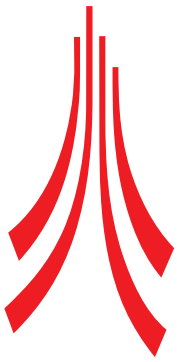


Computing Department
Lancaster University
United Kingdom

A Component-based Active Router Architecture

Stefan Schmid

Submitted for the Degree of
Doctor of Philosophy

November 2002

Abstract

Current Internet protocols and network services have been struggling to keep up with the fast evolution from traditional data to today's multimedia communication technologies and the changing requirements following these advances (for example, support for QoS, multicast, mobility, and security is still lacking in most networks). Active and programmable networking is a step towards enhancing the static and inflexible structures of current networks. As part of the general research trend, this thesis focuses on the design and development of technologies that allow rapid deployment of new functionality throughout the network (for example, customised network services and protocols), which allows network vendors and service providers to respond quickly to the changing requirements and keep up with the fast evolution in network and communication technologies.

This thesis starts off with an general introduction into the research area and a description of the basic mechanisms behind active networking. A critical examination of existing active and programmable systems and associated technologies is provided. These results together with previous experiences of developing an active network system lead to a set of fundamental requirements for active nodes. The core of the thesis presents the design and implementation of a novel active router architecture that enables flexible network programmability based on so-called 'active components'. This second generation Lancaster Active Router Architecture (LARA++) is designed to provide maximum flexibility for the development of future network functionality and services. Its comprehensive service composition framework enables flexible programmability through the transparent integration of active components into the router's data path. Finally, the success of the node architecture and its prototype implementation is evaluated by means of a few concrete applications. This shows that LARA++ offers sufficient flexibility and extensibility to augment the network in ways that suit today's fast evolving internetwork platform, and the prototype implementation confirms that the research platform offers acceptable performance for edge-routers of small-to-medium sized network environments.

Acknowledgements

Although my name is the sole name on the front page, this thesis has only become a reality due to the countless contributions of many people whom I want to thank here:

I would like to express my sincere thanks to my supervisor Professor David Hutchison for his professional guidance and great support throughout all my studies and my work at Lancaster University. He has taught me much – about academia and beyond – for which I am very thankful.

I am also very grateful to my colleagues of the Distributed Multimedia Research Group at Lancaster for all their inspiration and assistance over the last four years. In particular, I thank Professor Doug Shepherd who has always been a source of encouragement and support. In addition, I would like to express special gratitude to my close colleagues and friends Dr. Andrew Scott and Dr. Joe Finney for their constructive discussions and invaluable comments on my work.

Special thanks also go to my colleagues Tim Chart, Manolis Sifalakis and Daniel Prince for their useful comments on drafts of this thesis and their involvement in surrounding discussions. Tim in particular has helped a great deal during the final implementation phase.

Most of the research was carried out in cooperation with industrial research laboratories. I am particularly thankful to my sponsors Cisco Systems, Orange and above all Microsoft Research for their financial support. Through this I met many interesting people, and gained insights into their industrial experience and outlook, which had an important impact on the way I think about research.

I have also been very blessed to meet my fiancée Euline during this time. She has brought great happiness into my life. Her love and support mean the world to me.

Finally, thanks be to God who has been a great source of strength and motivation throughout my life.

I want to dedicate this thesis to my parents who have given me the chance of a good education, and so much love and support over the years. I probably owe them much more then they think.

Declaration

This thesis has been written by myself, and the work reported in it is my own. The documented research has been carried out at Lancaster University as part of the LandMARC project [Lan01] and the Mobile-IPv6 Systems Research Lab [MSR01].

The work reported in this thesis has not been previously submitted for a degree in this or any other form.

Lancaster, November 2002 Stefan Schmid

Contents

List of Figures

List of Tables

Selected Publications and Presentations

The LARA++ Active Network Architecture

Component-based Active Network Architecture
S. Schmid, J. Finney, A. Scott and D. Shepherd, July 2001.
In *Proceedings of 6th IEEE Symposium on Computers and Communications (ISCC)*, pages 114–121, Hammamet (Tunisia).

LARA++: A Component-based Active Router Architecture
S. Schmid and D. Hutchison, February 2002.
Invited presentation at *Dagstuhl-Seminar 02071: Concepts and Applications of Programmable and Active Networking Technologies*, Dagstuhl (Germany).

Implementation and Evaluation Aspects

Flexible, Dynamic and Scalable Service Composition for Active Routers
S. Schmid, T. Chart, M. Sifalakis and A.C. Scott, December 2002.
In *Proceedings of 4th International Working Conference on Active Networks (IWAN)*, Lecture Notes in Computer Science (volume 2546), Springer-Verlag, pages 253–266, Zurich (Switzerland).

Applications for LARA++ Active Routers

Active Component Driven Network Handoff for Mobile Multimedia System
S. Schmid, J. Finney, A. Scott and D. Shepherd, October 2000.
In *Proceedings of Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS)*, Lecture Notes in Computer Science (volume 1905), Springer-Verlag, pages 266–278, Enschede (The Netherlands).

An Access Control Architecture for Microcellular Wireless IPv6 Networks
S. Schmid, J. Finney, M. Wu, A. Friday, A. Scott and D. Shepherd, November 2001.
In *Proceedings of 26th IEEE Conference on Local Computer Networks (LCN)*, pages 454–463, Tampa (Florida, U.S.).

Chapter 1

Introduction

1.1 Overview

This introduction sets the stage for the research carried out for this thesis. It introduces the concept of "active networking" and outlines the general path of research that has been taken.

An analysis of the evolution of packet switched networks shows that the original static network architecture was not designed for the requirements of today's fast moving global multimedia data network – the Internet. This chapter outlines how the novel paradigm proposed by active networking could advance the current network and overcome its main design flaws. A discussion of the applications of active networks and its advantages highlights the potential beneficiaries of this new technology, namely third-party network software developers, service providers and most importantly the end users.

This chapter concludes with an overview of the main research challenges that are targeted by this research effort, followed by an outline of the thesis structure.

1.2 Network Evolution

The Internet was born in the late 1960s. It started as an experimental network to prove the viability of packet switching [HL98]. In a packet switched network, end nodes (*hosts*) communicate with each other by transmission of data units called packets. The packets are independently routed from the sender to the receiver via the intermediate nodes (*routers*) based on the destination address. A family of protocols (for example, the Internet protocol IP [Pos81b] and the Internet control message protocol ICMP [Pos81a]) define how the individual network nodes communicate (for example, how packets are formatted, how they are forwarded, etc.).

In the last three decades of its existence, the Internet has grown exponentially into a huge data network, spanning the whole planet. Especially since the emergence of

the World Wide Web (in the early 1990's) and the beginning of the commercialisation of the Internet (in the mid 1990's), the growth of the Internet has been beyond any expectations. It has grown from fewer than 5 million nodes in 1994 to over 100 million hosts in 2001, and the number of users has already exceeded half a billion.

The key to this success is most likely the simplicity and cheap deployment cost of Internet technology. Traditionally, routers in packet switched networks have been designed as simple store-and-forwarding devices that included only the minimal processing necessary for the forwarding of packets. For example, in contrast to traditional circuit switched networks, which establish a separate channel of fixed bandwidth between the corresponding users based on a dedicated signalling protocol, packet switched networks provide only a best-effort service, which tolerates queue overflows (and as a consequence packet loss) inside the network.

However, the requirements of the original packet switched network architecture have changed in recent years. These changes can be summarised as follows:

- The Internet has evolved from a pure research platform to a commercially dominated network. As a result, requirements such as network security, efficiency, and scalability, which were not sufficiently considered in the original design, are paramount now.

- The use of the Internet has changed from a pure data network towards a multimedia data network. While initially file transfer and remote terminal log-in, then e-mail, were the primary Internet applications, more and more continuous or interactive multimedia applications using audio and video have been emerging recently. Unfortunately, such Quality-of-Service (QoS) sensitive applications have quite different requirements and therefore benefit from network services that are richer than simple best-effort packet forwarding.

- The long-lasting success of the Internet is accompanied by the problem that the network technologies for which the network has been originally designed have become obsolete, and new technologies have been on the increase. For example, more recent technologies, such as wireless or satellite links, could not have been considered by the early Internet designers. These technologies have often quite different transmission characteristics from those of traditional wired links.

- And finally, the continuous performance increase of common computers has reached a level where it has become economically feasible to deploy off-the-shelf computers "inside" the network in order to improve network performance (for example, bandwidth utilisation or security).

As the number and complexity of network services is constantly increasing, a recent trend has been to take advantage of the processing capabilities inside the network. More and more user or application specific processing is carried out inside the network, where it is most efficient and effective. The following examples illustrate this tendency:

- Network caching as a mechanism to reduce network load has gained a lot of attention in recent years [Rac00]. For example, most Internet service providers (ISPs) offer today large clusters of Web caches at the edges of their service networks[1] to minimise the traffic in the core network. Recent developments further suggest that content providers should even replicate their content throughout the Internet in order to reduce network traffic and more importantly access times [Ver02].

- For security purposes it is now common practice for organisations to deploy firewalls at the border routers (*gateways*) to their network service provider. This enables network administrators to define traffic classes that are allowed to pass the 'gate' between a customer's Intranet and the service provider network.

- The need for qualitatively better communication mechanisms for real-time traffic (for example, interactive audio and video) than simple best-effort forwarding has led to the investigation of QoS mechanisms for the Internet. The two relevant IETF standards – the Differentiated Service (DiffServ) [BBC+98] and the Integrated Service (IntServ) [BCS94] framework – rely on special processing in every intermediate router along the transmission path.

- Application specific gateways (for example, media gateways that transcode continuous media in real-time [AMZ95]) and proxies (for example, the anonymizer Web proxy [MRPH01]), as well as application-layer network support (such as application layer routing [GFC00] and multicast [MCH01] protocols) that have been recently proposed, require processing capabilities inside of the network.

- Further network services and protocols, which rely on computation inside the network that differs from simple packet forwarding, include dynamic address allocation [Dro97], network address translation [EF94], and virtual private networks [G+00].

This discussion has shown a range of network-side developments and solutions that augment the current network infrastructure and services. Ongoing efforts in this area suggest that this trend of incorporating computational elements inside the network is far from over.

[1] Although Web caches typically run on standard server hosts (end nodes), from an end user's point of view they appear as intermediate nodes that operate inside the network.

The majority of these network-side services are implemented as individual ad-hoc extensions – all with the goal of improving the network design to account for today's requirements such as performance, QoS, and security. However, the fundamental problem, namely that the network provides no architectural support for flexible extensibility, remains.

This thesis therefore investigates novel active and programmable network mechanisms that consider flexible extensibility through programmability as part of the fundamental architectural design. The aim is to provide a uniform interface for adding new functionality (for example, caching, security, QoS, etc.) to the network infrastructure – independent of the service at hand.

1.3 Active Networking

Active networking is a recent approach whereby intermediate nodes inside the network can be involved in the customised processing of control and data traffic. Active networking augments the traditional *store-and-forwarding* model of packet switched networks with a programmable element. The resulting *store-compute-and-forwarding* model enables the processing of data packets on intermediate nodes as they travel through the network[2] (see Figure 1.1). A key characteristic of this technology is the ability to rapidly create, deploy and manage new network services in response to user demands. The approach is motivated by both the trend towards network services that perform user-driven computation at intermediate nodes (see the examples in section 1.2), and the emergence of mobile code technologies that make network programmability attainable.

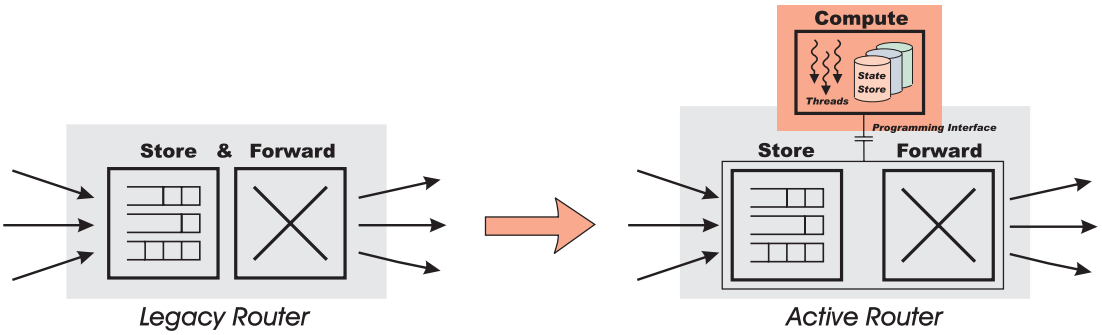


Figure 1.1: From Traditional Networking to Active Networking — The evolution of active networks enables the flexible programming of intermediate network nodes.

Changing the network, from a static entity that can do nothing more than routing packets from router to router into a dynamically programmable system, opens up a whole new realm of possibilities: customised data transmissions and application specific net-

[2]The form of computation is not exactly specified. But, in many cases the goal is to place as few restrictions on what can be computed as possible.

work support are just two examples. The increased flexibility and computational power inside the network leads also to a whole range of new applications, such as advanced network management and resource control.

While this may seem, at first, a revolutionary step, it is only a natural evolution. Today's network routers already contain a large variety of software components to support the growing demand of protocols and functions. A typical router, for example, incorporates software to support several routing protocols, firewalls, network address translations, virtual private networks and dynamic address allocation to name just a few. This increasing number of features makes the configuration and management of network devices more and more complex. In fact, one could argue that configuration management has become a form of programming. This illustrates that "programmability" is inevitable. The question simply becomes what is the best paradigm for network programming.

A central feature that distinguishes programmable engines from configurable ones is the programming model. While a configurable engine aims to establish a maximal set of high-level features that can be programmed (or configured) with a single action, a programmable engine focuses on identifying a minimal set of primitives (for example, system calls of host operating systems) from which one can compose (or program) a broad spectrum of features. Programmable engines provide a much higher degree of flexibility, which makes them the focal point of interest within most active network research.

Many different programming models have been suggested. A common approach is to provide a programmable engine at each intermediate node that can be programmed on a per-packet basis. Every packet contains in addition to the user payload (*data*) some form of active program (*code*) that is executed on each intermediate node as it traverses the network. Another major approach is to implement active networks without modifying the end-to-end protocols (i.e., packet format). In the latter case, active programs, or in other words any enhanced functionality or value-added services, are loaded onto the active nodes out-of-band prior to the transmission of data. Active networking approaches may also differ in various other aspects, such as the programming interface exposed to the network programmer, the degree of flexibility (i.e., whether code can be installed in the data path or only the control path), the programming languages, or the policies for installation of code and resource management.

1.3.1 Why do we need active networks?

Research into active networks is motivated by both *user pull* and *technology push*. The user pull can be perceived in the appearance of network-side services (for example, Web

or media caches, firewalls, multicast support, and mobile proxies) as a result of the changing network requirements (see section 1.2). These examples clearly show the need for services that reside inside the network.[3] The challenge of active network research is therefore to integrate generic processing elements in today's network architectures such that new network services can be implemented in a uniform manner. The programming interfaces provided by active networks offer a universal means to introduce new functionality into the network. This differs from today's practice of 'bolting-on' new network functionality to the existing architecture in an ad-hoc manner each time a new service is deployed.

The technology push, by comparison, is driven by the emergence of "active" technologies such as mobile code. These technologies comprise support for code compilation, encapsulation, transfer, and safe and efficient execution or interpretation of program fragments. Today, such mobile code technologies are mainly applied within individual end systems to provide application extensibility (for example, Web clients execute dynamically loaded Java applets) or remote programmability (for example, mobile agent systems provide a platform for the remote execution of agent programs). The challenge within active network research is to leverage and extend these technologies for use within the network.

The main drive behind active network research is the idea of developing communication networks that exhibit a degree of flexibility and customisability that is only known from programmable end systems. While the end nodes of today's networks (i.e., user PC's and workstations) are typically designed as open systems that can be flexibly programmed and hence upgraded with new functionality, intermediate nodes (i.e., routers and switches) are still closed, vertically-integrated systems whereby the node vendor provides both the hardware and software as a complete package. Thus, new protocols or services can only be tested or deployed when the manufacturer releases a software upgrade (as third parties cannot speedup the process). The development of new functionality is typically preceded by a long and awkward standardisation process. These different paradigms have created an increasing gap between the functions and capabilities of the users' end nodes and the intermediate nodes in the network.

Reconsidering the system architecture of those vertically-integrated network nodes is therefore a crucial step in network evolution. Enabling flexible extensibility through dynamic programmability, for example, has the potential to speed up the development and roll-out of multimedia support in the network. It may close the gap between the capabilities of today's end systems, which are already fully equipped with multimedia technologies, and the network, which still lacks basic support for multimedia applica-

[3]Note that although end users normally do not deliberately demand new services, these services are developed and deployed in order to satisfy the users' demands.

tions, such as resource reservation, real-time scheduling and multicast capabilities.

1.3.2 Who are the beneficiaries?

The advantages of a highly flexible and extensible network infrastructure are expected to benefit the networking landscape on various levels.

The fact that active network efforts aim at 'opening up' traditionally vertically-integrated network systems will enable third-party network software developers to compete with established network vendors. Separating the router hardware from the control software allows third parties to develop and sell network software (for example, router plug-ins or components) in much the same way as for end systems today. This will create a new competitive market for network software, services and applications.

For ISPs, the primary benefit of active networking will be the added flexibility that enables them to react quickly to changing network requirements. For example, it will allow ISPs to develop, test and roll out new network services in a time frame similar to those of end system applications and services (i.e., hours or days rather than years). The speed of service deployment will be especially important in this fast moving marketplace. In addition, ISPs will also benefit from better network management capabilities and more control over network resources.

The main beneficiaries of active networks will be the end users. The potential of network customisation on a per-user or per-flow basis (possibly even through user initiated and/or defined programs) opens a whole new realm of network applications. The wider range of network services and the faster time-to-market is expected to meet the users growing demands. At the same time, end users are expected to be the main driving force behind the progress in active networking and the actual deployment of the new technology. The market pressure resulting from the rising user demands might be the catalyst that makes ISPs, network software developers and equipment vendors move to a network architecture which provides the flexibility that is desired from today's network.

1.4 Research Challenges

The main aim of this work is to investigate flexible and extensible mechanisms that enable dynamic introduction of new functionality or value-added services into an operational network. This endeavour is pursued from the viewpoint of the end user and the network service provider as both have a great interest in network customisation and/or the processing of individual flows.

The key challenge of this thesis therefore is to design a novel active router architecture that provides the basis for flexible extensibility of network functionality. In order to verify the practicality of this architecture, prototyping an active node according to the new

design will be a major part of this undertaking.
The challenges of the architectural design are as follows:

- *Generic router platform (not tied to a specific application)*

  The design goal is to develop a generic programmable router platform to support the diversity of today's and future network services. The idea is to replace the numerous ad hoc approaches to provide specific services inside the network with a generic means that allows users (such as network administrators or trusted end users) to extend the network capabilities in a uniform way.

  Unlike most existing active router architectures, which are tied to a specific application domain (for example, network management or multicasting), the goal here is to start with a requirement analysis of a wide range of active network applications and services in order to consider the multitude of requirements in the architectural design.

- *Modular component-based architecture*

  Another key objective is to design an active router architecture that is truly component-based – not only the design of the active node, but also the active processing on the node, taking advantage of component features such as modularity, extensibility, and reusability. The active node can hence be programmed in units of active code called components, which are dynamically loaded onto the programmable nodes. These components will typically provide a new service or simply extend an existing service.

  The component architecture allows complex active programs and services to be split into simple and easy-to-develop functional components. This 'divide and conquer' approach eases the design and development of services. Moreover, it improves the granularity of service extensibility and reusability of components among services.

- *Fully programmable (not limited by a programming language)*

  A key advantage of programmable engines (over configurable ones) is that they can compose a huge spectrum of high-level features from a minimal set of low-level primitives. The architecture developed in this thesis focuses on the programmable approach to achieve maximum flexibility. Turing complete programming languages for the active programming are suggested for the same reason.

  Furthermore, programmable engines can be differentiated depending on the programming model that is supported. For example, programmability can be supported on the control or data path of a router or both. The design effort pursed

here focuses on full data path programmability. It enables components to be directly integrated into the packet forwarding path on a node.

- *Flexible service composition*

  The process of creating a service by assembling individual components is referred to as service composition. A key goal of this node architecture is to provide an elastic means that enables users to compose flexibly active services. A special focus is laid on dynamic and co-operative service composition. While the former enables extensibility of router functionality at run-time (i.e., without having to restart the node after an extension), the latter is concerned with the problems of feature interaction and interoperability of components from independent users. This is of special interest, as current active router architectures have barely addressed the problem of co-operative service composition among different users. For example, a congestion control mechanism installed by a network administrator must co-operate with the custom services loaded by other users (for example, a forward error control mechanism or a mobile handoff optimisation). Services of different users should not exclude each other as long they are not mutually exclusive by nature.

- *Configurable security*

  Safety and security are undoubtedly crucial aspects of the design of programmable network devices. Most active node architectures secure the nodes by limiting programmability through the active node design (for example, by forcing users to use safe code interpreters or programming languages). However, realising security through restrictions on the programming interface of the active nodes contradicts the goal of developing a fully programmable active node. Therefore, the architecture proposed here aims at maintaining maximum flexibility through a configurable policy based security mechanism. In this approach, the node configuration (i.e., security policies defined by the node administrator) determines the degree of programmability – not the architectural design.

- *Compatibility and transparency*

  The introduction of active network technologies in current networks, such as the Internet, depends largely on how easily they can be integrated with existing technologies. It is therefore a major objective to design the active router architecture in a way that enables seamless transitioning towards the novel networking paradigm. Most early active network proposals, for example, did not consider the transparent application of active computation a vital requirement, and hence, ended up with

solutions that rely on a network consisting only of active nodes or require modifications on the end systems. Such systems are obviously very hard to introduce in large networks such as the Internet.[4] Consequently, an important goal here is to design an active router architecture that allows transparent, and hence seamless, application of active computation on the transmission path. No change to the end systems and applications, or the intermediate nodes that are not directly involved should be required. Such transparent solutions have the advantage that a partial transitioning from conventional routers to active nodes – where value-added functionality and services are most effective – is possible.

- *Commercial feasibility*

  Another important factor for the success of active networking technology is its commercial viability. Many great technologies have failed in the past simply due to a weak business model. As a result, this work focuses on a solution that has evident beneficiaries and a likely commercial perspective.

  The challenge is to develop an active router architecture that enables third party development of network software. Breaking the closed or vertically-integrated systems design paradigm of current network devices decouples the role of the network software developer from the hardware vendor and thus opens up a new competitive market for third party router software. This is particularly promising as unhindered competition typically maximises the cost-performance ratio of products and services.

1.5 Thesis Structure

This first chapter of the thesis has introduced the concepts of active networking. It outlines how the new technology has emerged from traditional network technologies as a result of the growing demands of today's network users and applications. Furthermore, it provides the motivation for this line of research along with the main research challenges of this study. The remainder of this thesis is structured as follows.

Chapter two continues with an introduction of the general background and issues of active and programmable networks. It defines the basic methodology and introduces the main concepts. These include different architectural approaches towards network programmability, various programming models, and other important issues such as service composition and system integrity.

[4] Transitioning towards the next generation Internet protocol, IPv6, shows how difficult and long-winded the process of introducing a new protocol that relies on support throughout the whole network can be.

Chapter three provides a comprehensive overview of the current state-of-the-art in the field by introducing related work that is or has been under investigation at other research institutions. A special focus is placed on research into active network systems design and enabling technologies. Chapter three concludes with an overview of current work on active network applications and services.

Chapter four continues with a requirements analysis for active network systems. The requirements are derived from past experiences in active networking at Lancaster and a thorough study of related work as well as other influencing factors, for example commercial aspects such as the deployment of new technologies. From these general requirements a subset of requirements that forms the basis for the design of the LARA++ active router architecture and implementation is drawn.

Chapter five presents the design of the LARA++ active router architecture. This central part of the thesis describes in detail how LARA++ operates and how the component-based active node architecture enables network programmability through flexible integration and extensibility of network functionality. In addition to the basic node design, special focus is placed on the following key aspects: the service composition framework, the safety and security architecture, and the policing scheme.

Chapter six then describes the ongoing implementation efforts of developing prototype nodes of the LARA++ architecture. Due to the considerable extent of the LARA++ architecture, this chapter focuses primarily on validating the key aspects of the design through a 'proof-of-concept' implementation.

Chapter seven continues with a qualitative and quantitative evaluation of LARA++ and its prototype implementation. It evaluates how the LARA++ architecture satisfies the objectives and requirements identified in chapter 4 based on a case study and several example applications.

Finally, chapter seven concludes the thesis by drawing together the main arguments of this work and summarising the contributions that have been made. It also describes further work that could be carried out based on this line of research.

Chapter 2

Active and Programmable Networks

2.1 Overview

This chapter provides a general background on the field of active and programmable networks. It looks back to the initial developments of this trend in the early 1980's and shows how the field has evolved since.

The main focus however is to introduce the core concepts and issues of active networking as a basis for further discussions throughout the thesis. As such, this chapter defines the basic methodology for active networks and describes various approaches towards active programmability. Although the idea of active processing inside the network is not revolutionary (for example, packet routing is also a form of "in network" processing), dynamic programmability of the network by potentially arbitrary users requires architectural changes to the design and implementation of current network nodes. This chapter introduces several architectural approaches for the design of active network nodes and defines various programming models for dynamic network programmability.

Furthermore, the fact that active networks allow end users to program the network places increased safety and security concerns on such architectures. This chapter examines solutions within the context of active networks.

2.2 Background

The idea of performing active computation inside the network is not new. The earliest example of an active network dates back to 1981. Forchheimer and Zander at the University of Linkoping (Sweden) already envisioned a packet-based radio network, called Softnet [FZ81, ZF83], where every node of the network was programmable. The data-

grams transmitted in Softnet were considered small programs of a multi-threaded dialect of the FORTH programming language [Bro81].

Although the developers of this early amateur radio network did not use the same terminology – it was not referred to as "active" or "programmable" network – the key concept behind Softnet was the same. The nodes of their packet-switched radio network were programmable by the end users through the insertion of small network programs into packets which were processed by the nodes along the transmission path.

After that initial work in network programmability, research in the field disappeared for a long time. Researchers speculate [WGT98] that although the idea was considered to be very powerful and original, the lack of sophisticated mechanisms for mobile code and security hindered its acceptance. It can also be argued that the time was not yet right, as the need for network flexibility and extensibility was not evident in those days.

In the mid-1980s, a very simple form of network programmability was introduced for the control signalling in circuit switched telephone networks. The Intelligent Network (IN) marked the first attempt to separate the service logic (signalling protocols and services) from the switching system. The new system incorporated service specific "hooks" into the switching path, which enabled specialised processing of phone calls.

Apart from this distinct effort of the telecommunication industry, there was very little progress noted of research on network programmability for packet routed data networks for the rest of the 1980s. It was then in the mid 1990s when the idea of mobile agent systems arose that research in the field of network programmability was reanimated. The idea of mobile code that is loaded onto remote nodes in order to "program" the node is identical to the concepts behind programmable networks. In fact one could argue that network programmability is simply a subset of agent systems, as the nodes to be programmed and the computation are mainly restricted to network switches, routers and gateways, and the application domain of network services.

Finally, in 1994 and 1995 the concept of active networking emerged from discussions within the DARPA research community on the future directions of networking systems. In the following years, the new field has attained widespread interest throughout large parts of the networking community. The continuing funding of active network research by DARPA from 1996 onwards has had a significant impact on the development of the new community. For example, in 1997 DARPA funded already close to 30 active network projects. By 1999 this number has almost doubled.

2.3 Programmable Networks

The main drive behind programmable networks is the idea of developing communication networks that exhibit a high degree of flexibility, extensibility and customisability.

One of the key features that distinguish programmable networks from current configurable ones is the interface. While configurable architectures concentrate on establishing a maximal set of high-level features that can be configured through single atomic actions, programmable systems aim to identify a minimal set of primitives from which one can compose a large number of high-level features.

The aim is to make communication networks programmable to a degree that is known from programmable end-systems. It is expected that such a degree of programmability has the potential to close the gap between the capabilities of today's end systems and the network. While the former are already fully equipped with multimedia technologies, the latter still lack basic support for multimedia streaming, such as QoS and multicast.

However, programmability is a very generic term allowing many different interpretations and approaches. Likewise, network programmability has several facets. It encompasses already three fundamental approaches, namely Active Networks, Intelligent Networks and Open Signalling. Figure 2.1 illustrates these inherently different methodologies.

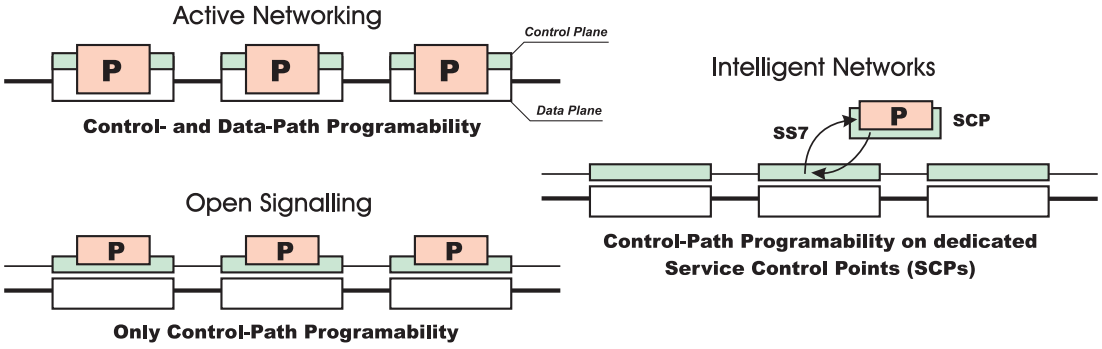


Figure 2.1: A Comparison of Programmable Network Approaches

2.3.1 Active Networks

Active networking is a new networking paradigm for data networks whereby the focus changes from pure packet forwarding to dynamic programming of the network nodes. Programmability is enabled through the exposure of a *programming interface*. It allows *users* (for example, network administrators, privileged users, or even end users) to dynamically modify the network behaviour and introduce new network functionality.

The scope of network programmability varies from control plane to data plane programmability and extends from very limited to highly flexible forms (depending on the programming interface). The ultimate objective in this respect, namely to have a general-purpose processing engine at each network node that can be flexibly programmed on a per-packet basis, however, is far from reality. The performance and security constraints of current networks and the lack of sufficient processing power inside the network make

this still a faint vision. Instead, current active network approaches have to balance the trade-offs between flexibility, security, and performance.

More than other programmable networks areas, active networks emphasise the need for mobile code and dynamic loading mechanisms to achieve on-the-fly programmability.

2.3.2 Intelligent Networks

The main objectives behind Intelligent Networks (IN) are similar to the aims of active networks, namely to provide a flexible extension mechanism (i.e., a programming interface) to speed up the deployment of new network services. It is mainly the target domain that is different. Intelligent networks focus on the introduction of value-added services other than basic telephony into the Public Switched Telecommunication Network (PSTN), rather than general data networks.

Intelligent networks use the Signalling System No. 7 (SS7) to incorporate service specific "hooks" into the switching path, such that enhanced processing of a phone call can be transferred to a Service Control Point (SCP) outside the switching fabric. The SCPs provide specific functionality, for example an enhanced service (such as local-call-rate or toll-free numbers).

While in the intelligent network an SCP could only provide a single advanced service, this restriction was lifted through the introduction of the Advanced Intelligent Network (AIN). It supports multi-service capable SCPs that enable the selection of a whole range of services via a 3-digit number. The same concepts have also been applied to the wireless telephone network in the form of the Wireless Intelligent Networks (WIN).

The intelligent network approach to providing extensibility of functionality inside the network corresponds to an active network architecture in which value-added services are executed on a nearby active server (or grid of servers).[1]

Despite the conceptual similarities to active networks, intelligent networks support only a very limited form of "programmability". The main restriction results from the fact that flexibility is only provided on the control path. Moreover, intelligent networks usually limit network programmability to rather coarse-grain and long-term services as service creation involves complex procedures.

2.3.3 Open Signalling

The open signalling community has been working on the standardisation of a flexible interface for the control plane of circuit switched data networks. Unlike the intelligent network, open signalling is not tied to a specific service domain such as telephony. The

[1] An example of such an active network architecture is the Alpine framework developed at Lancaster University [Ban01].

goal is to support a generic programming interface to network switches.

The idea behind open signalling is to define an abstraction from the physical network device in order to provide a switch independent interface between the switch controller and the switch fabric. This decoupling allows switch controllers and fabrics to be developed and evolved independently. The abstraction of a virtual switch and the exposure of programming interfaces enables third party software developers to build specialised or customised control architectures, while at the same time switch vendors improve their switching capabilities. This separation between the network hardware and the control algorithms is referred to as *virtualisation*.

The broadband kernel xbind [CHLL96] developed at Columbia University is a prime example of an open signalling architecture. The xbind service architecture is based on a distributed component based programming paradigm that allows modular construction of multimedia services. It includes components to implement mechanisms for distributed resource allocation, broadband signalling, and switch control and management.

Another example is the Tempest [VRLC97] framework developed at Cambridge University. It provides a flexible architecture that supports several independent control architectures on a single switch. These control architectures, called *switchlets*, can be safely installed on an operational ATM switch. The Tempest also supports refinement of services at a finer level of granularity through control plane modifications, called *connection closures*. In this case modification of services can be performed at an application specific level.

P.1520 [B+98] is an ongoing effort within the IEEE community to provide a reference model for open signalling (see Figure 2.2). The model identifies the key layers of the networking design space and illustrates the process of virtualisation. It defines clear interfaces between the various functional layers.

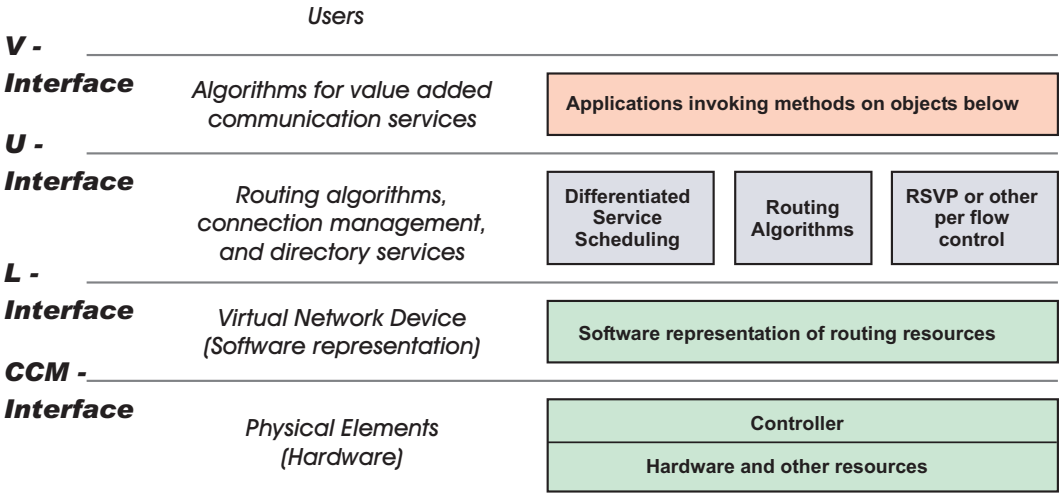


Figure 2.2: The P.1520 Reference Model for Open Signalling

Despite the flexibility gained through open signalling, its limitations are apparent: the flexibility is restricted to the control plane and targeted towards circuit switched data networks only. Furthermore, open signalling mechanisms support switch control only on the granularity of connection, not packets. Finally, open signalling does not envision control implementations being uploaded onto a switch (or removed) in a highly dynamic manner.

2.4 Active Network Methodology

As a result of the initial DARPA-funded research program for active networking [DAR98], an informal working group has been formed to define the scope and fundamental tenets of active networking along with an architectural framework [ANW98b].

According to this, an *active network* consists of a set of *active nodes* that are connected by a variety of network technologies. Each active node runs an *operating system*, responsible for managing the node's resources, and one or more *execution environments*. Each execution environment implements a virtual machine that processes the delivered active packets. An execution environment may provide general (or Turing-complete) computational services or simply a forwarding engine whose computation is controlled by the packet data.

Although this definition of an active network is very general, it can be argued that certain assumptions regarding the active node architecture are unnecessarily limiting. For example, assuming execution environments, or more specifically virtual machines that are tailored towards the processing of "active" packets (which comprise the code or simply a code reference as part of the data) or the explicit mapping between active packets and their execution environment, restricts the design space unnecessarily. As a consequence, this work proposes a more extensible active node architecture that does not limit "active computation" to code provided as part of the data packets and releases the explicit bonds between the data packets and the execution environments.

Besides the debatable definition of an active network, the DARPA-funded research community has agreed upon several fundamental tenets and objectives for active networks [ANW98b]:

Network API enables programmability: An active network must expose some form of application programming interface (API), referred to as the *network API*, to a group of or all network users (for example, administrators, service providers, and/or end users). The network API enables user's to "program" the network or individual nodes on a running network in order to achieve one or all of the following: deployment of new services, introduction of extended functionality into network

nodes, customisation of services for different applications, experimentation with new services.

Communication dominates over computation: The primary function of the active network remains as communication, not computation. Although computation services within the network are one of the main contributions of active networking, the active network platform is not designed to be a general-purpose distributed computing system.

Network packets are the data units for computation: The central data unit within active network computation is the network packet. The data packets are also the primary units multiplexed in the network as opposed to any higher-level entities such as circuits.

Minimal assumptions on underlying technology: The assumptions on the underlying packet-forwarding technology must be minimised, since active nodes might be interconnected by a variety of services that evolve over time.

Independent node administration: Active nodes need to be considered as independent administration units rather than units that are controlled by a common administration. The amount of global agreement required should be kept minimal. The partitioning of administration results in the need of explicit trust relationships between individual units.

Scaling to global active networks: The active network architecture must consider scalability issues and provision for very large global active networks. Appropriate management tools to administrate the overall network must be included.

Ensuring security and robustness of active nodes: Node-local and network-wide mechanisms to ensure security and robustness of the active network must be provided. Robustness should be considered independent of security, so that even the consequences of authorised actions are limited in scope. For example, an inadvertently defective active program should not cause any harm.

2.4.1 Taxonomy

Since the early days of active network research back in 1995, the field was divided into two different evolutions towards network programmability, namely the *active packet* approach and the *active extension* approach.

2.4.1.1 Active Packets vs. Active Extensions

The *active packet* approach supports network programmability on the granularity of individual data packet. Traditional or genuine data packets are replaced by so-called active capsules that carry the processing instructions (active programs) to be executed on their behalf. The program is processed by all network nodes along its transmission path that support the corresponding programming language. As a result of the active processing the packet payload, the forwarding behaviour (for example, packet scheduling or routing), or the state on the evaluating node may change. Once the packet execution has completed, or in other words, when the packet has been forwarded to the next hop, the active process on the node terminates and the resources of the active process (i.e., memory, CPU, bandwidth) may be released. Since the active code is distributed in-band with the data traffic and executed as the individual code packets pass the node, this approach is also referred to as an *integrated approach* to active networks [TSS+97].

The first active network approach leveraging this programming paradigm has been developed within the ANTS project [TW96, WGT98] at MIT in 1996. The main idea involved the development of a common communication model (as opposed to a special communication protocol) that enables dynamic and automated deployment of new network protocols.

By contrast, the second approach enables network programmability based on the so-called *active extensions*. Active extensions are active programs that are dynamically loaded and installed on network nodes to modify the behaviour of that node or in other words to enhance the functionality of the node. When executed on a node, active extensions provide extended services to the data streams passing through the node and other active extensions. They generally have a long-term impact that exceeds the lifetime of a packet by far. As active extensions are not tied to a particular data stream (i.e., they are loaded out-of-band), they can potentially be applied to multiple or all data streams. This enables active extensions to enrich functionality in a transparent manner. Due to the out-of-band distribution of active code and the separation of active extensions from the data packet, this approach is also referred to as a *discrete approach* [TSS+97].

The first active router architecture based on the concepts of active extensions has been developed at the University of Pennsylvania for the SwitchWare project [SFG+96, AAH+98]. The base layer of the programmable node (or switch) architecture supports extensibility based on active extensions, referred to as *switchlets*[2]. The motivation for this programming paradigm has been to enable flexible extensibility of a router's core functionality based on a system-level programming interface.

[2]Note that this term is also used within the open signalling approach Tempest (see section 2.3.3), but in a different context.

SwitchWare is also a good example of an effort that tries to integrate both approaches into a single active network architecture by combining the results of several projects. The active packet language PLAN [HK99] developed for SwitchWare allows the invocation and sequential composition of active extensions. For example, routing tables maintained by a switchlet (active extension) can be queried by PLAN programs (active packets) trying to find a path through the network.

2.4.1.2 Further Terminologies

Apart from those two fundamental categories of active network systems, active network architectures can also be classified according to the following characteristics:

- *Which OSI reference model layers are involved in the active computation?*

  Active network systems typically consider processing of the whole packet, or in other words all the protocol layers of a packet may be involved in the active processing – despite the restrictions imposed by the OSI reference model [ISO84] (see section 2.4.2 for further details). This is underlined by the fact that even the very first active network systems have assumed transcoding of media streams (which involves processing of application-layer data inside the network) a "lead user" application [TSS+97]. Hence, there is no need to explicitly define the OSI layer associated with the data processing.

  However, if an active node restricts active computations to a specific OSI layer (for example, transport-layer), the active network solution might be referred to as such (for example, layer-4 active networking).

- *How are packets associated with active extensions or active packets assigned to the appropriate execution environment?*

  A packet classification mechanism, which is typically implemented as part of the active node operating system, is required to associate data packets (or active packets) passing an active node with the appropriate active extensions (or execution environment respectively). For this reason, the classifier uses either *explicit* tags and/or packet headers as part of the actual packet data (for example, ANEP[3]). Alternatively, a more generic mechanism that allows classification on arbitrary packet data may be employed. In the latter case, packets can be *transparently* classified and assigned to active computations – without the need of explicit tagging. Hence active computation can be applied in a totally transparent manner without involvement of the end nodes.

[3]The active network encapsulation protocol (ANEP) [A+97] has been proposed as a means to assign packets passing a network node to active computations.

- *Where does the active computation take place?*

  Active network nodes are typically built on top of existing router platforms. In the case of a router platform built on a commodity OS, active processing can either be performed in protected mode (*user-space*) or system mode (*kernel-space*). System-level active network implementations on the one hand benefit from the flexibility of kernel space privileges (i.e., low-level data structures and system resources can be directly accessed), whereas user-level implementations on the other hand are typically more restrictive in terms of what can be processed and how efficiently.

  Most important however is the fact that system-level implementations of an active node (inclusive those that implement at least the packet capture and injection at the system-level) have the advantage that all types of active computations (i.e., integrated or discrete, and explicit or transparent processing) are supported, whereas a pure user-level implementation of an active node is limited to an *overlay* active network solution. The data must be explicitly addressed to the user-space application that hosts the active network implementation. System-level active network implementations, by contrast, enable layer-3 or *native* network data processing. Thus active computation can be transparently applied on the data packets passing through the node.

