AMnet 2.0: An Improved Architecture for Programmable Networks

Thomas Fuhrmann, Till Harbaum, Marcus Schöller, and Martina Zitterbart

Institut für Telematik
Universität Karlsruhe, Germany

Keywords: Programmable Networks, Active Nodes

Abstract. AMnet 2.0 is an improved architecture for programmable networks that is based on the experiences from the previous implementation of AMnet. This paper gives an overview of the AMnet architecture and Linux-based implementation of this software router. It also discusses the differences to the previous version of AMnet. AMnet 2.0 complements application services with net-centric services in an integrated system that provides the fundamental building blocks both for an active node itself and the operation of a larger set of nodes, including code deployment decisions, service relocation, resource management.

1 Introduction

The idea of active and programmable networks has been studied for several years already. But despite its great flexibility and potential, practical restrictions so far kept this idea from actual deployment. We believe that especially the tradeoff between general programmability and security considerations proves to be a major obstacle for actual deployment. Generally programmable active nodes have to offer full access to the network traffic, including inspection and modification of all the data routed through that node. Thus, deficient or malicious code loaded into such a node can threaten the network's integrity. Closely related to these security considerations is the lack of a sufficient amount of usage scenarios in which active and programmable networks provide an advantage over less precarious approaches like proxies and application-layer overlay-networks. Since we believe that both aspects have to be addressed in their mutual relation, the redesign of the AMnet architecture was guided by the consideration of usage scenarios for active and programmable networks that go beyond the mere support of heterogeneous group communication in AMnet 1.0.

The previous concept of AMnet [14,7,8] mainly pursued the scenario of a sender based heterogeneous multicast where different end-systems have different requirements. Following the multicast design principle, adaptation of the data was pushed from the sender to some node along the distribution tree. Signaling relied on session and service announcements conveyed along the distribution tree. Set-up of a service was solely triggered by the receivers. Thus, the application of AMnet 1.0 required both, sender and receiver, to be adapted to AMnet.

The new AMnet 2.0 concept, in contrast, extends the original scenario to cases where the sender, the receiver, or even both are not aware of AMnet. Typical examples for these, as we call them, *net-centric services* include services like

– Multicast in non-native-IP-multicast environments — One of the AMnet 2.0 service examples automatically and fully transparently pools HTTP music streams to save network and server resources.
– Congestion control for streaming applications — AMnet services can transparently reduce the data-rate of various media streams if congestion occurs.

We believe that such services will promote the deployment of active and programmable network technology by creating a seizable reward for installing active nodes. AMnet 2.0 (for which the described services are either already implemented or about to be implemented soon) can thus now provide gain to a network operator from the first installed node on. We believe that this empowerment of the entity that is required to install and maintain the active nodes is a key element of utmost strategic importance for the actual deployment of AMnet. Benefit from widespread deployment beyond an individual service provider will then automatically occur as a second step. This focus was not present in AMnet 1.0.

According to this extended usage scenario and strategic goal, the AMnet 2.0 architecture has to provide high performance and give control over the achieved security level to the respective domain's administration. Flexibility should primarily be viewed in the light of quick provision of novel services and easy maintenance. This shift in focus required a slight adjustment of the performance-security-flexibility tradeoff as compared to AMnet 1.0:

– AMnet signaling was originally closely bound to the evaluation process that selects a node for execution of a service module. It was also based on evaluation packets that carried the evaluation code (capsule approach). In AMnet 2.0, signaling and evaluation are separated, the capsules are omitted entirely, and the evaluation process is now based on modules drawn from the service module repository (cf. section 2.2). Thus, evaluation and service execution have become very similar. This fact reduces the node's complexity significantly. Together with the complete abandonment of the capsules this increases the overall security considerably.
– AMnet 1.0 employed its own mechanism to grab, mangle, and re-inject packets. With the advent of Linux 2.4 this mechanism became obsolete and could be replaced by a lightweight interface to the standard Netfilter mechanism of Linux. This allows AMnet to be installed most easily on standard Linux nodes. A fact that should also increase trust in the system.

