1998). Document Object Model (DOM) Level 1 Specification. W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium.

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

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

Campbell, A., Kounavis, M., Villela, D., Vicente, J., Meer, H. D., Miki, K., and Kalaichelvan, K. (1999a). Spawning networks. *IEEE Network*.

Campbell, A., Meer, H. D., Kounavis, M., Miki, K., Vicente, J., and Villela, D. (1999b). A survey of programmable networks. *ACM Computer Communications Review*.

Clark, J. and DeRose, S. (1999). XML Path Language (XPath) Version 1.0. Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116, World Wide Web Consortium.

da Silva, S. and Yemini, Y. (1999). NetScript: A Language and Environment for Programmable Networks . http://www.cs.columbia.edu/dcc/netscript/index.html.

Decasper, D., Dittia, Z., Parulkar, G., and Plattner, B. (2000). Route plugins: A software architecture for next-generation routers. *IEEE/ACM transactions on Networking*, 8(1).

Emmerich, W., Mascolo, C., and Finkelstein, A. (2000). Implementing Incremental Code Migration with XML. In Jazayeri, M. and Wolf, A., editors, *Proc. 22nd Int. Conf. on Software Engineering (ICSE2000)*, pages 397–406, Limerick, Ireland. ACM Press.

Fallside, D. C. (2000). XML Schema. Technical Report http://www.w3.org/TR/xmlschema-0/, World Wide Web Consortium.

Kakkar, P., Hicks, M., Moore, J. T., and Gunter, C. A. (1999). Specifying the PLAN networking programming language. In *Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

Knuth, D. E. (1968). Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145.

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The click modular router. *ACM Transactions on Computer Systems*, 18(4).

Mascolo, C., Emmerich, W., and Finkelstein, A. (2000). Xmile: An Incremental Code Mobility System based on XML Technologies. In *Poster Session of the. 2nd Int. Symposium on Agent Systems and Applications Mobile Agents*, Zuerich, Switzerland.

Mascolo, C., Zanolin, L., and Emmerich, W. (2001). XMILE: an XML based Approach for Incremental Code Mobility and Update. Submitted for Publication.

Sloman, M. and Lupu, E. (1999). Policy Specification for Programmable Network. In Springer, editor, *First Int. Working Conference on Active Networks (IWAN99)*, Berlin.

Smith, P. and Hutchison, D. (2000). New Telecommunication Services using Active and Programmable Networks. Tech. Report Univ. of Lancaster 00-04.

Wakeman, I., Jeffrey, A., Owen, T., and Pepper, D. (2000). SafetyNet: A Language-Based Approach to Programmable Networks. http://klee.cs.depaul.edu/an/spec/.