Table 2.1 summarises the basic operational modes of the different types of active network approaches and outlines their limitations.

	explicit processing	**transparent processing**
integrated approach		
native	native network-layer protocol takes care of active packet routing	n/a (in-band active code is explicitly bound to execution environment)
overlay	active packets are directly sent (addressed) to active nodes	n/a (packets are explicitly addressed to the overlay)
discrete approach		
native	packet tag or special header determines active extension(s) to be involved	classifier determines active extension(s) based on arbitrary packet data
overlay	packets are directly sent to active node; explicit identifier (packet tag or header) determines active service(s)	n/a (packets are explicitly addressed to the overlay)

Table 2.1: Summary of Basic Operational Modes for Active Network Approaches

What is not shown in this table are the dependencies arising from the different implementation approaches: kernel vs. user-space. Since transparent processing of the network data requires some form of kernel-space hook in order to access the network traffic in an efficient manner[4], a pure user-space implementation would not be rational. The same is true for explicit processing of network data in a native active network environment. Without a kernel-space implementation of the packet classifier that filters out the packets that are explicitly tagged for active processing, again all network traffic would have to be passed into user-space. However, for overlay active network solutions, where each active node has to explicitly address the next-hop node as part of the tunnelling mechanism, the advantages of a user-space implementation (for example, ease of development) outweigh the benefits of a kernel-space implementation.

2.4.2 Architectural Consideration

As active processing of data packets inside the network potentially changes the semantics of the network (for example, the routers can take on a very different role in an active network) which may have an adverse effect on end-to-end semantics, a discussion of these issues is included here.

Network communication is usually expressed in terms of the layered abstractions of the OSI Reference Model [ISO84]. The network layer (layer 3) provides support for transmission of packet data between end systems. It hides the lower-level network complexity (i.e., link layer data protocols and physical transmission of the data) from the transport layer (layer 4) and above. According to the OSI model, network routers are explicitly constrained to layer 3 processing as the transport-layer (or higher layers) provides end-to-end services, which are by definition the domain of end systems. Unfortunately, the implications for active networks would be that active computations inside the network are restricted to data concerning layer 3 (i.e., the IP header) and below.

Another well-established theory, known as end-to-end arguments [SRC84], comes to the same conclusion. The end-to-end arguments are a class of system design principles that impose a structure on the placement of function within a system. They provide a rationale for moving functionality upwards in a layered system, towards the applications. For example, a communication subsystem shared by applications with diverse needs should not provide functionality that is best implemented by the individual applications. Consequently, the end-to-end arguments also preclude the implementation of higher-level functionality inside the communication subsystem.

The rationale for this school of thought is based on the assumption that network

[4]Note that accessing the network traffic from a user-space application through a "raw socket" type interface would result in poor performance, as all the traffic would have to be passed into user-space just to see whether or not some form of active computation should take place.

nodes need to be very simple and very fast. However, these presumptions have changed due to the continuous increase of computational power and the advent of active and programmable network technologies (for example, specialised network processors). In contradiction to the end-to-end arguments, there are many situations where applying these principles is counterproductive. For example, many-to-many communications can be done by forwarding packets among multicast servers located outside the network, but it appears to be much less effective than providing multicast functionality at the network layer.

Likewise, there are many examples that show that active computation of transport layer or even payload information "inside the network" can significantly improve data communication between end-systems. Many applications, for example, place a large part of the functionality inside the network. Examples of such applications are: packet filters or firewalls, data caches, proxies, and congestion control or QoS mechanisms. And clearly, functionality that logically needs to be processed inside the network is best implemented and provided within the network. Moreover, there are other applications for which the performance or simply the design and implementation can be improved by coupling network-level processing with per-user or application-specific information. An example of this category is the mobile handoff optimisation described in chapter 7.

In conclusion, active networks do not contradict the end-to-end arguments. Instead, the design principles of the end-to-end arguments can facilitate the design conversation that leads to a more flexible and scalable architecture. For example, active network interfaces should be carefully designed such that only applications benefiting from specific functionality in the network are subject to the additional costs (for example, lower throughput or extra latency as a result of active processing).

As the desired level of computation within an active network node varies largely from one active application to another, all OSI layers above the the link layer should be accessible by active processes. Figure 2.3 illustrates how the difference between traditional network nodes (i.e., end-nodes and routers) and active nodes might be reflected in the OSI model. The marked areas (blue) indicate the OSI layers comprised by active processing within routers and end-nodes.

Figure 2.3 also suggests that active network architectures should be engineered in a *vertical* manner, whereby the network subsystem (protocol stack) above the link layer is collapsed into a single subsystem. The main purpose of the OSI layer model in this context is therefore to express the semantics of active computation on the various layers, rather than to direct the implementation of an active node.

On active end systems data traffic is typically also processed by the conventional protocol stack in order to provide the end-point functionality for network communications.

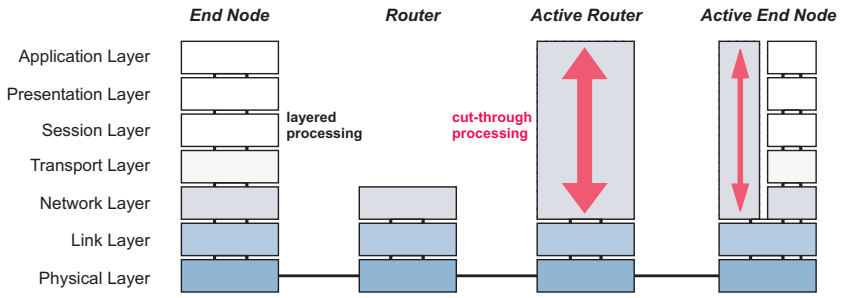


Figure 2.3: How active routers and end-nodes fit in the OSI reference model.

2.5 Architectural Overview of Active Nodes

This section provides an overview of the active node architecture defined by the DARPA Active Network Working Group (ANWG). The architecture is documented in a working group draft called "Architectural Framework for Active Networks" [ANW98b]. Although this architectural framework considers only one part of the design space for active networking (namely the integrated approach), it is considered by many to be a de-facto standard. The most likely reason for this misconception is the fact that the DARPA-funded ANWG, who has always focused on the active packet approach to active networks, has been the only formal working group in this area so far.

The proposed framework defines the fundamental components of an active node and how they interoperate. Figure 2.4 illustrates the architectural design.

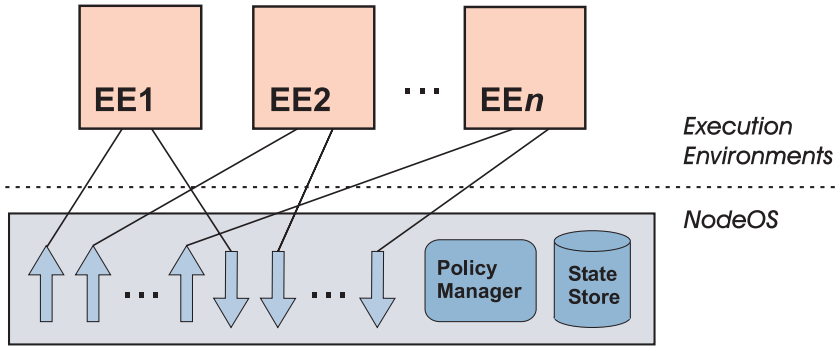


Figure 2.4: The Active Node Architecture according to the DARPA ANWG

One of the key design decisions of the architecture is that active nodes should not be restricted to a particular network programming interface or a fixed set of abstraction, but rather enable extensibility. For example, the architecture suggests allowing multiple active network programming interfaces to co-operate simultaneously on a single node.

As illustrated in Figure 2.4, functionality is divided into the active node operating system (NodeOS) and the execution environments (EEs). While the NodeOS manages and controls access to node local resources and system configurations, the EEs implement the active network APIs supported by the node.

2.5.1 Active Node OS

The NodeOS manages the physical resources (i.e., processor, network bandwidth, memory, etc.) of the active node and provides a basic set of abstractions for the EEs to implement an active network programming interface. These abstractions primarily include support for internal communication channels (required for passing packets between internal components), controlled access to node-local resources, and support for common services (for example, security policing and routing).

The internal communication channels can be either *anchored* within an EE or *cut-through*. Anchored channels are used to transfer packets between the EE and the underlying communication interface. Cut-through channels, by comparison, are used for packets forwarded through the active node without being processed by an EE. Channels are created during initialisation of EEs by specifying a set of *attributes*. These attributes include the modules that define the behaviour of the channel, as well as other properties such as addresses, direction of packet flows, MTU, and QoS requirements. In addition to the channel attributes, the EE needs to specify the packet *filter(s)* for the classifier.

An important task of the NodeOS is to schedule the communication channels and EEs for execution. The scheduling decision depends on both the computational requirements of the channels and the corresponding EEs, and the bandwidth requirements of the channels.

2.5.1.1 NodeOS Interface

In order to promote the wide deployment of active networks, the active network working group tried to identify a "common" interface between the NodeOS and the EEs. The results of this study have led to the specification of a "standard" NodeOS interface [ANW99].
The interface specification is influenced by the following underlying design principles:

- The common interface should be minimal, but extensible beyond the fixed point through the enclosure of special functionality provided by the underlying system and hardware.
- In accordance with the tenets of active networking, the interface is optimised towards packet forwarding rather than arbitrary computation.
- Functionality and mechanisms required for the active node interface that are not particularly unique to active networks are borrowed from an established interface. The interface specification is therefore designed in compliance with POSIX.

2.5.2 Execution Environments

The execution environments provide computational services (for example, a virtual machine or code interpreter) for active packets. Based on these active packets, or more precisely through the active code included within the active packets, users are able to control or program the active node and its forwarding behaviour.

Upon receipt of an active packet, a packet classifier identifies the matching EE and hands the data packet to the appropriate EE for execution. The execution of the active code instructions will normally alter one or more of the following: the packet content, the internal state of the EE and/or the operation of the active node. The function of the active code is not defined by the architecture and hence depends entirely on the provider of the EE and the developer of the active program. After the active code execution has been completed, the EE forwards the (altered) packet, sends one or multiple copies of the packet or simply drops the packet.

The EEs according to the DARPA active node architecture are considered to be entirely independent of each other and hence do no support any form of inter-EE communication and operation beyond the default communication interface (i.e., packet channels).

In order to enable network users to use an EE for the provisioning of active services, they must obtain the programming guide (i.e., specification) for the EE. Therefore, EEs require publicly available documentation on how to program active nodes (i.e., the specification of the programming interface exposed by the EE) and a description of the packet filter (for example, the ANEP packet identifier) that must be included in the active packets when addressing the EE.

Finally, the DARPA active node architecture does not encompass mechanisms for dynamic integration of EEs on an active node. It is left to the NodeOS implementation to decide whether or not such mechanisms are supported.

2.5.3 Summary

Although the DARPA active node architecture described in this section is widely respected and often considered to be the de-facto standard for active networks, the architecture is only partially useful, as it is largely tailored towards the active packet approach to active networking. The active extension approach to active networking is simply disregarded. As a result, the architecture restricts active programmability to the functionality provided by the programming interface of the high-level EE(s) available at a node and not beyond that. It will be shown throughout this thesis that the architecture lacks further important features, such as a transparent means to apply active computation to data streams of conventional applications and a flexible composition framework to ensure interoperability between active services.

2.6 Programming Models

This section describes in more general terms the various programming models that have been discussed in the context of active programmability. These programming models can be classified according to the code distribution mechanism and the program encoding used for active programs.

2.6.1 Program Distribution

Two fundamentally different models of program distribution have evolved during recent years of active network research. Active programs can be distributed either in an *in-band* or *out-of-band* fashion.

2.6.1.1 In-band Approach

The idea of in-band active code distribution has been pioneered by the ANTS project at MIT [TW96]. Their proposed communication model replaces traditional data messages with small programs, called active capsules. Since these capsules manage the payload simply as a data structure, the differentiation between packet headers and payload becomes superfluous.

In-band active code distribution typically used in conjunction with active packets (integrated approach) has several interesting implications for packet-switched data networks: First, as the routes through the network are generally not fixed within such networks, packets of a stream can follow potentially different routes. Therefore, to ensure that active programs are processed by every intermediate node along transmission path, the active code must be included in every data packet of a stream – not only the first packet. Second, active programs are limited in size[5] in order to sustain efficient use of the network bandwidth. These limitations clearly have an adverse impact on the capabilities (and hence the usefulness) of the active computations.

In general, however, in-band active code distribution is considered more universal than the active packet approach. It only requires that active code is delivered as part of the actual data stream, rather than through a separate control stream.

2.6.1.2 Out-of-band Approach

Out-of-band distribution of active code, which is typically used for loading active extensions in discrete active network approaches, allows users to dynamically upgrade the functionality of the router or to customise its behaviour. Once downloaded and installed, the active programs (or extensions) take part in the subsequent processing of the data

[5]The programs should be small enough so that appropriate space remains for the payload.

streams on the router. This approach has been initially developed and deployed as part of the programmable switch project at UPenn [AAH+98].

The mechanism for processing data packets passing through a node is here architecturally separated from the task of injecting active programs into a node. Users (or programmers) of the active network must first inject their programs into the network devices before the data streams can benefit from the extended functionality. Consequently, the packet format and transmission mechanisms for the actual data streams remain typically the same; no alteration of the packet type or header is required. The active programs involved in the processing of a data packet are usually determined through packet classification (based on the packet content – headers and payload).

On the one hand, the shortcomings of this approach are clear. The initial distribution of the active code must be explicitly performed before the data streams can take advantage of the new functionality. This may raise the problem that additional latency for loading the code is introduced when immediate programming of the active network is an issue. On the other hand, the separation of active code injection and program execution is advantageous if (1) high-level security checks need to be passed before a program can be executed on a node, or (2) active programs are relatively large in size. In both cases, code injection and loading on a per-packet basis cannot be performed efficiently.

2.6.1.3 Combination of In-band and Out-of-band Approach

The previously described program distribution approaches, namely the in-band and out-of-band approaches, define two distinct mechanisms for program distribution. However, since these approaches are complementary, many intermediate solutions have been developed over the years.

A technique that is commonly applied is to replace the code within the active packets with a *code reference*. Thus, upon receipt of such a packet, the active node loads the respective function from a local code store (i.e., cache). If the code is not available locally, it will be fetched from a remote location via an out-of-band loading mechanism. While both in-band and out-of-band code distribution are *explicit* loading mechanisms, this approach is referred to as an *implicit* or *on-demand* loading mechanism. It is triggered by a "program fault" (similar concept to a page fault in the context of virtual memory management) that occurs when the execution of a program, which is not yet loaded, is scheduled.

The main advantage of this approach is that the active code is only transmitted and loaded once on every active node. All subsequent packets can reuse the code without incurring additional network load (or processing load to instantiate the program).

Furthermore, this approach allows optimised code to be loaded upon a function miss. For example, active routers could specify their local environment (i.e., hardware platform and/or execution environments) when retrieving the active code from a remote server in order to load the best suited program version. Moreover, this approach has the advantage that vendors and network administrators can control the active program distribution (for example, loading can be restricted to trusted servers only). This allows, for example, active programs to be checked for errors and/or security problems prior to the loading and instantiation of the code. Offloading these time-intensive tasks has the potential to reduce the bootstrapping time of active programs significantly.

Further variations of program distribution mechanisms that are specific to individual research projects are outlined where appropriate in chapter 3.

2.6.2 Program Encoding

This section introduces various program encoding schemes that have been used for program encoding within active network research over the past few years. In particular, the encoding types for interpreted, intermediate, binary, source and self-specialising programs are discussed here. The study identifies their strengths and weaknesses in the context of active programmability and program distribution.

2.6.2.1 Interpreted Code

This category of active program encoding encompasses all interpreted programming languages that are used for active programmability, such as Safe-Tcl [Bor94] or NetScript [YdS96].

Interpreted programming languages have the advantage that safe processing of mobile code is easy to accomplish. Since the code must be interpreted (i.e., it cannot be executed outside the interpreter), safety is only a matter for the interpreter; and thus, the programming environment is safe as long as the interpreter is safe. Another key objective of interpreted code is platform independence. The high-level code representation is entirely independent from any platform specifics and therefore it is merely a matter of porting the interpreters to a specific platform in order to achieve platform independence.

The main drawback of interpreted code is the cost of processing involved with code interpretation (which typically is significantly higher than the cost of binary code execution). A secondary disadvantage of this program encoding in the context of active networks is the size of the code representation. Source code typically is significantly larger in size than for example binary code. However, the code size of interpreted code can be largely reduced through code compression at the expense of extra processing.

2.6.2.2 Intermediate Code

Intermediate code is probably the most commonly used encoding type for active programs. A very popular intermediate programming language used within active networks is Java [GM95, JAV]. Another example is the ML derivative Caml [CAM] used within SwitchWare.

Intermediate code, also referred to as *bytecode*, is produced through program compilation. Since intermediate code was mainly designed as a platform independent code representation, it is especially suitable for mobile programs. The mobile code is "interpreted" by a bytecode specific virtual machine on the target machine where the code is executed. As a consequence, safety of intermediate code systems relies entirely on the safe interpretation of the bytecode by a trusted virtual machine [Gos95].

Interpretation of intermediate code is typically significantly faster compared to interpreted code, because bytecode is already highly optimised towards fast interpretation through a platform specific virtual machine. However, since the virtual machine must still map the bytecode onto machine code at execution time, the performance of intermediate code interpretation is still considerably slower than binary code execution. In order to optimise the performance of intermediate code, many intermediate languages off-load the responsibility for operand validation from execution time to compile time. As a result, intermediate code is typically based on type-safe programming languages.

2.6.2.3 Binary Code

Active programs, directly compiled to platform-dependent binary code, have the best possible performance characteristics as the code is directly executed by the processor(s) of the target machine. However, in comparison to other encoding approaches, this is most challenging with respect to safety of active nodes. Since binary or machine programs "run" directly on the underlying hardware, sophisticated safety and/or security mechanisms are required to protect the active nodes from malicious active programs.

Safety with respect to protection from malicious active code and users can be simply achieved through conventional security mechanisms. Namely code signatures and user authentication techniques can be used to ensure that active code is not tampered with and users are authorised to execute a program. However, these security mechanisms rely on external trust relationships which make these measures typically less safe than node-local enforcement techniques. Furthermore, code safety with respect to robustness of an active node should be independent of security, so that the consequences of the actions of even the most-trusted users are bounded.

Binary code execution therefore requires system-level safety measures such as memory protection and fair scheduling of processing resources. These measures either verify

statically (at compile time) whether a program conforms to the safety regulations or enforce safety dynamically (at run time). While static or compile time measures can be more efficient with respect to run-time performance, run-time measures typically provide the highest-degree of safety[6]. The performance disadvantage of run-time safety is typically minimised through dedicated hardware support (i.e., virtual memory management and pre-emptive multi-tasking) available in most modern processor architectures.

For example, the Omniware mobile object-code architecture [ATLLW96] uses a mechanism called software-based fault isolation (SFI). It enforces a set of rules for instructions (for example, restrictions on how address arithmetic is performed) that are used to define a "sandbox" within which a program can do anything, but not escape. While Omniware comprises a special execution environment in order to enforce the sandbox at run-time, it has been shown that "rule compliance" of object-code can also be verified on the target machine before execution takes place by means of new techniques referred to as proof carrying code (PCC) [Nec97, Nec98] (for further details see also section 2.8.1.2).

The SPIN project[7] [BCE+94], by comparison, suggests the use of trustworthy compilers for mobile code generation. Herein, compilers are responsible for verifying that programs do not access any data outside their authorisation scope. Before a binary program is executed on a target machine, the active node must confirm that the compiler used to generate the binary code can be trusted.

2.6.2.4 Source Code

This approach suggests the distribution of active programs in the form of source code. The active nodes use a just-in-time compiler for "on-the-fly" compilation of active programs arriving at a node. As a consequence, source code encoding is not suitable for active packet based approaches, because additional latency resulting from the source compilation would occur on a per-packet basis. However, for discrete active network approaches, where active programs are loaded only once at the beginning, the delay caused by on-the-fly compilation of active source code is less critical.

Despite the need for an "on-the-fly" compilation mechanism, the distribution of active code in the form of source code has several valuable advantages:

- Source code is platform independent. Given that a compiler for the target platform is available and that any architectural differences (for example, byte order or memory alignment) have been considered by the programmer, the same active code fragment can be used across different active node platform.
- Safety techniques prior to compilation and compiler-based safety checks (for ex-

[6] Note, safety is enforced even in the case of an unexpected program failure.

[7] See also section 3.3.1.5 for further information on the SPIN operating system.

ample, type safety or range checking) can be applied on the target system.

- No external trust relationships are required as the compiler is part of the local system.
- On-the-fly compilation on the target machine allows the active node to customise the compilation (for example, to account for platform dependent optimisations).

The recently developed programming language 'C [EHK96] is an example of a safe on-the-fly compiler.

2.6.2.5 Self-specialising Code

Program specialisation is a technique that optimises a program with respect to the context in which the program is executed [JGS93]. This technique is used to generate mobile code that runs "anywhere". Instead of writing and maintaining a different program for every context in which the program may be executed, there is only one version of the code that adapts itself to the respective execution context. The concept behind self-specialising code is to transmit *code transformations* that enable it to generate customised code and types, rather than transmitting regular code.

A self-specialising code generator developed at the University of Pennsylvania [Hor00] has been designed with the objective of combining the concepts of program specialisation and program verification. Previously developed type systems [LL94, TS97, WLP98] can be used to provide safety guarantees for the self-specialising programs (for example, all generated code must be well-formed). Another technique considered for use in conjunction with certifying compilers is the verification of type-safety before program execution, but after the code is compiled [MDCG99].

2.6.2.6 Summary

The various program encoding schemes introduced in this section have different characteristics with respect to mobility, safety, programmability and performance. Table 2.2 compares the different code representations and evaluates their suitability for active network environments.

Each approach has its strengths and weaknesses: interpreted languages facilitate the implementation of safe execution environments for active code; intermediate encoding is largely platform independent and at the same time modest in performance; binary programs are preferable for computationally expensive or frequently used modules due to their superior performance; source code based mobile code overcomes the portability problem of binary programs, but at the expense of just-in-time compilation upon code arrival (which limits its usability to discrete approaches). Finally self-specialising code

Code Representation	Mobility	Safety	Programmability (Flexibility)	Performance
Interpreted Code	Yes	Simple	Limited	Slow
Intermediate Code	Yes	Not hard	Good	Medium
Binary Code	No	Hard	Complete	Very Fast
Source Code	Yes	Not hard	Complete	Very Fast, but compilation required
Self-specialising Code	Yes	Simple	Good	Very Fast, but "specialisation" required

Table 2.2: Comparison of Encoding Techniques (with respect to their qualities for mobility, safety and efficiency)

achieves mobility, safety and high performance through dynamic code specialisation at loading time.

2.7 Service Composition

Active node architectures that support sequential processing of multiple active programs in the data path (which is an obvious prerequisite for flexible and extensible architectures) must address the issues of service composition. A special working group has evolved within the DARPA active network program with the goal to investigate and standardise mechanisms for the composition of active services on a single node or EE [ANW98a].

A composite service is constructed from a set of components by means of a composition method. The composition method provides the syntax and semantics for creating services from components. It determines the set of software components needed to compose a service and the bindings to join these components. Composition methods vary from fairly simple or static means of defining a service composite to highly flexible and dynamic mechanisms. For example, a composition method may be as generic as a programming language and hence allows users to dynamically "program" a service composite.

Composition methods can be characterised according to the following criteria:

Sequence control: Mechanisms for controlling the invocation order of components are referred to as sequence control. Examples of sequence control approaches are *sequential* and *concurrent* invocation. More complex control mechanisms and

structures (for example, programming languages or dynamic graphs) are required when components and their composites can interact with one another.

Shared data control: Mechanisms for sharing data among components are called shared data control. Examples of methods for sharing data include *explicit parameters*, *shared objects*, and *inheritance*. Components executed concurrently can also share data based on *message passing*.

Binding time: Binding time refers to the time when a composite is created. Examples of the binding time of a composite are *specification time* (i.e., when the composite is defined/compiled), *creation time* (i.e., when the composite is created or initialised), and *run-time* (i.e., during execution).

Invocation methods: Invocation methods are the events that cause a composite or its components to be executed. The most common invocation method is the arrival of a network packet (i.e., *packet arrival*). Other possible invocation methods include *timers* and other *events* originating from hardware components, the NodeOS or active services.

Division of functionality: A composite is defined through a combination of content carried within the active packets and content residing on the active nodes. For example, a packet might carry a script program that includes calls to node-resident components.

An examination of existing composition methods has revealed that current methods typically lack support for multiple, independent entities. Most existing methods assume that only one entity (for example, a single user or one script) specifies the entire service composite for a packet or flow. However, in order to consider more complex scenarios (see for example section 5.2.1), the composition method must allow several entities to participate in this process. Therefore, the following characteristic to distinguish different composition methods has been added as part of this work:

Composition control: Active service composites that are created and controlled by a single entity are called *solitary*. Others that allow multiple, independent entities to partake in the composition process are referred to as *co-operative*.

In comparison to *implicit* composition, which is simply a result of executing a program that provides a composite service by invoking other components, *explicit* composition of services from independent programs or components (for example, active extensions) relies on an autonomous composition method. While the former depends solely on

the capabilities of the programming language and the interfaces provided by the available components, the latter depends entirely on the composition method(s) supported by the active node architecture.

The following basic models for explicit composition of active services have evolved within the past few years of active network research: (a) de-multiplexing, (b) static or dynamic plug-in approach, and (c) classification-based composition.

Model (a) is used within the DARPA active node architecture described earlier in order to de-multiplex active packets to the corresponding execution environment. It is rather restrictive as it limits packet processing to a single execution environment (or active extension[8]). Model (b), by comparison, allows composition of independent active extensions by means of a static graph. However, since static structures have proved to be too inflexible for many applications, dynamic means for defining the "slot logic" (for example, a dynamic graph or "underlying" program) have been proposed [MBC+99]. The last composition model (c), which has been developed as part of this work, is based on the concept of multi-stage packet classification (i.e., active services are composed on a per-packet basis depending on the packet content). A dynamically extensible classification graph structure ensures sufficient flexibility (for more details see section 5.6). Figure 2.5 illustrates the fundamental concepts of these three composition models.

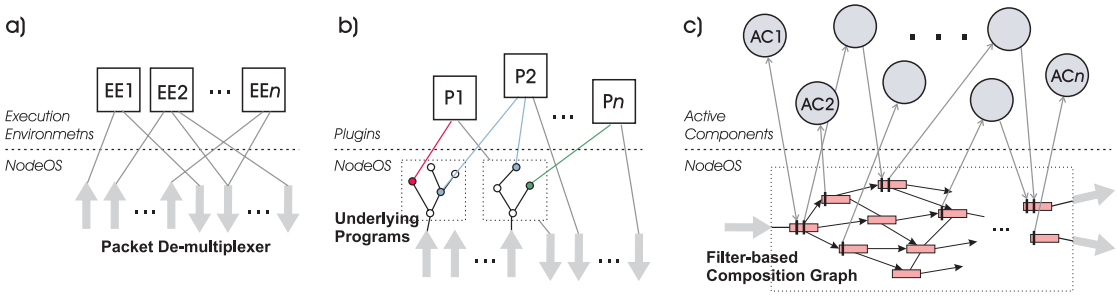


Figure 2.5: A comparison of several key explicit composition models for active node architectures: (a) de-multiplexing, (b) static or dynamic plug-in approach, and (c) classification-based composition.

2.8 Safety & Security

As is the case for most communication technologies, global acceptance of active networks relies heavily on adequate safety and security mechanisms. Both aspects of active networking – communication and computation – must be considered.

[8]It should be noted here that execution environments for integrated or active packet based solutions are only a specialisation of active extensions (which provides computational services to active packets) and hence should be considered as such.

Traditional packet-switched networks offer relatively little scope for security threats; routers merely store-and-forward packets, requiring only sufficient memory for packet queueing and insignificant processing resources for slow-path processing[9]. Conversely, active networks allow for a much broader range of attacks and safety problems resulting from active code execution. Providing a much higher degree of flexibility through programmability makes active networks clearly more vulnerable.

As a consequence, active network systems require sufficiently powerful safety and security mechanisms. While safety mechanisms are needed to ensure proper operation of the active node, security methods are required to control access to the programming interface and node-local resources, as well as more generic system services.

Key to the provision of safety and security within active networks is the satisfaction of the following requirements: (1) authentication and authorisation of users (i.e., active programmers) and mobile code, (2) validation of the code integrity (for example, through code signatures), (3) safe evaluation/execution of the active programs, and (4) run-time control of resource usage.

One of the primary characteristics of current network devices is robustness. Thorough testing is carried out before new devices are released. This indicates how important system reliability is for network devices, and even more so for active routers, where packets are not only forwarded, but also processed by active programs. In particular, since active programs may operate directly on the data path of a router or even in high-privileged mode to perform system-level operations, it is vital that active processes are restricted from consuming node resources in a way that would deny servicing other users or even cause the node to crash.

As a result, the provision of fault-tolerance through run-time safety measures is just as important as providing security. Precautions that ensure harmless execution of active code are essential because even trusted code that has been obtained securely can behave unexpectedly and hence result in a system failure.

2.8.1 Safety

Safety mechanisms for active node architectures should protect the nodes from both malicious and erroneous active code. Mechanisms to protect node-local resources, and prevent active programs from disrupting other active programs or even locking up the whole system due to resource misuse are essential.

This section introduces two key approaches towards safety, namely operating system and programming-language based protection mechanisms that are investigated within current active network research. Although these approaches are typically used separately

[9]Route lookups processed in software or processing of IP options [Pos81b] are commonly referred to as slow-path routing.

by most active node architectures, they are complementary in principle.

For a more comprehensive survey of safety mechanisms for mobile or active code, the reader is referred to a study by Wetherall [Wet95].

2.8.1.1 Systems Mechanisms

System-based techniques to protect an active node from malicious user code are also referred to as *software-based fault isolation (SFI)* or *sandboxing*. The idea is to execute code only within the restricted scope defined by a sandbox. Inside the sandbox, arbitrary user code – even malicious code – can be executed. The sandbox ensures that the code cannot break out of the safe processing environment. Malicious programs trying to exceed the sandbox boundaries are typically stopped and removed.

The user-space process model, common in recent operating systems, shows how protection based upon SFI works. A user process, or in other words the sandbox for a user program, defines the memory boundaries and processing quanta for safe program execution. A program executed by such a process can therefore not harm the operating system, or starve other user processes.

Sandboxes are typically enforced by means of software mechanisms. Modern operating systems also exploit the hardware capabilities of the CPU hardware (i.e., virtual memory, protection rings, etc.) to enforce the sandbox boundaries. For example, the virtual memory manager (VMM) defines the memory pages or segments a process can access. Recently published work [CVP99] has also demonstrated the viability of using hardware to segment not only user-space processes, but also kernel level modules. This is particularly valuable for active networks as it allows the safe execution of active code in privileged mode.

The main advantage of the system-based approach to safety is that active nodes executing mobile code are responsible for their own safety. The burden of having to use a particular (possibly new) programming language is taken from the programmer. Furthermore, the system does not rely on external components to ensure safety (for example, a trusted compiler for the code generation). Moreover, one could argue that in the case of active network nodes where robustness and reliability are vital, such system-based protection mechanisms should be mandatory anyhow in order to cope with unexpected run-time software failures.

Although operating system-based protection mechanisms can be very safe[10], context switching between concurrent processing environments (or sandboxes) is typically costly. Thus, in the case of active nodes, where efficient processing is vital, this approach is only applicable with lightweight optimisations (for example, user-level scheduling).

[10]Note that reliable operation of hardware components can be easily tested.

2.8.1.2 Programming Language Mechanisms

In contrast to operating system-based protection, research in the field of SFI and mobile code systems has developed various programming language-based sandbox mechanisms; for example, special language features are exploited to ensure safety during mobile or active code execution. The concepts behind these language-based protection mechanisms include type checking, type certification and program verification.

Type Checking

Type checking is a mechanism to ensure type safety for typed programming languages. Based on the type information provided by the programming language, type checkers can verify that a program uses the types (i.e., variables, data structures, functions, etc.) in a safe manner. Depending on the semantics of the language and the degree of type information, various levels of safety can be achieved. Examples of strongly typed programming languages used in the context of active networking are Java [JAV] and Caml [CAM].

Typed programming languages are designed for *static* or *dynamic* type checking, or both. Static type checking is done once at compile time, whereas dynamic type checking is done at execution time. Modula-3 [Nel91] is an example of a statically checked programming language. SmallTalk [GR93] and Caml, by comparison, are examples of dynamically checked languages.

The choice between static or dynamic type checking is a trade-off between performance and safety. In the context of active networks, static type checking has the advantage that no processing overhead for type checking is introduced during execution of the active code. Furthermore, it is beneficial as type errors, which may result in a run-time error, can be detected at compile time, before the code is distributed. A shortcoming of static type checking for mobile code systems is that the target machine (i.e., the active node) must trust the compiler of the code provider.

Dynamic type checking, by contrast, is advantageous as absolute type-safety can only be decided at run-time. Only dynamic type checks can spot certain type conflicts or run-time errors (for example, a reference to a wrong type or an array overrun) and take appropriate actions. A good example to illustrate the power of dynamic type checking is the strongly typed language Caml. It is capable of providing memory protection for active programs based on dynamic type and bound checking.

The limitations of static type checking (for example, in many cases type casts can only be decided at run-time) have led developers of several statically type-checked programming languages to include special run-time support that enables dynamic type-checking. For example, Java supports dynamic checking for type casts from a class instance to a

sub-class instance at run-time[11].

The main shortcoming of strongly typed languages in the context of active networking is that programmers have to bear restrictions regarding type conversions (i.e., references to data structures cannot simply be cast to similar types for the sake of efficiency) and memory operations (i.e., strongly typed languages lack the concept of bare memory), which make efficient processing of network data exceptionally hard.

Type Certification

Type certification is based on the same concepts as static type checking. Specialised type certifying compilers are used to extract static type information from the source code of an active program. This information enables active nodes to verify the type safety of active programs before program execution, when distributed along with the active code.

The assumption behind this approach is that programs that pass the static type check at compile time should be safe to execute. Preserving the type information at compile time by translating them into an intermediate language enables the target node (consumer) to validate the safety of the program before execution (without the source code). The important advantage of this approach is that the consumer does not have to rely on the correctness of the producer's compiler.

Examples of intermediate formats for type certified compilers are the Java Virtual Machine Language (JVML) [LY96], FLINT [Sha99], and several ML derivates [SA95, TMC+96].

Program Verification

The task of verifying the correctness of computer programs based on mathematical logic is usually called program verification. Although verification of a set of axioms based on reasoning principles is mechanisable and thus can be processed by computers, program verification is known to be very difficult[12]. There are still no practical solutions for the verification of large generic programs (such as "real-life" applications written in common, non-formal languages). However, progress in the field of program verification has led to an understanding that limited properties such as safety can be handled successfully. An emerging technology in the field of program verification that considers only the safety properties of the code has become known as proof carrying code (PCC). This recent discovery has been made by Necula and Lee [NL96, Nec97].

[11] Note that an exception is thrown in the case of a mismatching type cast in order to allow the program to recover at run-time.

[12] Program verification for non-formal programming languages relies on a formal specification of the program and a proof that the program is equivalent to the specification. The difficulty is finding the formal specification (since it is non-mechanisable) and proving its equivalence.

PCC allows a code producer to generate automatically a formal proof of safety along with a compiled program. The consumer can then verify the proof in accordance with its safety policy before installing or executing the code. PCC is designed to prove program safety rather than program correctness. Guaranteeing the execution of a program is safe is much easier than proving a program correct.

Although PCC has been developed as a generic safety measure for mobile code systems, it turns out to be very valuable for active networks for the following reasons:

- Verifying a formal proof for PCC is easy, even when the proof may have been very difficult to create. The burden of generating the proof is shifted to the code producer, where it is only performed once at compile time.
- Consumers of PCC (i.e., active nodes) minimise their trusted component base to the proof-verifying component on the local node.
- The concepts of PCC are not tied to a particular programming language or compiler, and thus can be applied to specialised active programming languages as well.

In summary, the main advantage of using a language-based approach to system safety is that minimum support is required in the NodeOS. However, this simply shifts the burden to the language compiler or interpreter, and the execution environment processing the active program. The disadvantages are that a special execution environment for the safe interpretation of the (intermediate) language must be provided on the target node. And although the use of intermediate languages typically simplifies portability, efficiency of the program execution certainly suffers from the evaluation of the bytecode. Furthermore, forcing active programmers to use a specific language because of system safety may restrict the problem domain that can be addressed or the efficiency of the implementation.

2.8.2 Security

Security within active networking is vital to prevent unauthorised users or active programs from gaining access to critical system services and configurations, and from consuming network and system resources (i.e., processing cycles, memory, and bandwidth). The security architecture of active nodes must therefore provide secure access control for a node's programming and system interface.

Two key components of the security architecture are the policy database and the enforcement engine. The separation of policy specification and storage from the policy enforcement maximises flexibility. For example, it enables dynamic policy specification at run-time.

The separation of security policies for different subsystems, namely the NodeOS, the EEs and the active code, is recommended to improve manageability. Allowing policies of a highly privileged subsystem to take precedence over a less privileged subsystem guarantees safety, besides the improvement in manageability. For example, it allows EE policies to specialise, but not override, NodeOS policies.

The notions of *principal* and *credential* have been introduced [ANW01] to facilitate security policing within active network architectures. A principal is the entity on whose behalf security decisions are made. It may be an individual user, a group of users, or an active program. A principal's request to access a certain resource or system service is granted or denied depending on its authorisation (i.e., its authenticity and access privileges). Consequently, since security within an active network is primarily a matter of authorising principals, the concept of credentials, which embodies both a description of the identity of a principal and its access privileges, has been proposed. Credentials are usually passed as a parameter to the interface function. The policy enforcement engine intercepts the function call and verifies the credentials. This involves checking the principal's identity (i.e., user, group or role) and access privileges against the policy database to determine whether or not the call is authorised.

Experience has shown that the attempt to describe security policies in terms of each individual principal and its privileges for a specific object does not scale. Instead, it is desirable to define groups of principals and objects with common privileges or security requirements such that policies can be aggregated. Credentials then include the group attributes associated with the principal, rather than the individual object attributes.

From the discussion so far, it is clear that the primary security mechanisms required are authentication and access control. Mechanisms for both are described in more detail below. In addition, a third security practice, known as "module-thinning" is introduced.

2.8.2.1 Authentication

Authentication is commonly achieved through cryptographic means [DVW92]. As the overheads of user and code authentication within active networks needs to be minimal[13], non-negotiated or one-way authentication mechanisms based on asymmetric encryption mechanisms (for example, RSA or DSA) are preferred [ANW01]. These methods normally require the consultation of some form of public key service or certifying agent to verify the cryptographic protection (i.e., digital signature or private key) [oST94].

Common cryptographic algorithms for authentication include public key signatures (e.g., RSA, DSA), (keyed) hashes (e.g., MD5 , SHA-1), and symmetric key encryption (e.g., DES, triple-DES).

[13]Note that active network systems have stringent QoS requirements regarding the latency and throughput that is introduced by intermediate nodes.

2.8.2.2 Access Control

Besides authentication, which is just a means to validate a principal's identity, a mechanism to control access to the node resources and system services based on credentials is required. Access control is typically split into two parts: (1) the control mechanism (policy enforcement) safeguards the interface by granting access only to authorised operations; and (2) the access control list (ACL or policy database) stores the access privileges for each principal or group of principals.

Although research into policing on active node architectures is progressing, a common agreement (for example, a universal language) for the specification of active network policies is still lacking. The main problem is down to the fact that the types of policies supported by a node depend to a large degree on the node implementation and the control mechanisms provided. Examples of policy and trust management systems are PolicyMaker [BFL96] and KeyNote [BFK98], both of which are exploited for resource and access control within the secure active network environment (SANE) [AAKS98] at the University of Pennsylvania. They offer a special-purpose language for expressing policies in terms of signatures of principals and delegation of trust. More recently, research at Imperial College London has proposed two new policy types for active networks, called *obligation* and *authorisation* policies [SL99], along with a generic and extensible specification language for policies, named Ponder [DDLS01].

2.8.2.3 Module-Thinning

Module-thinning is a special approach towards access control that has noteworthy advantages for active networks. Unlike conventional access control mechanisms that safeguard the interface, module-thinning controls system access by tailoring the interfaces exposed to a program during start-up based on the privileges of the program and/or the user instantiating the program. Thus, depending on the principal's authority, module-thinning dynamically links (at execution time) customised libraries to a program that exposes only interfaces for which the principal is authorised. This mechanism is most secure as unauthorised interfaces are completely removed and hence leaves no possibility for an attack.

The downside of module-thinning is that fine-grained access control (allowing/denying individual function calls on a per-principal basis) is not feasible. This would demand either a specialised dynamic linking technique, where link libraries are generated on-the-fly at execution time, or a large set of pre-compiled libraries, each exporting a fixed range of functionalities.

2.9 Summary

This chapter has introduced and put into context the field of active and programmable networks. It has set the basis for the advances and further research presented throughout this thesis.

In section 2.2, the background and chronological emergence of the research into active and programmable networks has been introduced. Section 2.3 then defined the scope of the field and divides it into conceptual different approaches to network programmability, namely *active networks*, *open signalling* and *intelligent networks*.

As the focus of this work lies in the design of a flexible and extensible active router architecture, the subsequent sections examined the issues and problems of active networks. While section 2.4 defined the methodology of this approach and describes the fundamental tenets of active network architectures, section 2.4.2 reflected on the impact of network-side processing with regard to the semantics of network communication and the end-to-end argument.

At the core of this chapter has been the division of active network research into two fundamental approaches: *active packets* and *active extensions*. It has been shown throughout this chapter and the following (chapter 3) that these different approaches have an important impact on the suitability for various application domains.

Finally, section 2.5 continues by introducing the de-facto standard for active node architectures as viewed by the DARPA funded active network community, comparing various programming models based on different code distribution and encoding schemes (section 2.6), introducing the concept behind service composition along with a comparison of several composition models (section 2.7), and discussing the issues of safety and security for this domain (section 2.8).

Chapter 3

Related Work

3.1 Overview

This chapter outlines important contributions already made in the area of active networks and shows the multitude of research areas that have been followed. Since the focus of the work presented in this thesis concerns the development of a novel active node architecture, this chapter focuses primarily on completed and ongoing work in active systems design and enabling technologies. A number of research projects that design and develop active network systems and services are studied. The chapter concludes with an overview of current work on active applications and services.

Since the beginning of research into active networks, many research groups have tried to develop active network architectures. The diversity of research groups has led to a large variety of different approaches. These approaches can be divided based on the following criteria:

- *Programming Interface:* The programming interface determines the programming capabilities of the active network. Programming interfaces range from very specific interfaces (for example, to control the forwarding behaviour) to general-purpose interfaces that enable full data path programmability.

- *Programming Model:* The programming model defines the type of code distribution mechanisms (in-band or out-of-band) and encoding schemes (source code, interpreted code, bytecode, or binary code). The encoding approach, in other words the programming language and/or compiler/interpreter, has an impact on the security architecture. For example, code interpretation relies only on a safe interpreter, while binary code execution demands system support to enforce security.