Besides this shift in focus, the new AMnet 2.0 design embodies many practical supplements and extensions of already existing ideas, e.g. a relocation mechanism, improved signaling for off-path node detection, execution environment extensions, etc. These, too, are described in the following sections.

1.1 Related Work

Active and programmable networks have been a field of intensive research for quite a while. Therefore, it is impossible to give a brief overview without being forced to arbitrarily choose among the many approaches in this field. We will hence only exemplarily present a few projects to place AMnet in context. Please see, e.g., [18,2,3] for a general

overview of fundamental concepts of active and programmable networks and [8] for work related to AMnet 1.0.

AMnet's active nodes are Linux based software routers. A similar approach has been pursued by the Click Modular Router project [12]. There, low-level extensions to the regular network protocol stack provide a router environment in which so-called *elements* perform the basic processing steps like packet classification and mangling. Compared to AMnet, which (mostly) runs in the user-space of an unmodified Linux installation, Click's direct hardware access trades security and programming ease against performance. Both projects' common objective, namely to benefit from existing operating system functionality, is also shared by SILK [1], in which a port of Scout [15] replaces the standard Linux protocol stack.

The NodeOS working group [16] gives a general specification of such an interface between the generic operating system and the particular active node functionality, whereas VERA [10] defines an interface between router hardware and software. An example for the use of flexible router hardware is the Dynamically Extensible Router (DER) [13]. Contrary to AMnet, where specialized hardware may optionally supplement the central processing unit of a software router, DER inserts *processing elements* between the *line cards* and the switching fabric.

The rest of the paper is structured as follows: Section 2 explains the main architectural concepts of the AMnode. Section 3 then presents the architectural structure. Section 4 describes our actual implementation of AMnet. Section 5 demonstrates the use of AMnet with a practical example of a service-module. And section 6 finally concludes our presentation with a short outlook on future work.

2 Conceptual Overview of AMnet 2.0

AMnet 2.0 consists of two main building blocks, the so-called *service module repository* and the active node, the *AMnode*, itself.

2.1 AMnodes as Active Nodes

AMnodes are mainly intended to be positioned within the network, preferably as edge-routers. Being placed on the path data is routed along, they are capable of analyzing and modifying the packets passing by regardless of their actual destination address. Additionally, with AMnet 2.0 AMnodes can also be placed off-path. In that case, AMnet signaling mechanisms provide means to start and control services on these nodes. If necessary, AMnet signaling thus enables multiple AMnodes to cooperate in order to, e.g. redirect traffic from AMnodes on the original path to an appropriate off-path node. Separating the AMnodes that are loaded with all the passing traffic from other AMnodes that have enough resources for computationally demanding services, provides a flexible means for scaling and allows AMnet to handle a broad range of services.

Such services that are flexibly installed within the network are the main guideline of our concept. They can be classified into two major categories: application (or end-system) services and the new net-centric services.

– *Application services* are initiated by an application on an end-system of the network. A typical example is media-transcoding. Here bandwidth-constraints or a request for a specific data-format are signaled from the end-device into the network. Other examples might be network support for peer-to-peer applications or the creation of application-specific overlay networks.
– *Net-centric services* are directly invoked on the AMnodes, e.g. directly during setup by a network administrator or indirectly by other AMnodes. Typical examples are network monitoring, security services like firewalls and intrusion detection systems, protocol-boosters, content-caching or multicast reflector services.

All services within the AMnet scenario are based on *service modules* that contain the actual code for the service. A service can be provided by a single service module alone, or by a combination of several service modules. This possibility to simply chain services modules to create more complex services opens AMnet to a wider audience of programmers whose main skills are scripting languages and not the full details of network programming. This goal is similar to the active pipe approach in [11]. AMnet, however, aims at combining modules within one active node, whereas active pipes are an overlay in the network. Service module programming and the improved chaining of AMnet 2.0 will be described in detail in section 5.

Technically, service modules are small portions of object code that are dynamically linked to the AMnode's *execution environment* upon invocation. Thereby, AMnet combines the high flexibility of on-demand installation of services with native execution speed making it thus feasible to use AMnet beyond mere "in-principle" demonstrations[1]. Besides this capability for native language code, typically C, wrappers for e.g. Java code are also available. We consider this openness to a broad range of languages and programming styles a key feature for a widespread deployment of programmable networking ideas.