- *Service Composition Model:* The service composition model supported by the active network architecture defines how active services can be composed. For ex-

ample, composition granularity and flexibility depends on the ease of active program/component interaction and the composition control (solitary or co-operative). They typically range from fairly static (creation time) to highly dynamic (run time) mechanisms.

- *Security Model*: The security model defines how safety and security is accomplished by the active node. Depending on the programming and composition model, different security measures are best suited. For example, in-band active code execution relies on lightweight security checking and hence interpreter or virtual machine based solutions are preferred.

The following section examines various active network architectures and compares their different approaches according to these criteria.

3.2 Active Network Architectures

The programming and composition model of an active network architecture has a key impact on the types of network services that can be provided and the granularity by which those services can be introduced. The spectrum of potential programming models ranges from simple configuration scripts to sophisticated software extensions and from conservative to highly dynamic and flexible levels of programmability.

Two of the first active network projects, namely the ANTS project at MIT [WGT98] and the SwitchWare project at the University of Pennsylvania [AAH+98], have strongly influenced the directions within active network research. The former has reanimated the idea of active packets carrying active code along with packet data. This so-called *integrated* approach represents a highly dynamic means of code and service deployment (i.e., on a per-packet basis). However, carrying the active code in-band imposes restrictions on the code size and hence the complexity of the active programs. The latter, by contrast, has introduced the concept of active extensions to programmable routers. This approach, which is also referred to as *discrete* approach, enables the injection of active programs separately from the actual data packets. As active extensions are typically delivered out-of-band, before the data is sent, this approach is usually less dynamic (i.e., active programming is not performed on a per-packet basis). As a consequence, however, the discrete approach does not impose any restrictions on the size or complexity of the active extensions.

These to some extent diverse approaches have led early work on active networks into different directions. However, as hybrid solutions have evolved over the years, the divergence has become less apparent.

The following sections comprise a survey of relevant active network projects and architectures. They are grouped by their fundamental approach – integrated or discrete.

3.2.1 Integrated Active Network Solutions

Much of the early work in active networking was stimulated by a paper of Tennenhouse and Wetherall on the concept of active capsules [TW96]. The Telemedia group at MIT has first pursued the idea of placing program fragments into IP packets as part of the ActiveIP project [WT96]. Initially, they studied the potential of placing small programs within the option fields of IP packets. These so-called *active options*, encoded in Safe-Tcl [OLW98] in their prototype implementation, were "executed" (i.e., evaluated) by modified network nodes as the packets traversed the network. Despite the limitations of this simple approach, namely lack of safety, security and resource management, the work demonstrated the potential of integrated or active packet based solutions.

Further work on ActiveIP has led to the development of the popular Active Network Transfer System (ANTS), which provides a base platform for many active network solutions today.

The remainder of this section introduces several key active network systems based on the integrated approach starting with the pioneering ANTS system.

3.2.1.1 ANTS – Active Capsules

The main objective driving the efforts at MIT was the development of a common communication model (as opposed to a particular protocol) that enables the dynamic and uncoordinated deployment of new communication protocols, as they become needed. Their active network toolkit, known as ANTS [WGT98], provides a set of core services including support for transportation of mobile code, code loading on demand and code caching techniques. These core services allow network architectures to deploy new network protocols or extend existing ones.

The two key concepts of the ANTS architecture can be summarised as follows: (1) traditional data packets are replaced by *active capsules* that carry the processing instructions to be executed by the active routers along the transmission path; (2) an intelligent *code distribution* mechanism ensures that missing service routines are automatically loaded and instantiated on-the-fly by network nodes along the transmission path. The operation of this mechanism is demonstrated in the following scenario: A capsule, dependent on certain functionality that is not locally available yet (i.e., cached), causes the distribution module to fetch the missing function(s) from the previous node (along the transmission path).

This mechanism has the advantage that code is loaded only when and where needed.

Furthermore, retrieving the code directly from the previous hop ensures rapid loading.

The ANTS capsule execution model supports highly dynamic and fine-grain network programmability on a per-packet basis. The intermediate network nodes along the transmission path are responsible for capsule execution and forwarding, distribution of capsule code, and maintenance of processing state between subsequent capsule executions. The atomic units of code sent as part of the active capsules enable programming of their forwarding behaviour and (re-)configuration of the nodes along the transmission path.

Capsules are executed within a restricted run-time execution environment provided by the active nodes. The ANTS execution environment exposes a network API for use with capsule programs. This run-time environment, for example, supplies primitives for configuring routing behaviour, controlling capsule scheduling, establishing inter-capsule communication, and storing soft-state information on the routers. The execution environment limits access to shared resources in order to protect the active nodes from malicious or erroneous capsules. For example, the notion of capsule processing quanta has been introduced as a metric for the management and control of node-local processing resources.

A prototype implementation of ANTS is available in Java. Java has been chosen because of its support for code mobility (i.e., dynamic loading/linking support for bytecodes is available) and safety (i.e., the Java virtual machine supports software fault isolation). The functionality of ANTS has been demonstrated with the development and deployment of several protocols and extensions, including a high performance reliable multicast extension [LGT98] and a TCP SYN-flooding defence protocol [Van97]. Furthermore, ANTS has also been incorporated at different sites of the ABone[1] as one of several active network technologies.

3.2.1.2 PLAN

The design and development of the Programming Language for Active Networks (PLAN) [HKM+98] was stimulated by the need for an active packet programming language as part of the SwitchWare project at the University of Pennsylvania (see section 3.2.2.1). The PLAN programming language and execution environment have been specifically designed for lightweight and simple programming of active packets. The functional programming language, which is based on the simply-typed lambda calculus, has been carefully designed in order to keep the language lean and secure. As a result, PLAN programs tend to be *very small* (to fit easily inside active packets), *strongly-typed* (to improve safety), and *functionally-restrictive* (to simplify security provisioning).

[1] The ABone [BR99] is a DARPA-sponsored experimental active network built on top of IP.

First of all, PLAN is designed considering a two-level programming architecture, in which PLAN serves as the high-level scripting language for active packets that 'composes' services from low-level functionality residing on the nodes. It provides the 'glue' to create value-added and customised network services from low-level SwitchWare services[2]. The distinction between high-level (lightweight) and low-level (heavyweight) programmability has been drawn by SwitchWare to help the design of the lean and efficient packet language.

Second, designing PLAN as a strongly-typed programming language has the advantage that programs can be statically type-checked before packets are injected into the network. This eliminates many potential type errors at compile time and also minimises the need for costly safety checks at run-time (see section 2.8 for more details).

Third, PLAN is carefully designed to keep the language lean and secure. It restricts the provided functionality deliberately in order to minimise the need for costly security checks. A new language feature has been added only when crucial functionality was missing and in that case the usability of the language was enhanced without compromising security (i.e., all existing security guarantees were preserved). As a consequence, PLAN has not evolved into a "complete" programming language; common functional programming constructs, such as recursion and mutable user-defined state, are missing. To compensate for these limitations, the PLAN execution environment exposes an interface to the extensible SwitchWare service routines.

Furthermore, PLAN provides explicit support for resource management, which enables the run-time environment to control resource usage during execution of the active packets. To facilitate this, the execution environment implements a fixed counter approach whereby the amount of processing and memory resources required to process a packet is counted until either the active program terminates, or a counter exceeds the upper bound for a resource. This causes immediate termination of the processes by the system. A garbage collection mechanism takes care of freeing unnecessarily bound resources upon process termination.

PLANet

The PLAN programming environment has been used to build an active inter-network, known as PLANet [HMA+99]. Although it is implemented as an overlay network based on UDP/IP for simplicity reasons, all packets include a PLAN program in the payload (just as "real" active packets).

A number of experimental applications, such as reliable and unreliable datagram delivery mechanisms, as well as standard network protocols, such as RIP-style routing

[2]Note that these low-level SwitchWare services can be fully programmed in a general-purpose language.

and address resolution based on ARP, are already supported. Experimentation with these protocols has shown that PLAN is sufficiently expressive to replace the protocol headers in the packets by PLAN programs. This illustrates the potential of PLAN for the design of new network protocols.

3.2.1.3 SmartPackets

The SmartPackets project [S+98] at BBN Technologies developed an active network solution based on active packets that encompasses special support for network management and monitoring applications. Schwarz et al. [SJS+00] argue that active networks technology is well suited for network management, as intelligent processing inside the network (i.e., closer to the nodes being managed) improves communication efficiency and event discovery compared to current techniques. Most traditional systems still rely on passive "polling" techniques to identify network problems (rather than on active "pushing" techniques).

Two fundamental design decisions were made to limit the complexity and simplify security on SmartPackets nodes. First, routers maintain no persistent state across packets. This implies that programs sent in smart packets must be completely self-contained and hence limited in size (i.e., smaller than the MTU). Second, the run-time environment or the virtual machine that executes SmartPackets code must provide a safe environment with well-defined and secure interfaces to critical system services and resources.

To this end, the project has developed two programming languages: Sprocket and Spanner. Sprocket is a high-level language similar to C, but without safety critical constructs such as pointers. In order to facilitate network programmability, Sprocket includes special features for network management related computations as part of the language (for example, special types for data packets and MIB[3] access). Spanner, by comparison, is an assembler language (for CISC[4] architectures) that can be compiled (assembled) into a compact machine-independent binary encoding. Although Spanner is largely similar to conventional assembly languages, several changes have been made to ensure safety. For example, all variables are declarative and no direct memory access is possible; only variables and a stack can be used for storage. The two languages complement each other as follows: Sprocket programs are first compiled into Spanner code, before this is assembled into the compact binary format. The Spanner intermediate language format allows for manual optimisation of the code before the machine code is produced.

The processing of "smart" packets on intermediate nodes is triggered by the Router Alert option [Kat97]. The ANEP encapsulation protocol [A+97] (introduced in chapter

[3]Management Information Base

[4]Complex Instruction Set Computer

2) provides the basis for the de-multiplexing of the packets to the correct execution environments. Four types of "smart" packet have been defined: program packets, data packets, error packets, and message packets. Program packets carry the code to be executed on the active routers along the transmission path. Data packets are used to report the results of the program execution back to the originating network management program. Message packets carry informational messages rather than code. And finally, error packets are used to indicate transport errors or execution exceptions.

Security within SmartPackets is achieved through authorisation and integrity checks on the "smart" packets. An authenticator sent in every packet allows the packets to be authenticated and checked for their integrity before the code is executed.

3.2.1.4 Related and Subsequent Work

A high performance active network node, called PAN [NGK99], is currently underway at MIT. PAN is a kernel based implementation of the active capsule approach. It is designed to support multiple mobile code systems. Currently PAN supports two code systems, one for Java bytecode and one for Intel x86 object code[5]. PAN is able to achieve high performance despite executing mobile code on a per-capsule basis. The following design decisions account for the performance improvement in PAN: in-kernel capsule processing, minimal data copying, and code caching for mobile code. A capsule in PAN contains both the data it transports and a reference to a code object that contains the active program to be evaluated at each node. If a code object is unavailable, it is dynamically loaded as proposed by the ANTS framework [WGT98]. PAN also provides a node-local state store that can be accessed by capsules traversing the node. The state store manages capsule state as soft state since nodes may be restarted from time to time or the network topology may change. Memory management within PAN nodes is achieved based on the idea of *software segments*[6]. Software segments provide a means for platform-independent buffer management which is equivalent to that found in most modern operating systems. It allows multiple mobile code systems to coexist and share memory without the need of costly data copy operations. Performance measurements on PAN have shown overheads as little as 13% for the processing of capsules based on native object code (which lacks safety, security, and portability) compared to standard packet forwarding. These results are based on their prototype implementation for Linux and a capsule size of 1500 bytes.

Another initiative, the SNAP (Safe and Nimble Active Packets) [MHN01] approach

[5]Although, at the time of publication, the Java code system was only available as a user-space implementation, a kernel-space Java VM is apparently in progress.

[6]The reader should note that despite the terminology overlap with segmentation in virtual memory systems, memory segments in PAN do not provide protection.

developed at the University of Pennsylvania, extends various existing active network technologies (i.e., ANTS, PLAN) by an integrated resource control mechanism. Moore et al. argue that the time-to-live (TTL) packet proliferation limiter and fixed resource bounds are not sufficient to limit resource consumption. For example, protection against denial-of-service attacks (based on many small active packets) requires resource bounds that are linear to the packet size. Like PLAN, SNAP limits the expressibility of the programming languages in order to restrict packet execution. However, SNAP goes further by limiting expressibility such that programs cannot exceed resource bounds (for bandwidth, CPU, and memory) that are linear in proportion to the packet length. This permits active nodes to enforce a strict upper bound on the resources used by a packet. The designers claim that SNAP fully retains the flexibility and performance of existing systems despite the language restrictions.

Finally, PANTS, proposed by Fernando et al. [FKFH00], extends the ANTS framework by a mechanism that enables dynamic changes to the active node at run time. For example, PANTS nodes are capable of dynamically changing the run-time execution environment, and capsules are able to dynamically rewrite their code. The latter is particularly challenging within Java (the programming language used within the framework) as it is statically typed. Furthermore, PANTS capsules are self-organising. They dynamically arrange themselves into groups of similar interest without the need of a central authority. These features make PANTS a flexible and dynamic active node architecture.

3.2.2 Discrete Active Network Solutions

The active network solutions described so far are all based on the principle of in-band active code distribution and packet processing. However, the processing of data packets is conceptually independent from the task of injecting programs into a programmable node. An out-of-band mechanism for program distribution and a discrete programming model is advantageous when the loading must be carefully controlled.

The main objective of the discrete or "programmable switch" approach is to provide a level of functionality close to that of a Turing machine. While active packet based programming is inherently restricted to lightweight programming (only small programs can be included in every packet), this approach has the potential for more fundamental extensions. In the context of active networking the idea of out-of-band injection of active code first emerged within the SwitchWare project (see below).

The remainder of this section presents a number of discrete active network solutions.

3.2.2.1 SwitchWare

The SwitchWare active network architecture [AAH+98] developed at the University of Pennsylvania provides an all-embracing active network solution based on three distinct layers: a *secure active router*, *active extensions*, and *active packets*. The layered architecture allows for different security mechanisms and programming models to be employed on each level while still meeting the challenge of flexibility and performance.

The secure active router layer – the bottom layer of the SwitchWare architecture – provides the foundation for the upper layers. It provides operating system support for the programmable layers above, whereby maximum security and performance are the primary objectives.

The active extensions form the middle layer of the architecture. This intermediate layer of programmability enables extensibility of a router's core functionality. The node-local extensions, also referred to as *switchlets*, are one of the key contributions of SwitchWare. Like active packets, switchlets can be dynamically loaded and executed on a switch. However, since they are not mobile[7], these extensions are not required to be lightweight and portable. Therefore, switchlets can be written in general-purpose programming languages. Nevertheless, security and safety still play an important role on this layer, and as a result, a variety of mechanisms, including sandboxing, authentication and program verification techniques, are deployed to ensure safety and security.

The active extension layer in SwitchWare provides an execution environment for switchlets. The prototype implementation of this layer, also referred to as the ActiveBridge [ASNS97], is based on a single-language environment. The strongly typed ML dialect, Caml, is used for the implementation of the ActiveBridge layer and the actual active extensions. Caml achieves module isolation by means of name space security (as opposed to address space security). Also, Caml bytecode is dynamically loadable and machine independent, which permits dynamic deployment of active extensions on different SwitchWare router platforms. These are many of the features also found in Java; however, Caml maximises performance by means of static type checking, which enforces type safety during compilation and linking, rather than at run-time.

The active packet layer, which constitutes the top layer of the SwitchWare architecture, enables network programmability based on small mobile programs delivered as part of the data packets. As previously discussed in section 3.2.1.2, active packet programming is based on the PLAN programming language. PLAN has been specifically designed to meet the requirements of active packet programming in environments where low-level functionality can be extended by means of active extensions. For example,

[7] Unlike active packets, those extensions do not travel through the network (i.e., they are only executed on a specific node).

limitations resulting from language restrictions in PLAN can be overcome by providing the functionality lacking through active extensions.

SANE

The Secure Active Network Environment (SANE) [AAKS98] is another outcome of the SwitchWare research project. Despite the fact that a prototype implementation of SANE was developed in the context of SwitchWare, SANE addresses the problem of security for active network environments in general.

SANE comprises a secure bootstrapping mechanism providing static integrity guarantees for active nodes (i.e., firmware and operating system components are checked). Once the system is safe and operational, SANE provides dynamic integrity checking mechanisms for the module loader and execution environment. Moreover, SANE includes a secure key and certificate exchange mechanism that enables code authentication. The cost of safety and security provisioning based on SANE has been examined by Alexander et al. [AAA+99].

3.2.2.2 NetScript

The NetScript project [YdS96], which started in 1996 at Columbia University, was another pioneering project in the area of active networks. The project investigated three distinct areas: a programming model for active networks, a programming language for network programming, and a programmable node architecture.

NetScript proposes a distributed programming model whereby the script programs determine the processing of packet streams on individual network nodes. The script programs provide the "glue" for customising node-resident functionality and services. These NetScript programs are also referred to as *agents*, which accounts for the fact that the programs travel around the network in order to program the nodes.

The network architecture developed for NetScript comprises Virtual Network Engines (VNEs) and Virtual Links (VLs). VNEs are interconnected by VLs to create a NetScript Virtual Network (NVN). An NVN may correspond only loosely to the underlying physical network; a node might be responsible for executing several VNEs, and a physical link may relate to a collection of VLs and vice versa. The architecture provides great flexibility regarding interoperability with existing network architectures and protocols. An NVN can be either overlaid on top of other networks (for example, a VL may be implemented on top of IP), or NetScript can be deployed to implement an existing protocol stack (for example, a NetScript agent may implement IP routers in VNEs).

The script language NetScript [dS98] – a small and simple, object-oriented dataflow language designed specifically for the programming of stream-based computation – is

used to program the agents. It is particularly suitable for programming network tasks such as packet routing, traffic analysis, control signalling, and network management. The language has three important features. First, it provides a *universal* abstraction of a programmable node. The VNE hides the heterogeneity of the nodes by providing a cross-platform virtual machine abstraction. Second, NetScript supports *dynamic* programmability by allowing new code to be loaded and executed on-the-fly without disrupting processing on the VNE. Finally, NetScript is a *dataflow-driven* language, which means that computation is triggered through the arrival of packets rather than normal program control structures.

The NetScript node architecture is made up of several modules, namely the agent service layer, the connectivity service module and the resource manager. The agent service layer provides a multi-threaded execution environment for agents based on the SMARTS Operations Server (SOS) implementation [ART94]. The connectivity service module allows agents and VNEs to interact with the underlying physical environment in order to allocate and maintain VLs. Finally, the resource manager enables agents to control the allocation of VL resources, and the scheduling and transmission of packets.

The communication model chosen for NetScript allows a single stream to be processed by multiple agents. When a packet arrives at a VNE, the NetScript specific encapsulation header, which contains information on how to process a packet (for example, the sequential order of agents that need to be invoked), is used to dispatch the packet to the corresponding agents. As such, NetScript can be seen as a hybrid solution that is based on a discrete programming model regarding the distribution of active code and the type of active services provided (i.e., NetScript agents offer persistent functionality) while relying on the integrated model for the in-band processing of the encapsulation header.

More recent work on NetScript has resulted in a service composition framework [dSFY98] that enables interoperability and extensibility of existing protocols and end-to-end service composition.

3.2.2.3 Bowman & CANEs

Researchers at Georgia Tech have been developing an active network architecture comprising the Bowman NodeOS and the CANEs execution environment [MBC+99]. The Bowman NodeOS layers active network-specific operating system functionality, namely abstractions for communication, processing and storage, and an extension mechanism to enrich the functionality, on top of a standard host operating system. The CANEs execution environment provides a composition framework for active services based on the selection of a generic underlying program, which defines the basic type of service pro-

vided (for example, forwarding), and the insertion of customised code into well-defined slots of the program.

Bowman offers three basic abstractions to support active programmability through the CANEs execution environment:

Channels represent communication end-points that support sending and receiving packets via an extensible set of protocols. For this abstraction Bowman exposes functions to create, destroy, query and communicate over channels. A special type of channel, called a *cut-through channel*, is provided for packets that do not require active processing on the node. This enables "fast-path" processing for non-active traffic.

A-flows are the primary abstraction for computation; they encapsulate processing contexts and user state. Each a-flow consists of one or multiple threads and executes on behalf of an identified principal. A set of functions to create, destroy and run a-flows is provided. Furthermore, the a-flow concept is extended to provide a generic timer mechanism. A dedicated timer thread is deployed to execute user-defined callback functions upon timer expiration.

The *state-store* provides a mechanism for a-flows to store and retrieve state. It provides functions to create, store, retrieve, and remove data. The state-store also provides a mechanism for data sharing between a-flows without sharing program variables.

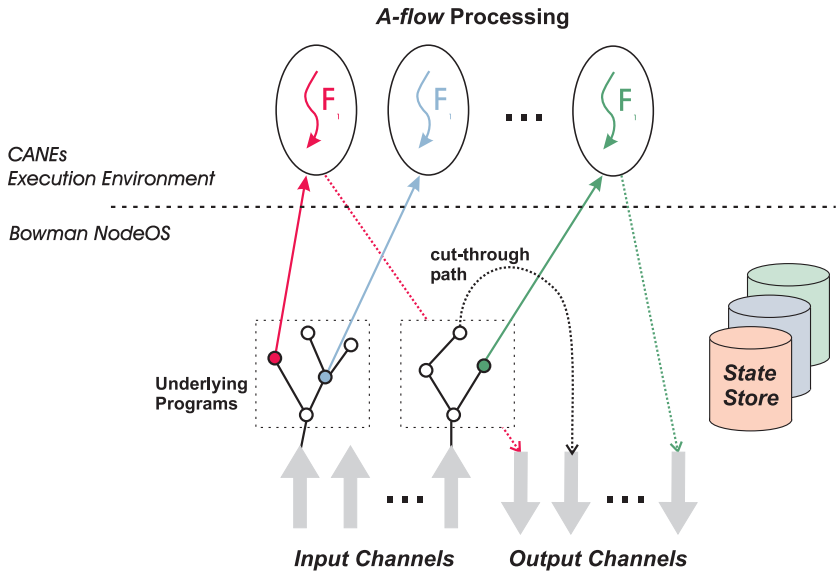


Figure 3.1: The Bowman NodeOS – Illustration of the Bowman node abstractions: input/output *channels*, *a-flows* and *state-store*, and the service composition approach.

The way in which these abstractions are used is illustrated in Figure 3.1. It shows a schematic of the packet processing path. First, incoming packets undergo an initial classification step that identifies a set of channels on which a received packet should be processed. Second, once channel specific processing completes, the packet undergoes a

second classification step that identifies the a-flow or cut-through path to be processed. A-flow processing is determined by the active code specified at creation time. A-flow code can be dynamically introduced into a Bowman node by means of a dynamic code loading mechanism.

Since Bowman provides only the low-level node OS functionality of an active network node, a high-level execution environment providing a programmable network interface to the users (for the a-flows) had to be incorporated to attain a complete systems solution. As a result, the Georgia Tech group developed the CANEs execution environment.

The main thrust behind CANEs is the *slot processing model*. It comprises two parts: an *underlying program* (fixed part) that represents the uniform processing applied to every packet, and the *injected programs* (variable part) that represent user specific computation on packets. The underlying program defines specific points (or *slots*) where user programs can be injected. Service composition within CANEs is therefore a two step process. First, an underlying program providing a basic service (for example, packet filtering or forwarding) is selected from amongst those currently installed on an active node. In the second step, a set of injected programs is selected to customise the underlying program. These injected programs are either already available at the active node or can be downloaded from a remote site.

Performance measurements with the Bowman NodeOS have shown that its performance is slightly below standard gateway applications. The overheads resulting from the frequent system calls and context switches, and the massive data copies required for normal user-space processing of active packets precludes Bowman from sustaining high throughput. However, it has been demonstrated that Bowman is able to saturate a 100 Mbps Ethernet for packet sizes close to the maximum Ethernet frame size [MBZC00]. Nevertheless, it is clear that the processing load as a result of the frequent context switching between user-space execution environments and the active NodeOS in kernel-space, and particularly the massive copy operations involved in shuffling the packet data up into user-space (and back to kernel space) is excessive.

3.2.2.4 Joust

Researchers at Arizona University have designed and developed Joust [HBB+99] – a Java-based platform for *liquid software*. The term "liquid software" [HMPP96] refers to mobile, communication-oriented software or code that is able to "flow" through a network. Joust is therefore primarily designed as a platform for communication-oriented systems, such as active networks.

Liquid software encompasses an entire infrastructure for dynamically moving functionality inside a network. It can be considered as a dynamically configurable remote

procedure call (RPC) system, which enables clients to define the interface and semantics of the RPCs by downloading the appropriate code onto a server prior to the calls. This allows programming of network nodes through customised interfaces, which are specifically tailored to the task at hand. A key functionality of liquid software is therefore the ability to support mobile code throughout the network, between end-hosts and network nodes. Active networks, which allow users to customise dynamically the network by injecting code, are considered a key application of liquid software.

The Joust programmable node is built from three major components: the Scout OS, which will be further described in section 3.3.1.1, a Java run-time system, and a just-in-time (JIT) compiler.

The Joust Java run-time system [JOU] implements a custom Java Virtual Machine (JVM) as a Scout module. A virtual machine based execution environment for mobile code has the advantage that platform heterogeneity throughout the network is hidden by a common programming interface. In order to overcome the resource access and control limitations of the JVM, Joust adds special support for lower-level abstractions for real-time scheduling to the JVM. Furthermore, since high performance of mobile code execution is a key issue for liquid software (and active networks in general), Joust has optimised the JVM at several levels: First, the core JVM has been optimised so that potentially slow functions are made more efficient. Native implementations in C of performance critical functions have been incorporated. Second, the Java API has been extended to support Scout specific functionality, such as operations on Scout paths. Evaluation experiments with a prototype Joust implementation have shown that the Joust JVM performs in the order of 2.3 times faster than the Sun JDK 1.1.3 [HBB+99].

The Joust JIT compiler is required to achieve high performance within liquid software. The JIT dynamically translates Java bytecode to the native instruction set of the host processor at loading time of a Java class. This enables the dynamic integration of mobile code that implements new or improved services on a Joust node without losing the performance benefits of statically compiled code.

A demonstration of the Joust system is based on a port of MIT's ANTS framework (see section 3.2.1.1). The ANTS execution environment has been translated to C code and compiled to a native Scout module. The Joust JIT compiler is required to compile the protocols carried by the active capsules and to dynamically link them to the ANTS module. A performance comparison with the original ANTS systems has indicated gains up to 3-5 times faster than the original JDK based implementations.

In conclusion, Joust has revealed valuable results for the field of active networks. It provides not only a flexible and programmable platform, but also gives evidence that communication-oriented operating system support is an essential component for efficient active networks.

3.2.2.5 LARA

The Lancaster Active Router Architecture (LARA) [CFSS99] developed from 1998 onwards proposes a low-cost, scalable high-performance active router platform. The architecture encompasses both hardware and software design. The initial architecture consists of four parts, namely the Cerberus hardware architecture, the LARA Platform Abstraction Layer (LARA/PAL), the LARA MANagement component (LARA/MAN) and the LARA Run-Time execution environment (LARA/RT).

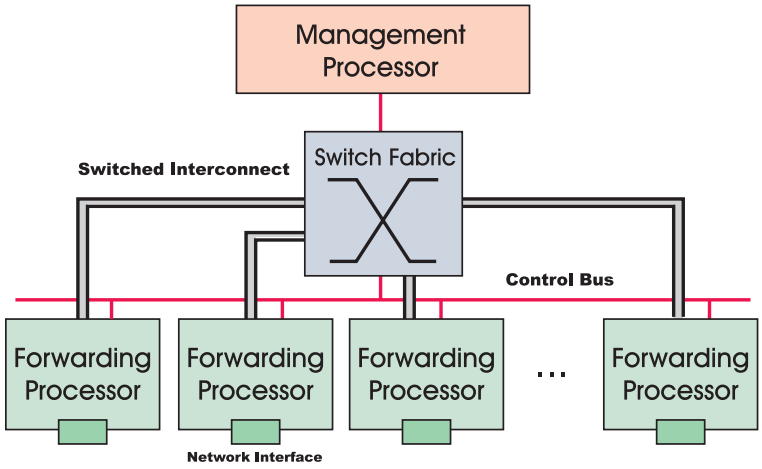


Figure 3.2: The LARA Hardware Architecture

The Cerberus hardware architecture forms the basis of a LARA node. It suggests a platform that can be built from off-the-shelf *low-cost* hardware components. Cerberus achieves *high performance* due to the following design decisions: (1) a cluster of high-performance processor units is used for the active computation and (2) active processing is carried out in kernel-space which enables fast processing as expensive copy operations and context switches are avoided. *Scalability* is achieved by use of dedicated processor units for each network interface. Extending the number of router interfaces therefore increases the processing power proportionally. A scalable high bandwidth backplane (for example, switched ATM) interconnects the interface processing engines and ensures high-speed communication between them (see Figure 3.2). A single management processor is responsible for just-in-time active code compilation, configuration, authentication and policy management. The prototype Cerberus implementation uses a dedicated Intel Pentium II 266 MHz processor equipped with 128 MB RAM for each forwarding engine. The backplane interconnecting the forwarding engine consists of a SCSI bus, which enables transfers of up to 500 Mbps.

The LARA Platform Abstraction Layer provides a platform independent layer that enables the deployment of LARA execution environments on various hardware platforms. It exports a set of programming primitives (i.e., API to control and manage scheduling,

memory, network bandwidth, and policy enforcement) for use by the execution environments. This abstraction layer with its well-defined programming interface facilitates the development and deployment of divergent execution environments (i.e., SwitchWare, ANTS etc.) on LARA active nodes. LARA/PAL has been included in order to account for the assumption that no single execution environment and/or active programming language will fully satisfy all types of active applications and services.

The LARA Management component is responsible for ensuring security on LARA nodes. Based on a policy infrastructure and an authentication mechanism, LARA/MAN ensures that only authorised active code (originated from a trusted source and installed by an authorised user) is loaded and executed by the active processing engines.

Finally, the LARA Run-Time environment is the implementation of a particular execution environment based on an extension of the Linux 2.2.x kernel. Active code for the LARA/RT is provided in the form of loadable kernel modules. Code loading and unloading is based on the standard kernel module loading mechanism provided by Linux. Active code for the LARA/RT is written in C[8] and distributed either in the form of a binary module or source code[9]. In order to ensure fair scheduling of the processing resources on a LARA node, LARA/RT includes a dedicated pre-emptive scheduler for the active module threads. This also prevents faulty active modules from locking the system. The decision to place the active processing in kernel-space simplifies the data handling on the node and thus ensures high network throughput. The downside, however, is that the LARA/RT, which adds merely kernel threading and packet capture support to the Linux kernel, is difficult to program.

Although only a subset of the LARA architecture has been implemented, it has provided a useful input into prototyping the LANode active router (see section 3.2.2.8). The completion of the LARA implementation has been superseded by the developments presented in this thesis. The redesign of the LARA software architecture as proposed herein has led to a new active router architecture, called LARA++. Despite the comprehensive redesign of the LARA software architecture, LARA++ considers the scalable and high-performance Cerberus architecture as one possible hardware platform.

3.2.2.6 Click

MIT's Click [MKJK99] is a software architecture for building flexible and configurable routers. A Click router is "configured" from packet processing modules called *elements*. Individual elements support simple router functions such as packet routing, queueing, or

[8]Note that C is typically the only programming language (apart from assembler) that is used for Linux kernel programming.

[9]LARA includes a JIT compiler for just-in-time compilation (at module loading time) of active code that is distributed in the form of source code.

scheduling. A complete router *configuration* is defined by a directed graph whose nodes are the elements. The arcs between the elements are called *connections*. They represent a possible path for packets traversing the router. The endpoints of the connections between elements are called *ports*. An element can have any number of input or output ports.

A Click router configuration is determined at compile time. The elements are internally represented by C++ objects that are inter-linked with each other through object references. Packet passing between functional elements is thus simply a matter of passing memory pointers between objects.

Each element has an element class associated with it (like objects in object-oriented programs), which determines the fundamental behaviour of the element.

Click provides two types of connection between elements: *push* and *pull*. While a push connection simply passes packets on to the next downstream element, a pull connection request the upstream element to return a packet. When an element receives a packet from a push connection, it must store, discard, or forward the packet to a connected element for further processing. Most elements forward packets by calling the push function of the next downstream element. Since packet forwarding is just a virtual function call, the CPU scheduler cannot suspend the processing at arbitrary points; elements must cooperatively choose to stop the processing. In order to allow the system to schedule different tasks, Click supports special elements, called *queues*. Those queues provide the transitory storage for packets received by a module.

Click configurations are written in a simple language by specifying the elements and the connections among these elements. The language supports constructs that allow users to define new element classes composed from existing ones. This enables users to build their own compound element classes.

The current Click implementation has limitations resulting from the default system scheduler (i.e., the standard Linux scheduler), which handles the scheduling of competing push and pull paths despite the lack of application specific semantics. Furthermore, a notion for element types is missing. In addition to that, Click currently also misses a static type system, which prevents configurations connecting, for example, a link-layer element (e.g., an Ethernet interface element) directly with a transport-layer element (i.e., TCP or UDP) rather than a network-layer element.

3.2.2.7 Router Plugins

Router Plugins [DDPP98], developed by a collaboration between ETH Zurich and Washington University, suggests a modular and extensible software architecture for next generation routers. The primary goal of this project is to build a flexible network subsystem

that offers the ability to select implementations (or even instances of the same implementation) of router components, called *plugins*, on a "per-flow" basis. The concept of a *flow* therefore plays a fundamental role, as it defines the unit of granularity by which network services can be assembled.

Plugins are binary code modules that implement a specific extended service, such as a special packet scheduler, a routing component, etc. They can be dynamically loaded and unloaded into the router kernel at run-time. NetBSD, which is used as the base platform for Router Plugins, provides appropriate kernel support to load modules into the kernel. The Plugin Control Unit (PCU) provides the "glue" to bind individual plugins to the network subsystem and offers a control interface to manage plugins.

For maximum flexibility, Router Plugins support the creation of multiple instances of the same plugin, whereby an instance is a specific run-time configuration of the plugin. This allows efficient configuration of services on a per-flow basis without the overhead of loading multiple plugins for different configurations. Since the notion of instances needs to be supported by the plugins themselves, developers must expose a standardised interface to control instance creation and configuration.

The notion of a *gate* is used to define a "socket" in the core network subsystem where the plugins can be "plugged in". These gates define the points in the packet processing chain where the flow of execution passes to an instance of a plugin. Thus, gates are placed wherever interaction with plugins is desired. The architecture however does not support dynamic creation of gates; they are assigned at compile time of the kernel.

The framework supplies an efficient mechanism to map individual data packets to flows and to bind individual flows to plugin instances. Flows are specified based on packet *filters*, which are defined as six-tuples including the source and destination address, protocol, source and destination port and the incoming interface.

The Association Identification Unit (AIU) is responsible for maintaining the bindings between flows and plugin instances. Upon receipt of a packet at a gate point, the AIU checks based on the installed filters whether or not any plugins wish to process the packet. Since packet classification is performed on a per gate basis, an inefficient implementation will seriously impact performance. Hence, Router Plugins exploit two optimisation techniques to speed-up packet classification. First, a flow lookup cache that allows very fast lookups of packets belonging to a "cached" flow is introduced. And second, a fast filter lookup algorithm based on Directed Acyclic Graphs (DAGs) is used which has a worst case performance in the order of microseconds[10].

[10]This result is based on the fact that the DAG requires a maximum of 24 memory accesses in the worst case, and where a single memory access takes in the order of 60 ns. This leads to an upper bound of approximately 1.4 us. Note that the processing cost required for the computation of the hash is negligible compared to the memory accesses.

Research into Router Plugins has yielded several important results. It provides not only a flexible and programmable router platform, but also gives evidence that filter-based, multi-stage packet classification provides a flexible means for assigning active computation to packet flows. It also demonstrates that despite multiple classification stages, the overall classification performance can be good if carefully designed.

3.2.2.8 Related and Subsequent Work

The Protocol Boosters [FMS+98], developed at Bellcore, aim to provide methods that support dynamic protocol customisation in heterogeneous environments and rapid protocol evolution. To accomplish both of these goals, modifications have been designed which can be transparently incorporated within existing protocols to improve (or "boost") their performance. These protocol boosters are designed as supporting agents that add, modify or delete protocols, but never terminate (or convert) them. They consist of one or multiple elements. While single-element boosters interact with the "boosted" protocol only once, multi-element boosters interact several times. Multi-element boosters have the advantage that they can introduce additional message types/headers for the transmission among the boosters. However, since message types/headers added by a booster are unknown at the end nodes, the last booster involved in the transmission path has to strip off this information. For example, a two-element protocol booster could introduce forward error correction (FEC) codes to the IP protocol. The error correction codes would be transparently added and removed by the booster without changing the IP protocol at the end-systems. The protocol booster model suggests that protocols are initially designed for the default case (for example, a homogeneous network) and later be fine-tuned or tailored for specific environments (i.e., heterogeneous network) based on booster modules.

The Active Network Node (ANN) project [DPP99] at the University of Washington developed a scalable, high performance active network node. Like LARA (see section 3.2.2.5), ANN adopted a combination of hardware and software techniques. The development of the router hardware follows three design principles. First, each network port should be able to use several processing elements. Second, a close coupling between those processing elements and the network is required. And finally, the processing power of the node must be scalable. Therefore, ANN also suggests a system composed of multiple active network processing engines (ANPEs) that are interconnected using a switched backplane. Scalability is promoted by using a load-balancing algorithm that distributes the processing load amongst the available ANPEs. High performance is supported through a cut-through forwarding mechanism for non-active packets and kernel-space active packet processing. The current ANN implementation supports up to

eight ports with data rates up to 2.4 Gbps. It consists of an ATM Port Interconnect Controller (APIC), a Pentium class CPU, up to 4 GB of RAM and a field programmable gate array (FPGA). In addition, ANN has been complemented by an active code caching mechanism, called DAN [DP98]. The distributed code caching system has been included to minimise the download latency for active code.

The application level active network (ALAN) system [FG99], developed at Sydney University, introduces value added network services by means of a virtual or overlay active network infrastructure. The system is assembled from standard IP applications and servers connected to the Internet. Active processing "inside" the network takes place in so-called dynamic proxy servers (DPS). The active programs, called *proxylets*, act as communication proxies for data streams dispatched through the dynamic proxy servers. Unlike typical active networking schemes, which intercept packets directly on the forwarding path (network layer), ALAN requires the data streams to be explicitly addressed to the proxy servers. This approach suits Web based scenarios well as the use of HTTP proxies is a common practice in this domain and is supported by most Web browsers. For other types of applications, however, this approach is not very practical since active services cannot be applied in a transparent manner to the end systems (i.e., applications or end-systems must address packets explicitly to a DPS). The current implementation, known as FunnelWeb [Gho00], is based on Java 2. The DPS forms the active node execution environment for proxylets. It exposes a control and monitor interface for proxylets over Java RMI[11]. This out-of-band control mechanism enables third parties to control the downloading and execution of proxylets, and to observe the state of active proxylets. Experiences with FunnelWeb have shown that the choice of a user-space Java execution environment is inefficient and ill suited for applications with heavy data-plane processing requirements (for example, transcoding).

The LANode [SBSH01] active network node developed at Lancaster University strives to be one of the first router implementations for the emerging P.1520.3 programmable IP element architecture [B$^+$98]. The focus of the design is to experiment with the development of the programmable API abstractions ('L-' interface) and service control features ('L+' interface) that are defined by the architecture. The LANode architecture divides the router into two planes: the active control plane (ACP) and the active data plane (ADP). All control functionality that is off the data path and not directly concerned with packet forwarding resides in the ACP (for example, code loading and service composition). This separation has a direct impact on the programmability of the node. On the one hand, data plane functionality in LANode is implemented in form of Linux kernel modules, since the LARA active router kernel (see section 3.2.2.5) has been adopted as the programming environment for the ADP. Such ADP modules expose an 'L-' interface

[11] Remote method invocation (RMI) is a Java-specific RPC mechanism.

to the ACP for control purposes. On the other hand, the ACP programming environment is based upon a user-space Java virtual machine, similar to the one proposed by the ALAN (or FunnelWeb) system described above. The active control interfaces used for programming or configuring the forwarding behaviour are thus implemented in the form of Java proxylets. These proxylets expose a 'L+' interface through a CORBA/IIOP middleware infrastructure to facilitate control and management of the node as well as associated active applications through independent third-parties.

3.2.3 Comparison of the Integrated and Discrete Approach

Integrated active network solutions typically suffer from loss of performance due to the high costs involved in providing safety and security on a per-packet basis. In an effort to reduce this burden, custom programming languages that restrict the functionality of active programs to simple tasks (for example, configuration or control) were developed. This approach, however, counteracts to some degree the principal objective of active networks, namely to support flexible network programmability.

The discrete approach to active networks, by contrast, circumvents this performance problem because heavy-weight security operations usually occur only during program set-up. It has been shown that dynamic (or on-demand) code distribution mechanisms can be beneficial for discrete solutions in order to preserve maximum flexibility. Unfortunately, on-demand loading of active extensions inevitably introduces extra latency during loading of the programs up-front. In view of that, it has been demonstrated that code caching schemes can be very valuable, as they greatly reduce loading delays. The fact that programming takes place out-of-band prohibits fine-grain programmability (i.e., on a per-packet basis), albeit there is no evidence that programmability of such granularity is required. Out-of-band code distribution approaches do not limit active programming in terms of code size or loading time.

In order to benefit from the features of both approaches, namely flexibility, usability, security and performance, a combination of both ideas seems to be most attractive. A common way of integration is based on a layered architecture. The discrete approach supporting flexible programmability at the bottom layer can satisfactorily provide an integrated solution offering fine-grain programmability at the top layer. For example, the SwitchWare architecture (see section 3.2.2.1) layers the PLAN execution environment (integrated approach) on top of the ActiveBridge (discrete approach).

Another way of integrating the best of both approaches is by providing a modular or component-based programmable environment that supports fast module loading and at the same time provides flexible extension and integration mechanisms for modules. This approach is at least to some extent pursued within Bowman and CANEs (see section

3.2.2.3), Router Plugins (see section 3.2.2.7) and Click (see section 3.2.2.6), and is also one of the primary goals in the work presented here.

3.3 Enabling Technologies

Active networks encompass a large variety of mechanisms and technologies ranging from operating systems techniques such as memory protection, resource control and system interfaces – over communication systems comprising network protocols and routing mechanisms – to software engineering techniques for mobile code. Furthermore, safety and security mechanisms must be considered as a fundamental aspect in those areas due to the vulnerable nature of network systems and mobile code execution.

The challenge of active network research is to bring these diverse technologies together in order to construct systems solutions that enable flexible network programmability without compromising any of the fundamental features of network systems (i.e., performance, security, reliability). An analysis of the development of active networks has shown that advances of these technologies led to a continuous evolution in the development of active network systems. For example, the progress of mobile code technologies has notably influenced and advanced the developments in network programming.

The remainder of this section introduces a number of operating systems and support technologies as well as safety and security mechanisms that have proven to be useful for building active network solutions.

3.3.1 Operating System Support

Although operating systems do not seem to have evolved much in the last decade – they still perform the same basic tasks, namely hardware abstraction, resource partitioning and protection, and access control functionality – some of the recent advances have proved to be important for active networks. These advances concern mainly system design rather than its functionality (for example, the modular approach of micro-kernels and the adoption of software engineering techniques).

In recent years, a number of specialised operating systems that explicitly targeted programmable and network devices have been developed in research laboratories. Several of these systems, which have particularly been used within active network research, are described in the following sections.

Finally, another recent advancement in operating system design is the appearance of dynamic extensibility support, which will be further explored in section 3.3.1.6.

3.3.1.1 Scout

Scout [MMO+94] is a communication-oriented, configurable operating system developed at the University of Arizona. Scout OS[12] is composed of low-level communication primitives, customised for the purpose of a particular kernel.

Configurability of Scout is done in units of *modules*. Each Scout module provides a well-defined and self-contained functionality. Typical examples of modules are networking protocols, such as IP, UDP, or TCP. A complete kernel configuration is formed by connecting individual modules into a *module graph*. Such a configuration is defined at build time, and a number of configuration tools assemble the selected modules into a Scout kernel. As a result, Scout enables flexible configuration of customised kernels for network-attached devices and routers (for example, active routers). For example, the Joust system introduced earlier is simply a Scout configuration that includes a module, which implements the Joust execution environment, along with underlying services, such as IP, TCP, etc.

In addition, Scout provides a communication-oriented abstraction (*path*) to provide a more fine-grained means to configure the behaviour for individual flows at run-time. A path can be thought of as a logical channel through the module graph over which I/O data flows. It is defined by a sequence of modules that are applied to the data as it passes through the system.

3.3.1.2 Pronto

The Pronto platform [Hj00], developed at AT&T Laboratories, claims to provide a commercial strength platform to support active network research in general. It is hoped that the platform provides a universal solution, which can be used by many institutions as the base building block for their active network research.

A user-level library provides a programming interface to access generic kernel-level active node services and to interface with the forwarding path. The architecture does not impose a specific programming model or execution environment; instead it is execution environment independent and facilitates adoption of new execution environments.

A unique feature of the Pronto platform is that it supports four different models of interaction with the data plane:

- *Programmable control-plane mode:* Active applications are only given access to control-plane functionality
- *Data-plane peeking mode:* Active applications are able to snoop packets and queues in the data-plane but cannot alter those packets.

[12] Currently available as a native (stand-alone) system for Intel Pentium and Digital Alpha processors.

- *Local multicast mode:* Packets are both processed by the normal forwarding engine and relevant active applications running in user-level execution environments. The packet duplicates that are "multicasted" to the active programs for processing may be altered.

- *Programmable data-plane mode:* Active applications execute directly upon every packet in the data path. This mode is computationally the most expensive one due to the high cost of executing user-level code on every packet.

Due to the interaction limitations with the data plane in the first three approaches, it can be argued that only the last provides sufficient flexibility for generic active network programmability. Unfortunately this model is the least efficient one since every packet must be first copied to user-space before the active processing can take place and then back to kernel-space upon completion.

The Pronto implementation is built around the Linux operating system. It is mainly based on the Linux module model and thus requires only a number of small changes to the Linux kernel. Since Pronto is otherwise based on "standard" Linux, an off-the-shelf Java VM can be adopted as the execution environment. Furthermore, this enables the platform to take advantage of the well-maintained process, memory, thread and file-system support, and the wealth of freely available compilers, debuggers and software libraries.

3.3.1.3 OSKit and Janos

The OSKit [OSK] developed as part of the Flux group at the University of Utah is a framework and set of modularised code libraries which aims to facilitate the construction of new operating systems. For example, Janos [THL01] – a Java-based active network NodeOS – is currently under development based on this framework.

Active network and mobile code research at Utah [BH99] has shown that systems using user-level software mechanisms for protection such as the Java VM are not sufficient to offer safety for concurrent program execution (for example, several mobile code modules running in an active router). Instead, a clear separation between kernel and user space privilege levels is required. For example, systems that use type-safe languages for extensibility [BCE+94] and language run-time systems [HCC+98, TL98] to enforce protection usually lack the implementation of process termination, enforcement of resource control, and safe inter-process communication.

As a result of these limitations, Back et al. started the development of Janos, a specialised Java OS with support for resource control, safe inter-process communication and process termination. Janos takes advantage of the OSKit component library and Kaffe [BTS+98] – a multi-process Java virtual machine. Kaffe introduces a process

abstraction and a clean separation line between kernel and user space concerns in Java. This separation made feasible the implementation of the added functionality within Janos.

3.3.1.4 Genesis Kernel

The Genesis Kernel [CDK+99] developed at Columbia University supports spawning of virtual networks on-the-fly. Spawned networks inherit architectural components from their parent networks, but enable refinement of their operation. For example, a Cellular IP [CGK+00] network (child) can be spawned from a standard IP network (parent) in order to overcome the high handoff latencies present within Mobile IP [Per96]. The proposed architecture is generic in that architectures can be built, for example, to spawn virtual active networks.

Genesis supports spawning of virtual networks on three levels. At the lowest level, a *transport environment* delivers packets through a set of open programmable virtual nodes, called routelets. Routelets represent virtual layer-3 routers of distinct virtual networks. The Genesis Kernel, providing the execution environment for the virtual routers, exposes a programmable interface to the control algorithms (for example, routing) of each routelet. The intermediate level enables control of routelets through a separate *programming environment* for each virtual network kernel. The top level provides an open programmable interface, called the *binding interface base*. It offers access to a set of routelets and virtual links that constitute a virtual network.

3.3.1.5 SPIN

At the University of Washington, Bershad et al. have developed SPIN [BCE+94], an extensible general-purpose operating system. SPIN allows applications to safely add system extensions to the kernel and adapt the interface to the operating system accordingly. These extensions allow an application to specialise the underlying operating system in order to achieve a particular level of performance and functionality.

SPIN extensions are written in a type-safe language, and are dynamically linked into the operating system kernel. This approach enables dynamic extensibility of the operating system functionality without compromising the security of the core system code. SPIN and its extensions are written in Modula-3 and run on DEC Alpha workstations.

3.3.1.6 Dynamic Kernel Extensibility

Many commodity operating systems such as Linux, NetBSD, or FreeBSD have recently[13] been enhanced with support for dynamically loadable kernel modules that allow the kernel to be augmented with additional functionality at run-time. The idea is to modularise the kernel so that only the required functionality at any given point in time must be loaded into memory. For example, a plug-able device such as a particular PCMCIA network card that might be used rarely does not need a driver for the device to be permanently compiled into the kernel image. Instead, the respective device driver can be dynamic loaded at run-time when needed (in form of a kernel module) and unloaded when unused.

This technology has proven to be very useful in the context of active network implementation. The module approach provides a clean separation between value-added functionalities such as special device support or active services, and the core functionality of the kernel. The fact that kernel modules can be dynamically inserted and removed at run-time without affecting the rest of the system (if the module is well-behaved) introduces a degree of safety and reliability. This allows, for example, an active NodeOS to re-initialise or remove an active module that misbehaves without disruption of the entire system operation. Therefore, it is not surprising that dynamic kernel extensibility based on loadable modules has formed a key building block within several active network implementations. For example, it has been deployed within LARA, Router Plugins and Pronto (see previous sections) as a means to dynamically load and run trusted active extensions inside the kernel.

Although Microsoft's Windows NT, 2000 and XP platforms do not support kernel extensibility by means of loadable modules, the Windows Driver Model (WDM) [One99] enables dynamic loading of drivers and execution of such extensions in kernel-space instead. Since the WDM is more general than only for writing device drivers, this technique can be deployed under Windows to dynamically extent kernel functionality in much the same way as with the previously mentioned modular kernel architectures. It is shown later in chapter 6 how this approach is applied in the Windows 2000 implementation of LARA++.

3.3.2 Safety and Security Mechanisms

According to section 2.8, the main focus for providing safety and security within active networks is the safe execution of active code, and the secure control of resource usage and access to privileged operations such as system configurations. This maps to the following

[13] Modular kernel extensions have been introduced by several operating systems in 1994/95 (for example, NetBSD before version 1.0 release, or Linux since kernel version 1.2).

requirements: (1) a safe execution environment (for example, a sandbox) and/or methods for static analysis of active code in order to verify safety prior to program evaluation (for example, type safety); (2) authentication mechanisms to securely verify users identity and code authenticity (for example, public key and code signing mechanisms); and (3) secure access control mechanisms to safeguard privileged interfaces and control resource usage. Most of these mechanisms can be implemented either implicitly or explicitly. For example, safe code execution could be simply based on code interpretation, and resource control could be neglected if fixed resource bounds are enforced by the execution environment. Also, user authentication could become superfluous if the system does not support user-based privilege levels. The remainder of this section describes related work which focuses primarily on the safety and security aspects of active networks.

Shapiro et al. [SMSF97] suggest the use of the Extremely Reliable Operating System (EROS) for providing safety for active code execution. A code certification mechanism is supplied for the authentication of injected programs. Safe code evaluation is achieved through isolation and control of the execution environment for every application. Moreover, their system proposes resource allocation control based on the concept of *capabilities*.

RCANE [Men99] developed at Cambridge University proposes a resource control framework for active networks. Router resources such as CPU, memory and bandwidth have to be restricted to prevent malicious active programs from causing denial-of-service attacks. RCANE is based on the OCaml [OCA] run-time environment and the PLAN interpreter to provide a restricted execution environment. Security techniques approached by the framework include resource accounting and scheduling (i.e., CPU and network input/output), memory and service management functions (for service creation and manipulation), and session authentication (i.e., the owner of a session must be authenticated).

PLAN-P, proposed by Thibault et al. [TCM98], extends PLAN (see section 3.2.1.2) to a generalised active network programming language running over IP. It provides enhanced programmability while maintaining safety and security. For example, PLAN-P guarantees program delivery and termination. PLAN-P was designed with the aim to improve performance compared to its predecessor. This is achieved through program specialisation. The idea here is to specialise the interpreter/compiler according to the input PLAN-P program at run-time. Performance measurements show that this technique approximately doubled the speed. PLAN-P has been successfully demonstrated for adaptation of distributed applications in extensible networks [TMM99].

SafetyNet [WJGO98], developed at Sussex University, addresses the safety and security issues of active networks at the language level. Wakeman et al. propose to design security policies that protect the integrity of an active network into the type system of a

new programming language dedicated for active programming. The language is strongly typed such that any form of run-time checks required by other languages (for example, Java) can be avoided. SafetyNet also provides a formal model of the active network language which shows the correctness of the type systems and the safety policies. This enables the system to ensure that programs passing the type-check totally conform with the policies.

Yeh et al. [YCN99] have pursued research into the interoperability issues of security policies for end-to-end communication when different administrative domains are involved. Their work proposes a dedicated signalling and path repair mechanism for connection-oriented active network sessions. The path repair mechanism is needed to setup quickly an alternative path when a path failure is detected by the nearest router. A path failure is typically caused either due to congestion on a link or a change in security conditions.

Finally, the reader is also referred to the numerous systems and architectures previously presented throughout this chapter for further information on the safety and security measures used within active networks, such as SANE (section 3.2.2.1), PLAN (section 3.2.1.2), SNAP (section 3.2.1.4) and Janos (section 3.3.1.3).

3.4 Applications and Services

The potential of active networks in solving problems of today's networks (as previously discussed in chapter 1) as well as allowing them to support new types of applications and services is still overlooked by many researchers and industry. The successful deployment of active networks relies on the development of genuine applications and services that demonstrate the potential of active networks to its depth. Fortunately, there is currently an increasing interest in the active network research community to work on applications and services that will benefit from additional computation and/or state in the network. Although the design and development of the 'perfect' active network system is yet under way (novel prototype active network systems are still emerging, and existing platforms are continuously enhanced), many systems are already being used within experimental networks to develop and test applications. For example, the ABone [BR99] is a representative example of such an experimental testbed network. The active overlay network enables testing of different execution environments and active applications across research labs.

The remainder of this section introduces a range of applications and services that are currently under investigation within the active network community. These example applications and services can be divided into the following areas:

Packet Forwarding and Routing

Active network applications of this type exploit the capabilities of active nodes to support custom processing of data flows. The programming interface offered by the nodes allow users to implement special algorithms for packet routing and to define custom forwarding behaviours. Work in this area comprises the following projects: reliable multicast such as the active reliable multicast (ARM) developed at MIT [LGT98]; ICSI's robust multicast audio and layered multicast video protocol [B+97]; application-layer multicast as investigated by the Alpine project at Lancaster [MCH01]; application-specific routing [HMA+99]; and handoff optimisation for Mobile IPv6 [SFSS00a] (see section 7.3.1.3).

Protocol Deployment

Support for dynamic deployment of new functionality throughout the network is undoubtedly the strongest drive for active networking. It opens up the network and allows third-parties to roll out new services and protocols dynamically within a very short time. Instead of being held back by lengthy standardisation processes and router development cycles, new protocols can be dynamically tested and deployed right after the development phase (or even as part of it). Example applications are the active bridging project at the University of Pennsylvania [ASNS97] and the active multicast network (AMnet) project at the University of Karlsruhe (Germany) [MHWZ99]. The latter provides heterogeneous group communication services based on dynamic protocol and service deployment. Furthermore, Legedza et al. [LWG98] studied the opportunities for end-to-end performance improvements of distributed applications as a result of active protocol deployment.

Network Signalling

Research at USC Information Sciences Institute (ISI) explores the feasibility of exploiting active network technologies for network signalling [Lin00]. A special execution environment for signalling, the active signalling protocol (ASP) EE [BCF+01], has been developed to explore the potential of active networks for on-the-fly deployment of network protocols and management software, and dynamic customisation of the network. Initially, the project focuses on well-known signalling protocols, such as resource reservation, alternate path-routing, and dynamic provisioning of differentiated services. For example, a Java-based implementation of the resource reservation protocol (RSVP) [BZB+97] was developed. The Java-based implementation (Jrsvp [B+01]) ensures portability across different active node platforms and enables dynamic extensibility of the base protocol through active network techniques. These features have been demonstrated by starting off with a basic version of the reservation protocol, which has been enhanced by means of a dynamic loading mechanism at run-time.

Congestion Control

Network congestion control is another area in which active networking can be beneficial. Research at the Georgia Institute of Technology [BCZ96] proposes an intelligent frame dropping mechanism that reduces congestion inside the network at the point of occurrence rather than through rate control at the end systems. Since congestion occurs inside the network, usually far away from the actual applications that cause the congestion, congestion control inside the network (where the congestion occurs) exemplifies an excellent application for active networking.

Network Management

Network management has also shown to be a very convincing application domain for active networks. While management systems of traditional networks use polling as a means to periodically acquire status information from the network nodes (for example, to check on smooth operation of the network) [Mar94], active networks enable delegation of management tasks to active programs that report failures immediately upon detection (as an event driven approach). In contrast to the former approach, which is ineffective as repeated polling leads to increased bandwidth usage and undetected failures until the next poll takes place, the latter is highly efficient (i.e., bandwidth is only used when necessary) and responsive (i.e., events are immediately reported). In addition, active programs running locally on the network node that is monitored can exploit all information available at the point of failure in order to derive a meaningful report (i.e., additional requests from the management system may not be needed) and/or take immediate actions to resolve critical problems if appropriate. A range of research projects have emerged in this area including the SmartPackets project at BBN [SJS+00], Columbia's work on network management by delegation [GY98], and the Darwin project at CMU [C+98].

Data Caching and Storage

Data caching and storage 'inside' the network is an example of a totally new application domain. Unlike conventional caching and storage applications, which are implemented within specialised end-nodes, active networks enable data storage within the actual network nodes. Examples of caching applications are the self-organising wide-area network cache developed at Georgia Tech [BCZ98], the dynamic RAM-based network-level cache for high quality video built at Lancaster University [RWS00], and the distributed Web caching, which transparently redirects Web requests to nearby caches, developed at MIT [LG98]. Research at Carnegie Mellon University, by comparison, investigates the integration of storage technologies with active networking [Rie99] in order to provide a whole

end-to-end computing infrastructure. The challenge of network-side data caching is to intelligently partition functionality between clients, network storage and active nodes such that performance, reliability, and scalability of the service is optimal.

Value Added Service

Recent developments show that service support demands of today's networks increase both in quantity and complexity. As active and programmable networks provide sufficient flexibility to deploy not only conventional network level services (such as routing or forwarding), but also enhanced services on higher layers of the OSI reference model that can be tailored towards individual applications and/or users, they are suitable to cope with this demand. Examples of value added services based on active and programmable networks include packet filtering and firewalls [AR94], active email [GBB+01], active content distribution [MF01], distributed on-line auctions [LWG98], and network-level access control for public wireless networks (see section 7.3.1.3 for further details). These examples suggest only a small sample of potential services, which clearly widen the spectrum of traditional network services.

3.5 Summary

This chapter has introduced selected work and developments across the broad spectrum of active and programmable network research. It provides a comprehensive overview of the state-of-the-art in the field and exhibits a number of open problems that drive continuous research into active networks. The various projects described throughout this chapter introduce the relevant aspects of active networking including code distribution and loading techniques for mobile code, safe execution of active code, specialised programming languages and compilers, operating system support, safety and security measures, and novel network architectures.

At the core of this chapter is the division of active and programmable technologies in two main approaches, namely *integrated* and *discrete* active networking. Although their fundamental goals are identical and the timescale when they have emerged is related, the underlying technologies are inherently different. While the former aims to "open up the network" by providing a programmable interface for transient in-band active packet code, the latter aims to do this through support of persistent out-of-band programmable extensions.

The discussion of several representative projects for each approach has revealed the advantages and disadvantages as well as their limitations. It has become clear that the active packet approach to active networks is primarily useful for network signalling and control as the programming capabilities are fairly fine-grained and restrictive, whereas

active extensions enable much more fundamental and coarse extensibility of network capabilities (i.e., they are far less constrained regarding program complexity and service lifetime). They can even support the deployment of an active packet based programming environment across an active network (compare section 3.2.2.1).

In accordance with a study by Hicks and Nettles [HN00], the analysis of current active network approaches presented in this chapter comes to the conclusion that the majority of today's active network systems do not provide sufficient extensibility for future network evolution. They lack a service composition method that allows for extensibility at run-time. In other words, extensibility is limited unnecessarily by assuming a static underlying graph structure or program that cannot be changed or extended at run-time without recompiling, reinstalling, or reconfiguring the active node. This weakness was the main source of motivation for the development of yet another active node architecture.

The diversity of system requirements for active networks is reflected in the large range of projects described throughout this chapter. Common requirements that are investigated by most active network solutions are flexibility, adequate performance, safety and security. The following chapter discusses the whole gamut of the system requirements indicated in this chapter and attempts to classify the variety of requirements into stringent and less-stringent requirements.

Chapter 4

Active Network Requirements

4.1 Overview

This chapter discusses the requirements for active networks in general and derives the specific requirements for the LARA++ active router architecture that is proposed within this work (see chapter 5).

Most common requirements of an active network have already been mentioned in the last chapter. It has become apparent that active network research deals largely with trade-offs. For example, many of the active systems introduced in chapter 3 trade-off system security for performance or flexibility for simplicity.

Research into active networks so far has shown that no single solution will meet all possible requirements of active networks, and thus, multiple systems for domains with different demands must be able to co-exist and interoperate. The challenge in designing active network solutions therefore is to draw the optimal line between trade-offs depending on the requirements at hand. For this, it is crucial to understand fully the requirements of a given domain (for example, an active router designed for core networks has totally different performance requirements from an edge router).

4.2 Requirements

The general requirements for active networks can be divided into two categories, namely those that are fundamental to an active network architecture and those that enable enhanced features. This section therefore introduces the following categories:

1. *Class A requirements* encompass all requirements without which a solution would either not classify as an active network or lack feasibility to deploy in real environments:

Class / No	Requirements
A.1	Programmability
A.2	Flexibility
A.3	Safety
A.4	Security
A.5	Resource Control
A.6	Adequate Performance
A.7	Sufficient Manageability

2. *Class B requirements* include those requirements that are commonly considered to be valuable for active networks and those that will become crucial for large-scale active network deployments:

Class / No	Requirements
B.1	Interoperability
B.2	High Performance
B.3	Scalable Manageability
B.4	Business Model
B.5	QoS Support

The remainder of this section describes these requirements in depth and explains their significance in more detail. Relevant active network research (previously introduced in chapter 3) will be cited when appropriate in order to show how those requirements are dealt with by others.

4.2.1 Class A Requirements

4.2.1.1 Programmability

The primary requirement of an active network is by definition programmability. In order to qualify as an active network, network nodes require at least some form of programming interface that permits users to remotely control or program the network (i.e., forwarding behaviour, management operation, etc.).

As we have seen in chapter 2 and 3, different programming models (discrete or integrated), program distribution approaches (in-band or out-of-band), programming languages (for example, PLAN, Java, and Caml), and network APIs have evolved – all with the one goal, namely to enable network programmability.

Network programmability as suggested by active networks requires mechanisms to dynamically download, securely authorise, and safely evaluate mobile code on network nodes. The different approaches presented in chapter 3 demonstrate that the choice of

programming model and implementation has a great impact on the overall performance of the active node.

Key to active programmability is the choice of the programming model and execution environment. Depending on the application domain at hand, it has to be decided whether in-band active code distribution and execution is advantageous over out-of-band programming and what kind of execution environment (i.e., code evaluation in kernel or user-space; or code evaluation through interpretation or execution) is preferred. As these decisions are fundamental for the implementation of an active node, they have far reaching consequences. For example, in-band active code distribution limits the architecture to fairly small active programs, while out-of-band programmability is typically far less dynamic.

4.2.1.2 Flexibility

Flexibility is, like programmability, a fundamental property of active networks. For a particular active node architecture the question to be asked therefore is what degree of flexibility is required for the given application domain? The requirements of flexibility and programmability are closely related. In fact, the choice of programming model determines to some degree how flexible a system can be.

Active network approaches that choose high flexibility as a key requirement typically attempt to provide a general-purpose (or *Turing complete*) programming environment. Examples described in chapter 3 that follow this line are ANTS, CANEs, and LARA. However, as other research has indicated, providing a high degree of flexibility typically results in complex programming environments, which are hard to secure. PLAN, for example, is an approach that sacrifices flexibility in order to achieve a lightweight programming environment. Projects that concentrate on a particular application domain (for example, SmartPackets only support network management operations, and Protocol Boosters focus on protocol customisation) typically restrict the flexibility of the programming environment in order to simplify safety and security procedures.

Finally, an analysis of different programming interfaces has shown that flexibility in a programmable environment (independent of its degree, but more importantly for high levels of flexibility) must be constrained to useful programming abstractions in order to provide developers with a practical API. For example, without special APIs to process network packets (i.e., receive, send, fragment, etc.) or deal with security (i.e., user authentication, access control capabilities, etc.), it can be very expensive to program active code.

4.2.1.3 Safety

Safety within active networks is primarily concerned with the reliability of the active nodes. It ensures that active code loaded on an active node executes safely without causing the system to malfunction. Safety and reliability of active network nodes are particularly important, as the consequences of a system failure can be far reaching – beyond the scope of the user or active program that caused the problem. In fact, erroneous behaviour on a shared network device may impair many users or take down the whole network node.

As network users typically do not tolerate degradation of service reliability with the advent of new technologies, safety is another vital requirement for active networks. The challenge here is to offer a similar level of reliability to that known from conventional networks despite the fact that active network nodes execute user-defined mobile code as part of their normal operation, which makes them far more vulnerable.

Given the importance of this property, most active network architectures consider safety as an integral element of their design. However, it has been shown in the previous chapter that different approaches to ensure safety have been favoured. (*a*) Some approaches provide safety for active computation based on safe code evaluation by means of a virtual machine or interpreter. For example, Java based systems such as ANTS provide safety based on the Java virtual machine; PLAN is an example of an interpreted language. (*b*) Others, for example Bowman or LARA[1], exploit operating system mechanisms such as virtual memory and multi-tasking to ensure safety. This latter approach has the advantage that the system has low-level control over the active processing and no additional overheads occur because of an extra layer (i.e., virtual machine or interpreter) on top of the node OS. It also allows hardware support offered by most modern processors to be exploited for efficient protection. The downside of operating system-based techniques is the complexity involved. Developing such systems is far more challenging. (*c*)Recently, a third approach based on specialised programming languages has emerged. The idea here is to exploit language features to ensure safety. Examples of such solutions are Caml as used within SwitchWare and SafetyNet. The advantage is that safety is more or less embedded into the active programs themselves, but typically at the cost of reduced performance during execution. In a way similar to the first solution, however, this approach usually lacks control in the case of unexpected failures of the language run-time system or virtual machine respectively.

[1]Note that LARA provides only partial safety (i.e., protection of the processing resources) based on a specialised kernel-space active thread scheduler.

4.2.1.4 Security

The recent emergence of network and transport level security mechanisms, such as IPSec [KA98] and SSL [SSL], is an indication of the importance of security within today's networks. In the context of active networks, security must demand at least as much attention considering the extra capabilities and complexity. The challenge for active networks therefore is to maintain an equivalent level of security as in conventional networks while enabling flexible programmability for network users (potentially even unauthenticated users) at run-time .

The fact that active network nodes expose some form of a programming interface, which allows remote users to customise the packet processing path or extend the node functionality, calls for strong security mechanisms. The administrators of an active network must be able to securely control who can program the active nodes and to what extent (for example, which programming interfaces or node configurations user-provided active code can access, and how many resources these programs can consume).

As we have seen in previous chapters, security is a key topic within active network research. Many active network projects take security very seriously. SANE, for example, is an active network architecture that is primarily concerned with security. Most projects however simply restrict the flexibility of their architecture in order to ease security provisioning. It is not yet clear whether or not security and flexibility is in fact a trade-off. But more likely, it is simply more convenient to reduce the flexibility of a system in order to reduce the complexity of developing the security mechanisms. Examples of such projects are PLAN or SNAP. A few projects, such as PAN or ANN, even omit security entirely (or leave it to external mechanisms), as they focus mainly on the performance issues of active nodes.

In accordance with previous discussions, security within active networks requires mechanisms to control access to the system (i.e., to system configurations, the code loader, the programming interface, and system resources). This is typically done by means of security policing (see section 2.8.2). While the security policies define who has access to the various controls and to what extent, the enforcement engine ensures that these policy rules are strictly obeyed at all times. In addition, a mechanism that enables auditing a node in order to track down security problems or potential threats is useful considering that complex security systems are always vulnerable and conceal loopholes.

4.2.1.5 Resource Control

Resource control within active networks is generally concerned with the scheduling and provisioning of node-local resources such as storage, memory, bandwidth, and processing resources. It ensures that active programs running on a node receive a fair (equal or

prioritised) or reserved share of the available resources.

A minimum degree of resource control is required to protect a node's resources from malicious users or active programs. Since resource control prevents individual programs from starving others or even the node OS, it embodies an important aspect of safety and security. Resource control is hence considered another vital requirement.

4.2.1.6 Adequate Performance

Performance is an important requirement in networking. In fact, it could be argued that performance is, besides robustness, probably the foremost property of a network. Consequently, this is equally true for active networks. Moreover, the fact that active networks add computational overheads besides the normal forwarding intensifies the need for good performance.

In view of this observation, it seems absurd to even consider the idea of active networks since current network devices are already struggling to cope with the rapidly growing performance requirements of today's Internet. However, before jumping to a hasty conclusion, it is important to put this in the right perspective. In order to provide 'adequate' performance, it is important to consider the actual network (or network segment) of interest. For example, performing line-speed active processing on edge networks with up to 100 Mbps has been demonstrated based on many approaches (for example, PLAN, SmartPackets, or Protocol Boosters). In contrast, the provision of active network solutions for core networks is a completely different issue. So far, only a few projects, namely ANN, PAN and LARA, have aimed for high performance active routers.

Ideally, active network performance should be comparable to the performance of traditional networks. For a general acceptance of active networks, it is vital to provide adequate performance for the environment where the new technology will be deployed. For example, a simple active router running a Java based execution environment in user-space is not acceptable for core networks, where throughputs up to many Gigabits per seconds must be processed. However, the same solution might be perfectly acceptable as a programmable device on a small to medium sized edge network.

4.2.1.7 Sufficient Manageability

Active network solutions have been successfully demonstrated as facilitators for network management applications. A few of them have been previously mentioned in section 3.4. However, an active network itself demands a substantial amount of management and administration. In fact, the move from passive to active networks brings along many new administrative obligations that need to be addressed sufficiently in order to make this technology a success.

Manageability of active networks at the micro level is concerned with the management of individual network nodes. As the control of active nodes is typically achieved through some form of policing (see section 2.8.2), node manageability is mainly a matter of managing policy rules. For example, functionality to define, add, and remove policies for system and resource access is required.

As active networks grow beyond the boundaries of individual research labs, the issue of global management (across multiple administrative domains) must also be addressed. However, manageability of active networks at the macro level is yet another hurdle to overcome. With the exception of a few projects (for example, ABone [BR99] or Alpine [Ban01]) not much research has been carried out in this area. The main reason for this is probably that active networking is still in its infancy and very few larger scale experiments have been carried out. The ABone virtual active network with less than 100 registered nodes[2] is still the largest scale deployment of an active network.

Nevertheless, for the general acceptance of active networks, it is crucial to provide sufficient manageability that enables a smooth roll-out of the technology. This implies, for example, that mechanisms are in place to conveniently deal with a moderate to large user base. With respect to the user management, for example, it would be utterly impractical to tie security or resource access policies only to individual user accounts [Ber00]. Rather, mechanisms to aggregate users into user groups (for example, network administrators, privileged users, all or unauthorised users) and associate policies to user groups must be considered in order to reduce the management overhead of active networks.

4.2.2 Class B Requirements

4.2.2.1 Interoperability

As active network solutions are accepted and deployed more widely, interoperability among the different active network systems and applications becomes important.

Early work in active networks [VRLC97] has already recognised that the range of potential applications is too large for a single architecture to satisfy sufficiently all their requirements. Clearly, an application that extends the network by uploading a new network protocol has entirely different demands from a network management application. Thus, depending on the problem domain at hand, different programming models (programming languages and interfaces) and code distribution mechanisms (in-band or out-of-band) are best suitable. It is therefore crucial as active networks are deployed beyond research networks that different active network architectures, tailored towards

[2] According to a report of the ABone Coordination Center at http://www.isi.edu/abone/abocc.html in August 2002

certain types of applications, can co-exist and interoperate in the global network.

Interoperability among active network architectures and applications provides end-users with the flexibility to select the most appropriate service relative to their current needs. Early active network research proposed two protocols, namely ANEP [A+97] and ANON [Tsc99] to support interoperability among different active network nodes and solutions.

Furthermore, interoperability within active networks is not only an issue of enabling different active network approaches to interoperate, but also about considering the problem of unsolicited interactions among active programs and services running on a single system. This type of interoperability is still largely underestimated, as no wide-scale deployment of active networks has been carried out yet. But it is expected that the increase in flexibility as a result of using active networks leaves a massive feature interaction problem behind that needs to be solved as active networks evolve.

4.2.2.2 High Performance

When active network solutions in today's research labs and small-scale edge networks will have proven to be a successful evolution of passive networks, they will certainly gain a more general acceptance among network vendors and service providers. This will probably initiate a larger-scale deployment of the new technology. However, for this to be successful, active network solutions must be scalable in performance towards large amounts of traffic. Although users and service providers would probably accept a small drop in performance for a significant increase in flexibility and functionality, they would reject the technology if the network became too slow.

While current active networks are primarily used by researchers and possibly a few test users, a wider deployment would drastically increase the user base and consequently the throughput requirements for these network devices. The challenge will be to accommodate the additional overheads caused by active computations and continue processing packets at speeds approaching the line speeds of the surrounding networks. As previously shown, it is absolutely vital for active networks to support adequate performance for a given environment.

Finally, high-speed active network nodes suitable for high-throughput core networks may become of interest in order to provide programmability through all parts of the network – from the core to the edge. Although current research implementations of active network systems are still far away from providing performances close to that required by today's core networks, research at the University of Washington (see section 3.2.2.8) and Lancaster University (see section 3.2.2.5) already investigate scalable, high-performance active network architectures.

4.2.2.3 Scalable Manageability

Active network solutions demand scalable mechanisms for management and administration when widely deployed across large inter-networks.

The challenge will be to administer large numbers of active programs and users across the whole inter-network. For example, a global naming scheme for active code and users or appropriate mappings between different domains, and adequate aggregation mechanisms (i.e., user groups, privilege levels) will be compulsory. Furthermore, the definition and distribution of security and resource access policies across administrative domains will demand scalable solutions.

The idea of macro-level manageability, whereby large administrative domains are managed as a single entity, seems to be an important step towards scalable manageability of active networks.

4.2.2.4 Business Model

A key factor as to whether or not active networks will become a real success depends on a sound business model for the new technology. Without a promising business perspective, network vendor and service providers will have no urge to move towards an active network in the first place. In fact, the risk of losing some of their revenue to third parties (for example, active service providers or active component providers) forces network vendors and service provides to remain cautious. Such behaviour, for example, can be observed by looking at traditional router vendors (for example, Cisco or Nortel Networks). They show no immediate interest in active network research. Obviously, allowing third parties to program and extent router functionality directly conflicts with the business interests of such companies.

From this, we can conclude that a wider deployment of active networks relies very much on a promising business model for current stakeholders and additional business opportunities for new players in the area.

4.2.2.5 QoS Support

The issue of quality-of-service (QoS) provisioning within active networks is another long-term requirement that becomes more and more important as the users of the shared resources take it more seriously and the start relying on it. This effect can currently be observed in the Internet. Since the Internet is now predominantly used for commercial applications and services, which rely on appropriate QoS for their customers, QoS provisioning has become a key issue. Efforts are currently underway to provide QoS within the Internet, namely integrated services (IntServ) [BCS94] and differentiated services (DiffServ) [BBC+98].

In the context of active networks, the challenge of QoS provisioning is closely related to the problem of resource control. However, unlike resource control, which is mainly concerned with the appropriate scheduling of resources, QoS support is a broader undertaking that also involves admission control and some form of resource reservation or class-of-service (CoS) mechanism.

QoS provisioning within active networks differs from that of conventional data networks primarily as the node-local processing resources in every intermediate network node play a significant role. The interaction between node-local QoS and end-to-end QoS yields interesting requirements. For example, an end-to-end reservation for a stream demands node-local resource reservations for the active processing of the stream on all active routers along the transmission path. Although the problem of mapping end-to-end reservations onto internal QoS mechanisms is not new, in the context of active networks the problem is intensified. Active computation, potentially within several active routers along the transmission path, accounts for a greater share of the overall transmission time than packet forwarding in conventional routers.

Similar to DiffServ in the Internet, *soft* QoS within active routers requires the ability to define and support different service classes for active computations based on absolute or relative QoS attributes. Likewise, *hard* QoS relies on the concept of resource reservation in much the same way as in IntServ. Active routers with hard QoS support therefore require a means for resource reservation, admission control and resource scheduling. In either case (soft or hard QoS), the active router QoS framework must define a mapping from end-to-end resource reservations or service classes to node-local reservations or service classes. While the mapping between soft-QoS related service classes is relatively simple when the appropriate internal service classes are supported, the mapping between hard-QoS reservations is less straightforward. External resource reservations can only be granted if sufficient processing resources are available internally to the nodes along the transmission path to perform the active computations within the given QoS bounds. Following the example presented above, the respective active routers must be able to reserve sufficient processing resources for every active program involved in the processing of the stream.

4.3 LARA++ Design Requirements

The primary goal of the work presented here is to develop a generic active router architecture that can be used to design and build *real* active network devices. However, since the prototyping of such systems is impeded by financial resources and time constraints in the context of a PhD project, the objective is rather to use low-cost, commodity hardware and software as a base platform and to focus on the development of a minimal, but

extensible edge router that may serve as a flexible platform for future network research. Hence, the main challenge for the design is to come up with an architecture that scales well from low-cost systems up to high-performance routers with specialised hardware support.

The remainder of this section introduces the key design objectives of the LARA++ architecture and shows how these relate to the general requirements of active networks described in the last section. Moreover, several additional, LARA++ specific requirements (labelled L.1, L.2, ...) are drawn from these design goals.

4.3.1 Flexible Extensibility

In agreement with the primary goal of active networks, LARA++ strives first and foremost to provide maximum flexibility to allow maximum choice in terms of how the network can be programmed and services tailored to suit the needs of user applications. Maximum flexibility is especially of great importance since LARA++ aims to provide a research platform that offers a high degree of flexibly for the end users and their applications, and for future development.

Building on lessons from the development of LARA (see section 3.2.2.5), we have learned that the ability to extend flexibly the functionality of available network protocols and services is crucial. Since most applications for active networks expect merely simple modifications or extensions to existing network protocols and services (for example, support of a new protocol option or extension header), the ability to extend or replace available software components flexibly, rather than having to re-implement significant parts of the protocol stack or service every time, is important. This feature demands individual software components to provide extension interfaces, or active routers to support flexible extension mechanisms (for example, plug-in or composition support). The fact that many active network applications aim to modify or extend the behaviour and functionality of current protocol suites within network nodes, leads to the conclusion that active routers should provide an extension mechanism for existing protocols and services. This reveals two specific requirements for LARA++: a flexible extension mechanism and support for expandability of existing network protocols and services.

Flexible extensibility of router functionality also demands that active programming is not limited by language restrictions (i.e., specialised languages that are limited to dataflow processing, small code size and/or security) and lacking programming interfaces (i.e., restrictive APIs or lack of support libraries). As a result, two further requirements for LARA++ can be derived: *Turing-complete* programmability and support for interface extensibility and support libraries.

4.3.2 Moderate Performance

As we have seen in section 4.2.1.6, adequate performance is a vital requirement for active networks. However, as LARA++ strives primarily to provide maximum flexibility, which is mainly needed at the edge of the network, adequate performance is less demanding.

An analysis of active network applications shows that in the core of the network active processing is mainly tied to "simple" layer 3 processing, such as multicast, congestion control, and routing. At the edge of the network, however, active applications range from packet filtering, network access control, data compression and media conversions to data caching and storage (for more details see section 3.4). This shows that maximum flexibility is predominantly needed near the edge of the network (close to the user) where performance is less crucial.

In conclusion, since LARA++ aims to provide an architecture for flexible edge devices, moderate performance is adequate. LARA++ therefore strives to support active processing at speeds approaching the line speed of typical edge networks, which equates to speeds up to 100 Mbps today.

4.3.3 Highly Dynamic Programmability

According to the whole concept of active networking, dynamic programmability is clearly an inherent requirement. As previously discussed in section 2.6.1 and 3.5, the code distribution approach used (in-band or out-of-band) has a great impact on the granularity of programmability. While in-band active code distribution enables very fine-grained programming (on a per-packet basis), out-of-band programming is usually less dynamic and involves uploading of rather heavyweight active programs.

The choice of the active programming approach (integrated or discrete) also affects flexibility. On the one hand, integrated programming maximises flexibility through support of fine-grain programmability, but it limits flexibility due to the hard restrictions on code size and instantiation time. Discrete programming, on the other hand, maximises flexibility through unrestrictive programming, but restricts flexibility through more coarse-grain program distribution.

Consequently, a hybrid approach that tries to consolidate the best features of both worlds is favoured: unlimited programmability through out-of-band program distribution and highly dynamic programmability through fast program loading/instantiation.

4.3.4 Easy Usability

Usability of active networks can be viewed from the perspective of the end users who are the actual beneficiaries of the active network, or the programmers who develop the active programs.

In order to make LARA++ a success, it is clear that the programming (i.e., code loading and instantiation) of the network devices must be hidden from the end user's view. End users should not have to deal with such peculiarities; instead, the applications should take care of setting up and configuring the active services. From a usability aspect, the discrete approach to active programmability has the advantage that active services can be loaded and initiated through external applications, whereas integrated solutions rely on intrinsic support by the user applications.

From a programmer's perspective, usability of an active network architecture is defined by the programming capabilities and the ease of use. This is largely dependent on the programming language, the APIs and library support, the execution environment (user or kernel-space), and the debugging and testing facilities provided.

In summary, ease of use of an active network architecture from both the end user and the programmer perspective is another important design criterion. The LARA++ architecture makes this a key design objective.

4.3.5 Safe Code Execution

In order to satisfy the vital security requirements of active networks, LARA++ must provide security measures that protect active nodes from malicious users and their hazardous code, without restricting flexible programmability for end-users. This is especially important for active edge devices (like LARA++), where programmability is primarily offered to the end-users of the network.

According to the discussion in section 2.8.1, there are mainly two approaches towards safe active code execution, namely operating system and programming language-based protection mechanisms. While the latter approach has the advantage that no special support is required in the NodeOS, the former is favourable in cases where maximal flexibility and reliability are required. Operating system-based safety measures do not restrict programmability with respect to expressibility (i.e. potentially any programming language can be used) and performance (i.e. the most suitable language can be chosen). Furthermore, operating system-based protection is considered to be highly reliable, as it enforces safety at run-time (i.e. it allows system protection even from program failures) and typically makes use of the protection mechanisms supported by the system hardware.

4.3.6 Secure Programmability

Key techniques to provide security on active nodes, without compromising flexible programmability for end-users, are access control, user authentication, and code signatures. Access policies, which form the access control list, define who has access to a node, or in other words, which user can program the node and to what extent.

In this context, user authentication plays the important role of ensuring the integrity of user identities. The origin of active code (i.e., the programmer or code provider) is also considered. This can be accomplished by means of code signatures, which authenticate the programmer or code provider, and verify the integrity of the code (i.e., make sure that the code has not been tampered with along the transmission path). As a consequence, the final decision of whether or not an active program is considered secure depends on both the user loading the code and the code origin. For example, an average user may only be allowed to install an active program signed by a trusted company, whereas a network administrator may be permitted to install even its own programs.

4.3.7 Scalable Manageability

Scalable manageability on active nodes is mainly concerned with the handling of access and security policies for potentially large numbers of users. It is therefore important to introduce the concept of user groups and level-of-privilege classes as such aggregates can greatly reduce the number of policies needed. It allows access policies to be defined on a per-group rather than a per-user basis.

Moreover, defining a "default" group and privilege class for standard users without special permissions should largely reduce the number of policies since the majority of end-users will likely belong to this group or privilege class. Consequently, the use of a default privilege class with minimal access rights (for example, to install active code that can only operates on the user's data streams) for unauthenticated users makes manageability scalable. In summary, this practice allows limited but global programmability by anybody without additional management cost.

4.3.8 Summary of LARA++ Requirements

The following table summarises the primary requirements of the LARA++ architecture. These requirements have been derived from the fundamental or class A requirements of active networks (see section 4.2.1) and tailored towards the specific design objectives of LARA++ as described above.

Class / No	Requirements
L.1	Flexible Extensibility • Extensibility of existing protocols and services • Support for interface and library extensibility • Complete programmability
L.2	Moderate Performance • Active processing near to lines speeds of typical edge networks

Class / No	Requirements
L.3	Highly Dynamic Programmability • Fast active program loading and instantiation
L.4	Easy Usability • Convenient active "programming" for end-users • Debugging support for active program developers
L.5	Safe Code Execution • High reliability and performance through system-level protection mechanisms
L.6	Secure Programmability • Flexible end-user programmability without security hazards
L.7	Scalable Manageability • Scalability through user aggregation and default access policies • Restrictive default privilege class for unauthenticated (unknown) users

4.4 Summary

This chapter has examined the requirements for active networks and active node architectures in particular.