Since native code does not run in a sandbox, additional security measures have to be implemented to assure the node's integrity against malicious or erroneous code. AMnet 2.0 introduces a wrapper layer in form of a library between the service module and the node's operating system to impose constraints on the execution of native code. This allows the AMnode to enforce certain restrictions on the service modules [9]. Another key element of AMnet's security strategy is the use of a service module repository.

2.2 Service Module Repositories

Service module repositories in AMnet are administratively managed and provide the module descriptions and the modules' object-code, potentially for various platforms. This aspect of the repository is similar to the code caching approach of [4]. Module descriptions are currently based on a centralized naming scheme only. A more flexible description scheme is under development.

Additionally, service modules can also come in multiple flavors to allow for different resource restrictions or operating conditions. So service modules can be optimized

[1] For a detailed analysis of the achievable system throughput see [6].

for low memory consumption, high-throughput, or other criteria, like the use of specialized hardware. The AMnet signaling mechanism will select the appropriate flavor of a service module depending on the respective AMnode's capabilities.

Besides the service modules, the service module repository contains service dependent modules for the evaluation and relocation process (as described below). The evaluation and relocation modules have to be service dependent since different services may have different requirements such as the preferred location for the service (close to the receiver or close to the sender) or the number of running service instances on distributed AMnodes (e.g. for the semi reliable multicast service). This mechanism enables the administrator to directly set up preference hierarchies as to which service modules are used within its domain.

The use of a service module repository tackles a major security issue within AMnet. Since the repository is administratively managed and therefore the domain administrator is the only one who can add new services to that repository, only tested and verified code gets executed on the programmable nodes within that domain. Network security protocols (IPSec, SSL) and cryptographic signatures protect the data transfer from the repository to the node from data manipulation and man-in-the-middle attacks.

Service module repositories may be linked to form a network of trust. If a repository cannot satisfy the service request of an AMnode, the search can be extended over all trusted repositories. This implies that service modules can be introduced and updated rather easily at a single insertion point if admitted by the administrative policies. E.g., repository administrators of a given administrative domain could decide to trust a certain software vendor's repository. Code released by this software vendor would then automatically become available on all repositories of that domain. A drawback of this linkage is the danger of quickly spreading faulty code. If employed carelessly, this mechanism of trusted links can bring down a service throughout the network. Care must hence be taken of how trust is established within AMnet, especially if repositories mutually trust one another. Thorough step-by-step testing of new service modules, using AMnet's mechanisms to restrict trusted links by administrative policies, can minimize this threat.

In the following section we present the AMnode's functional architecture in more detail. Implementation specific details are described in section 4.

3 Architecture of AMnodes

As already mentioned, AMnodes are intended to be positioned as routers at the edge of the network. Their purpose is to provide services that primarily comprise analysis, conditional forwarding and general modification of the bypassing packets. To achieve this, while observing the already mentioned security requirements, the AMnode architecture forms a three-layered structure containing the following fundamental functional building blocks: packet filtering, signaling and resource monitoring, the execution environment, and support for node selection and service relocation.

At the network-layer a *packet filter* parses the packet stream, transfers packets that match the service modules' filtering rules to the respective modules, and forward non-matching packets according to the node's routing table. This hook into the operation

system's kernel is necessary, since AMnet needs to override the basic networking functionality of a standard router. This building block is fully embedded in the operating system's kernel.

3.1 Signaling and Monitoring

The next-higher functions are located in the user space. There the usual Linux APIs apply and simplify the implementation and maintenance of the AMnode.

The *resource monitor* keeps track of the modules' individual resource requirements and initiates countermeasures upon menacing overload. The countermeasure depends on the kind of service module, its priority and the node's administrative policies. Countermeasures include service relocation and forced shutdown of a service. Ideally, relocation should be initiated well before the node's resources are exhausted giving the service enough time to relocate gracefully. In any case, un-cooperating modules must be kept from jeopardizing the node's functional integrity.

The *resource access control* controls the allocation of resources on a per service basis. The security policies are again service-dependent and also stored in the service module repository. The rule set either limits or entirely prohibits the use of certain resources. This access control allows to minimize the threat originating from both, the service modules themselves (e.g. memory exhaustion, malicious operations) and the data processed by the node (e.g. denial-of-service attacks).

The basic *signaling mechanism* provides applications (and thus end-users) with the possibility to request services [17]. It also handles the communication with the service module repositories and allows an AMnode-to-AMnode communication, e.g. for evaluating service requests or to negotiate service relocation.

As mentioned above, the communication between the AMnode and the service module repository is secured either by network or by application security protocols. The same mechanisms can be applied to the inter-AMnode-communication. The communication between end-users and AMnet must include an authentication mechanism to legitimate the request.

Further signaling functionality can then be introduced on a per-service-module level. This is e.g. necessary for protocol-boosters that need to understand the signaling of the respective protocol.

3.2 Execution Environment

On top of the AMnode's resource monitor and basic signaling mechanism resides the *execution environment*. Its location in user space with potential additional protection by the mechanisms described in [9] secure the AMnode against malicious service modules.

The *execution environment* hosts the service modules and supplies basic functionality, e.g., for setting and revoking packet filtering rules, access to (parts of) the kernel API, and to the packets captured by the filter. It thus provides the framework within which service modules are executed. Multiple service modules can potentially be combined to provide more complex services. Technically, the execution environment (EE) is a basic user-space program which the service modules are linked to on demand.

Being located in user-space the EE and thereby the service-modules underly the standard Unix security restrictions for processes. Together with the security mechanisms already mentioned, this ensures that erroneous or malicious service modules can neither manipulate other services nor endanger the AMnode's integrity.

The EE of the first AMnet implementation was limited in various ways, e.g. in the way service modules could be linked together and in the fact that neither branches nor any other kind of packet re-routing, duplication etc. was possible inside the module chain. With the increasing complexity of services implemented in AMnet, a need arose for the flexible combination of multiple modules and the possibility to dynamically change the path on which packets are handed from module to module. An example for such services is a traffic analyzer that dynamically installs new services in the module chain in order to change the behaviour of the service with respect to changing traffic conditions.

Further problems arose with the integration of the resource monitor into the AMnet node architecture. In AMnet 1.0, complex service modules often forked child processes in order to provide proxy or server like functionality. With the introduction of the new resource monitor, forking had to be limited to allow effective resource monitoring. AMnet 2.0 implements several callback handlers that help with common problems usually solved by forking server and proxy child processes and thus avoids many fork operations. A service module can register callback functions, e.g. for timer events as well as for socket and file input-output, allowing a service module consisting of one single process to handle various timers, network sockets, file handles etc. at once. Furthermore, this abolished the need for interprocess communication making the design of complex service modules even simpler and more efficient.

3.3 Support for Node Selection and Service Relocation

Three special mechanisms closely related to the service modules themselves, namely evaluation, relocation, and resource management, complete the AMnode architecture:

The *evaluation mechanism* determines on which AMnode a requested service will be started. Primary criteria are administrative policies and the AMnode's current resource availability. If the service request does not have sufficient rights or if the AMnode's available resources fall below the service module's specified threshold, any further evaluation is immediately rejected. Otherwise, a potentially more complex evaluation procedure is started.

The *relocation mechanism* determines to which AMnode an already running service will be relocated upon an overload situation. In most cases, this is very similar to the initial evaluation procedure. For some services however, differing evaluation and relocation mechanisms are better suited, e.g. when a service needs to be quickly started on an in-path AMnode before it is (more time-consumingly) relocated to a better suited AMnode that might be found off-path.

The *resource manager* mediates between the AMnode's resource monitor, the service modules' requirements and the evaluation and relocation mechanisms. Through it, a service can define e.g. its relocation strategy.

In contrast to the fundamental components listed above, these latter functionalities can mostly be built within the service module framework, i.e. they can be coded as ser-

vice modules that are executed within the node's execution environment. The AMnode itself only needs to provide basic support functionalities for signaling and interaction with the resource monitor. This allows a much greater flexibility than if these mechanism were fixed parts of the AMnet architecture. In most cases however, standard mechanisms can be used. These are realized by ready-made modules available in the repositories.