Section 4.2 discussed the general requirements for active networks and systems. These requirements have been derived from related work and acknowledged publications in the field. They summarise the general requirements of today's active network solutions. A closer examination of the requirements has suggested dividing them into two categories: those that are fundamental to the design of functional and acceptable active network solutions, and those that will become important as active network systems get deployed on a larger scale in the future. Section 4.3 then derived the LARA++ specific requirements and tailored them according to its particular design goals, namely to constitute a highly flexible and extensible programmable edge router. Section 4.3.8 has summarised these requirements in tabular form.

The following chapters present the design (chapter 5) and implementation (chapter 6) of the LARA++ active router architecture. Both chapters show how LARA++ fulfils the requirements defined in this chapter by design and implementation. Finally, chapter 7 evaluates to what extent LARA++ has succeeded in meeting these requirements.

Chapter 5

The LARA++ Architecture

5.1 Overview

The actual realisation of an active network architecture has been revealed to be non-trivial. Chapter 3 has described a large variety of active network solutions. However, most of them are very much tailored towards a specific application or application domain. Moreover, with the exception of very few (for example, ANTS) hardly any of these platforms have been used outside their own research environment. The development of a more generic active network platform therefore requires a wider and more thorough look at the requirements. Chapter 4 has discussed the multitude of requirements and has defined the specific requirements for the development of the Lancaster Active Router Architecture (LARA++).

This central chapter of the work introduces the design of Lancaster's second-generation active router architecture and discusses how it fulfils the requirements that have been defined above in principle and design. This design chapter and the subsequent realisation and evaluation chapters therefore contain the key contributions made in this thesis.

The structure of this chapter is as follows: section 5.2 starts with a summary of the motivation behind the development of the LARA++ active node architecture. Section 5.3 then summarises the background work that has been carried out a-priori as part of the LARA project. Preceded by a brief design overview of LARA++ (section 5.4), section 5.5 introduces the main components of the LARA++ active node architecture. Section 5.6 then continues with a thorough description of the unique service composition approach proposed by LARA++. Finally, a discussion on safety and security (section 5.7) and policing (section 5.8) is provided before section 5.9 concludes this chapter.

5.2 Motivation

The 'killer application' for active networking is probably the "unknown application". The ability to provide services that are not yet known to the device manufacturers and system vendors at the time of network deployment is certainly considered a desirable feature of an active network. Such a degree of flexibility would enable the rapid deployment of new services by third parties, and provide an unprecedented level of customisation by network administrators and end users.

Although many active network services and applications require only simple modifications or extensions of existing protocol stacks and/or services (for example, to optimise or improve the operation of a service), a study of available active network architectures has shown that most existing platforms do not offer the degree of flexibility required for the dynamic deployment of such services. For example, the lack of flexibility is often caused by drives for simplicity and/or performance, or as a result of targeting the platform towards a narrow range of user applications.

Furthermore, the closer examination of extensible active router architectures has revealed that the majority of existing platforms lack sufficient extensibility to account for true evolution. In accordance with a study by Hicks and Nettles [HN00], the problem has been narrowed down to: plug-in based architectures typically limit the scope of future changes through pre-defined interfaces and an inflexible "underlying program" (core system). Instead, true extensibility requires the the underlying program (providing the service composite) to be minimal. For example, assuming an underlying program with a static interface that is not necessarily suitable for the lifetime of the system unnecessarily limits extensibility. While the needs of users are hard to predict and hence change frequently in the fast evolving world of network communication, it is desirable to provide a flexible composition framework with a minimal inter-component and composition interface, but a rich general-purpose API for active computations.

Moreover, the dynamic roll-out of network protocols and services in a real environment demands a sophisticated service composition framework, which allows services from independent users to interoperate and collaborate on the shared network nodes. Section 5.2.1 describes an example scenario for which such mechanisms are required. Unfortunately, most current active network systems lack sufficient support for collaborative service composition as they have simply been designed to prove the concept of dynamic service provisioning, rather than for practical use.

The lack of flexibility found in current active router designs and the limited support for collaborative service composition gave inspiration for the design and development of a novel component-based active router architecture, which enables flexible and extensible network programmability based on service composition. The following reference scenario

illustrates the kind of interactions among services on a single active node LARA++ intends to support.

5.2.1 An Example Scenario

Before we describe the LARA++ architecture, we outline a scenario that adequately encompasses many of the problems of service composition. This reference scenario precipitates the desire to provide a flexible and scalable composition model as part of the LARA++ architecture.

The scenario considers the case where both end-users and the administrator of a corporate network want independently to program an active edge router. The administrator in this example wants to enable a simple congestion-control mechanism, which supports differentiated classes of service based on a proprietary packet marking technique (for example, based on a new IP option). The administrator would therefore need to upload active code onto the active router which intercepts all relevant packets in order to apply the local congestion control algorithm. At the same time, an end user on the corporate network may want to upload fast handoff support[1] for data streams destined to its mobile terminal (see Schmid et al. [SFSS00a] for further details). The challenge here is to make active applications and services of various users (with different privilege levels) co-exist and co-operate. In this scenario, this means that the congestion control mechanism and the fast handoff service can both be active on the edge router and that packets streamed to the end-user's mobile device benefit from both services.

This example scenario demonstrates the type of active services LARA++ aims to support. It also serves as a reference scenario throughout this chapter to illustrate the problems and challenges of the LARA++ architecture.

5.3 Background Work

As the name indicates, LARA++ evolved from the LARA architecture, which was previously developed at Lancaster University. This section emphasises several LARA design features and flaws which had a direct impact on the design of LARA++. A more general overview of the LARA architecture has already been provided in section 3.2.2.5.

The initial drive for the LARA++ architecture arose from the results of a usability study of LARA. The fact that LARA requires 'active programmers' to develop low-level system code, which differs substantially from typical user-space programming (i.e., different APIs must be used, development and debugging support are minimal, etc.) and is highly critical with respect to system failures (i.e., a program error typically leads to

[1] This active application temporarily optimises the routing to a mobile node's new network location (on the node where the route change takes place) until the mobile routing protocol converges.

a total system crash), led to the reconsideration of the original software architecture.

These usability drawbacks of LARA in conjunction with the more general limitations of active programmability, namely the lack of flexibility for active code integration and interaction on routers, underlined the intention of developing a new software architecture from scratch. As discussed in more depth later, a fundamental step towards this goal is to introduce high-level processing environments (allowing safe and easy development of active code) and high-performance communication channels between the low-level NodeOS and these processing environments.

The remainder of this section introduces a few further aspects of the LARA architecture, which either have had a direct impact on the LARA++ design, or have been simply adopted by the new architecture.

One of the key contributions of the LARA architecture is its innovative hardware architecture. LARA proposes a scalable, high-performance architecture based on inexpensive and off-the-shelf hardware components. The use of dedicated processing engines on a per-interface basis provides substantial processing power and thus high-performance for the active computation. A switched interconnect technology (for example, an ATM network between the interface processors) enables the scalable extension of network interfaces (see Figure 3.2).

LARA++, by comparison, lays the main focus on the software design of the active router architecture. Its software architecture is designed to be largely independent of the underlying hardware, which makes it easily adaptable to a new platform. In particular, the design of LARA++ considers compliance with LARA's Cerberus architecture an important issue. However, the fact that Cerberus nodes are already distributed systems in themselves, opens a "Pandora's box" of new questions and issues (for example, how to distribute state across the distributed platform, where to store and load active code, etc). Future work on LARA++ is expected to deal with these issues and to implement the software architecture proposed herein on a Cerberus node.

Another contribution of the LARA architecture is the concept of source code based active program distribution and just-in-time (JIT) compilation at loading time. The main benefit of this approach is that active programmability can be language independent to the extent to which language specific JIT compilers and APIs are provided on the active nodes. LARA++ extends this approach by also enabling the distribution of binary code. This method is especially valuable in conjunction with on-demand-loading of active code, as the request message can enclose a target platform identifier allowing the code server to upload the correct binary format.

5.4 System Design

This section provides a high-level design overview of the LARA++ architecture describing the principal design decisions and the reasoning behind it.

5.4.1 Edge Device

LARA++ is designed as an active edge router for use in small to medium sized networks. Despite the fact that LARA++ has continued the work previously carried out under LARA, the focus has shifted from developing a high-performance hardware architecture to the design of a flexible software architecture. As a consequence, LARA++ redefines the software architecture in order to promote flexibility and extensibility in the underlying design. Although the new software architecture deals carefully with performance critical operations and preserves basic compatibility with the Cerberus hardware architecture, the trade-off between performance and flexibility limits LARA++ to moderate performance. However, since LARA++ is primarily intended to be used within small to medium size edge networks, moderate performance is sufficient. The second application domain of LARA++, namely as a research platform for experimentation with novel network protocols and services, also does not require support for high-performance.

The new software architecture has been designed independent of the underlying hardware architecture. In fact, LARA++ tries to scale from single processor to multi processor machines and from centralised to distributed router architectures. Therefore, the software architecture accounts for deployment on a single-interface, single-processor node as well as on the distributed Cerberus hardware architecture (see Figure 5.1).

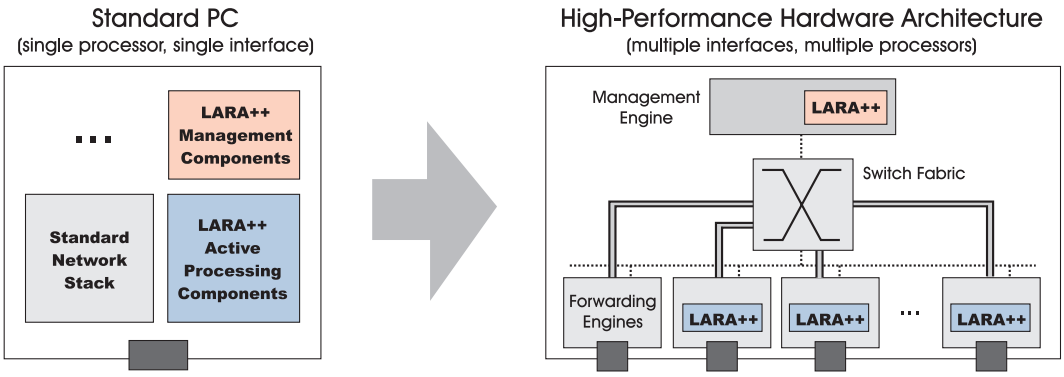


Figure 5.1: The LARA++ software architecture is scalable from low-spec router up to specialised high-performance hardware architectures.

Like conventional edge routers, LARA++ operates on top of the link-layer protocol. It intercepts link-layer frames arriving on one of its interfaces and forwards them on any other interface. Active LARA++ programs or services who indicate interest in processing a frame that is passing through the node, receive the frame for processing.

Depending on the active application and its privilege level, it may simply process the frame (i.e., apply some form of active computation or update router local state) and pass the frame on for further processing, or it may decide the frame's fate (i.e., whether the frame is dropped or forwarded or if a special forwarding behaviour is applied) and take the appropriate actions. Standard IPv4/v6 routing and forwarding serves as a fall-back solution for the case where none of the active applications deal with that issue.

The advantage of this active node design is that active computation can be applied transparently to the end-systems and applications. Conventional routers can therefore be directly replaced with LARA++ active routers – without the loss of functionality[2], but with the prospect of flexible extensibility and programmability.

5.4.2 Programming Model

Conventional active nodes are based upon one of two programming models – either the *integrated* approach, where in-band active programs are executed within the context of specialised execution environments, or the *discrete* approach, which enables users to load and execute active programs out-of-band, prior to sending the data. According to the discussion in section 2.6.1, the former approach tends to be fairly restrictive because of the limited programming capabilities (for example, active programs are very limited in code size, and/or execution environments offers only a small, fixed programming interface), while the latter often lacks adequate service compositing capabilities for software components. LARA++, by contrast, tries to resolve these limitations by extending the discrete programming model by a flexible composition framework for 'active' software components. The service composition framework enables complex active programs or services to be divided into many simple and easy to develop functional components, which are then dynamically re-assembled or composed on the active router at run-time.

In the same way as other discrete approaches, LARA++ routers are programmed individually through out-of-band loading and instantiation of active programs, referred to as *software components*. Its composition framework allows for flexible integration of these components on the router. Since these components are not restricted in size, they can provide substantial extra functionality on the nodes.

Out-of-band distribution of active code also allows programmability through distribution of active service 'configurations'. Active nodes receiving such a configuration try to load and initiate the active components required by the service from their code cache. If the code is not yet locally cached, an on-demand code fetching mechanism (as for example proposed by Wetherall et al. [WGT98]) takes care of loading the code from the component's source host or any intermediate caches. This approach can be especially

[2]Assuming that the base platform for our LARA++ architecture, or in other words the fall-back solution, supports the same functionality as the replacement router.

efficient if active code is likely to be available at the active nodes.

Furthermore, the fact that LARA++ components are (unlike in-band active network programs) not transient by nature, they can provide long-term functionality. As a consequence, the LARA++ programming model also enables extensibility of the programming capabilities of a router through provisioning of a new programming interface.

5.4.3 Component Architecture

> *"We are not certain what form a new model might take, but suggest that it will be more component-based than layered."*
>
> – *Tennenhouse, 1996 [TW96]*

Although Tennenhouse believed right from the beginning of his involvement in active network research that a component-based approach is most suitable for active networks, none of the early approaches have truly adopted a component model. The work presented here (along with the Bowman & CANEs architecture, which has evolved at the same time), constitutes one of the first componentised active router architectures. Both provide a flexible service composition framework that enables dynamic deployment and creation of active services.

The logical decoupling of the underlying router platform from the "active component software" and the implicit self-contained nature of components made this the most appropriate choice for the construction of the LARA++ architecture whose principal aim was flexibility. As such, LARA++ profits from the general advantages of component-based design, namely code modularity, reusability and dynamic composition, to facilitate development and deployment of custom network services.

The component architecture allows complex active programs and services to be split into simple and easy-to-develop functional components. This 'divide and conquer' approach eases the component design and development, because only "small" programs with limited functionality need to be built, and the composition of the active services is taken care of by the composition framework. Furthermore, this modular approach also simplifies extensibility of router functionality. Since components typically have a well-defined and tidy component interface, dynamic extensibility and replacement of individual components is straightforward.

Figure 5.2 provides a conceptual overview of the approach taken. The vision of the LARA++ platform is to provide a framework upon which complete router functionality can be provided in the form of individual components (for example, a routing component, a filtering component, etc.), which are then composed into network services at runtime. Composition is achieved through packet classification. Packet filters define the processing "route" through the component environment. This allows routes appropriately tailored to the packet type or content.

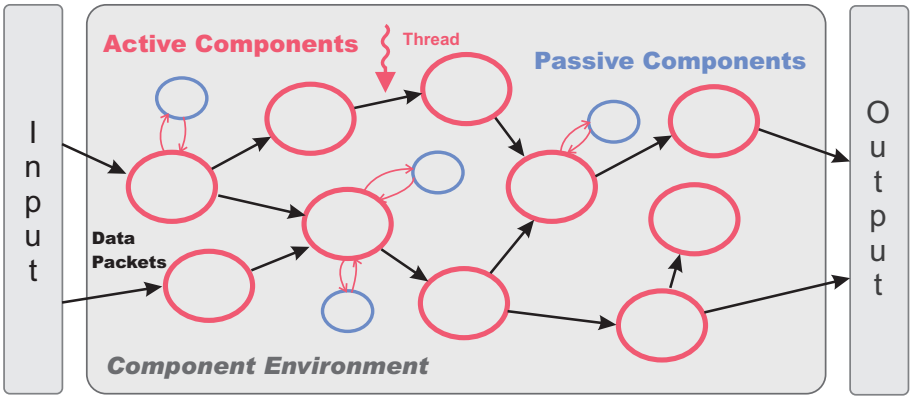


Figure 5.2: A Conceptual View of the Active Component Space

Finally, the extra flexibility obtained by the dynamic composition framework also facilitates the development of customised and adaptive network services that better meet the requirements of end-user applications.

5.5 Node Architecture

This section outlines the LARA++ active router architecture and illustrates how it can be built on a real router platform. The subsequent sections in turn describe the individual aspects of the architecture.

A key difficulty in designing active network systems is to allow network nodes to process user-defined and flexibly specified programs while providing reliable network services for everyone. Active nodes must therefore protect co-existing network protocols and services from each other and securely control shared resources. Consequently, LARA++ applies the layered active node architecture specified by the DARPA active networking group (see section 2.5). However, in order to provide safety for user-defined active code, LARA++ extends this architecture with safe execution environments. Figure 5.3 illustrates the building blocks of this layered architecture:

The *Active NodeOS* provides low-level system service routines and policing support to enable controlled access to node-local resources and system services (for example, device configurations). It serves as a platform abstraction, thus allowing the LARA++ architecture to be implemented on many existing configurations of hardware and software.

Three further abstractions model the conceptual architecture above the system level of the LARA++ platform. *Policy Domains (PDs)* form the management units for resource access and security policies which are enforced on every active program executed within the PD. *Processing Environments (PEs)* provide the protected environments for the safe execution of active code. Active programs with mutual trust relationships are

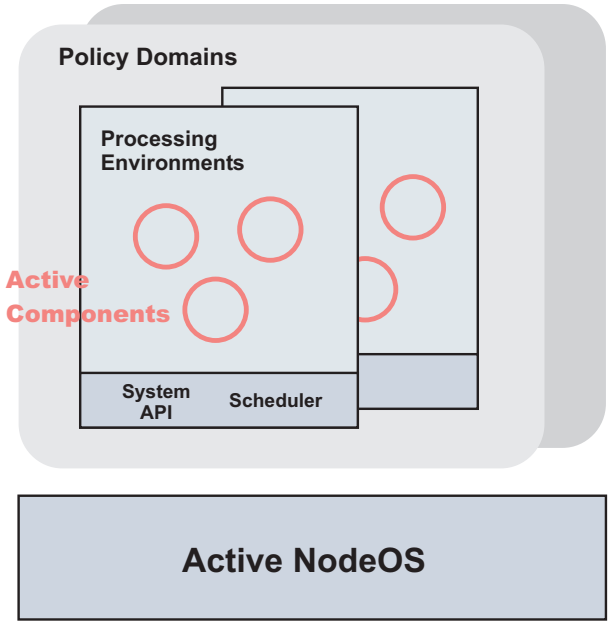


Figure 5.3: The layered active node architecture of LARA++

executed within the same PE. *Active Components (ACs)* comprise the actual active programs. They are the units of active code processed within the PEs. Active components are executed by means of one or more LARA++ threads.

LARA++ introduces the concept of processing environments as an extra protection layer between the active programs and the NodeOS. The idea behind the PEs is to provide a safe execution environment for active programs of arbitrary (potentially even malicious) source. Apart from language based safety measures, which would restrict the system to a specific programming language, this extra level of safety can be achieved through a number of other mechanisms previously introduced (see section 2.8.1). In particular, we suggest the use of operating system-based safety measures (see section 2.8.1.1) as they are highly reliable and provide maximum flexibility.

The LARA++ architecture is designed to extend existing routers by layering active network-specific functionality on top of the router operating systems. A generic high-level active network layer enables cross-platform programming and processing of active programs. To unify access and programmability of the active routers, well-known interfaces (for example, the system API or PE interface) are exposed by this layer. Low-level functionality of the active router architecture as provided by the active NodeOS is directly integrated with the router OS in order to maintain good performance for the active processing.

The architecture is generic in the sense that it can be implemented on any software-based router platform. To assure platform independence, there are currently two prototype implementations for both MS Windows 2000 and Linux being developed.

5.5.1 Components

Components are the program units developed and distributed within LARA++. They are dynamically loadable onto LARA++ active routers in order to extend the functionality of a router or provide a new service. Components can either provide a self-contained service or simply add functionality to a service composite.

As illustrated in Figure 5.2, LARA++ components can be either *active* or *passive*. Active components perform the active computations on a node based on one or more execution threads, whereas passive components provide merely static functionality similar to support libraries. Components can be further differentiated by their function into *user components* and *system components*. User components are those active programs or libraries that are injected by users of the active network. System components, by contrast, are the components that constitute LARA++ (for example, the packet classifier, the active thread scheduler, the programming API).

User components within LARA++ have a minimal well-known interface similar to the **IUnknown** interface known from the Microsoft COM component model [COM]. It enables other components to query what other interfaces are exposed by the component. Figure 5.4 illustrates an example user component with several interfaces.

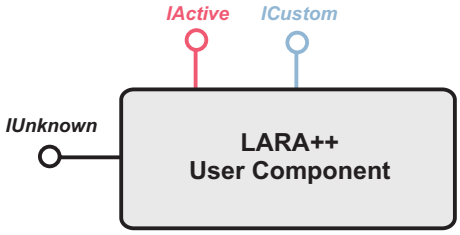


Figure 5.4: LARA++ Component Interfaces

Active components can be distinguished from passive components based on their **IActive** interface, which is used to initialise (**IActive.Initialize**) and activate a component (**IActive.Main**).

LARA++ components are built like normal shared or dynamic link libraries. Specific functionality (for example, the LARA++ system API or active thread scheduler) is dynamically linked to the components at compile time like normal system libraries. At installation time, the component loader (CL) extracts the active component code from the link library and loads it directly into a processing environment.

Finally, LARA++ components are distributed in the form of pre-compiled machine code or as source code. In the latter case, the component code is just-in-time (JIT) compiled the first time the the component is loaded (see also section 5.3).

5.5.2 Processing Environments

The LARA++ processing environments provide a safe and yet efficient execution environment for groups of active components maintaining a mutual *trust relationship*. LARA++ defines trust relationships in terms of several criteria, but initially limits them to user and code identities. Thus, depending on the code producer of a component and the user who installs the component, the component might be regarded as trustworthy for execution within a particular PE or not.

Although the LARA++ PEs share a very similar function as the execution environments defined by the DARPA active networking group [ANW98b], they are rather a specialisation of the EEs with useful additional properties:

First, the PEs provide a *safe* code execution environment to protect the active node from arbitrary active code based on the principles of software fault isolation (see section 2.8.1.1 for further information), resource management, and system call control. The "visibility" boundaries defined by the PEs prevent malicious active code from breaking out of the protection environment. As a consequence, malicious active code can only disrupt the operation of the PE and the ACs executed within, but not beyond. This is the reason why ACs executed within the same PE must have a mutual trust relationship.

Second, the PEs are designed for *efficient* active code execution of trusting ACs. A light-weight thread scheduler as part of the PEs enables very efficient scheduling of AC threads within the same PE. The high-level scheduling mechanism exploits the advantages of trust relationships, namely that trusted active components can be executed within the same protection environment without creating a security threat, in order to minimise scheduling latency.

Protection based on the principles of software fault isolation implies that the system API, or in other words the gate to low-level system routines and services, must be provided as part of the protection environment. Therefore, the LARA++ API plays an important part in the design of the PEs.

LARA++ System API

The LARA++ system API provides the programming interface for active and passive components to program the active nodes. It has been designed with the following two general design objectives in mind:

1. The main role of the API is to support packet forwarding, as opposed to arbitrary computations. Therefore, the interface provides special functions to support data flow processing (i.e., packet processing, accounting and admission control for resource usage, routing, etc.).

2. System calls that are not particularly unique to active networks are borrowed from established interfaces specifications, such as POSIX.

As a result, the LARA++ system API encompasses active network specific system calls in addition to standard system calls. The active network specific interface includes routines, for example, to load, install and configure components, to register packet filters with the classifier, and to access network packets and node resources. The standard system interface must also be provided through the LARA++ system API (as opposed to directly through the underlying operating system) in order to enable control through the active NodeOS. For example, the standard system call for memory allocation: `malloc()` must be checked by the policing component prior to execution to ensure that active components do not exceed their memory resource quotas.

It is thus the responsibility of the processing environments to expose the LARA++ API within the protection domain of the PEs and to force components to use it. This may imply to explicitly disable the use of default system interfaces and to provide equivalent wrapper functions within the PEs.

From this, it can be concluded that the LARA++ programming interface does not impose any restrictions on programmability. Safety and security is entirely a policing issue. Finally, it is noteworthy that the specialisation of the execution environments into PEs also accounts for the application of fine-grained *module thinning*[3] techniques. Each PE can provide a specially customised system API depending on the trust relations between the active components being processed and the active NodeOS.

5.5.3 Policy Domains

The Policy Domains (PDs) proposed by the LARA++ architecture define the legitimate scope for security and resource access policies for active computation. This purely logical layer of the LARA++ architecture forms the management unit for node-local policies. This implies, for example, that every PD maintains its own policy rule base. All active components in the scope of a PD are policed according to the same policy rule base – independent of their PE.

The idea behind the PDs is to offer different classes of service for active programs on an active router. Depending on the PD of an active component, the component may have a different level of resource access (for example, more memory, more CPU cycles) and security restrictions (for example, packets can only be read, not modified or dropped).

The choice of the PD is taken at loading time of an active component. Based on node-local loading rules, or an explicit selector within the component configuration, the

[3]Module thinning secures the programmable system by individually tailoring the programming interface exposed to a software process based on the privileges of the user or program (see also section 2.8.2.3).

component loader assigns the active component to an appropriate PD. A default PD is provided for the case that an active component does not specify a particular PD or qualify for a high-privilege PD.

Although the PD provides the scope for policy management, policy enforcement is carried out by the underlying active NodeOS, since policy enforcement is uniform across all PDs. Depending on the active component that is policed at a given point in time, the NodeOS switches to the policy databases of the corresponding PD. As a consequence, the PDs are purely logical entities that do not involve any processing.

5.5.4 Active NodeOS

The LARA++ active NodeOS provides access to low-level system services (for example, device configurations) and node-local resources. It is therefore responsible for controlling access for user components based on the component's privilege level and the system policies defined by the policy domain.

The LARA++ NodeOS layers active network specific functionality on top of the router operating system. Although it is designed as an extra layer on top of the router OS to achieve platform independence, the architecture is tightly integrated with the router OS for performance reasons.

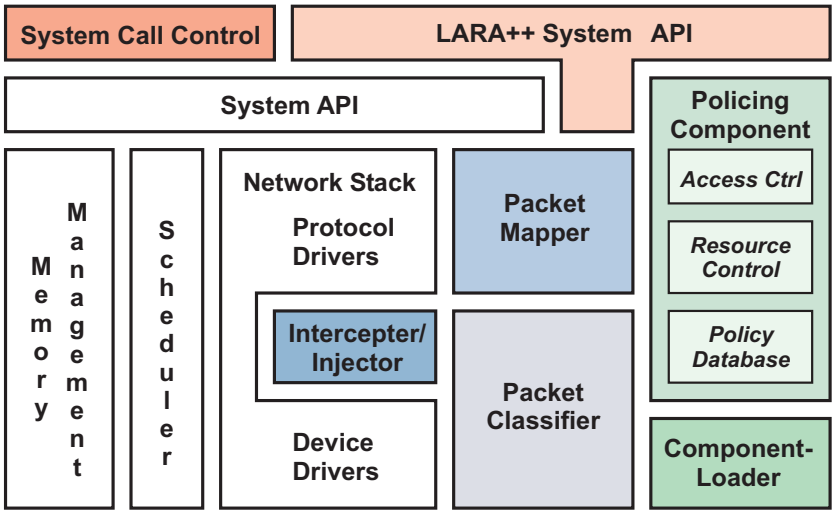


Figure 5.5: Illustration of the LARA++ Active NodeOS. Active network specific functionality is layered on top of the router OS such that standard operating system functionality (i.e., memory management, scheduling, etc.) can be easily integrated and reused by the active NodeOS.

As illustrated in Figure 5.5, core components of the LARA++ active NodeOS are the packet interceptor/injector, the packet classifier, the packet mapper, the system interface, the system call control, the policing component and the component loader.

The *packet interceptor* and *injector* are directly integrated with the network stack of

the router OS. Depending on the active network services to be provided and the actual implementation, data packets or frames can be intercepted (and injected) either on the link-layer level (for example, as Ethernet frames) or on the network-layer level (as IP packets).

The *packet classifier* determines whether or not packets passing the node need to be processed by any of the active components. It finds out which components need to process the packet and controls the order of component processing. A multiple stage classification mechanism allows flexible ordering of packet filters and hence active components (see section 5.6.2 for further details).

The *packet mapper* provides an efficient means to pass data packets between active components. The mechanism was included to speed-up the packet handling on the node. It is vital for the overall performance of the node when the protection scheme prohibits memory access outside the processing environment boundaries (for example, in the case of virtual-memory based memory protection). In this circumstance, memory mapping enables lightweight access to packet data across protection boundaries through avoidance of heavyweight copy operations (see Figure 6.2 for further details).

The LARA++ *system interface* provides access to the standard system interface for normal system calls and active network specific system services. While standard system calls are merely forwarded to the system interface of the underlying OS, LARA++ specific system services are directly implemented by the NodeOS component of the system interface.

The *system call control* ensures node safety by restricting active components to the interface exposed by the LARA++ API. It safeguards the system interface by checking each direct call to the router OS for its origin. Only system calls that have been invoked through the LARA++ system API (or non-LARA++ processes) are allowed. As illustrated later in Figure 6.3, malicious system calls that try to circumvent the LARA++ system API (i.e., through direct calls to the system interface or bogus software interrupts) are strictly blocked.

The *policing component* is responsible for ensuring that active components conform to the safety and security policies defined by the policy domain. The policing component encompasses an access and resource control unit besides the policy database. While the access control unit enforces the access policies by checking every system call for its authorisation (based on the calling component and the corresponding policy domain), the resource control unit manages the accounting of resources usage by the individual components and performs admission control. See section 5.8 and 6.3.5 for further details.

The *component loader* is responsible for the dynamic fetching of missing active code and the loading and installation of the active components. Since LARA++ supports active code distribution 'by reference', which implies that active code packets merely

include a reference for an active component rather than the code itself, the component loader may first have to download the component code before it can load it. A component reference[4] uniquely identifies a component (or its description file) through specification of the location where it can be retrieved. As a result, when the router cannot find the code for an active component in its local cache, it can retrieve the component either from the source server using the HTTP standard (or any intermediate component cache such as a web cache), or from the 'last hop' active router by means of the dynamic code loading mechanism proposed by Wetherall et al. [WGT98]. While the former approach is better suited for directed component distribution (i.e., only to specific active nodes), the latter is more efficient for general component distribution (for example, along a whole transmission path).

Once the active code is downloaded, the component loader must authenticate, instantiate and initialise the component. It is the loader's responsibility to select a "trusting" processing environment for the execution of the active component. A processing environment is considered trusting if all the instantiated components have a mutual trust relation with the new component. Trust relations among LARA++ components are defined by node-local trust policies. They define which sets of components trust each other based on the identities of the code producer and the user loading the component.

5.6 Service Composition

One of the key contributions of the LARA++ architecture is the novel service composition framework for active services. It plays a central role in the overall architecture, as it provides the foundation for the flexible and dynamic extensibility within LARA++. The remainder of this section describes the concepts and operation of this framework in full detail.

5.6.1 Operational Overview

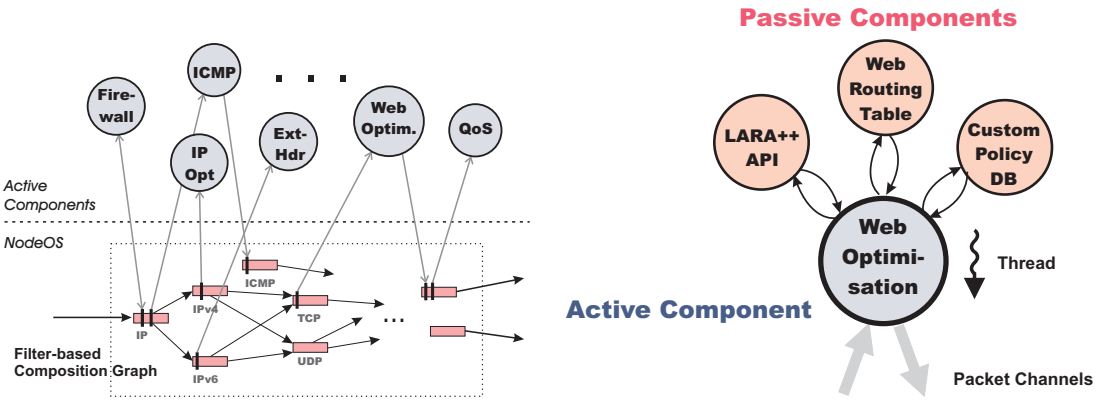
Service composition on LARA++ nodes is carried out on two levels – the *service level* (macro-level) and *component level* (micro-level). Different composition models are applied on the different levels:

Macro-level composition: On the service level, a comprehensive filter-based composition model allows active components to dynamically integrate themselves into node-local service composites. By inserting packet filters into the classification graph, active components define the traffic pattern upon which the component should be

[4] The universal resource identifier (URI) [BLFIM98] format is used to reference LARA++ components (for example, lara++://compserver.lancs.ac.uk/lcc.dll).

invoked and the order of execution relative to other components. Figure 5.6(a) illustrates the use of this composition model.

Micro-level composition: On the component level, a lightweight, plugin-like composition model enables the composition of active components from passive components. The fact that active components can be largely composed from functionality provided by passive components greatly facilitates active component design. Figure 5.6(b) demonstrates how active components can glue together passive components to provide a new service component.



(a) Macro-level composition – Composition of active services

(b) Micro-level composition – Composition of active components

A closer look at these composition models shows that both have their strengths and weaknesses. While the former is highly flexible and dynamic (i.e., packet filters can be inserted at run-time), the latter is more efficient (i.e., no context switches are involved for the invocation of passive components). As a consequence, the application of either composition model is a trade-off between flexibility and performance. It is still unclear to what extent those diverse approaches should be applied, or in other words, how much composition should take place at the micro-level or macro-level. The challenge is to find the right balance between component-level and service-level composition, such that the overall service provides sufficient flexibility and performance.

Since the micro-level composition model is very well-studied and widely used for service composition in conventional component systems, the remainder of this section focuses on the macro-level composition model. The latter is also of particular interest as it is specifically tailored towards service composition of active network services.

Macro-level service composition within LARA++ is largely packet driven. Dependent on the packet content, a different overall service may be composed for the processing of that packet. The *packet classifier* plays therefore a central role in the service composi-

tion process. It determines based on the set of *packet filters* currently installed whether or not a packet passing through the active router requires active processing, which active component(s) are involved, and in which order they should process the packet. The *classification graph*, which is managed by the packet classifier, maintains the key data structures for the composition framework. It organises the packet filters of the active components according to their computational function, and thus, provides the basis for the classification process. The following sections describe each of these elements in more detail.

5.6.2 Packet Classifier

The packet classifier defines the "route" through the active component space for packets passing a node. The classifier filters incoming (and outgoing) network packets based on the component filters installed in the classification graph. Figure 5.6 presents an example classification graph that is used here to illustrate the operation of the packet classifier.

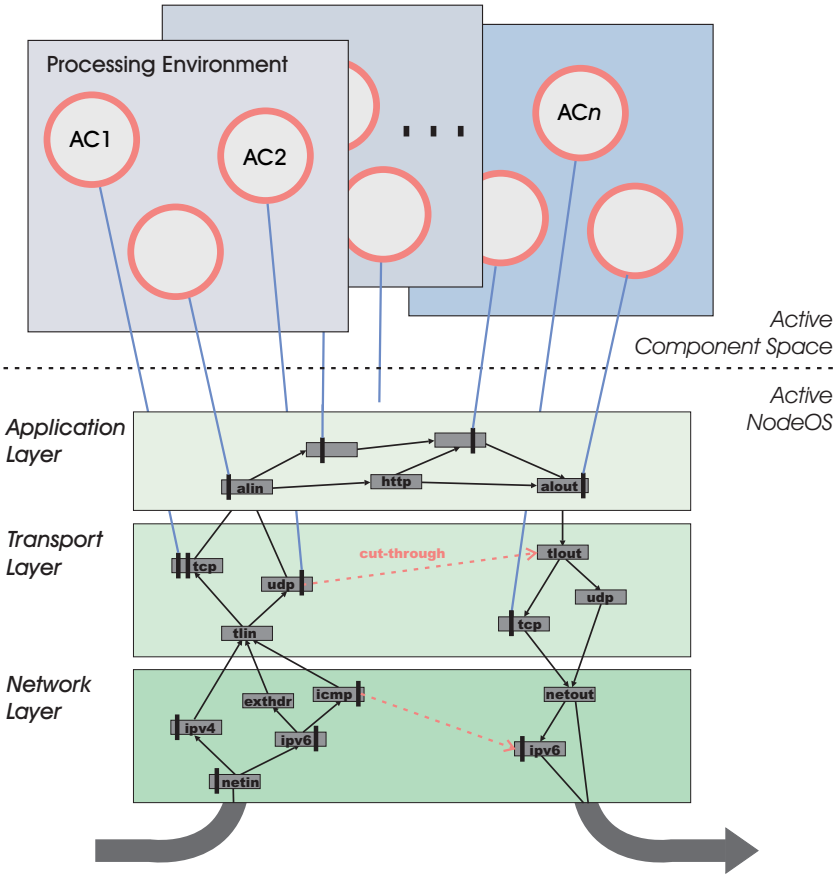


Figure 5.6: A Simplistic Classification Graph

The classification mechanism traverses the classification graph starting at the root

node (i.e., **/netin**). On every node in the graph, the classifier tries to match the packet filters installed by the active components at this point in the graph. Packets that match any of those filters are passed to the corresponding active component. After completion of the active processing, the classifier continues the multi-stage classification mechanism at the same point (or an optionally specified point defined by the packet filter[5]) in the classification graph. When the classifier has applied all packet filters that have been installed by the active components at a node, it follows the classification graph based on the "default" or graph filters (for example, **/netin/ipv4** or **/netin/ipv6**) and continues the classification process there. The classification process finally terminates when the classifier runs out of filters branches. That is typically when the packet is sent off to the next hop.

An examination of the classification process shows the similarities between the general concept of "packet routing" in data networking and the way service composition is achieved within LARA++. The problem of service composition within packet driven systems (such as routers) can be largely reduced to the general problem of "routing". The question simply becomes how to route a packet through the "network" of components. Hence, similar mechanisms to those found in normal routed networks can be applied to solve the composition problem within LARA++. For example, the graph filters at each node in the classification graph are used like the 'routing table' in packet switched networks. These filters define how to forward packets to the next nodes in the component network.

A reflection on the classifier's role for the packet processing on LARA++ nodes emphasises its importance for the overall architecture. The classifier is therefore a core system component of the active NodeOS. The fact that the packet classifier requires low-level access permissions (for example, to pass packets to active components and to awake active component threads) mandates a system-level implementation of the component. Furthermore, since service composition within LARA++ relies on extensive packet classification, the integration of the classifier into the low-level active NodeOS has also a positive impact on the system performance.

5.6.3 Packet Filters

The packet classifier distinguishes two types of packet filter: active component filters and graph filters. Figure 5.7 illustrates how these filters are used within the classification graph.

Active component type filters are used by active components to define the network packets of their interest. These filters are registered with the packet classifier at instan-

[5] Availability of this option depends on the user privileges and filter type (i.e., protocol or flow specific).

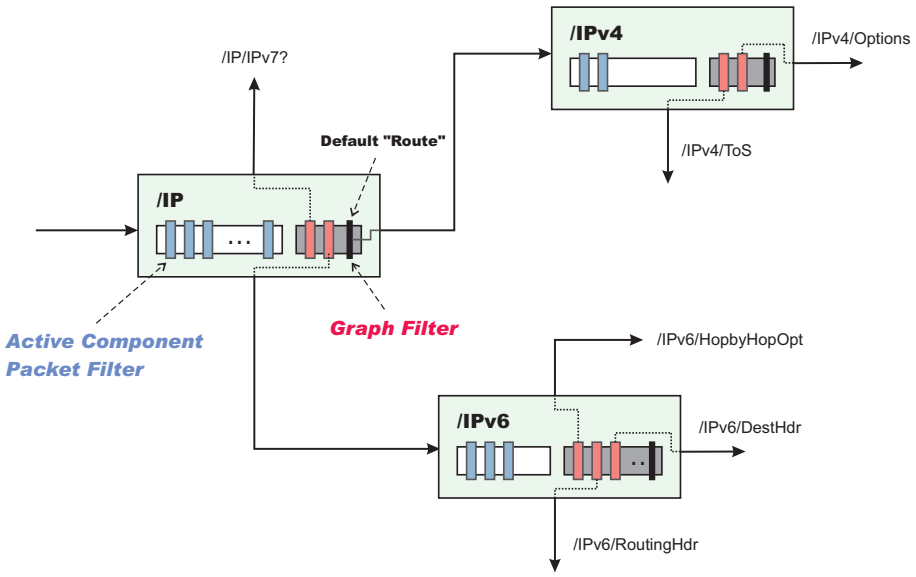


Figure 5.7: The classifier manages packet filters within a filter graph structure, called classification graph. Active component filters are used to dispatch network data to ACs for processing, whereas graph filters are used to define the structure of the graph.

tiation time of the components and at run-time if necessary. The classifier uses these filters to determine the active components to which network traffic is sent for active processing. The graph filters, in comparison, are used by the classifier in order to define the structure of the filter graph. These filters are used to define the branches in the classification graph. Appendix A describes the common properties of both filter types.

For reasons of scalability, active component filters can be further subdivided into *general filters* and *flow filters*. Given that a component may install a significant number of filters (for example, to identify individual end-to-end flows), it would be very costly to check each of these packet filters on a per-packet basis. Flow filters have been introduced to overcome this scalability problem. Like general filters, they can filter on any arbitrary bits (fields) in the packet, but are restricted to packets of specific end-to-end flows [6]. The advantage is that they can be hashed based on their end-point properties, which permits instant filter lookups even in the case of large numbers of flow filters.

To enforce restriction to a specific end-to-end flow, the classifier adds the obligatory filter patterns to the flow filter at installation time. Although this may seem very restrictive at a first glance, it is an extremely useful "limitation". It enables secure active processing within routers even for unauthenticated end-users. Limiting the data stream to a user's own end-to-end flow(s) reduces the security problem of active processing to the problem of providing a safe and secure processing environment.

[6] An end-to-end flow is defined by the source, destination, or both end-points. An end-point may be identified by network fields such as the network layer addresses and transport layer ports, or any other flow labelling techniques (e.g., the flow ID in IPv6).

5.6.4 Classification Graph

As previously illustrated in Figures 5.6 and 5.7, the classification graph provides a flexible structure for packet filters to be installed and structured on an active router. Each node in the classification graph consists of a set of active component filters and graph filters. While the former are used to "plug in" ACs into the packet processing chain (through insertion of the filters at an appropriate location in the graph), the latter are used to define the structure of the graph itself. Accordingly, the structure of the classification graph can be dynamically altered simply by inserting or removing graph filters, which provides a high degree of flexibility.

This extensible graph structure offers great flexibility for the composition of active services (i.e., new nodes in the classification graph can be easily introduced as new network protocols become available). However, the basic structure of the graph must be known to the active component developers and/or user to define where in the classification graph a component should be integrated. For example, an AC that wants to process HTTP requests should be executed subsequent to active components providing network and transport level services. Thus, in this example, the best location within the packet processing chain would probably be: **/netin.(ipv4|ipv6).tcp.http**.

5.6.5 Classification Graph Table

The classification graph table (CGT) provides the means to describe the structure of the classification graph. Its main purpose is to make the graph structure globally available across a LARA++ active network. In order to support flexible extension of the classification graph (and hence the packet processing chain on the active routers), an "elastic" means to describe the graph structure is required. For this purpose, a simple notion to defines the nodes (for example, **ipv4**, **tcp**, and **udp**) and the branches of the graph (for example, **ipv4** → **tcp** or **ipv4** → **udp**) that provides sufficient flexibility to incorporate new protocols and extend current protocol stacks has been introduced.

The basic structure of the classification graph described in the CGT conforms to the TCP/IP layer model [Ste94], which ensures that active components providing low-level services are processed before components dealing with higher-level computations. For example, network protocol options must be processed prior to transport protocol operations (see Figure 5.6). The fine-grain structure accounts for the layer-specific protocols. A specific example is that the extension headers in IPv6 must be processed in a well-defined order.

As part of the classification node properties, the CGT specifies the types of filter (i.e., general filters, flow filters) that may be installed inside the classification nodes. For example, it would make no sense to allow general-purpose filters to be processed before

filters pertaining to a firewall component, or otherwise the node would be vulnerable to a denial-of-service attack. Classification nodes also maintain permissions for fine-grained control of data access at the node. Currently defined permissions are **no** (no access), **ro** (read-only), **wo**, (write-only), and **rw** (read-write) access permissions.

Since the CGT is expected to change occasionally (for example, a new node in the classification graph might be introduced when a new protocol becomes established), an automated mechanism to update the CGT across the active network is desired. The fact that the CGT defines the structure of the component graph (which provides the basis for the packet "routing" inside the active node) in much the same way as the IP routing table defines the network topology, suggests that a routing protocol like mechanism is suitable.

At a first glance, it may seem that the overhead of updating the CGT every time a new protocol is introduced is heavyweight and makes the system inflexible. However, it should be noted that a CGT update is only required if a new protocol or protocol extension is "standardised" (i.e., globally announced such that it can be extended by others). The CGT does not require a global update in order to deploy and test the new protocol or extension at first.

5.6.6 Characteristics

Service composition within LARA++ is a co-operative process based on dynamic and conditional component bindings. It is a *co-operative* process as LARA++ nodes allow independent network users (controlled by the local security policies) to install active components that match the same data streams or subsets of streams. The classification graph provides the means for independent users to integrate new active functionality or services in a "meaningful" way without having to know or worry about the component interfaces of ACs installed by other users.

The fact that active services are composed through insertion or removal of packet filters (when components are instantiated or removed) at run-time makes the component bindings highly *dynamic*. The decoupling of component bindings among ACs through the filter concept and the classification graph hides changes from all but the component directly involved. Neighbouring components in the classification graph are not disrupted when another AC is removed.

Since the service composite depends on the data in the packets, the bindings are called *conditional*. A binding is only in force when the AC filter for the binding matches a data packet. Service composition within LARA++ is therefore a process that takes place on a per-packet basis.

The service composition approach proposed by LARA++ can also be described ac-

cording to the general characteristics of composition frameworks as introduced in the active network working group draft on composable services [ANW98a]:

Sequence control is based on the dynamic structure provided by the classification graph. The classifier determines based on this structure which active components need to process a packet passing the node and in what order. Both sequential and concurrent executions of active components are supported.

Shared data control is implemented by the packet classifier. It passes network data and associated state between those ACs that indicate interest in processing the packet.

Binding time is at run-time. Since the actual binding of ACs depend on the packet data (or in other words whether or not a packet filter matches the packet), composition takes place at packet processing time on a per-packet basis.

Invocation method is based on the arrival of active code. Upon receipt of active code (i.e., a packet including an active component), the component loader invokes the component bootstrapper of a "trusting" processing environment[7] in order to instantiates the active component. The component in turn plugs itself into the service composite by inserting one or more packet filters.

5.6.7 An Example

This section demonstrates how the LARA++ service composition process and in particular the packet classifier operates based on the example scenario introduced in section 5.2.1.

The local congestion control mechanism introduced by the network administrator requires access to all IP packets labelled with the CoS tag in the IP options field. The congestion control component would therefore define a packet filter that matches the respective tag and insert it into the classification graph. Since the congestion control mechanism drops "low priority" packets in the case of congestion, it is best executed early on the incoming path (before much processing is done). A suitable point to insert the packet filter in the classification graph would thus be **/netin.ipv4.options** or **/netin.ipv6.exthdrs**.

The handoff optimisation component, by comparison, redirects packets sent to a mobile node's previous location to its new location (after a network handoff). This end-user initiated service requires therefore access to all IPv6 traffic destined to the mobile node's previous care-of-address. Hence, a suitable classification graph node to insert the packet filter for this component would be **/netin.ipv6**.

These exemplary applications illustrate how independent active components (devel-

[7] Note that if none of the existing PEs consider the new component trustworthy, the component loader creates a new PE for this component

oped by different people or companies and installed by different users) can co-exist, and yet be part of the same active service composite – even without knowledge of one another.

5.7 Safety and Security

The LARA++ safety and security architecture is based on a combination of known mechanisms. While safe processing environments protect the system from malicious or erroneous active code, user and code authentication and access control mechanism enforce security.

5.7.1 Goals

The following high-level goals have led the design of the LARA++ safety and security architecture:

1. Safety and security mechanisms should incur minimal overheads during normal operations.

2. Overall system safety and security should be maximised by reducing the trusted component base (TCB).

The former is achieved by migrating expensive operations to infrequently called procedures such as the initialisation phase. For example, heavyweight public-key authentication of users and active code is carried out only once during the instantiation of an AC, while a lightweight authentication mechanism is used internally for frequent system calls.

The latter is accomplished by reducing the trusted component base to the active node implementation. However, since user and component authentication is based on public key encryption, LARA++ also needs to trust a public key server (as the only remote entity to be trusted).

5.7.2 Safe Execution of Active Code

The LARA++ architecture provides safety for the execution of arbitrary active code based on the processing environments introduced in section 5.5.2. However, the architecture does not define how the processing environments provide safety; it merely requires the processing environments to provide “process-like” protection. The PEs must therefore ensure that active code being executed cannot harm the low-level active NodeOS (or even the system OS) or any other PE (or ACs executed by other PEs).

As a consequence, the LARA++ processing environments must at least include protection mechanisms for memory and computational resources.

5.7.2.1 Memory Protection

LARA++ protects active processes from each other through a technique commonly referred to as software fault isolation (see also section 2.8.1.1). The "visibility" boundaries defined by the PEs prevent malicious active code from accessing memory outside its protection environment. This can be achieved through a number of mechanisms ranging from language based techniques (for example, strongly typed languages without pointers) to virtual memory management (as used by conventional operating systems).

The use of language based protection is typically easy to realise (as the problem is offloaded to the language designer) and fairly lightweight (no heavyweight[8] context switches are required), but at the cost of flexibility. System based protection mechanisms such as virtual memory based approaches, by contrast, are language independent and allow the use of typical system languages such as C, which support pointer arithmetic, user controlled memory allocation and omit run-time checking of types and ranges.

Since context switches between protection domains can be relatively heavyweight (for example, in the case of virtual memory based protection), LARA++ exploits the benefits of trust relations among active components to minimise the number of these context switches. For example, components with a mutual trust relationship can be executed within the same protection domain.

5.7.2.2 Pre-emptive Thread Scheduling

Besides the protection of memory resources, LARA++ must also protect the processing resources from fraudulent or erroneous active programs. LARA++ must prevent active programs from consuming more than their share of the processing resources (or even fully 'locking' a processor). As a consequence, LARA++ requires some form of pre-emptive scheduling mechanism that allows the low-level system to interrupt active programs that exceed their scheduling quantum.

The LARA++ architecture proposes the use of two thread schedulers – a system thread scheduler to warrant safety, and a component thread scheduler for performance reasons:

The **system thread scheduler** is needed to protect the processing resources on the active node. Since LARA++ requires separate system threads (at least one for each

[8] For example, a context switch between virtual address spaces typically involves a system call (i.e., two context switches between kernel and user space) and requires the invalidation of cached memory pages.

PE) to share the processing resources among the PEs and other systems tasks, a system thread scheduler must be provided by the underlying OS or the active NodeOS.

The **component thread scheduler**, by comparison, has been introduced to the processing environments for efficiency reasons (see also section 6.5.3). Since thread scheduling of trusted components within the same protection domain does not involve heavyweight context switches, scheduling of component threads can be very fast (see section 7.4.1 for measurement results). Although the trusted components of a single PE could also share the processing resources through cooperative scheduling without weakening system safety, LARA++ suggests a pre-emptive algorithm for convenience and fairness reasons. Pre-emptive scheduling is fully transparent to the programmer and allows the scheduler to fully control resource scheduling as desired.

5.7.3 Security Mechanisms

The LARA++ security model is primarily based on user and code authentication, and authorisation of security critical operations. These operations encompass the installation of active components, registration of packet filters and access to low-level system services and resources (through the LARA++ API). The general principle of this security approach is: *"who developed the active code and who installed the component determines whether or not an operation can be carried out."*

The remainder of this section discusses how LARA++ uses the following security mechanisms: authentication, code signing and access control.

5.7.3.1 Authentication

LARA++ uses authentication techniques to identify network users and active code producers securely. Based on these identities LARA++ determines whether or not an operation can be authorised.

Authentication within LARA++ can be built on public key encryption mechanisms such as RSA or DSA (see section 2.8.2.1). The user installing an active component (or code producer developing an active component) encrypts its identity (i.e., username or company name) with its private key. Public key encryption ensures that the encrypted message can only be decrypted with the public key assigned to the user. Therefore, a LARA++ node that receives an authentication message (for example, as part of an active code packet) can securely verify a user's identity based on the user's public key.

The fact that authentication based on public key encryption relies on trusted key servers and secure key exchange mechanisms, obliges LARA++ to provide such a service across the active network or to enable the (re-)use of an existing public key infrastructure.

5.7.3.2 Code Signatures

LARA++ uses code signatures (1) to identify the code producer of a component and (2) to verify the code integrity.

Code signatures are created from the checksum (for example, CRC or message digest) of the active component code. The code producer encrypts the checksum with its private key and includes it into the component's configuration file. When a LARA++ node receives the active component for installation, it decrypts the checksum with the code producer's public key and verifies it against the self-computed checksum. If the checksums match, the integrity of the active code and the identity of the code producer are validated.

LARA++ can use standard algorithms, such as those based on CRC or a specific technique like MD5, to calculate the code checksum/digest. Encryption of the code checksum can be based on the same public-key encryption algorithms used for user authentication.

5.7.3.3 Access Control

Access control is the primary means through which LARA++ achieves security. The policy enforcement component (see also section 5.8.1) controls access to system services and resources based on the user and component identities. It blocks unauthorised active system calls and controls resource usage accordingly.
Access control is used to secure the following operations:

Active component execution – The installation of active components is firmly controlled by the component loader. Access control regarding active code execution is based on the identity of the network user installing the component and the code producer.

Network data access – The insertion of packet filters into the classifier is also subject to access control. Again, depending on the respective user and component identity, the classifier accepts or rejects packet filters. However, access control with respect to network access must also take into account the location of a filter within the classification graph (for example, **/netin** or **ipv6**), the desired access permissions (i.e., **ro, wo, rw**) and the filter type (i.e., general filter or flow filter).

System API access – Controlled access to the system API is again based on the user and component identity. However, since system calls, such as calls to send or receive packets, occur very frequently, LARA++ introduces a lightweight "authentication" mechanism for internal use. The mechanism uses a unique and (pseudo) random 'identifier' to efficiently identify and authorise active components (see also section 5.8.1). The identifier provides the means to efficiently lookup a component's access

privileges during a system call. Depending on the particular system call and the access privileges of the component at hand, access is either granted or denied.

Resource usage – Access control to system resources (for example, memory, CPU and bandwidth) further demands an admission control component, which accounts for resources as they are consumed. Nevertheless, since these resources are also accessed through the LARA++ system API at run-time, resource usage too can be controlled through the system API access control mechanism described above.

5.8 Policing

LARA++ uses policing as the main measure to achieve security on active nodes. In order to maximise flexibility for policing, LARA++ manages a separate policy database for each policy domain. Allowing active components to be associated with different policy domains, and hence with different system and resource access policies, based on the user downloading the component and/or the code producer, enables fine-grained control.

The remainder of this section describes the policy enforcement and specification mechanisms in further detail.

5.8.1 Policy Enforcement

Policy enforcement must strictly control all security threatening operations on the active router. Within LARA++ this implies that policy enforcement must control the installation of active components, the insertion of packet filters, and run-time access to system services and resources through the system API.

Since policing within LARA++ depends primarily on the user and/or code producer identity, secure authentication mechanisms for active components (as described in section 5.7.3) are vital. However, authentication based on public key encryption and code signatures is computationally expensive and therefore unsuitable for run-time policy enforcement on a per-packet basis.

As a result, LARA++ introduces a lightweight "authentication" mechanism for internal use. The mechanism uses a unique and (pseudo) random active component identifier (ACID) that is generated during component installation. Since component installation is only performed once per component (as opposed to on a per-packet basis), full user authentication and code signature check is carried out before the ACID is created.

Along with the ACID, the active NodeOS creates an active component data structure (ACDS) which comprises all component specific, run-time critical information (for example, user id, user group, code producer, component group). In particular, the ACDS defines which system APIs can be accessed by an active component. The ACDS also

holds the information required for resource control, namely the current resource usage and the resource limits of an active component.

The ACID is used during a system call as a hash key to lookup a component's ACDS.

$$H(ACID) \rightarrow ACDS \tag{5.1}$$

The ACDS is needed to check whether or not the calling component has permission to access the respective system call. In the case of a system call that consumes any controlled resources, the ACDS is needed to check whether or not the component still has sufficient resources available to complete the respective operation. Consequently, any resource usage (i.e., allocation and release) must update the component's 'resource meters' in the ACDS.

In summary, the ACIDs and ACDSs provide the means for fine-grain access control for active components on a per system call basis. Furthermore, they enable highly efficient run-time policy control due to (a) the fast lookup of the components control structure, and (b) the optimised representation of the ACDS for fast policy verification.

Since the security of this approach is mainly based on the correctness of the ACIDs, they must be sufficiently long such that malicious components cannot easily guess the ACID of a high privileged system component. In order to defend against brute force attacks, the policy enforcement component immediately destroys active components using invalid ACIDs. In addition, a mechanism is required that prevents impostors from repeatedly instantiating such components.

Dynamic changes of run-time policies while the active node is operating demand special consideration. Since policy enforcement is not directly performed upon the policy rule base, but upon the optimised representation provided by the ACDS, the active NodeOS must re-compute the affected ACDSs subsequent to changes in the policy database.

5.8.2 Policy Specification

This section provides an overview of the different policy types that are supported by LARA++. Since a complete definition of all policy rules would be beyond the scope of this document, only a brief description of the policy specifications is provided.

Policy specification within LARA++ is based on the concept of *domains* and *groups*. While domains define the classes or categories of entities upon which policies are defined (i.e., users, components, resources), groups identify collections of entities that are grouped together for simplicity and scalability purposes (i.e., to aggregate common policies).

LARA++ policies make use of the following domains and groups:

User Domain: This domain defines the users of the active network for which policies are needed. Users can either be identified by their individual names or by their group name. User groups can be easily created and individual users can be flexibly assigned. The default set of user groups include: *system administrators*, *network administrators*, *privileged users*, *authenticated users*, and *others*. A special group, referred to as *all*, encompasses all users (known and unknown).

Code Producer Domain: This domain defines the code producers of active components (for example, Cisco or Microsoft). It enables policies that consider the producers of active code in addition to the users loading the code. Again, the use of groups, such as *trusted*, *authorised* or *authenticated producers*, enables aggregation of policies, which can greatly simplify policy specification. Active programs that have either none or an unknown code signature are assigned to the group of *unknown producers*.

Component Domain: This domain is defined by the properties of active components (for example, trusted or untrusted components) and the component types (for example, user or system components). Again, policy specification can be simplified through the use of component groups. The following default component groups are defined: *system* or *user components* and *trusted*, *privileged*, *authenticated* or *unknown components*.

System Call Domain: The system call domain defines different classes of system calls. For example, such classes could include system calls for memory allocation, persistent memory access (reading and/or writing), or routing table update. Yet again, the idea here is to group similar systems call in order to facilitate policy specification. The grouping of system calls enables aggregation of policies (i.e., policies do not have to explicitly address individual system calls).

Resource Domain: This domain encompasses all the resource types that can be accessed through the LARA++ API (for example, persistent memory, memory, bandwidth, processing). The resource domain also includes logical resources, such as the routing table, number of files, etc. Consequently, this domain enables the definition of policies that limit (or grant) active components a certain quantum of a resource. Depending on the resource type (whether or not it is scheduled), different scheduling policies may be provided. For example, network and processing resources can be scheduled according to *best-effort* mode (default) or based on *priorities* (see section B.3 for an example).

Policy specification within LARA++ is divided into three parts. Each part governs the policies for one of the following protection realms: (1) instantiation of active com-

ponents, (2) installation of packet filters, and (3) run-time control of active components. The various policy types along with examples on how to use them is further discussed in Appendix B.

5.9 Summary

This chapter has presented the design of LARA++, a novel, component-based active router architecture. The primary focus of this design is to provide a highly flexible and extensible active node architecture, suitable as a research platform for edge networks, that can form the basis for further research into active networks and networking in general. The motivation behind the new approach and the design overview of the novel architecture has been described in section 5.2 and section 5.4 respectively.

The work presented here has built on lessons learned from the design and development of Lancaster's first generation active router architecture, called LARA. The background work on LARA has been introduced in section 5.3. On the one hand, LARA++ greatly improves programmability of active nodes regarding flexibility, extensibility, usability, safety and security. On the other hand, the LARA++ software architecture has been designed bearing in mind the possibility of adopting the LARA scalable high-performance hardware architecture later.

In contrast to LARA, the LARA++ architecture emphasises mainly the software architecture of the active router. The key building blocks, namely the active NodeOS, the policy domains, the processing environments, and the active and passive components, have been presented in section 5.5.

The advances regarding flexibility and extensibility are primarily a result of the component-based programming model proposed by LARA++. The filter-based service composition framework described in section 5.6 enables flexible and co-operative programmability and extensibility of active node functionality at run-time. Usability has been improved due to the introduction of the user-space processing environments. For example, they facilitate component development because standard development environments and programming interfaces can be used. Furthermore, the introduction of the processing environments has a significant impact on the safety aspects of active programming. The software fault isolation techniques provided by the safe processing environments have been discussed in section 5.7. Finally, the security procedures have evolved to a highly flexible and configurable framework based on policing. Section 5.8 has presented the various policy types and rules, and the corresponding policy enforcement mechanisms.

The following chapter continues this thesis with an overview of the LARA++ prototype implementations. Due to the broad scope, the prototype implementations serve

mainly as proof-of-concept for key mechanisms and design decisions of the LARA++ architecture.

Chapter 6

Implementation

6.1 Overview

This chapter describes the ongoing efforts to engineer a prototypical realisation of the LARA++ active router architecture. The previous chapter has presented the design of this novel architecture. Due to the extent of the LARA++ architecture, the prototype implementations focus primarily on validating the key aspects of the architecture by implementation.

After a discussion on potential router platforms for the implementation of LARA++ in section 6.2, this chapter describes the implementation of the main components of the layered architecture. The details of the implementation are discussed layer by layer, starting with the active NodeOS in section 6.3. This is followed by the implementation of the policy domains (section 6.4) and the processing environments (section 6.5). Finally, section 6.6 discusses the development of LARA++ components along with several example components.

6.2 Router Platforms

As mentioned previously in section 5.5, the LARA++ architecture layers active network specific functionality on top of an existing router platform. The LARA++ active NodeOS is tightly integrated with the router OS in order to obtain full control of the system (for example, to change scheduling policies) and maximise performance (for example, for resources access, packet classification, data handling). As a result, the use of an open-source or source-available operating system is vital.

Since conventional routers are typically closed commercial systems, it is virtually impossible to get source-level access to the software. The source code of the router software is considered the manufacturer's secret asset and therefore well protected. On the one hand, router manufacturers fear that it could fall into the hands of their competitors,

which would give them cheap access to new developments. On the other hand, router manufacturers do not want third parties being able to extend the router software in order to secure the company's future business. As a consequence, these systems are not pragmatic for use as an underlying platform for LARA++.

Instead, router systems based on commodity operating systems have been considered as a starting point for the prototype implementation of LARA++. The integration of active network support with a commodity operating system is also advantageous as LARA++ can evolve in parallel and still take advantage of enhancements made to the standard operating system.

Consequently, the LARA++ prototype implementations outlined in this chapter are being built upon Linux and Microsoft's Windows 2000 (both of which support basic routing functionality as part of the core OS). Despite the fact that the latter is a commercial, non-open source OS, source code access has been gained through close collaboration with Microsoft Research in Cambridge as part of the LandMARC [Lan01] project.

Since the work presented here was primarily funded by Microsoft Research, the initial implementation work has been carried out on the Microsoft platform. However, as platform independence is important for active networking (which typically involves many network nodes) a second implementation under Linux was initiated recently. Although the implementation and integration of the LARA++ active NodeOS functionality into these operating systems is not the same due to the differences in the OSs, the principles and functionality remain the same.

In both cases, a modular approach has been taken for the design of the active NodeOS. While in Windows 2000 the NodeOS modules are implemented as a series of 'virtual device drivers', under Linux the kernel module support is used (see also section 3.3.1.6). The modular approach provides a relatively clean separation between the active NodeOS and the remainder of the kernel. However, more importantly, this decoupling also introduces an element of safety and enables dynamic extensibility. Both virtual device drivers and kernel modules can be dynamically loaded and removed without affecting the rest of the system. So, if trouble ensues, the malfunctioning LARA++ component can be removed and re-initialised without disrupting the entire system.

6.3 Active Node OS

The implementation of the LARA++ active NodeOS is integrated with the base operating system for reasons such as control and efficiency. A tight integration with the host operating system grants the NodeOS full control to implement system-level functionality such as packet handling, resource scheduling and policing, and enables high performance for the active processing.

This section describes the LARA++ active NodeOS modules that have been implemented so far for our Windows 2000 and/or Linux prototype systems.

6.3.1 Packet Interceptor / Injector

The packet interceptor and injector provide the interface between the LARA++ active network environment and the data path on the node. The packet interceptor is responsible for intercepting the network traffic traversing the node and passing it to the active network environment for processing. The packet injector, by contrast, re-injects the network data back into the default forwarding path on the node or sends it directly through one of the outgoing interfaces.

Under Windows 2000, the packet interceptor/injector is implemented as an intermediate network device driver [IDD]. The Windows network driver interface specification (NDIS) [NDI] enables the placement of so-called intermediate drivers between the lower-layer network interface card (NIC) driver and the upper-layer network protocol driver. Thus, all traffic that is received by the node is passed through the intermediate driver to the network layer protocol for the default processing (for example, routing) and vice versa.

Under Linux, the packet interceptor/injector is integrated with the classification mechanism. The packet classifier, which is based on an extension of the Linux netfilter module [NET], also intercepts link-layer frames and tries to match them against the active component filters.

Interception of traffic at the link-layer has the advantage of making the system network protocol independent. As a consequence, it maximises the flexibility of the active node by enabling extensibility of active router functionality at the network layer (for example, LARA++ components could provide the functionality for a hypothetical Internet protocol version 7).

In both implementations, the packet interceptor/injector can be loaded (or unloaded) dynamically at run-time (without interrupting the entire system). This allows LARA++ to dynamically activate (and deactivate) the active network functionality on a node. Note that removing the packet interceptor completely disables any LARA++ specific processing on the data path.

6.3.2 Packet Classifier

The packet classifier is a key component of the LARA++ architecture. It is responsible for the dispatch of classified packets to active components. Since the classification process is very much self-contained and does not directly interface with the router OS, a platform independent implementation has been developed.

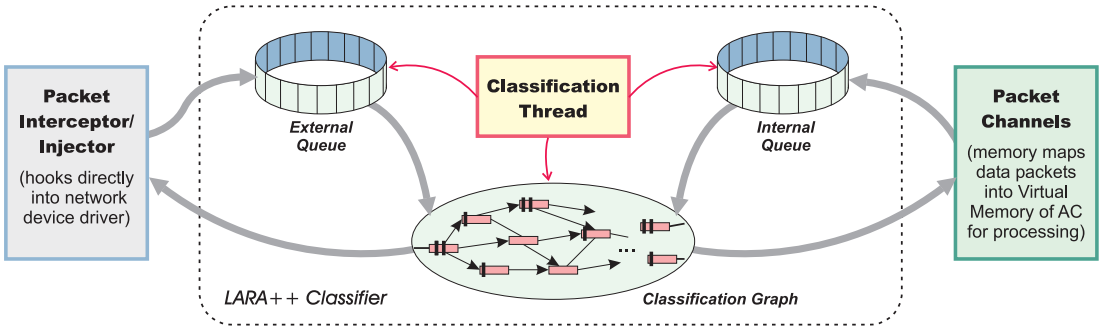


Figure 6.1: The LARA++ Classifier Architecture

Figure 6.1 illustrates the architecture of the LARA++ classifier. The classifier is the first and last element through which packets pass when being processed by a LARA++ router. Incoming packets, intercepted by the packet interceptor component, are asynchronously queued on a circular buffer known as the *external queue*. The classification thread sequentially takes packets from this queue and performs an initial classification.

After a packet has been classified (i.e., a component has been selected to perform active processing on the packet), the packet is immediately queued in the packet channel of the active component and the classification engine may continue to classify more packets until all packets are classified or until its scheduling quantum is over. This asynchronous approach allows the classifier to efficiently classify packets (avoiding the overhead of synchronisation between the classifier and active components). Once classified packets have been processed by the corresponding active components, they are returned to a second queue known as the *internal queue*.

Many components will often be identified to process a packet over the course of its passage through the active router. However, it is not possible to identify all components to which a packet will be sent in advance because components could change the content of the packet. Therefore, re-classification of packets between the processing of active components is crucial. Packets requiring reclassification (i.e., waiting on the internal queue) are separated from packets that are awaiting initial classification (i.e., waiting on the external queue) so that the classifier may discriminate, to the benefit of the performance of the node. By processing packets waiting on the internal queue in preference to those on the external queue, the classifier minimises the latency of the individual packets.

Packets removed from the external queue for classification are given a classification context that records the progression of the packet through the classification graph. This context remains with the packet for the duration of its passage through the active router. In order to facilitate the processing of a packet by flow filters, the flow keys for the packet are computed only once on arrival at the node and then stored in the packet context.

Packets taken from either of the two classification queues resume their classification at the same classification node from which the last classification had been made, starting with the subsequent filter. If the packet is undergoing its first classification, it begins at the first graph node and with the first filter.

Each packet removed from a classification queue is processed as follows until an active component filter has been matched or until the classifier passes the last filter in any given classification node. Filters are processed in the order: flow filters, general filters, graph filters and finally, the default graph. Flow filters are checked by looking up the flow in a hash table. Each classification node has its own hash table. The hash value calculated for each packet based upon the flow characteristics on arrival is used as the lookup key, and the table then yields a list of candidate flow filters. The list of flow filters must then be checked to find the correct flows. General filters and flow filters whose flow is identical to that of the packet are then checked against the packet. If any filter rule matches the packet, a successful classification is deemed to have been made.

Graph filters are processed last in each classification node. This ensures that all active components that have placed filters matching the packet in the classification node are processed. Graph filters contain matching rules similar to flow and general filters. If a graph filter is matched, classification terminates at the current classification node and resumes at the next node defined by the graph filter. If no matching graph filters can be found but the current classification node has a default graph, then classification continues at the start of the classification node indicated by the default graph. Otherwise, the end of the classification node is reached and the packet is deemed to have finished classification, and hence active processing, and is passed to the packet injector component for forwarding.

Filter Processing

The creation of a service composite for each packet is based on the packet filters. Section 5.6.3 introduced the notion of filter patterns. Each filter type (general, flow and graph filter) contains such a pattern as one of its attributes. While the expression of packet filters in this way is convenient and extremely flexible, it comes with an inherent overhead. The position of fields that might be identified by the filter pattern (for example, **TCP_HEADER**) can change from packet to packet. This means that the absolute offset must be recalculated for each packet. The impact of the operation can be somewhat lessened if the classifier maintains a list of packet characteristics (for example, protocol headers) that have been identified during the classification of the packet in its journey through the active router. These "features" can then be used in the offset calculation, rather than having to parse the packet to locate these features each time they are re-

quired. For example, the classifier could store the offset of the IPv6 header for each packet when the header is encountered so that subsequent filters can use it in offset calculations. Because of this approach, it is not a coincidence that most headers have one or more dedicated classification nodes in the classification graph; this is a property of the composition model. Therefore, the classifier can store an offset of a feature processed at a classification node upon arrival at that node.

In order to take advantage of this optimisation, graph filters are given an additional property known as the *focus translation*. The packet context contains a stack of foci. If a graph filter is matched or default graph is encountered, a new focus is pushed on top of the stack. The new focus increases/decreases the previous offset by the focus translation of the matched graph filter or default graph. For example, the focus translation of a graph filter branching between an IP header and a TCP header would be the size, in bytes, of the IP header. Therefore, a focus that previously pointed to the offset of the IP header would point to the start of the TCP header subsequent to the processing of the graph filter. Consequently, the problem of finding a packet feature for use in offset calculation is reduced to one of searching for the desired feature (identified by the classification node) on the stack of foci. The focus stack model was chosen because it is likely that most attempts to examine features of packets will be made closest to the focus of the packet in the current classification node. Since the most recent foci are placed at the top of the stack, searches for foci usually find a match within a few attempts.

Another potentially heavyweight task in filter processing is the computation involved in calculating the offset of packet fields. The fact that packet headers are not always of constant length (for example, IP options can cause the length of an IP header to vary), creates a need for flexible expressions in order to specify the focus translation. Given the frequency of the evaluation of offsets (nearly every filter pattern uses an offset) and the fact that packet filters do not change after filter installation, it is best to move the overhead of evaluating the semantics of the expression to the installation time of the filter. As a result, LARA++ uses a just-in-time compiler that translates the safe, machine-independent offset expressions into native machine code at the time of filter installation. Consequently, execution of the compiled expressions is very lightweight (only a few CPU cycles). This allows focus translations and packet offsets to be calculated efficiently and flexibly on a per-packet basis.

Note that classification algorithms based on the notion of data flows are well studied. It has therefore not been deemed necessary to take part in these efforts as part of this work. Recently published work in this field [WVTP97, NK98, SVSW98, SSV99] demonstrates the viability of high performance packet classification scaling to a large number of packet filters (in the order of millions).

6.3.3 Packet Channels

The active NodeOS establishes so-called packet channels to every active component being instantiated on the node. These channels provide the means to transfer network data (i.e., frames or packets) to and from the active components. Two unidirectional channels are employed – one for streaming data to the active component (*input channel*) and one to transfer the data from the components back to the active NodeOS (*output channel*).

Since the implementations of the LARA++ architecture described here are split across kernel and user space (see section 6.5 for an in-depth discussion)[1], standard communication mechanisms between kernel and user space components would involve heavyweight copy operations. More specifically, two copy operations would be required for every packet being passed to an active component in user-space – one for passing the data "up" into user-space and one to get them back "down" again. Clearly, a performance hit of such a scale is not acceptable for network devices such as routers (see also section 4.2.1.6).

In order to overcome this deficiency, LARA++ implements efficient, zero-copy packet channels based on the principles of memory mapping. The mechanism behind the packet channels relies on specific operating system support, namely it requires a means for the operating system to make a physical memory area (i.e., segment or page) visible within the protection domain of an active component (i.e., sandbox or process) and to withdraw it again. Fortunately, most current operating systems provide such a mechanism through the concept of virtual memory.

Based on this mechanism, the active NodeOS can "map" the memory area where a matched packet is stored into the address space of the active components, and "unmap" it after completion of the active processing (without the need for copying the data). The Windows 2000 implementation of LARA++, for example, directly maps the physical memory, where the packet data were stored by the network device driver, into the virtual address space of the active component's processing environment (see Figure 6.2).

For the management of the input and output channels, the active NodeOS establishes a shared memory segment with the active components during component initialisation. A separate packet queue is maintained for every channel. The queues are used to indicate new packets to the active components and vice versa. Since the data packets are in fact not copied, the queues pass only lightweight control structures of the packets up and down. The control structures tell the active component where the actual packet data are stored in memory. These internal structures include the packet length, the number of

[1]Note that the LARA++ architecture does not demand this split. This path was mainly chosen to achieve language independence for active programmability and to enable integration of (standard) operating system protection mechanisms.

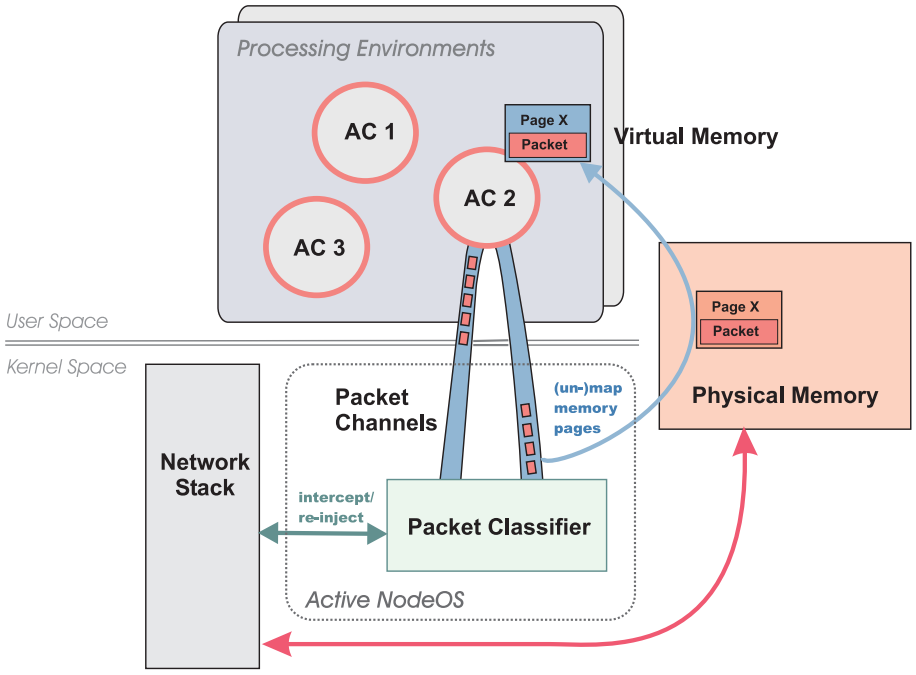


Figure 6.2: Zero-Copy Packet Channels – Packet memory is directly mapped into the virtual address space of the active component(s) processing the data.

packet buffers[2], an array containing the addresses of the mapped buffers and the length of each buffer, and the reference to the kernel-level control structure.

6.3.4 System Call Control

The system call control mechanism is vital to safety and security on LARA++ nodes. It is the means by which LARA++ enforces security policies on active components accessing the low-level system interface. As illustrated in Figure 6.3, malicious system calls trying to avoid the LARA++ system API (i.e., direct calls to the system API or bogus software interrupts) are strictly blocked.

The system call control must therefore intercept all system calls and check whether the call has been invoked by a LARA++ component. If so, it must handover the call to the policing component to verify the authorisation of the calling component. Otherwise, if the system call did not originate from a LARA++ component, it is simply passed on.

To enforce that no unauthorised system calls can take place, the system trap module (which implements the gate via which system calls enter kernel-space) must be altered. A system call can be directly triggered by any user application simply by initiating a software interrupt (i.e., `int 2Eh`). The same mechanism is used by the user-space stub of the system API (for example, kernel32.dll) to switch context into kernel-space.

[2]Within most current operating systems, network packets are typically managed as a list of packet buffers (for example, *mbufs* in Linux or *NDIS buffers* in Windows 2000).

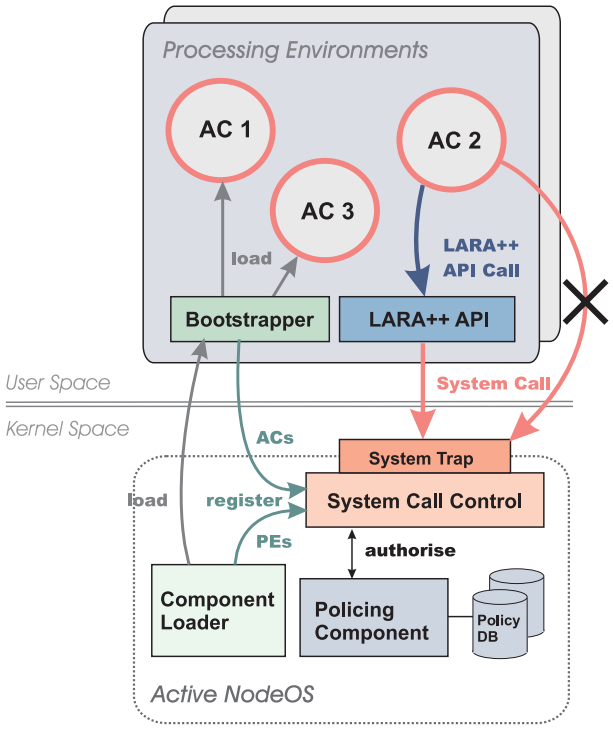


Figure 6.3: System Call Control Mechanism – Malicious system calls avoiding the LARA++ interface are blocked to ensure security.

At the time of writing this chapter, the system call control has only been implemented for the Windows 2000 implementation of LARA++. It checks the process handle of the calling thread to find out whether the call originates from within a LARA++ processing environment. If so, the call must be first authorised by the policing component (prior to passing it on to the actual system routine); otherwise, the call is allowed to pass. To enforce that all LARA++ system calls (i.e., all system calls originating from active components) are controlled by this mechanism, the NodeOS must registers every processing environment with the system call control at instantiation time.[3]

A tight integration of the system call control with the system trap module is also advantageous for performance reasons. It minimises the performance overhead caused by the control mechanism. Since the system API is heavily used, such performance optimisations are crucial for the overall system performance. For this reason, the system trap module is completely implemented in assembler language. As a result, the implementation of the system call control has also required the use of assembler. However, this has benefited the overhead caused by the control mechanism, which turns out to be bound to a few assembler instructions.

[3]Note that the processing environments can be trusted to register active components before they are executed.

6.3.5 Policing Component

The policing component is responsible for applying node-local security policies to the active processing on LARA++ nodes. This section describes the implementation details of this component. A more general discussion on policing within LARA++ is provided in section 5.8.

As the policing component is in charge of authorising critical system calls – i.e., system calls that provide access to controlled resources (for example, memory and bandwidth) or system service routines (such as `LSetThreatPriority()` or `LAddRoute()`) – this is a key element of the LARA++ safety and security architecture.

Since our LARA++ implementation exploits the concept of virtual addressing for software fault isolation, active components can access critical routines and resources only through the LARA++ system API. It is therefore sufficient to safeguard the system trap gate. The system call control described in the previous section is therefore sufficient to restrain the system interface. It intercepts all system calls originating from active components and checks with the policing component to whether a call is authorised or not. As a consequence, the policing component must merely check the access privileges of the calling component and compare them with the actual system call.

The following paragraph describes the policing mechanism of the prototype implementation in more detail. When a LARA++ system call is identified by the system call control, it requests the policing component to authorise the call before it is passed on. The authorisation request includes the system call identifier (SCID) and the active component identifier (ACID). While the former is used to identify the system call, the latter identifies the active component that has originated the call. The ACID (which is essentially a sufficiently large random number that makes it hard to guess) also serves as a secret credential that is used to authenticate the active component with the policing component (see also section 5.8.1).[4] In the case of a resource related system call (for example, `LWriteToDisk()` and `LSetThread-Priority()`), the resource quantity or the scheduling mode/priority must also be included within the authorisation request. The policing component then looks up the active component data structure (ACDS) and verifies whether or not the active component has permission to place the system call. The response to the system call control indicates whether the system call has been authorised or not.

As a result, the main task of the policing component is to look up the access privileges of a calling component and to validate the component's authorisation for placing

[4]Note that system calls with an invalid ACID are strictly blocked. Since a false ACID indicates a malicious (or erroneous) active component, the system call control immediately terminates such components. This prevents malicious components from falsifying component identities which could give them unauthorised access to system services and resources.

the system call. Since this is primarily a problem of fast data lookup, which can be implemented based on standard database mechanisms, a 'proof of concept' implementation has not been considered mandatory. Hence, a complete implementation of the policing component has been deferred to a later time.

6.4 Policy Domains

The policy domains proposed by the LARA++ architecture are considered to be a pure management unit that defines the scope for security and resource access policies on a node. These policy domains thus provide a means to offer a different 'class of service' to active programs on a single active router. Depending on the policy domain, active components can have a different resource quantum or access priority (for example, a higher thread priorities and more memory) and security restrictions (for example, packets can only be read, not modified or dropped).

Since the LARA++ policy domain layer is only a conceptual layer for the management of node-local policies, no implementation is involved. Instead, this logical layer must be supported by several other components:

First, the component loader, which is responsible for assigning active components to a policy domain during instantiation of the components, must support this concept. Based on the component (i.e., the component identity, its producer, and the user loading the component) and the node-local security policies, the loader chooses a suitable policy domain for the component. The loader selects the policy domain that offers the best class-of-service for the component. Once the component is assigned, the loader initialises the ACDS[5] according to the policies defined by that domain.

Second, the policing component, which is responsible for enforcing the node-local policies on the active processing, must also support the concept of several policy domains. However, the fact that policy enforcement is primarily based on the optimised policy representation kept in the ACDSs, hides the use of different policy databases from the policing component. It is the component loader that is responsible for initialising the component ACDSs with the policies of the right policy domain.

Although the concept of providing different classes of service for active computations is an important element of the LARA++ architecture, the current prototype implementations do not yet support multiple policy domains. Both implementations provide only the default (or fall-back) policy domain. The reason for this limitation is simply the lack of time. The implementation is expected to be straightforward as illustrated above.

[5] As previously described in section 5.8.1, the ACDS is the data structure that is used for the run-time policing of active components. The data structure is optimised for lightweight authorisation of AC system calls.

6.5 Processing Environments

The LARA++ processing environments provide a safe and efficient environment for active component execution. While safety is provided based on the principles of software fault isolation, resource management and system call control, efficiency is achieved through the use of optimised binary code for active computation and lightweight context switching between trusting active components.

Both prototype implementations of LARA++ are based on a standard commodity OS as the underlying router platform. Hence, the concept of virtual memory that is used by typical user-space processes within Windows 2000 and Linux is the obvious choice to achieve software fault isolation for the active processing. The virtual memory manager (VMM), which is typically realised in hardware on the processor, enforces the protection boundaries between the processing environments and the active NodeOS. Such a hardware supported solution has the following advantages: reliability and efficiency. It incurs hardly any processing overhead to achieve safety, and more important, hardware based solutions are known to be highly reliable.