The following section describes our actual implementation of AMnode. A more detailed documentation together with sample service modules can be obtained with the source code of our implementation [5].

4 AMnode Implementation Based on Linux

The central functionality of the AMnode is to provide an interface between the networking core of the host operating system and the service modules running within the AMnode's user-space execution environment. The whole setup is depicted in figure 1.

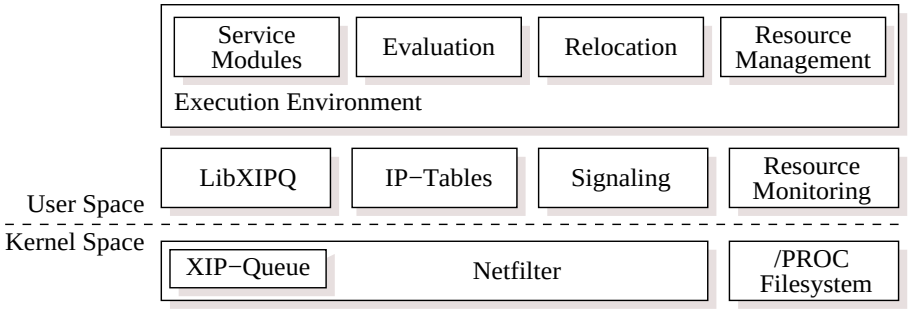

Fig. 1. AMnode setup on Linux 2.4

4.1 The Use of Netfilter

The first version of the AMnode kernel integration was based on a proprietary hook into the Linux kernel code. The installation of these hooks was available for the 2.0 and 2.2 series of the Linux kernel and required the recompilation of the networking core functions and, hence, a re-installation of the kernel and a system reboot. With the release of the 2.4 Linux kernel, a new flexible interface to the packet processing paths inside the networking core was introduced. This so-called *netfilter interface* forms the basis of functions like packet filtering used e.g. for firewalls and packet modification required e.g. for network address translation (NAT) and IP masquerading. The flexibility of the netfilter interface allowed many third party extensions of the basic routing functionality, e.g., IPsec and QoS.

By using the netfilter interface as the basis of the AMnodes access to the packet processing, it is now possible to use unmodified Linux 2.4 kernels as a basis for the

AMnode. This allows the installation and integration of all of the AMnet functionality into a running standard Linux 2.4 system as operated by various ISP's without even the need to reboot the system. AMnet can even be used in combination with other netfilter based extensions[2].

4.2 Extended IP Queue (XIPQueue)

The user-level status of the execution environment equips AMnet with a high level of security. On the other hand, this concept of service module based packet processing requires the transfer of network packets from their usual processing space inside the system kernel into the user space where the service modules are located. After the processing in user space, the packets have to be re-injected into the system kernel for further processing by the networking core[3].

The netfilter implementation distributed with standard Linux 2.4 kernels contains an interface module for packet transfer between kernel and user-space called *IPQueue*. Among the many limitations of this IPQueue are the inability to interoperate with several user space processes simultaneously and the inability to inject additional data packets into the data stream. Therefore, we extended the standard IPQueue implementation to a more general packet-queuing interface between the kernel and the user space. This new netfilter module named *XIPQueue* (extended IPQueue) integrates into the netfilter concept and works independently from the IPQueue. It may even be used simultaneously with the standard IPQueue. The XIPQueue allows the kernel to tag packets for different user space processes. Additionally, it can tag packets destined to the same user process with different markers to forward packet classification information derived from other netfilter functions to the user-space process.

Through the implementation of the XIPQueue, further extensions were made possible. These extensions include a second interface to the user space via the proc file system. This interface is used to control and supervise the operation of the XIPQueue and to access queuing specific parameters from user space.

The access from user-space to the XIPQueue netfilter interface is done directly from a user-space application. In order to allow flexible extension and modification of the XIPQueue interface and reuse of the the XIPQueue in other, non-AMnet scenarios, an additional interface layer was implemented. This *LibXIPQ interface library* can be accessed by general user space applications and provides all functions required to access the XIPQueue netfilter module. Further extensions were made to the *iptables* program which is used to install and maintain netfilter modules in the linux kernel. Extensions to this program are also made via user-space libraries and can be introduced into the existing IPTables installation in a working system.