As a result, both prototype implementations take advantage of the concept of user-space processes in order to implement the LARA++ processing environments. Unfortunately, on the one hand, this approach has the drawback that the network data have to be passed back and forth between the active NodeOS in kernel-space and the active components in user-space. Although this typically requires a heavyweight copy operation for each crossing of the protection boundary, the efficient packet channel mechanism described in section 6.3.3 provides a solution to circumvent this deficiency through a technique called memory mapping.

On the other hand, the use of standard user processes for the processing environments has several noteworthy benefits: First, software fault isolation based on virtual memory enables the safe execution of binary code. This circumvents the need for a safe programming language (i.e., any programming language for which the LARA++ system API is available can be used) and avoids the need for costly run-time checks (for example, type or range checking) or language based constraints (for example, no use of pointers). Second, the use of standard processes as a basis for the processing environments enables the use of existing threading mechanism for the active component processing. However, as will be further discussed in section 6.5.3, the use of an optimised user-level thread scheduling mechanism is suggested in order to minimise the cost of context switching between active components.

The remainder of this section describes the implementation of additional processing environment mechanisms, namely the component bootstrapper, active component scheduler, and system API.

6.5.1 Component Bootstrapper

The bootstrapping mechanism of the processing environment is responsible for loading LARA++ components into memory, and to initialise and start them. The bootstrapping of a component is initiated by the component loader – the system component responsible for verifying the integrity of a component (i.e., authentication and authorisation) and choosing an appropriate processing environment for the component (based on the trust relation of other components).

The component loader initiates the bootstrapping of a component by passing the component URI to the bootstrapper. The bootstrapper then loads the component into memory like a shared or dynamic link library. Once loaded, the bootstrapper initialises the component, and in the case of an active component also activates the component. For this to work, the components must implement a well-known interface – either the *IPassive* or *IActive* interface. The initialisation routine `Initialise()` passes the handle for the LARA++ system API to the component in either case. The interface handle is required for the components to gain access to the system interface.[6]

Active components will also receive a secret identifier (ACID) during initialisation. This run-time identifier is required for the component to authenticate itself when accessing the system interface. In the event of a system call, the ACID is used by the policing component to (hash) lookup the ACDS of the calling component in order to authorise the system call. Moreover, active components also register with the active NodeOS and establish the communication channels with the packet classifier.

Finally, if the initialisation of the active component was successful, the bootstrapper will activate the component by creating a new active thread using the component's main routine (`IActive.Main()`) as the entry function. A component's main routine typically starts off with the installation of an initial set of packet filters and then loops to service the packet channels. This involves the continuous processing of packets received on the input channel and returning them through the output channel once processed. Like normal programs, active components terminate when returning from the main routine.

Note also that a watchdog mechanism is frequently used to control the operation of the active components. If an erroneous or malicious AC (for example, a component that fails to process/forward packets) is detected, the component is terminated by the active NodeOS in a safe manner.

[6]Note that the LARA++ system API is not directly addressable from within the components (i.e., dynamic re-linking of the interface libraries would be required).

6.5.2 System API

The LARA++ system API exposes the programming interface of the processing environment and the active NodeOS. It enables user components to program the routers through a set of standard and active network specific system calls. While the standard system API provides common operating system functionality (i.e., functions to access node-local resources and configurations), the active network specific API encompasses system routines, for example, to load, install and configure components, to register packet filters with the classifier, and to access and send network packets. For further details on the LARA++ system API, the reader is also referred to section 5.5.2.

Since both prototype implementations of the LARA++ architecture exploit the concept of user-level processes to achieve safety for active components execution, the need for an interface stub that is part of the processing environment's address space arises. For this reason, a user-space stub of the LARA++ system API is provided in form of a shared or dynamic link library, which is dynamically linked to the processing environment at start-up time.

At initialisation time of a newly loaded active component (i.e., `Initialize()`), the processing environment passes the handle of the user-space stub of the system API to the active component. The interface handle enables the component to access the system API through the user-level stub. System calls are either directly resolved within the stub functions (for example, functionality implemented as part of the processing environment is directly addressed), or mapped onto standard system calls or LARA++ specific NodeOS routines in the kernel by means of a software interrupt.

Since the LARA++ system call control component in kernel-space polices all system calls of a LARA++ process (even standard operating system calls such as `WriteFile()`, `SetProcessPriority()`, etc.), there is no need for the active NodeOS to re-implement these service routines.

The current prototype implementations of the LARA++ architecture support only a small set of system calls in order to demonstrate the basic functioning of the system API. In the long term, however, it is planned to fully implement the NodeOS system API specified by the DARPA active network working group [ANW99], which provides a richer interface (for example, for file-system and memory access) and better error reporting.

6.5.3 Active Component Scheduler

Since the LARA++ architecture envisages an active node to provide the functionality of conventional routers and beyond based on the composition of many active components, the need for a lightweight scheduling of active components is inherent.

It is expected that many of the active components will be developed by well-known

and trustworthy component providers (for example, by companies like Cisco or Microsoft). This enables the grouping of active components with mutual trust relationships into a single and hence efficient processing environment. The fact that context switching among components running in the same address space is far less expensive than context switching among different address spaces (see section 7.4.1 for experimental results), allows LARA++ to offer a very lightweight solution for active components sharing a processing environment.

As a result, the LARA++ architecture proposes the use of a very lightweight scheduling mechanism for components executed within the same processing environment. The scheduler minimises the cost of context switching between active components by avoiding expensive boundary crossings between the processing environment and the system kernel for every scheduling decision. The scheduler rather operates internally to the processing environment that directly implements the lightweight component scheduler.

The component scheduler supports three complementary scheduling algorithms that can be used in conjunction with each other to achieve maximum performance.

Cooperative Scheduling: This enables active components to actively de-schedule a thread in order to release the processing resources immediately (before the scheduling quantum expired). The scheduler, in turn, allocates the processor straight away to another 'waiting' thread of the same processing environment. Cooperative scheduling is triggered by calling the component scheduler directly through the LARA++ system call `LDeScheduleNow()`.

Pre-emptive Scheduling: This mode ensures that scheduling of active components running within the same processing environment is not impeded by misbehaving components that do not cooperate in the scheduling process. Such components would otherwise lock the overall processing environment. Support for pre-emptive scheduling also takes the burden of explicitly releasing the processing resources off the active program developers. Note, however, that pre-emptive scheduling is slightly more costly than cooperative scheduling, as the entire processing state of the thread (i.e., all processor registers) must be captured, whereas only a subset of the processing state (for example, the index registers such as the instruction and stack pointer) is required in the cooperative case.

Restart Scheduling: This scheduling mode is a special case of cooperative scheduling. The idea here is that short tasks (for example, the processing of certain events or packets) that easily finish within one scheduling quantum do not have to capture the state at the end of the task, as the thread can simply restart (or start at a pre-captured check point) each time it is activated. The advantage is that the cost

of scheduling is further reduced by avoiding state capture at the end of the task. Due to the special nature of this mode, however, it is only useful for the processing of short tasks.

The Windows 2000 implementation of LARA++ provides the active component scheduler in form of a link library that is dynamically linked to the processing environments. The library exports functions to create, terminate and (de-)schedule user-level threads. These functions can thus be directly invoked through the user-level stub of the LARA++ API – without the need of a system call.

Pre-emptive scheduling is enabled by means of a special kernel-level timer component[7]. The component registers with the system timer to get called within the interrupt service routine of the timer interrupt. When called by the timer interrupt service routine, it checks whether or not the current process is a registered LARA++ processing environment. If so, it sets the instruction pointer of the active thread to the address of the active component scheduler. Consequently, when the interrupt service routine completes, the active thread resumes its processing at the start of the scheduling routine. This in turn captures the state of the current user-level thread and schedules another 'waiting' thread.

6.6 Active and Passive Components

Active and passive components are the software modules that are downloaded onto LARA++ nodes. They are the units of active code that are executed within the processing environments. While active components are the actual active programs that are executed on the node, passive components provide merely support functionality for active components like software libraries.

As previously described, LARA++ components are either distributed in the form of source or binary code. In the former case, components are just-in-time compiled upon arrival on the target node, whereas in the latter case, components are distributed in binary form as shared or dynamic link libraries. Once checked for code integrity (based on code signatures), components are loaded into a processing environment alongside other components that share a mutual trust relationship (see also section 5.5.2).

Since the LARA++ prototype implementations exploit standard link library techniques and execute active code within user-space processes, components can be developed and tested based on standard development tools (for example, the Visual Studio IDE, or the GNU compiler and debugger).

[7]The kernel-level timer component is implemented as a standard Windows 2000 device driver.

6.6.1 Implementation Process

LARA++ components are implemented as shared libraries under Linux or as dynamic link libraries under Windows 2000. Developers can therefore choose among a range of developments tools (for example, compilers and debuggers) available for these platforms and pick their favourite programming environment.

As active components can be loaded and executed in the form of binary code, the implementation of the components is conceptually independent from any particular programming language. Note that the LARA++ architecture does not build safety and security upon a specific programming language or certain language constraints (such as strong typing, range checking, etc.). However, since the LARA++ system API must be linked to the component at build time of the shared library or DLL, only programming languages for which the system API is available can be used. At the present time, the system APIs of the current LARA++ prototype implementations are only available for C and C++.

In order to allow the component bootstrapper to initialise and activate the components, they must implement a minimal well-known interface. Depending on the type of component (active or passive), a different interface must be exposed. Active components must implement the *IActive* interface, whereas passive components need to implement the *IPassive* interface (see section 5.5.1 for further details). Both interfaces expose an initialisation routine (`Initialise()`), which is called by the bootstrapper at loading time. The *IActive* interface must also implement a main routine (`IActive.Main()`).

In order to support LARA++ component developers in the implementation process, template source files for active and passive components are provided for C++. The component templates implement stub functions for the *IActive* or *IPassive* interfaces. They include routines for component registration with the active NodeOS, packet filter installation, and standard packet channel processing.

6.6.2 Debugging and Testing

A special processing environment and active NodeOS that can operate on the development machine (for example, standard user work station) have been developed in order to enable component programmers to test and debug their components using standard debugging tools. The idea here is that developers run a minimal LARA++ environment on the development machine to directly test and debug the software.

Providing such a minimal environment in the case of the current LARA++ prototype implementations is relatively easy as both prototypes are based on standard operating systems, which allows the development platforms to be identical to the router platform. As a result, component developers can simply install the debugging processing envi-

ronment (like a normal user-space application) and active NodeOS (which consists of a set of device drivers under Windows 2000 or kernel modules under Linux) on their development machines.

However, since the computation of an active component is typically driven by the data passing through the active router, additional software that generates data packets matching the filters of the tested active component is required. This can be achieved either by a local traffic simulator or through an external program that produces the respective traffic patterns and routes them through the development machine.

These debugging and testing capabilities have proven to be a very useful feature. It allows the developer to carry out fundamental testing of active components on a real system and with genuine data traffic rather than simply through emulation or even simulation. Nevertheless, since the testing is carried out in an isolated environment on the developer's workstation, the level of testing is still limited. Remote testing (and possibly even debugging) of active components within real active network nodes, where further unexpected problems such as unpredicted feature-interaction issues are expected, is subject to future research.

6.6.3 Example Active Components

This section briefly outlines the implementation of three example active components. These examples demonstrate how simple LARA++ components, implemented as normal user-space libraries, can provide useful network services. The examples also illustrate the scope of active programmability based on the LARA++ component architecture – ranging from a general network service to the implementation of a building block of the LARA++ architecture itself.

6.6.3.1 Local Congestion Control

This section demonstrates how LARA++ active components are implemented based on the local congestion control example introduced in section 5.2.1.

The trivial congestion control mechanism simply drops packets of lower priority service classes when congestion builds up. A queue length threshold for each service class is used to determine which packets to drop during congestion. Thus, if the queue length of an output queue exceeds the threshold of a particular service class, all packets of a lower priority service class that would be sent on this interface will be dropped. For example, if the threshold rises above the threshold of the *premium* CoS (i.e., `THRESHOLD_PREMIUM`), packets of lower CoS will only be sent if the threshold of the corresponding CoS is not (yet) exceeded.

The following code fragment shows the core sections of the source code for the ex-

ample active component. It illustrates how the LARA++ system API is used. The `IActive.Main()` function is called when the active component is instantiated. After the registration of the active component with the active NodeOS and the installation of the packet filter(s), the active thread loops around the packet servicing functions (i.e., receive, send or drop routines) until the component terminates. In the `while{}` loop, the component blocks on the `LReceivePacket()` call until any packet that match the packet filter (i.e., packets that include the specific CoS mark) is received and then decides whether to send (`LSendPacket()`) or drop (`LDropPacket()`) the packet based on the congestion condition and the packet's CoS.

```
ACDLL_API int ACMain(void)
{
    // Initialise variables & define packet filter(s)
    [ ...  ]
    if (LRegisterAC(&ACInfo) == LARA_FAILURE)
        return LARA_FAILURE;
    if (LInsertPacketFilter(pFilterList) == LARA_FAILURE)
        [ ...  ]
    while (Run) {
        pLaraPacket = LReceivePacket(&ACInfo);
        pBuffer = LGetPacketBuffer(pLaraPacket, &bufLen);
        pIPHeader = pBuffer + sizeof(TEthernetHdr);
        // get output interface
        OutputIF = LRouteLookUp(pIPHeader);
        // get current length of output queue
        QueueLen = LGetQueueLength(OutputIF);
        if (pIPHeader->Type == IPV4)
            CoS = GetCoSMarkIPv4(pIPHeader);
        else
            [...]
        if (QueueLen < THRESHOLD_PREMIUM)
            // no congestion; service all packets
            LSendPacket(&ACInfo, pLaraPacket);
        else
        if (QueueLen < THRESHOLD_GOLD && CoS == PREMIUM)
            // output queue fills; still serve premium CoS
            LSendPacket(&ACInfo, pLaraPacket);
        else
            [...]
        else
            // output queue full, drop low CoS packets
            LDropPacket(&ACInfo, pLaraPacket);
```

```
    }
    LUnregisterAC(&ACInfo);
    return LARA_SUCCESS;
}
```

6.6.3.2 Server Load Balancing

A common problem for today's businesses and corporations is to provide scalable online services (such as Web or FTP services) to their customers. Providing appropriate network bandwidth to the servers is in many cases not sufficient, as the servers become the bottlenecks.

This section describes an example active LARA++ component that allows an active edge router to load balance network traffic, or in other words server requests, among a number of servers. The basic idea is to exploit the LARA++ active router technology to distribute server requests (i.e., TCP connections or UDP flows) among available servers based on round-robin scheduling. Since load balancing must be fully transparent to the client applications, the servers in the cluster must have an identical setup. The active router provides a separate sub-network for each of the servers on one of its interfaces using the same network address. Such a peculiar network setup, however, confuses standard routing towards these sub-networks. As the server networks use identical addresses, the router cannot route packets to the server hosts based on the network address anymore. Instead packet routing to these specialised sub-networks is based on the source address of the packets for the lifetime of a "connection" or flow.

So, when a "new" server request (i.e., connection or flow) is received at the active router, the router selects a server (based on a round-robin approach) and adds a new entry for this client into the special routing table. The entry consists of the triple *(client IP address, interface, timestamp)*, where the client IP address is used as the hash key. Thus, all packets received from the corresponding client will be routed out the same interface and hence reach the same server. The router detects a "new" request/connection based on the hash table. If no entry or a stale entry for the client address exists, a new server is determined and another route is added. An entry is considered stale when the *timestamp* has not been updated for a configurable timeout period. To ensure proper behaviour even for long-lived connections, the router must update the *timestamp* of the routing table entry each time it uses a particular route.

It should be noted that that this simple approach does not claim to 'exactly' load balance server requests. It is rather an approximation. Since there is no mechanism to determine precisely the end of a request, some error might be introduced with respect to the load balancing accuracy when the same client initiates several requests within the lifetime of a route table entry. However, proper operation is guaranteed at all times.

6.6.3.3 Component Loader

The LARA++ component loader can be implemented as a "standard" active component. Since it would be this LARA++ component that enables the loading of other components, it would have to be installed by default on each node. The bootstrapping of the loader component could be done automatically during start-up of the active router.

The component requires access to privilege service routines (for example, to authenticate, authorise and load components) within the active NodeOS, and must therefore be assigned to the system component group. Consequently, the loader component will thus request registration as a system component with the active NodeOS during initialisation.

At start time, the component inserts the packet filter(s) required to intercept the data traffic of the supported code distribution protocol(s). The filter based network access mechanism allows the component loader to simply integrate support for multiple code distribution protocols into a single "application". Conversely, the LARA++ component architecture also encourages the use of several independent loader components (i.e., one for each distribution mechanism).

Upon receipt of a code stream, the loader reassembles the component code in permanent memory and verifies the code checksum. It then checks the authentication and authorisation of the component by calling the respective system service routines. Components that are distributed in source code form must also be just-in-time compiled at this point.

Finally, if the component is valid and approved, the loader initiates the bootstrapping of the component. As part of this process, the NodeOS identifies an appropriate processing environment for the new component based on the trust relationships to components currently running on the node. If no appropriate PE exists, a new one will be spawned. In turn, the component loader signals the bootstrapper of the selected PE to instantiate the new component.

Note that although the main program logic of this component can be implemented as an active LARA++ component in user-space, the component relies heavily on system service routines and the policing support that is provided through the system interface. For example, signalling the bootstrapper of another PE to initiate the instantiation of a new component is achieved by means of a system call. This has the advantage that access control can be done through the default policing mechanism of the system interface and thus makes the need for a special access control mechanism within the PE superfluous.

6.7 Summary

The realisation of the prototype active router platforms described in this chapter does not attempt to provide a complete implementation of the LARA++ architecture previously described in chapter 5. For example, significant parts of the security and policy framework, which are both key to the active router architecture, have not been implemented due to the overall complexity of the system and the time constraints. The objective was rather to demonstrate the feasibility of the architecture through a 'proof-of-concept' implementation of the LARA++ specific mechanisms such as the packet classifier, the packet channels, the system call control, the active component scheduler, etc. (the major components).

The implementation of the various mechanisms is described according to the overall structure of the LARA++ architecture, namely the active NodeOS (section 6.3), the policy domains (section 6.4), the processing environments (section 6.5), and the active and passive components (section 6.6). The last section focuses in particular on the implementation process, and the debugging and testing of LARA++ components. It also describes a range of example components.

Chapter 7 proceeds with the evaluation of the architecture. The LARA++ prototype implementation described here serves as the reference platform for the quantitative evaluation of the architecture.

Chapter 7

Evaluation

7.1 Overview

This chapter presents the evaluation of the LARA++ active router architecture and prototype implementation as described in chapter 5 and chapter 6 respectively.

Since the main objective of this work was to design a novel component-based active router architecture from the ground up, it has not been feasible to fully realise such a system. As a consequence, the evaluation of the LARA++ architecture is to a large extent a theoretical analysis. Based on a concrete case study and several example applications, section 7.3 evaluates how LARA++ fulfils the objectives and requirements presented in chapter 4.

The final part of this chapter (section 7.4) evaluates the prototype implementation of the LARA++ architecture. The performance analysis examines the individual components and mechanisms that constitute the LARA++ platform first, and then combines these results analytically in order to estimate the overall performance of the prototype active router.

7.2 Evaluation Methods

In general there are two methods used for the evaluation of research contributions such as those presented within this thesis, namely qualitative and quantitative evaluation. Since the contributions here include both an architectural design and its implementation, a combination of the two approaches has been employed throughout this chapter.

The main focus of the evaluation lies in the qualitative aspects of the LARA++ architecture. In particular, since the main contribution of this thesis is the novel component-based architecture for active routers, a qualitative evaluation of the concepts and design of the LARA++ architecture has been regarded as more meaningful.

Since it has been feasible to implement only a subset of the overall LARA++ architecture, a quantitative evaluation of the entire system cannot be provided at this stage. Nevertheless, section 7.4 provides a quantitative evaluation of the key components and mechanisms. Finally, the results of this micro-level analysis are combined to estimate the overall system performance of the LARA++ prototype implementation.

7.3 Qualitative Evaluation

This section presents a qualitative evaluation of the LARA++ architecture. Issues regarding the design and concepts behind the LARA++ active programmability are considered here.

Firstly, the case study that is used to evaluate the features and usability of the LARA++ architecture is examined. Subsequently, LARA++ is evaluated according to the success or failure of the requirements introduced in chapter 4.

7.3.1 Case Study – An Evaluation Scenario

In order to evaluate the LARA++ architecture, an example case study is introduced. The case study on ubiquitous Internet service provisioning within the city centre of Lancaster, and the University campus, uses LARA++ active nodes for the deployment of new network service and the integration of enhanced router functionality. In particular, advanced network services that provide secure access control for the wireless network and fast handoff support for roaming users are considered.

7.3.1.1 The Setting

The underlying network infrastructure for this undertaking is currently being deployed as part of the Mobile IPv6 Testbed collaboration between Cisco Systems, Microsoft Research, Orange Ltd., and Lancaster University [MSR01].

As illustrated in Figure 7.1, the approach of a wireless overlay network has been taken. Broadband wireless LAN technology based on IEEE 802.11b [LAN99] is used as the primary network access technology (due to the high data rates). GPRS [Nok98] and Bluetooth [For01b] are used alternatively. While GPRS provides a low-bandwidth fall-back solution for "remote" locations without wireless LAN coverage, Bluetooth is used for short range communication in hot-spot areas.

It is envisioned that LARA++ active routers will form the core elements of this service network. A dedicated active router will be deployed per network *district*[1] for fine-grained control of the network (for example, access control, accounting, and caching).

[1]Districts define the boundaries for administrative domains.

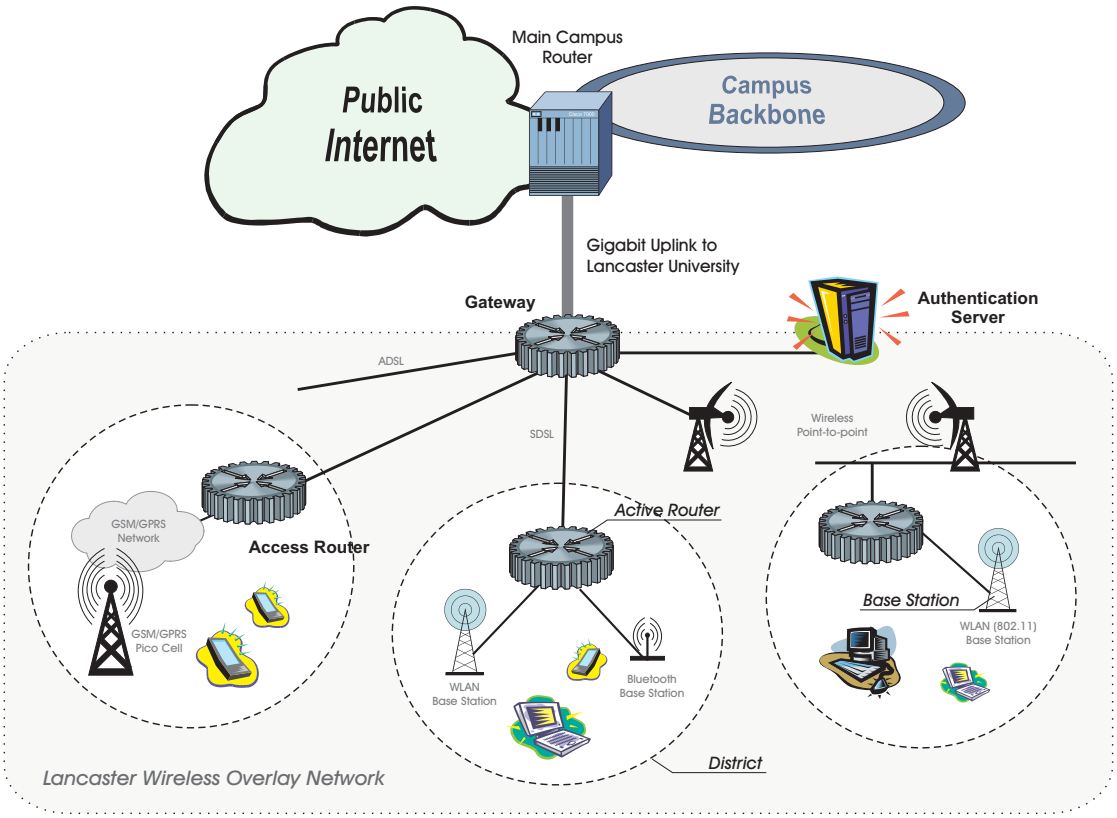


Figure 7.1: The Mobile IPv6 Testbed – A wireless overlay network environment for research in to next-generation mobile systems, services and application.

As shown in Figure 7.1, these *access routers* are linked back to active routers further up in the hierarchical backbone structure by means of point-to-point wireless links, DSL, or FastEthernet. The top-level router in the hierarchy, called the *gateway router*, connects the access network with the core router of the campus backbone via a GigabitEthernet link. The use of an active router based gateway is expected to be beneficial as firewall controls can be flexibly customised and new control mechanisms can be dynamically deployed.

GUIDE II [GII01], the successor project of the well-known GUIDE project [DCMF99, CDM+00], is one of the research activities making use of the Testbed infrastructure. Unlike the initial GUIDE project (which aimed at mobile services and applications for Lancaster tourists as they explore the historic city), the goal of GUIDE II is to provide general-purpose, ubiquitous services and applications to Lancaster citizens whilst on the move around the city centre.

7.3.1.2 The Challenges

A key challenge of the GUIDE II project is to "open up" the network and to provide public services (including Internet access) to the general public. Since the deployment

of wireless network technologies in public places bears the danger that unauthorised people use the network resources, it is important to introduce appropriate access control mechanisms. A secure user authentication and authorisation mechanism, and a reliable access control mechanism, are vital – particularly for wireless LANs, where the absence of comprehensive security provision has been a hindrance to its widespread adoption.

Another fundamental challenge of the GUIDE II project is to provide ubiquitous network connectivity. The various link technologies deployed to achieve this goal are integrated using IPv6 as the common inter-network protocol. IPv6 serves also as the inter-district network protocol that connects the individual administrative domains. The mobility extensions for IPv6 [Per01] allow users to roam between the different administrative domains and/or network technologies. The concept of a permanent "home address" is applied to achieve location transparency. However, since real-time applications such as real-time audio and video streaming have very stringent QoS requirements, special network support to enable smooth handoffs for roaming users is required.

Since existing proposals for both of these example problems rely on enhanced support in network routers (which is not yet provided by most router manufacturers), it can be argued that they can be addressed more elegantly through the use of active network technology. The following section will demonstrate how the LARA++ active router architecture provides the flexibility and extensibility required to cope with these problems in an efficient and elegant manner.

7.3.1.3 The Solutions

This section presents a solution based on the LARA++ active router architecture for both of the problems outlined above. The focus here lies not on the solution itself, as the concepts behind the solution are not specific to active networks and therefore can be realised otherwise, but on the fact that LARA++ provides a generic platform that is sufficiently flexible and extensible to resolve those problems (and many more). The combination of the two problems is of special interest as it demonstrates how different users or groups of users can program a LARA++ router and thereby take part in the cooperative process of service composition.

Network-level Access Control

The access control mechanism developed for the Mobile IPv6 Testbed is based on the principles of packet marking and packet filtering. Data packets are tagged on the client terminal before they leave the node. Based on the presence of the tag (access token) and the credentials associated with the token, access to the network is either granted or denied. A complete description of the access control architecture and its realisation based on LARA++ active routers has been published by Schmid et al. [S$^+$01, SFW$^+$01].

Although this access control mechanism involves several elements (i.e., authentication protocol, authentication server, extension of client network stack, etc.) in order to achieve a high-level of security, reliability, scalability, and performance in a roaming network, the key component in this context is clearly the access router. These routers are responsible for controlling the data traffic passing the node ("access gate") based on the access token in the *upstream* packets (originating from the clients) and the destination address in the *downstream* packets (destined for the clients).

In conventional networks, where routers are closed commercial systems, such value-added functionality cannot simply be introduced when needed. The service provider is stuck until the router manufacturer provides a new software image with the required functionality. Using the LARA++ active and programmable network technology, however, enables the service provider in this scenario to roll out such a service without the delays caused by slow standardisation processes and software production cycles of the router manufacturer. Customers of conventional router technologies are simply tied to the manufacturer as no other company can develop software for the "closed" system, whereas active network technologies encourage third party software development (and hence competition) by defining a programmable interfaces to the router hardware. For example, the LARA++ model enables end users to develop new network functionality themselves (if they have the know-how and resources) or to simply buy the required functionality from an active component provider.

The access control functionality of the access routers can be implemented in various ways, ranging from single to multi component solutions. A solution based on three active LARA++ components is described here:

- The *Control Component* implements the access control protocol that is required to communicate the access control information (i.e., valid access tokens, session keys, etc.) from the authentication server to the access router. Based on periodic access control updates, the access router maintains an access control list that provides the basis for the filtering components.

- The *Upstream Filter Component* ensures that only authorised client systems gain access to the network beyond the access router. It therefore intercepts all traffic originating from the wireless clients and checks the validity of the access tokens against the access control list. While packets with an invalid or out-dated token are strictly discarded, authorised packets are forwarded after the removal of the access control information.

- The *Downstream Filter Component* is responsible for controlling the traffic sent towards the wireless terminals. It ensures that only traffic to authorised client nodes

is forwarded by the router. For this, the component intercepts all traffic destined to any of the wireless networks and checks whether or not the final destination address has a valid authorisation.

This example application exploits only a subset of the LARA++ features. For example, since the whole access control architecture is a managed and long-term service provided by the network service provider/administrator, LARA++ is mainly used as a means to add value-added functionality (i.e., packet filtering based on the proprietary tagging approach) to the routers. Features such as the remote loading mechanism and fast component instantiation support provided for highly dynamic service deployment, or the high-level security architecture incorporated to enable end-user programmability are not necessary here. Conversely however, these features clearly do not weaken the system. The foremost feature, for example, simplifies the dynamic roll-out of the service in the first place and enables dynamic service extensibility through replacement of components when newer versions of the software become available. For example, this dynamic service deployment capability enables the service providers to extend the access control service dynamically by an accounting component or to introduce QoS support throughout the access network at later times.

Network Support for Fast Mobile Handoffs

A key characteristic of the Mobile IPv6 Testbed is the segmentation of the network into many (small) administrative domains (*districts*). The key benefits of this design, namely high scalability and fine-grained access control (i.e., potentially on a per-cell basis), have been fully explored by Schmid et al. [SFW+01]. Moreover, such a micro-cellular network structure enables the Testbed to function as a "real" simulator of third and fourth generation mobile networks.

However, as a consequence of the highly segmented network infrastructure, support for smooth network handoffs becomes especially important. As users may frequently change layer-3 networks when moving between the wireless cells, location transparency and smooth network handoffs are critical for sustaining any active communication channels. Smooth network handoffs (which imply minimal delay) are especially indispensable for QoS sensitive communication, such as multimedia streaming or interactive services (for example, VoIP).

A thorough analysis of the mobility support within the Mobile IPv6 protocol has revealed that Mobile IPv6 is purely a routing protocol for end systems; network routers (apart from the home agent) are not involved. As a result, the delays caused by network handoffs can be unnecessarily high due to the fact that they require a correspondent node

(any IPv6 device communicating with a mobile device) to receive a *Binding Update*[2] message, before this can adapt the routing to the mobile's new location. Note that this adds at least the delay of one round trip time (between the mobile node and the correspondent node) to the handoff latency. As typical latencies of wide area links can be as high as 400 ms, this can result in a handoff performance that is unacceptable for multimedia applications. Figure 7.2 illustrates this inefficiency and suggests a solution based on network-side computation. The router in this example could simply reroute the traffic sent to the mobile node's old location to its new location based on an approach called network address translation [EF94].

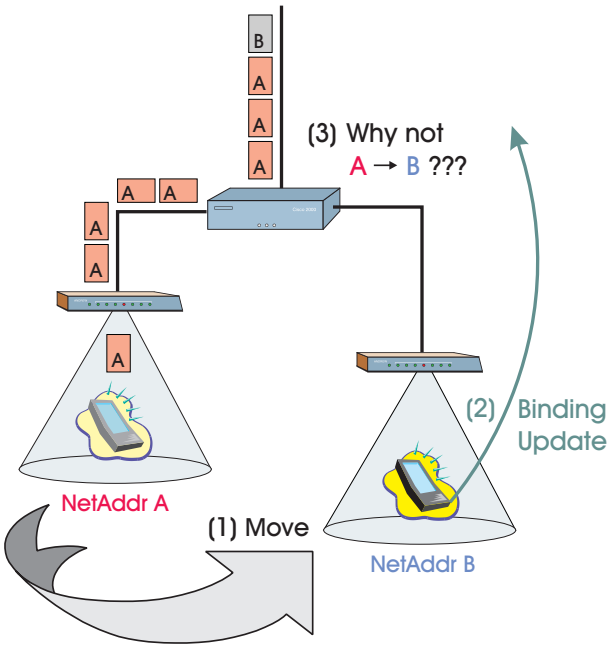


Figure 7.2: Mobile IPv6 Routing Convergence Time: After move of mobile device (1), it sends binding update to inform correspondent node (2). It takes $\geq$ RTT until data flow will reach the mobile's new location. Why can't the router immediately reroute the traffic to the new destination (3)?

Although the solution outlined in Figure 7.2 is straightforward and seems to be an ideal solution for smooth handoffs, for conventional networks this would require the standardisation of a protocol extension to Mobile IPv6 that involves the routers of the access network. Since the Mobile IPv6 protocol does not yet comprise normal routers (apart from the home agent), such an extension would drastically change the overall protocol semantics. Furthermore, the protocol extension would first require the standardisation body to approve it, router manufacturers to implement it, and network administrators to deploy the new software releases before the fast handoff mechanism could be used.

[2] A Mobile IPv6 control message sent to the correspondent node and the home agent to indicate the mobile node's new location (care-of-address) [Per01].

Other proposals that address this inefficiency [C+96, C+00, MS00] also require extensions to the Mobile IPv6 standard and/or specialised support within the network (for example, proxy nodes).

Conversely, active networks allow this problem to be resolved *transparently* on the router where the root change occurs (without any modification to the network or streaming protocol) when a smooth handoff is really needed. Note that fast handoffs are only needed for certain QoS sensitive applications. Active routers can be dynamically programmed on a per-handoff basis to reroute any data traffic incorrectly routed to the mobile device's old location to its new point of attachment without delay. This simple technique provides a means to repair the incorrect routing locally on the router where the route change takes place, until the Mobile IPv6 protocol responds to the route change caused by the mobile node's movement.

The remainder of this section presents the design of the smooth handoff solution based on a simple active LARA++ component that minimises the handoff latency of mobile devices in roaming networks:

The *Flow Routing Component (FRC)* introduces short-lived Mobile IPv6 functionality into active routers, such that they can actively take part in the mobile routing (i.e., optimise the handoff performance) until the standard Mobile IPv6 routing protocol converges. Thus, upon a network handoff the mobile device injects (or simply invokes) the FRC into active LARA++ routers along the reverse transmission path[3]. After instantiation on a LARA++ router, the FRC checks whether or not the route change for the particular flow takes place on this router. If not, it terminates immediately. Otherwise, it starts re-routing any packets misrouted to the mobile node's old location to its real location by means of network address translation [EF94]. Since the component offers only a temporary route optimisation (until the mobile routing protocol converges), the FRC is very short-lived. It terminates typically a few seconds after instantiation (as soon as packets for the mobile node's old address stop arriving). For a more thorough description of this smooth handoff mechanism and its realisation based on the LARA++ component technology, the reader is referred to prior publications of the author [SFSS00b, SFSS00a].

This second example application for LARA++ differs significantly from the access control application introduced above. The flow routing component provides a very specific (tied to a particular flow) and short-lived (only for the convergence time of the mobile routing protocol) service on the network nodes. Because of its nature, the appli-

[3]Note that in the case of a wireless network, the access network is typically structured as a hierarchical network, which implies that the route change for a mobile device roaming between cells (of the same provider) typically takes place on a router of the hierarchical access network (i.e., close to the mobile device).

cation has very stringent requirements regarding the dynamic loading and instantiation of the component. Since the service has very rigid timing constraints, dynamic loading of LARA++ components and code instantiation must be very fast. Furthermore, the fact that the service is user initiated (upon a handoff of the user's mobile terminal) requires support for safe and secure programmability of LARA++ nodes for common end-users.

This example again demonstrates the fundamental capabilities of the LARA++ architecture, namely flexible and transparent extensibility of router functionality based on the concept of packet filters. The component-based handoff optimisation neither relies on any modifications to the MIPv6 protocol nor requires installation of any application specific support on the active router.

7.3.1.4 Discussion

This section analyses the advantages and disadvantages of the LARA++ active router architecture compared to other active network solutions based on the case study introduced above.

The foremost contribution of LARA++ over other active network solutions is its flexible service composition framework. It is this enabling technology that allows active solutions for both problems described above (and many more!) to cooperate with each other on a single active node. It enables independent user groups (i.e., service providers and end users) to extend the functionality on the active router and provide a cooperative service composite (without knowing about each other's services).

The use of packet filters and the classification graph structure as a transparent means for the integration of active services allows independent users to dynamically insert and remove components into and from the service composite. The classification graph provides the semantics for a structured integration of services. It has the potential to reduce the feature-interaction problem resulting from collaborative use of independent active components.

In order to emphasise the actual contribution of LARA++, the remainder of this section will analyse other active network approaches and compare how they would perform in the example scenario.

While the integrated approach to active networking replaces traditional data packets by so-called active capsules (which include both code and data), such active services require (by definition) modifications to the end systems. Depending on the implementation, either the end applications or the end nodes' OS need to be altered in order to insert the active code in the data streams. And although the active capsules approach seems to be a perfect means for the upstream access control application (i.e., extension of end system software is required anyway in order to accomplish the packet tagging),

downstream access control based on active capsules is totally infeasible. It would demand that all correspondent nodes of the mobile systems (i.e., potentially all nodes on the Internet) require amendment to their application or system software. As a consequence, it becomes clear that integrated active network solutions are not suitable for our example scenario. In fact, this general limitation of the integrated approach suggests that it is only practical in controlled environments and for specialised applications.

As a consequence of this general drawback of integrated active network solutions, the remainder of this section compares LARA++ only with other discrete approaches (see section 3.2.2) regarding their performance in this example scenario.

An examination of SwitchWare (section 3.2.2.1) – the pioneer programmable switch approach – has revealed that a sophisticated service composition framework for the active extension is missing. It uses a simple de-multiplexing technique based on the active packet content (for example, identifier of active extension). In fact, since the active extensions are primarily designed for use in conjunction with its integrated active packet approach, SwitchWare would suffer from the same drawbacks as fully integrated solutions.

In theory, both the Router Plugins architecture (section 3.2.2.7) and the CANEs/ Bowman active node architecture (section 3.2.2.3) provide sufficient service composition capabilities to compose active services for either of the example scenarios described above. Both architectures apply a plug-in approach to support service composition, whereby an underlying data structure or program defines the logic ("glue") for the plug-in bindings. However, the fact that the slot logic is defined by a static structure or program (see also section 3.2.2.3), makes those approaches ineffective in practical terms. For example, each time a new type of protocol or service is required, a new underlying structure or program must be introduced. Unfortunately, both architectures lack a dynamic mechanism for that. While the Router Plugins architecture limits extensibility of the plug-in model to the compile time of the kernel, CANEs/Bowman allows manual integration of new underlying programs. In the latter case, it is still unclear though to what extent the classifier, which assigns arriving packets statically to one or more underlying programs, limits the flexibility of the composition framework, as links between those conceptual divides are not considered.

An analysis of application level active network approaches such as FunnelWeb (section 3.2.2.8) has revealed similar problems to those of integrated active network solutions. These systems create a virtual overlay network on top of the existing IP network, whereby the active routers are simply implemented as user-level applications. As a consequence, specialised end-user applications (or modifications to existing applications) that explicitly address the first hop application level router are required. While such systems have proved to be useful for research purposes and experimentation with ac-

tive network applications[4], they are arguably hard to deploy effectively in real network environments if a complete end-to-end network solution is needed.

Another discrete approach called Joust (section 3.2.2.4), which provides the underlying platform for liquid software [HBB+99], suffers from a similar problem. Extended network functionality or services must be explicitly addressed within the actual data packets. The fact that liquid software provides a service equivalent to a dynamically configurable RPC, which allows the applications to tailor the client/server interfaces to the task at hand, ties this service to specialised liquid software applications. The application-controlled service compels the application software to encapsulate RPC related data (for example, RPC identifier and call parameters) into the actual data flows.

The modular Click router (section 3.2.2.6) is another very flexibly configurable router architecture that enables extensibility of router functionality as required for the example active services. However, since configurability of the Click router functionality is limited to the compile-time of the router image, dynamic introduction of new services is not possible. For example if the service provider would like to add an accounting mechanism to the access control service later, it would have to re-compile the router software image first.

The LANode active router platform (section 3.2.2.8) differs from the other approaches as it makes a clear distinction between the control plane and data plane of the router and limits the programmable interface to the control plane. Unfortunately, this has an adverse impact on its usability in this context. Since the core functionality of the example applications (i.e., packet filter, network address translation) must be provided on the data plane, which does not provide a programmable interface, LANode is not an ideal platform. The required data plane functionality would have to be manually introduced through system administrators installing new software modules on the routers.

The Protocol Booster approach (section 3.2.2.8) supports transparent integration of active processing elements inside the network in order to improve the performance of a protocol. Consequently, this approach would provide an ideal solution for the handoff optimisation application as it is specifically designed as a means to "boost" the performance of network protocols inside the network. A drawback of the protocol booster architecture is that program modules are directly executed in kernel-space (without sandboxing). As a consequence, strong authentication mechanisms are absolutely vital when loading the boosters in order to prevent malicious code from disrupting the network node. Unfortunately, this would conflict with the fact that in this case study arbitrary end users (without a security association) would need to download the smooth handoff booster. In addition, it is unclear from the design documentation how the protocol

[4]The simplicity of application level active router implementations usually facilitates fast prototyping.

booster approach deals with booster interaction problems since explicit composition structures (for example, an underlying program or a booster graph) are lacking.

The remaining discrete active router architectures introduced in chapter 3, namely LARA and ANN, are both neglected in this comparison as the main focus of these architectures lies in the hardware design of the routers.

7.3.2 Requirement Fulfilment

This section recaps the architectural requirements of active routers previously introduced in chapter 4 and evaluates whether they have been successfully met by the LARA++ architecture. The list of general active network requirements, consisting of the vital (or class A) requirements and the long-term (or class B) requirements, has been reduced as the LARA++ specific requirements (L.X) encompass most of the general requirements. Table 7.1 lists the relevant requirements and indicates to what extent the individual requirements have been satisfied. Since the table gives only a rough indication of success or failure, a more thorough discussion complementing these findings follows:

Requirement	**Description**	**Satisfied?**
L.1	Flexible Extensibility	Yes
L.2	Moderate Performance	Partially
L.3	Highly Dynamic Programmability	Yes
L.4	Easy Usability	Yes
L.5	Safe Code Execution	Yes
L.6	Secure Programmability	Yes
L.7	Scalable Manageability	Yes
A.5	Resource Control	Yes
B.1	Interoperability	No
B.4	Business Model	Yes
B.5	QoS Support	Under Investigation

Table 7.1: LARA++ Compliance with relevant Active Network Requirements

Requirement L.1, which demands flexible extensibility of router functionality through active programmability, is one of the primary goals pursued by the architectural design of LARA++. While support for data plane programmability through transparent integration of active components in the packet processing chain makes LARA++ a highly extensible platform, the dynamic composition framework proposed by the architecture

provides a very flexible means for that. The fact that LARA++ introduces the concept of safe processing environments for active code eliminates the need for a specific or constraint programming language and thus enables "Turing complete" programmability. Furthermore, the LARA++ programming interface does not impose any restrictions on programmability. As node security is entirely controlled through policing, the LARA++ system API could expose the full low-level system interface to the active components without weakening the security on the node. Finally, LARA++ also provides a means to extend the programming environment on the active nodes by allowing users to create and download support libraries in the form of passive components.