---

[2] Care must be taken concerning the order of rule installation. This determines e.g. whether the AMnode runs before or after a given other netfilter extension.

[3] For many services (e.g., video-transcoding), the delay caused by the two copy operations is negligible compared to the actual processing time. For high throughput services, we are currently considering improvements that either perform a quick copy based on memory remapping or execute simple operations directly in kernel space.

None of these extensions require the modification or replacement of existing programs or libraries on the system. The XIPQueue netfilter module as well as the LibX-IPQ interface library and the extension library for the iptables program can easily be installed on the running system without the risk of influencing other applications.

4.3 The Resource Monitor

The resource monitor is tightly bound to the operating system's kernel. It reads its respective current state and attributes resource usage to the individual services (i.e. modules or module chains). In our implementation this is done via reading Linux' standard proc-filesystem. In order to also include the packet filter status, we provided our packet filtering mechanism with an appropriate interface to the proc-filesystem. Since different service-modules run as different processes, resource usage can be traced via the process identification (PID). Our implementation currently monitors three general parameters: memory, bandwidth, and CPU cycle consumption. Since processing power is difficult to valuate in a general and platform-independent way, we employ an indirect mechanism that has the advantage to yield exactly the required information: Being user-space processes, service modules share CPU time according to their scheduler priorities, i.e. the AMnode can control the modules' individual share of CPU time in the first place. By observing the modules' capability to hold pace with the incoming packet stream the node can determine whether the attributed processing power suffices. A growing queue in the packet filter thus indicates an unsatisfied demand for processing power.

4.4 The Execution Environment

A service module (or module chain) is installed and operated by one instance of the user-space based execution environment (EE). The EE is user or script controlled and the service modules are implemented as system shared libraries. So they can be installed on user or script demand, even into an already running EE. Running the modules in the context of their EE allows to control the resource usage of the service modules by observing the behavior of the respective EE user-space process only. Even more important is the fact that a service module in user-space is subject to the same limitations as any other user-space process. Unlike code executed in the kernel, a user-space application usually runs with very limited rights and the risk of critical malfunctions is much smaller when using user-space code than with kernel-space code.

The EE provides (among others) the following fundamental functions:

- *Registration and release of packet filtering rules.* These rules can be composed out of the usual criteria: protocol, address and port range. If necessary, further header fields can be included into the rules.
- *Packet passing, both from the packet filter to the service module and back again into the kernel.* Packets may be modified by the service module. Packets handed back into the kernel can be marked to be forwarded regularly as indicated by the routing table, to be sent through a specified interface, or to be discarded.
- *Creation of new packets.* Service modules can execute standard system calls unless these calls are prohibited by the security mechanisms described in [9]. This includes

the creation of new packets, regardless of their kind. I.e., UDP packets, whole TCP streams, and raw IP packets are equally feasible.
– *Helper Functions.* The EE also provides various helper functions for module chaining, inter-module communication, relocation, and various other tasks common to many service modules.

This concept has major advantages for the service-module developer. A very simple framework forms the core of a service module and helper functions provide additional functionality for all service modules.

5 Service Module Example

AMnet draws much of its flexibility from the way modules can be combined out of multiple object code pieces. We will illustrate this mechanism by the following example which sheds light onto AMnet service module programming in general.

5.1 Scripts for Module Initialization

The central element of an AMnet service module is a script that controls the module's start-up in the execution environment: it determines which code pieces have to be loaded, where packet filters are to be hooked into the kernel, and which filter rules have to be established.

Figure 2 gives an example of a module that receives all port 80 TCP traffic from (and to) some given server from the kernel's IP pre-routing hook. Note that we normally have to process both directions (i.e. from and to an address/port pair) since content modification typically requires us to modify the acknowledgments, too.

These filter specifications can also be issued or changed with the EE helper functions after module start-up, i.e. from within the module's object code. But the possibility to put such parameters into the start-up script makes service module programming easier for unexperienced programmers. This script based programming idea is further supported by the possibility to specify individual module parameters in the start-up script. In our example we might want to specify a string that can later be used for content-based filtering.

Once the service module is installed in the EE, it receives all packets matching the specified filter rules. In our example, we will receive all port 80 web-traffic from *www.some-server.net* passing through the AMnode. These packets can now be modified in the service module. For inspection of the TCP content, we can, e.g., redirect the data stream to a local socket. This interception mechanism is described in more detail in [5]. It equips our example module with an easy way to inspect and modify TCP content. Information about intercepted TCP connections can e.g. be stored in the EE using the appropriate helper functions for list management. This has the advantage that an administrator has direct access to these lists and can easily maintain them from the EE's command line interface. The service module can e.g. now rewrite content as indicated by the parameter in the start-up script. This short example might be used e.g. as a building block for a transparent web-proxy.

```
loadmodule tcp {
  file "libtcp_example.so";
  myparameter = "some string";
};

listen {
  protocol "tcp";
  nfhook PREROUTING;
  source {
    name "www.some-server.net";
    port 80;
  };
};

listen {
  protocol "tcp";
  nfhook PREROUTING;
  destination {
    name "www.some-server.net";
    port 80;
  };
};
```

Fig. 2. EE script example

5.2 Improved Module Chaining

Chaining of service modules was already implemented in AMnet 1.0, but it was limited to a fixed linear combination of multiple modules and mainly used to overcome limitations in filters. For example, in AMnet 1.0 an MPEG2-to-AVI filter could be obtained from chaining an MPEG2-to-MPEG1 filter with an MPEG1-to-AVI filter. Reuse of software components in multiple service modules (e.g., Huffman decoding, IDCT etc.) was possible in principle but required additional communication paths between different service modules.

AMnet 2.0 directly addresses these issues by extending the chaining capabilities and providing several ways of inter-module communication. On top of these new functions lies a powerful chaining scheme that allows to address modules either by function, position in the chain or some unique identifier attached to the modules during its installation [5]. This way, a AMnet 2.0 service module can request additional packets to be forwarded from the kernel to specific service modules. Additionally, service modules can cause packets to be rerouted inside the module chain in order to provide a special packet specific processing. A service module may at any time duplicate packets and schedule duplicated packets for different processing, allowing e.g. to duplicate a data stream and transform the duplicate stream into a different format or to influence the further processing of the duplicate stream by changing destination addresses etc.

Module chaining can be used e.g. to splice together a TCP connection that does no longer need to be intercepted at a local socket. In this, several code pieces are loaded

via the start-up script. Packets caught by the packet filter are first handed to a classifier that decides whether this packet needs to be inspected with the help of a local socket or not (cf. figure 3). If it has to be handeled, it is forwarded to the module code described above and handeled as described there. Otherwise it is passed over to another module that does not need to employ a full TCP socket but just rewrites sequence numbers to accommodate for potential modifications in earlier parts of the stream. After having modified a packet, the service module uses yet another helper function to correct the TCP checksum to ensure proper further processing of the packet.

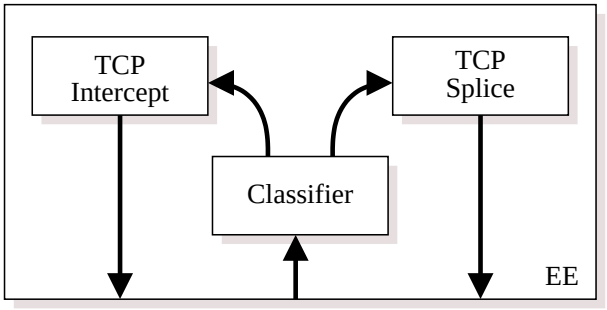

Fig. 3. AMnode setup on Linux 2.4

An example for another problem that could not satisfyingly be solved with AMnet 1 was e.g. demultiplexing while splitting a MPEG2 stream with included audio into separate audio and video streams. With the new chaining capabilities of AMnet 2, modules for multiplexing and demultiplexing can easily be implemented at any complexity by changing the routing inside the chain on a per packet basis or by linking several packets paths inside the chain to the same service module.

A service module may also communicate with other service modules or the EE, allowing this module to replace parts of the chain or to change the behaviour of some parts of the chain. This is required e.g. to implement dynamic services that are influenced by the e.g. current traffic situation or specific payloads being observed by a service module. An example for this may be a service detecting a wireless link specific loss pattern on and dynamically loading a service module to apply forward error correction on the affected link or apply protocols especially suited to handle the current situation on the link.

6 Conclusion and Future Work

In this paper we gave an overview over the current state of the AMnet project. We described the basic concepts of AMnet and the architecture of its active node, the AMnode. The three-layered approach of kernel-level packet-filter, fundamental node support (signaling and resource monitoring), and execution environment was motivated, and the interaction of EE and service modules illustrated by a short example.

Our future work aims at increasing the number of available service modules, thus producing building blocks for individual services and applications. We believe that the experience with productive uses of the AMnet architecture can provide valuable experiences that help us in further extending AMnet towards a practical programmable network infrastructure.

Acknowledgments The work described here has been performed in the framework of the FlexiNet project which is funded under grant number 01AK019E by the Bundesministerium für Bildung und Forschung. The authors would like to thank the reviewers of the paper and its shepherd for the helpful comments.

References

1. Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, and Per Gunningberg. SILK: Scout paths in the Linux kernel. Technical Report 2002-009, Uppsala Universitet, February 2002.
2. Kenneth L. Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 36(10):72–78, October 1998.
3. Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, 29(2), April 1999.
4. Dan Decasper and Bernhard Plattner. DAN: Distributed code caching for active networks. In *Proceedings of INFOCOM'98*, San Francisco, CA, April 1998.
5. Thomas Fuhrmann, Till Harbaum, Marcus Schöller, and Martina Zitterbart. AMnet 2.0 source code distribution. Available from http://www.flexinet.de.
6. Till Harbaum. *Rekonfigurierbare Routerhardware für adaptive Dienstplattformen*. PhD thesis, Insitut für Telematik, Universität Karlsruhe, 2002.
7. Till Harbaum, Anke Speer, Ralph Wittmann, and Martina Zitterbart. AMnet: Efficient heterogeneous group communication through rapid service creation. In *Proceedings of the 2nd International Workshop on Active Middelware Services (AMS'00)*, Pittsburgh, Pennsylvania, August 2000.
8. Till Harbaum, Anke Speer, Ralph Wittmann, and Martina Zitterbart. Providing heterogeneous multicast services with AMnet. *Journal of Communications and Networks*, 3(1):46 – 55, March 2001.
9. Andreas Hess, Marcus Schöller, Günther Schäfer, Adam Wolisz, and Martina Zitterbart. A dynamic and flexible access control and resource monitoring mechanism for active nodes. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH'02)*, New York, NY, June 2002.
10. Scott Karlin and Larry Peterson. VERA: An extensible router architecture. *Computer Networks*, 38(3):277–293, February 2002.
11. Ralph Keller, Jeyashankher Ramamirtham, Tilman Wolf, and Bernhard Plattner. Active pipes: Service composition for programmable networks. In *Proceedings of Milcom*, Washington DC, October 2001.
12. Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
13. Fred Kuhns, John DeHart, Anshul Kantawala, Ralph Keller, John Lockwood, Prashanth Pappu, David Richards, David Taylor, Jyoti Parwatikar, Ed Spitznagel, Jon Turner, and Ken Wong. Design of a high performance dynamically extensible router. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, San Francisco, May 2002.

14. Bernard Metzler, Till Harbaum, Ralph Wittmann, and Martina Zitterbart. AMnet: Heterogeneous multicast services based on active networking. In *Proceedings of the 2nd Workshop on Open Architectures and Network Programming (OPENARCH'99)*, New York, NY, USA, March 1999.
15. David Mosberger. *Scout: A Path-based Operating System*. PhD thesis, Department of Computer Science, University of Arizona, July 1997.
16. Larry Peterson. *NodeOS Interface Specification*. Active Networks NodeOS Working Group, Department of Computer Science, Princeton, January 2002.
17. Anke Speer, Marcus Schöller, Thomas Fuhrmann, and Martina Zitterbart. Aspects of AMnet signaling. In *Proceedings of the Second International Networking Conference*, pages 1214–1220, Pisa, Italy, March 2002.
18. David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 25(1):80–86, January 1997.