The aim of requirement L.2 is to achieve router performance close to the line speeds of typical edge networks. Since performance of a LARA++ active router is obviously highly dependent on the implementation of the architecture, this requirement is hard to evaluate universally. Later in this chapter (in section 7.4.5), a performance estimate of our prototype implementation shows that LARA++ certainly has the potential to fulfil this requirement. From the architectural point of view, there is certainly a trade-off between modularity and performance. Since LARA++ tries to maximise flexibility through the concept of (de-)composition of active services, it trades off performance. For example, data packets traversing a node must be passed to all the components that indicate interest in the packet. However, LARA++ can also be programmed in ways that trade modularity for performance. Active service developers can balance between the comprehensive filter-based composition model provided for active components and the lightweight plugin-like approach used for passive components. Furthermore, the LARA++ architecture has been carefully designed to maximise performance where possible. Because of this, the idea of application level active networking has been rejected, and a true network level approach has been employed instead. For the same reason, LARA++ enables the safe execution of efficient binary code rather than having to rely on code interpretation. Also, the LARA++ safety and security architecture has been optimised by moving expensive security tests (for example, public key authentication) to the initialisation and start-up routines, whereas very lightweight mechanisms are used at run-time. For example, the active component identifier (ACID), which is assigned at start-up time after proper authentication of the component, in conjunction with the active component data structure (ACDS) allows very fast lookups of security policies at run-time (see also section 5.8.1).

Highly dynamic programmability (L.3) is accomplished through the filter-based composition model proposed by LARA++. Since active programmability is merely a matter of inserting (or removing) active components into (from) the packet processing chain on the node, which is essentially done through loading (or unloading) component code into the PEs and inserting (or removing) packet filter(s) into the classification graph, it is a

highly dynamic process that can be efficiently carried out at run-time without the need for restarting or even reconfiguring the system. For example, it will be shown that loading and instantiation of LARA++ components can be less than a millisecond assuming the code is already stored and cached on the node or less than fifty milliseconds if the component is loaded the first time from permanent storage (without the caching effect). Note that when active components are initially (up)loaded onto a LARA++ node, additional checks regarding the code's integrity (i.e., verification of the code signature) and the authorisation of the loading operation are required.

Requirement L.4, which demands easy usability, must be considered from the point of view of the end-users and developers. A key architectural decision that benefits end-user usability is that LARA++ enables transparent service enhancements and customisation – without the need to modify the end-user applications or systems. LARA++ programmability can simply be carried out by means of additional software or software extensions for existing programs. Since this approach of network programmability can be totally transparent, the correspondent nodes or applications of a user will not require any changes to the protocol stacks or applications. As the LARA++ framework completely separates 'active router programmability' from 'component development'[5], it tries to maximise usability for both tasks. While the former is essentially done through the provision of a simple configuration file that specifies the actual component code, potential loading properties (i.e., the provider), run-time configurations, etc. and a download/distribution mechanism for the configuration file, the latter involves the development of the component software. Even though the programming of a LARA++ router by end-users is simply a matter of adapting the configuration file, it is expected that support tools and/or application extensions will assist the users in this process. Usability from the developers' point of view, by contrast, is very much dependent on the actual implementation of the LARA++ architecture and the support tools provided for component developers. In the case of our particular implementation of LARA++, the development task is greatly facilitated by the design decision to execute active code within the user space processing environments, which allows convenient development of user-space code. This in conjunction with the fact that the LARA++ architecture is conceptually language independent prevents developers from having to learn specialised programming languages and development tools (i.e., compilers, debuggers).

Safety for active code execution (L.5) within LARA++ is achieved through software fault isolation. Specialised processing environments are used to protect the active NodeOS and other active components from malicious or erroneous active code (see also section 5.5.2). These processing environments form the management units for system-

[5]In fact, the LARA++ framework encourages third party development of active and passive components, and establishes a business role for component service providers.

level resource control (A.5). They prevent LARA++ components from using memory allocated to other processing environments or locking up a node through consumption of all processing resources. More fine-grained resource control is achieved by means of the system call control and policing component. They allow LARA++ to control access to all resources including logical resources, such as routing tables and other system configurations. In our particular implementation of the LARA++ architecture, high reliability and performance for the safe processing environments is achieved through the use of hardware supported system-level protection mechanisms, namely the virtual memory manager and the processor task scheduler.

Secure programmability (L.6) of active routers for end users is commonly controlled by means of user authentication (i.e., only authorised users can program a node) or code signing techniques (i.e., only code with a valid signature is accepted). The LARA++ architecture combines both approaches to enable the specification of flexible policies (for example, network administrators are allowed to install unsigned code, whereas user X can only install code signed by Y). The fact that LARA++ supports safe active code execution (L.5) – even for potentially un-trusted code[6] – enables active programming for unknown users and/or code. This is especially valuable for scenarios with a large user base where authentication of individual users is hard to manage, as limited programmability can be granted despite the lack of authentication.

While support for safe programmability in addition to the security measures (i.e., user authentication and code signing techniques) benefits scalability of the architecture (as no per-user state is necessarily required for all users), scalable manageability (L.7) is also favoured through a policy notation that supports aggregation (whilst retaining the flexibility of fine-grain policing). Policy aggregation through grouping of policies of a domain (i.e., users, code producers, etc.) largely reduces the number of policies required to express the rules for common domain members. This is particularly important as scalable manageability within active networks is mainly concerned with the management of security policies for potentially very large domains of users and active code. The provision of default policies (for unknown users or unsigned code) enables manageability even for extremely large domains.

Interoperability (B.1) among active network architectures and applications has been recognised as a long-term requirement for active networks that becomes important as active router solutions are more widely deployed. Today, the only wide-spread deployment of various active network technologies takes place within the ABone [BR99] virtual active network. The ANEP protocol used for active packet encapsulation in the ABone provides the basis for interoperability. Unfortunately, this explicit de-multiplexing ap-

[6]Note that the LARA++ processing environments fully protect the active NodeOS and other processing environment (even from malicious active code).

proach is very inflexible (see section 2.4.1) and demands explicit alteration of the data streams (i.e., encapsulation). Since this restricts the evolution of execution environments (support for versioning is lacking) and limits service composition (only one execution environment can be targeted), it conflicts with the LARA++ requirements L.1 and L.4. Nevertheless, despite the fact that the LARA++ architecture does not promote ANEP for those reasons, it could certainly be used to provide ANEP functionality. The filter-based interaction with the data path obviously enables de-multiplexing of active packets based on the ANEP identifier. The individual ABone execution environments would simply have to be ported to LARA++ active components. Furthermore, the ability to transparently integrate active network functionality and services into existing networks enables LARA++ to co-exist and complement other active network systems – even without formal interoperability through ANEP.

The component-based approach to active networks proposed by LARA++ lays the foundation for a new business model (B.4). It divides the market segment of current router vendors into those of *active router vendors* (selling the programmable node hardware and ActiveOS), *active component developers* (selling the active component software), and *active component providers* (selling the service of providing the active components). The component model proposed by the LARA++ architecture promotes the fast evolution of active network software. The fact that LARA++ components are modular and can be transparently integrated into the data path minimises the need for standardisation and facilitates dynamic software upgrades (i.e., new versions of a component can simply be loaded and instantiated on the routers). The short design-development-deployment cycle of LARA++ components provides quick feedback on the performance of a component, which accelerates the evolution of the software and reduces the time-to-market. Also, as LARA++ components do not necessarily rely on standardisation or 'formal' cooperation with others, they can be quickly developed by a small number of people at low cost. In conclusion, LARA++ promotes a free market approach to the traditionally closed market segment of network software, as individual software components compete in economic terms on the free marketplace (which is also accessible to small companies) rather than politically in standardisation committees (where small companies have less impact). Although network vendors will most certainly defend their current market dominance as long as possible, transformation will become inevitable as soon as the first active and programmable routers become established and the market starts to open up. Businesses will soon stop investing in proprietary solutions that exclude them from the benefits of the free market.

QoS support (B.5) within active networks is another less stringent requirement at this stage as long there are not yet wide-scale deployments. The evolution of the Internet, for example, shows that QoS support was not an issue for a long time and has just

recently started to be deployed in real environments. However, since network vendors and service providers have now started to support and roll out QoS within the networks, a solution to provide an equivalent level of service through active network nodes will be required. Unfortunately, the problem of providing QoS in the context of active networks is exacerbated, as active computation potentially within several active routers along the transmission path must be considered in addition to the QoS properties of conventional data networks. Although QoS support within the LARA++ architecture has not yet been fully defined, an early study [SS00] has been carried out to investigate the peculiarities of QoS support within active routers in general and in particular the LARA++ architecture. The paper identifies the need for node internal service classes and reservation mechanisms that are at least as "strong" as those of external QoS mechanisms (i.e., DiffServ [BBC+98] and IntServ [BCS94]). In order to simplify the mapping between external and internal QoS classes, the work proposes scheduling mechanisms for node-local network and processing resources that are semantically equivalent to those used along the end-to-end communication path. A skeleton for the implementation of such a system for the LARA++ active node architecture has been provided as part of this study.

7.4 Quantitative Evaluation

This section presents the experimental results of the LARA++ prototype implementation. Sections 7.4.1 to 7.4.4 evaluate the performance of individual components and mechanisms, namely the active component scheduler, the packet channels, the packet classification, and the component loading. Finally, section 7.4.5 tries to add up these results to estimate the overall performance of the prototype implementation.

The performance results presented throughout this section are based on the prototype implementation for Windows 2000 simply because this implementation was further advanced than the Linux implementation at the time of experimentation. Similar results are expected from the Linux based prototype due to the high resemblance in the implementation approaches.

The hardware architecture of the prototype LARA++ router is based on a standard PC consisting of a single Athlon XP 1600+ processor running at a clock speed of 1.4 GHz, 256 MB of RAM and three 100 Mbps Fast Ethernet interfaces (unless stated otherwise).

When considering this evaluation, it should be kept in mind that the results are based on the initial prototype implementation without any special hardware support. It is expected that the performance could be substantially improved when developed as a commercial product.

7.4.1 Active Component Scheduler

This section evaluates the performance gain resulting from the LARA++ active thread scheduler based on the prototype LARA++ implementation for Windows 2000 (see also section 6.5.3). The aim of the active component scheduler is to reduce the scheduling times for active threads by providing a specialised user-level thread scheduler as part of the processing environments. Low scheduling overhead for active component processing is considered especially important, as the LARA++ processing model assumes that active services or enhanced network functionality are decomposed into (many) lightweight active components. The reduction of scheduling overhead benefits the execution of trusted active components within a single processing environment.

The measurements presented below compare the performance of the LARA++ active thread scheduler with the standard Windows 2000 system thread scheduler. Figure 7.3 shows the performance gain resulting from the lightweight active thread scheduler. These results indicate that the LARA++ scheduler performs context switches between active threads approximately one order of magnitude faster than Windows 2000 switches between normal user threads. Even context switches among kernel-level threads are about 5-6 times more expensive.

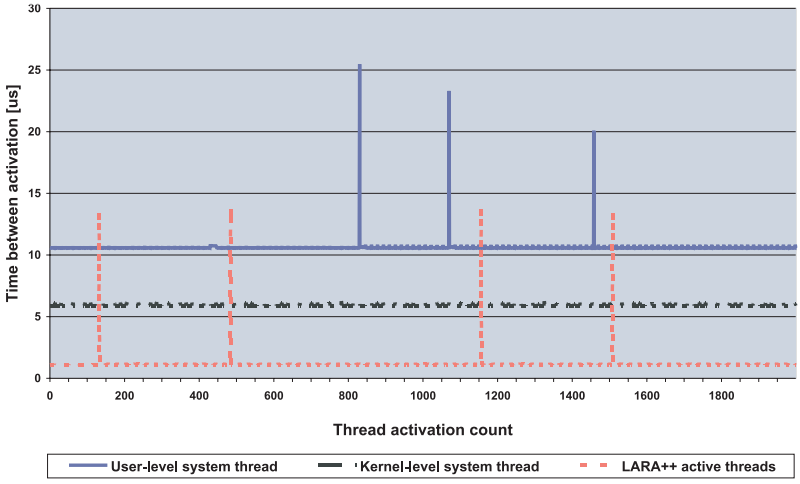


Figure 7.3: A comparison of context switch times for different threading approaches: While Win2K user-level threads take about 10.5 μs (avg) to schedule a thread out and in (inclusive of housekeeping), kernel-level threads require half the time, approximately 5-6 μs (avg). LARA++ active threads, by contrast, require only in the order of 1 μs (avg).

The measurements are based on simple test programs (i.e., a standard Windows 2000 application, a Windows 2000 kernel-level driver and a LARA++ processing environment) that measure the times between activation of two competing threads of the same kind (i.e., user-level system threads, kernel-level system threads and active threads respectively).

The results of this experiment indicate that the performance of the LARA++ component architecture, which strives to decompose large active applications into several lightweight active components (each running in a separate active thread), remains adequate in case of an efficient implementation of the scheduling mechanism. The fact that context switches among active threads can be approximately one order of magnitude faster than among ordinary threads shows that increasing the number of software components of a similar order should not decrease the performance much.

It should be noted that this experiment was carried out on the initial hardware platform used for LARA++ development and testing – a standard Intel Pentium II 233 MHz PC equipped with 128 MB of RAM. However, since the focus of this experiment lies on the relative performance gain of the LARA++ component scheduler, rather than absolute performance results, repeating the experiment on the current hardware platform is unnecessary.

7.4.2 Packet Channels

This section evaluates the performance of the LARA++ packet channel implementation under Windows 2000. The packet channels, providing the data transfer mechanism between the classifier and active components (see section 6.3.3), play a central role within the LARA++ architecture. Since all packets that require some form of active processing need to be transferred between the classifier and the active component(s), the performance of the packet channels is critical. The fact that LARA++ executes the active components within the safe processing environment in user-space aggravates the problem as the packet channels must pass the data between different memory protection domains. However, since the processing environments exploit virtual memory mechanisms to provide safety, LARA++ is able to circumvent expensive copy operation for passing data between the different protection domains by applying a virtual memory mapping technique (see section 6.3.3 for further details).

To evaluate the performance of the packet channels, the following measures are considered: processing load and latency. While the former measures the load increase on the node as a result of the channel processing, the latter captures the average channel processing latency for packets passing the node. The measurements try to quantify the processing time and load required to (1) pass a packet "up" to an active component, (2) perform the minimum amount of housekeeping required to receive and send packets[7], and (3) transfer the data back "down" again.

The first set of experiments estimate the performance penalty arising from the mem-

[7] This includes the time an active component requires to receive and send the data; it involves copying a packet descriptor from the input channel to the output channel. It also accounts for the average waiting time for the active thread to be activated.

ory mapping mechanism. For this, we compare the processing load on a Windows 2000 based router in two cases: (a) *without* LARA++ support whereby packets are simply forwarded by the default routing mechanism (Figure 7.4), and (b) *with* LARA++ support (Figure 7.5). In the latter case, the active NodeOS intercepts all data packets[8] and passes them to the test component (i.e., the packet memory is mapped into the address space of the corresponding PE). The test component performs only the minimum house keeping operations to process a packet before it passes the packets back down to the NodeOS, which then un-maps the virtual memory and finally forwards the packet as in the first experiment.

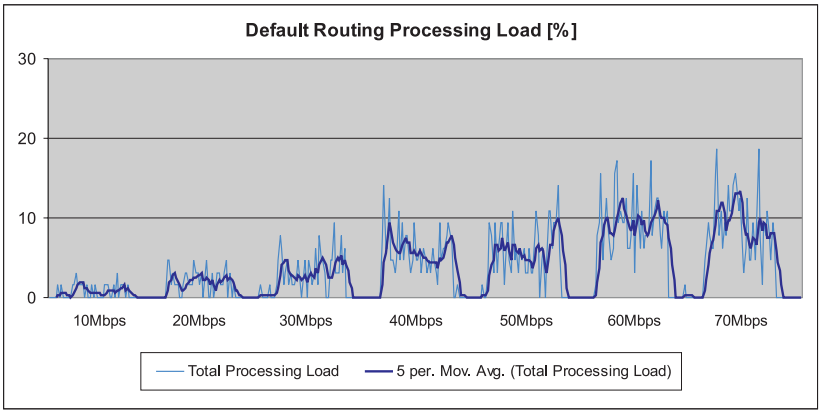


Figure 7.4: The processing load on a standard Win2K router while forwarding packets under various network loads.

A comparison of Figures 7.4 and 7.5 shows that mapping network traffic demanding active computation into user-space approximately doubles the processing load on the router compared to default forwarding. The fact that LARA++ enables full data path processing in user-space for approximately the same cost as the router takes for standard packet forwarding shows that the packets channel implementation is very efficient. However, it also shows the limitations of the current prototype. Assuming 50% of the processing resources would be reserved for the active computations on an active node, only the remaining 50% would be available for the packet handling by the LARA++ component framework. Extrapolating the graphs in Figure 7.5 to about 50% of the total processing load shows that a single processor active node of such power (i.e., Athlon XP 1600+) could only cope with approximately 150 Mbps (or 12500 packets per second[9]). For the current prototype implementation this would imply that on average only approximately 1.5 active components could be processed for every packet if line speed of typical edge networks (of the order of 100 Mbps) is required.

[8]Note that the classifier is completely circumvented in this experiment in order to estimate the processing load of the packet channel processing, not the packet classification.

[9]Assuming a packet size of 1500 bytes.

Nevertheless, considering that the current implementation is merely a research implementation without any performance optimisations or special hardware support (for example, the LARA hardware architecture Cerberus), the results seem to be reasonable as a solution for edge networks (see section 7.4.5 for further discussion).

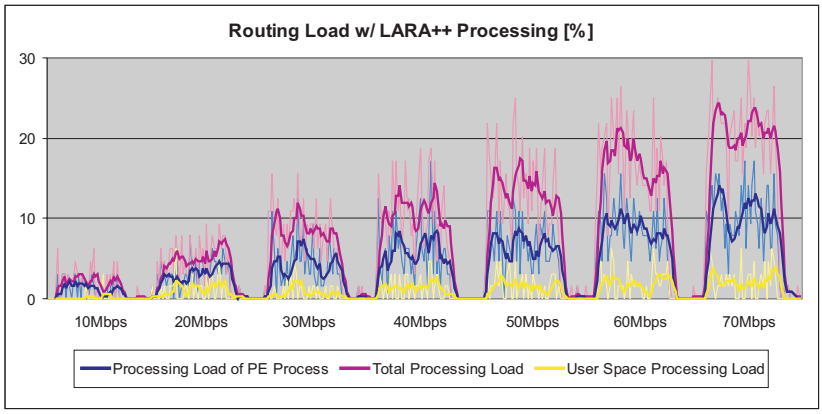


Figure 7.5: The processing load on a Win2K LARA++ router that passes all data to a user-space processing environment for active processing before it forwards the packets as in the first experiment (see Figure 7.4).

The second experiment tries to capture the latency introduced by the packet channel mechanism. Since this latency correlates with the overall processing load on the node, the experiment will estimate the average latency under various loads.

The latency is measured by adding a timestamp to the internal packet data structure when the packet is queued in the input channel of an active component. Upon completion, or when the active NodeOS continues to process the packet, a second timestamp is taken. The latency is simply computed as the time difference between these timestamps. Figure 7.6 presents the results under various processing loads.

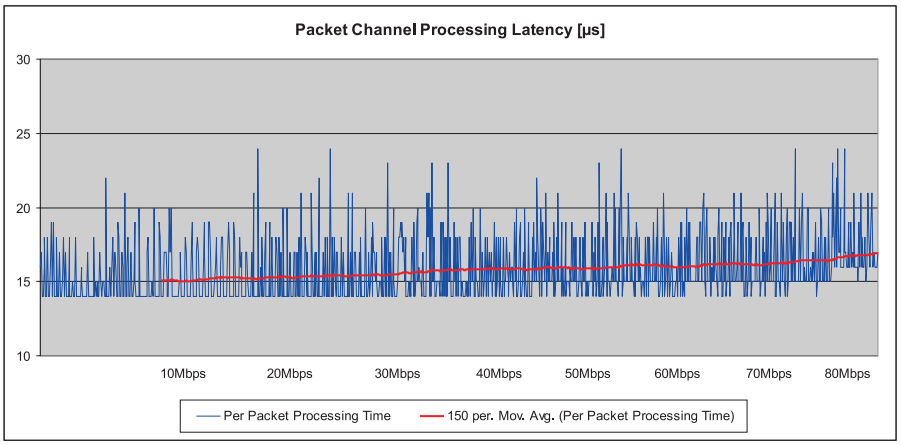


Figure 7.6: Average Latency caused by Packet Channel Processing (under increasing loads)

The figure shows the per-packet processing time averaged over 1000 packets. While the average latency increases from 15 μs to 17 μs, the upper bound is consistently just under 25 μs.

A closer analysis of the time distribution is provided in table 7.2. This shows that the processing times are not much affected by the throughput. Even under high load (i.e., 80 Mbps throughput), more than 95% of the packets are processed in less than 25 μs, and the number of packets exceeding 225 μs increases only by approximately 0.2%.

Throughput	Average Processing Time [μs]	< 25	25-49	50-74	75-99	100-149	150-224	> 225
10 Mbps	15.13	987.23	5.23	0.73	0.59	1.32	4.39	0.50
20 Mbps	15.43	983.82	8.07	0.59	0.15	0.29	5.56	1.52
30 Mbps	15.83	975.53	15.66	1.04	0.18	0.32	6.14	1.13
40 Mbps	15.95	972.09	18.81	1.01	0.29	0.54	5.53	1.73
50 Mbps	16.03	971.61	19.88	1.46	0.30	0.61	4.14	2.00
60 Mbps	16.08	968.53	22.06	1.18	0.55	0.64	4.91	2.13
70 Mbps	16.44	962.90	27.25	1.30	0.50	0.28	5.71	2.06
80 Mbps	17.04	954.11	27.63	5.61	3.50	2.80	3.90	2.44

Table 7.2: Distribution of Packet Channel Processing Times (averaged over chunks of 1000 packets)

The fact that standard edge routers introduce delays of the order of 1 ms for normal packet forwarding indicates that the delays introduced by the LARA++ packet channels are reasonable. Even if a packet is processed by up to 20 active components, the processing cost of passing the packet up into user-space for active processing and back down again for re-classification (inclusive minimal housekeeping) for each active component delays the packet less than 1/2 ms on average.

7.4.3 Packet Classification

This section evaluates the performance of the LARA++ packet classifier. As the classifier is the central component of the service composition process, its performance is critical for the overall system performance. The classification performance is especially important, as it is the means to determine which active components must process a packet, and therefore all packets passing through a node must be classified (as opposed to only those that require active processing).

In order to estimate the processing cost of the classifier, an experiment has been set up to measure the average classification times of packets as they pass through the

classification graph. To get the net cost of the classification process (without any channel processing or active computation involved), the classifier filters the packets as normal, but circumvents the processing.

The experiment encompasses three different scenarios. Each of the scenarios operated over the same populated classification graph, albeit with different processing characteristics for each one. Scenario one involved a type of packet chosen so that the traffic passes through 5 classification nodes in the classification graph. The packets are checked against 10 general filters and 500 flow filters. The second scenario used a packet content that causes the traffic to pass through 10 classification nodes in the classification graph. The number of general filters was doubled in order to impose roughly the same processing load per node (as in test one), whereas the number of flow filters on the path was kept constant at 500. By comparing the throughput of the first and second tests, we expected to find the processing load to be proportional to the number of classification nodes through which the packet travelled. The third scenario involved the same packet format and number of general filters on the packet path as the second test, but the number of flow filters on the classification path was doubled. The objective of this experiment was to confirm that adding extra flow filters does not proportionally decrease performance, all other things remaining equal.

Figure 7.7 presents the average results of these three experiments calculated as an average over 5 million packets. As expected, the throughput roughly halved between scenario one and two because the number of classification nodes on the packet path doubled. The results show that the graph filters and general filters do not scale well. Fortunately, their numbers are not related to the number of users of the active router, and hence do not have to scale to large quantities. By increasing the number of users of an active router, mainly the number of flow filters will increase proportionally. Between scenarios two and three, the number of flow filters doubled, but performance was virtually unaffected. With an average drop in throughput of less than 1% between these scenarios, we have shown that the use of flow filters allows the classification model to scale well as the number of users rises.

Further experiments were performed with the aim of measuring the average and the maximum latency of a packet travelling through the classifier. Figure 7.8 illustrates a breakdown of the time taken to perform different stages of classification over an average of 5 million packets. Six states of processing have been selected to represent the complete passage of a packet through the classifier, and the figure shows how these states account for the total latency of a packet.

The sum of the processing time in each of the stages is equal to the total latency imposed on a per-packet basis by the classifier. The total latency imposed by the classifier on an individual packet is 9.4 μs on average. The maximum latency measured over the

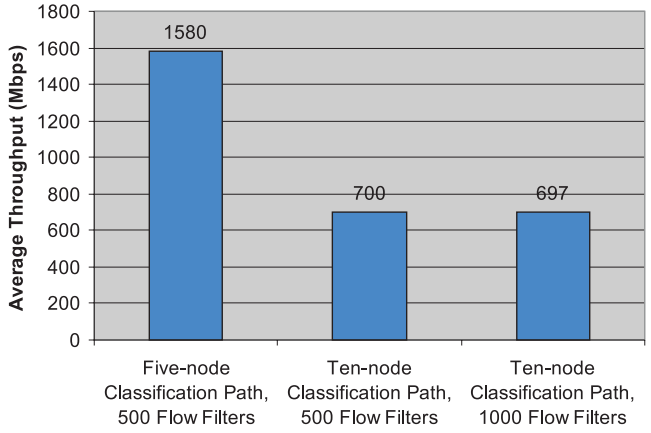


Figure 7.7: Classification Throughput (for pre-defined packet paths)

5 million packets was 273 μs. Since the average values measured during this experiment are almost identical with the minimum latency, it can be assumed that high processing latencies of the order of the maximum latency must be rare.

Unfortunately, the fine-grained timing mechanism we employed had the effect that it reduced the throughput of the classifier by approximately 35%. Therefore, we believe that it would be a fair assumption that the true latency figures would be a proportionate fraction of those presented in Figure 7.8 (i.e., approximately 6 μs for the average latency of a packet and about 180 μs as an upper bound).

Of the total latency, more than half is accounted for by the complexity of the classification graph. The complexity of the path through the classification graph undoubtedly has a direct impact upon the latency. Thus, the main determinant of the packet latency is the complexity of the packet itself. The majority of packets have only a MAC header, a network header, a transport header and a payload but rarely have many options. They would therefore take a "shortcut" through the classification graph, avoiding the extra classification nodes required to process these optional headers.

The average latency of the classification stage processing flow filters is comparatively small ($\sim$ 800 ns) in the context of the total packet latency. The measurements described above show that doubling the number of flow filters in the classification path reduces the throughput by less than 1%. The impact of such a proportion added to the latency of this classification stage would barely be noticeable. The classification latency of packets is therefore largely unaffected by a change in the number of flow filters installed on an active router.

The results show that the processing latency imposed by the classifier on an individual packet is as little as ten microseconds (depending on the complexity of the classification graph), which is a tiny fraction of the latency introduced by a normal edge router.

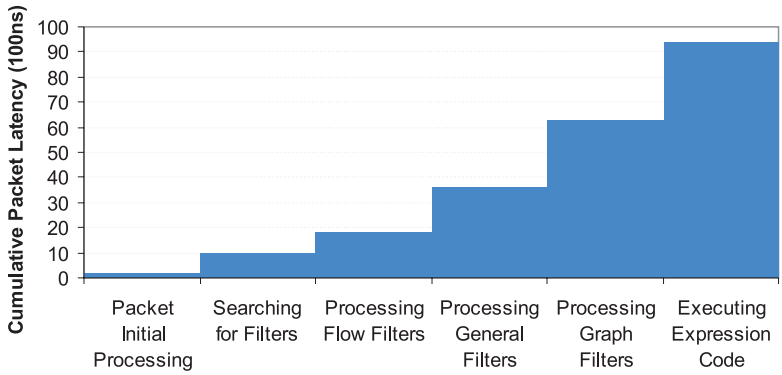


Figure 7.8: Classification Latency – Breakdown of the stages of packet classification, measured in tenths of microseconds. The classification graph and the population of the graph was chosen such that 500 flow filters, 10 graph filters and 12 general filters were checked. Of these, 2 flow filter, 2 general filters and 4 graph filters were matched.

This shows that the inclusion of the LARA++ composition framework has a negligible impact on the overall latency of the packets passing such a node. The results also demonstrate that the introduction of the flow filters allows the composition model to scale exceptionally well in terms of both the throughput of the active router and the latency of packets being routed and does so without the number of users significantly reducing the performance of the active node.

7.4.4 Component Loading

The quantitative evaluation of the component loader is impeded by the large variety of loading procedures. Depending on the situation and the active application at hand, different loading mechanisms are advantageous. Active code is either delivered in-band with the data flow or fetched out-of-band upon receipt of a loading request for an active component not yet loaded (cached) by the node. In the latter case, the active node contacts the active component server specified by the component URI to fetch the component code.

Active code that is loaded the first time must undergo a full security check. The code signature is verified to make sure that only original (or unaltered) code from trusted producers is accepted. Once the code is locally stored, the component must still be loaded (i.e., instantiated and initialised) by a processing environment. For this, the component loader checks the user's and code producer's authorisation, identifies a trusting processing environment (if none exists, a new processing environment must be created), and finally loads the component code into the processing environment and starts the active processing.

In order to evaluate the active component loader, an experiment has been set up to

measure the instantiation times of active components. Figure 7.9 shows the experimental results for varying component sizes based on our prototype LARA++ implementation. The experiment shows the average loading times over 50 runs. The results indicate that the loading times for LARA++ components increase only slightly with the code size. For typical component sizes (i.e., 50-1000 KB), the average loading times seem to be in the range 20-60 ms. The maximum value measured was of the order of 300 ms. This exceptionally high value was taken under full system load while accessing the disk.

The experiment was difficult to carry out since the second loading of a component (i.e., a cached component) resulted always in times less than a millisecond. Consequently, the test machine had to be rebooted each time the experiment was carried out in order to obtain representative results. Fortunately, this effect has a positive impact on LARA++ when deployed in a real-world scenario. It allows the instantiation of cached LARA++ components (for example, components that have been previously loaded) of the order of a few milliseconds.

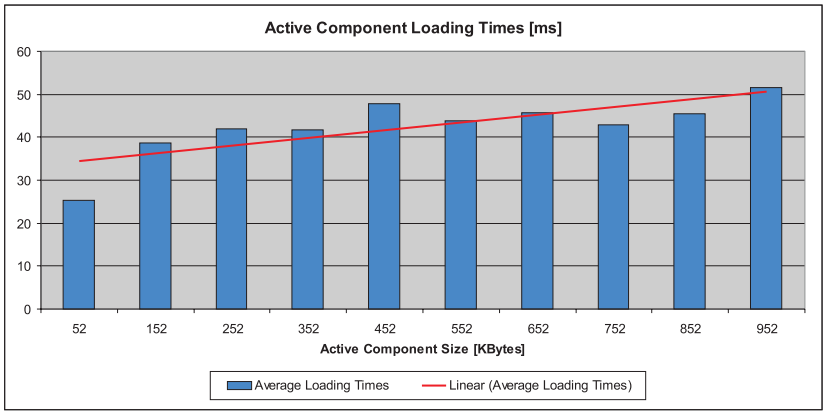


Figure 7.9: Average Component Loading Times

Although the experiment does not consider the times required for fetching component code and verifying code integrity (i.e., code signature), this lower bound is still valuable, as applications that demand very short loading times usually have the code stored or cached on the active node. This is especially significant for applications that rely on very fast instantiation of active components such as, for example, the fast handoff mechanism described in section 7.3.1.3. In such cases, active code should be pre-loaded or cached such that code authentication and integrity tests can be suppressed.

7.4.5 Discussion

This section concludes the quantitative evaluation of the LARA++ prototype implementation. Previous sections have shown the performance results of the individual subsystems or mechanisms that constitute the LARA++ platform. For example, the

evaluation of the active thread scheduler (section 7.4.1) has shown that the cost of scheduling among active components can be very low. This enables LARA++ routers to cope with many concurrent active components, and hence supports the idea of a component-based architecture.

Finally, this last section tries to combine these results for an overall system analysis. In particular, it tries to estimate the overall processing cost (time and load) of the LARA++ active node framework. In order to estimate the overall packet handling cost for packets passing a LARA++ node, the following formula is used:

$$\begin{aligned} C_{Total} = {} & C_{Intercept} + C_{Classification}(P_i) + N(P_i) \times C_{ChannelProcessing}(P_i) \\ & + \sum_{n=1}^{N(P_i)} (C_{Scheduling} + C_{ActiveProcessing}(n, P_i)) + C_{Inject} \end{aligned} \tag{7.1}$$

The processing cost C can be either the processing time (T) or load (L). $N(P_i)$ expresses the number of active components to be processed for packet P_i.

The formula shows that the packet processing cost is vastly dependent on the packet itself, i.e. the number of active components that are involved in the processing of the packet and the nature of the active computations. Since the overall packet processing cost depends largely on the active computations ($C_{ActiveProcessing}$), which cannot be approximated as they in turn depend entirely on the actual active application (for example, active component tasks could range from simple monitoring to heavyweight data transcoding), only the "housekeeping" cost of the packet handling (see also section 7.4.2) is considered from now on. This simplification is feasible as the aim here is to evaluate the LARA++ active component framework (i.e., the performance of the packet processing path through a LARA++ router), rather than a particular active network application.

Furthermore, since the aim here is primarily to provide an estimate of the overall processing cost, the following additional simplifications of the cost function C_{Total} can be proposed:

- The cost of intercepting ($C_{Intercept}$) and re-injecting (C_{Inject}) packets is neglected[10] as both are marginal compared to the overall processing cost of a packet.
- The classification cost $C_{Classification}(P_i)$ is approximated by an average cost value

[10]Note that both operations involve only a few simple instructions (i.e., pointer arithmetic) in order to either intercept a packet from the network stack or to re-insert it back into the stack.

computed over a large set of packets in order to drop the packet dependency[11]:

$$\bar{C}_{Classification} \approx \frac{\sum_{i=1}^{j} C_{Classification}(P_i)}{j}; for j >> 1 \quad (7.2)$$

- Likewise, the channel processing cost $C_{ChannelProcessing}(P_i)$ is replaced by an estimated average $\bar{C}_{ChannelProcessing}$ for the same reason.
- The cost introduced by the component scheduler $C_{Scheduling}$ is disregarded as it has been found negligible compared to the active processing cost of a packet (see section 7.4.1).

As a result, the remaining "housekeeping" cost of the LARA++ packet handling is estimated as follows:

$$C_{TotalPacketHandling} \approx \bar{C}_{Classification} + N(P_i) \times \bar{C}_{ChannelProcessing} \quad (7.3)$$

Based on this cost estimation, the remainder of this section analyses the LARA++ system performance with respect to processing time and load. According to the experiment described in section 7.4.3, the classification times ($\bar{T}_{Classification}$) seem to be fairly small compared to typical packet processing times within routers. The results have shown that even for a moderate classification graph with many packet filters, the average classification times do not exceed the order of 10 μs. The average channel processing time ($\bar{T}_{ChannlProcessing}$), by comparison, turns out to be approximately double (see section 7.4.2).

The performance of the packet channels is of special interest as the channel processing cost occurs once for every active component that is involved in the processing of a packet (i.e., $N(P_i)$ times). Moreover, the channel processing time is also dependent on the system load. According to the results of section 7.4.2, the channel processing times can vary substantially. The times have ranged from the order of 10 μs to over 225 μs. Fortunately, the number of packets that have required more than 225 μs was as few as 0.2% under high load. Also, the overall channel processing times have shown to be fairly stable with respect to the system load. For example, the average processing time rose only by approximately 2 μs whilst increasing the throughput from 10 Mbps to 80 Mbps.

As a result of this, one can conclude that the packet processing time introduced by the LARA++ component framework is primarily dependent on the number of active

[11] Note that although the classification cost depends heavily on the type (or rather content) of packets, an average cost estimation is sufficient to determine the order of the overall packet processing cost, as this has only a small impact on the total cost.

components being involved in the packet processing path.[12] According to the measurements in previous sections, the average processing times can range from as little as a hundredth of a millisecond when no active components are involved (for classification only), to over approximately a tenth of a millisecond (with a few active components in the packet processing path) up to approximately half a millisecond (in the case of many active components being processed). Table 7.3 illustrates the order of processing latencies introduced by the current prototype implementation of LARA++. These estimates are computed according to formula 7.3 and the experimental results of section 7.4.2 and 7.4.3. A moderate classification graph and a medium load on the active router is assumed.[13]

	Number of ACs ($N(P_i)$)	Average Latency	Upper Bound (with a 95th percentile)
Classification only	0	$\sim$ 6 μs	$\sim$ 8 μs
Few ACs	5	$\sim$ 86 μs	$\sim$ 133 μs
Moderate # of ACs	10	$\sim$ 166 μs	$\sim$ 258 μs
Many ACs	20	$\sim$ 326 μs	$\sim$ 508 μs

Table 7.3: Approximate Processing Latencies introduced by LARA++

The processing load introduced by the LARA++ component framework is also primarily a result of the packet classification ($\bar{L}_{Classification}$) and channel processing ($\bar{L}_{PacketChannel}$). According to the results of section 7.4.3, the load of the classifier is comparatively small for data rates in the order of line speed of typical edge networks (i.e., 100 Mbps). Thus, the processing load is mainly determined by the packet channel mechanisms. Fortunately, it is the computationally less expensive classification procedure that is involved in the processing of every packet passing through the router, whereas the more heavyweight task of passing the packets to the active components is only introduced when some form of active computation is required.

As a result, one can conclude that the processing load on the node is primarily dependent on the number of packets being processed by active components and less on the total number of packets streamed through the node. According to the cost estimation (see formula 7.3) and the experimental results of section 7.4.2 and 7.4.3, the current Windows 2000 prototype implementation of LARA++ can handle of the order

[12] Note again that the type of active computation and the time for the active processing are completely omitted here in order to evaluate the LARA++ component framework independent from any particular active application.

[13] According to the results of section 7.4.2 and 7.4.3, this implies an average for $\bar{T}_{Classification} \approx 6$ μs and $\bar{T}_{ChannelProcessing} \approx 16$ μs, and an upper bound (with a 95th percentile) of approx. 8 μs for $T_{Classification}$ and about 25 μs for $T_{ChannelProcessing}$.

of 250 Mbps if every packet is on average processed only once by a lightweight active component.[14] However, assuming that 50% of the processor resources might be reserved for active computation, this would drop to about 125 Mbps. Table 7.4 estimates the number of active components that can be processed for each packet passing the prototype active router for different throughputs. It is assumed that 50% of the processor resources are reserved for active computation.

Throughput	Packets (per second)	$\bar{L}_{Classification}$ (measured)	Remaining Processor Resources	$\bar{L}_{ChannelProcessing}$ (measured for one AC per packet)	$\bar{N}(P_i)$ (Average # of ACs)
250 Mbps	$\sim$ 20800	$\sim$ 16%	$\sim$ 34%	$\sim$ 83%	$\sim$ 0.4
125 Mbps	$\sim$ 10400	$\sim$ 8%	$\sim$ 42%	$\sim$ 42%	$\sim$ 1
100 Mbps	$\sim$ 8400	$\sim$ 6.3%	$\sim$ 43.7%	$\sim$ 33%	$\sim$ 1.3
75 Mbps	$\sim$ 6250	$\sim$ 4.7%	$\sim$ 45.3%	$\sim$ 25%	$\sim$ 1.8
50 Mbps	$\sim$ 4200	$\sim$ 3.1%	$\sim$ 46.9%	$\sim$ 16.7%	$\sim$ 2.8
20 Mbps	$\sim$ 1670	$\sim$ 1.3%	$\sim$ 48.7%	$\sim$ 6.7%	$\sim$ 7.3
10 Mbps	$\sim$ 840	$\sim$ 0.6%	$\sim$ 49.4%	$\sim$ 3.3%	$\sim$ 15

Table 7.4: Estimation of how many active components can process a packet on average for different throughputs.

From these results, it can be concluded that without performance improvements (for example, through hardware upgrades or a specialised hardware such as proposed by the LARA Cerberus design), LARA++ is either restricted with respect to its performance or the degree of modularity. Figure 7.10 illustrates this trade-off.

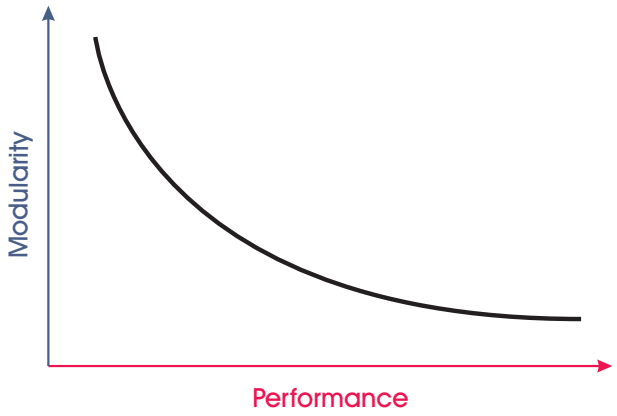


Figure 7.10: Trade-off between modularity and performance

As a result, LARA++ allows administrators to configure a node's performance and

[14]Note that according to section 7.4.2 and 7.4.3, the packet channel mechanism uses approximately 80% of the processing resources to handle 250 Mbps, while the classifier takes somewhat less than the remaining 20% to classify this amount of data per second.

modularity attributes flexibly along the trade-off graph. A description of two reasonable configurations for LARA++ routers at either end of the scale between performance and modularity follows:

Performance optimised (close to line-speed): This configuration assumes that the bulk of the traffic is simply forwarded by the router, and active computation is applied only to a fraction of the overall traffic (i.e., only to certain packets or data streams). In order to achieve performance of the order of line-speed, the router must support mainstream protocols such as IPv4 as standard network stacks, but may provide experimental functionality such as Mobile IPv6 through active components.

Fully componentised (fraction of line-speed): A router configuration on the other end of the spectrum provides all functionality (i.e., even mainstream protocols) in the form of active components. While this approach facilitates flexible extensibility (i.e., even extension to mainstream protocols is simply a matter of uploading active components), the performance is significantly lower than line-speed.

As a final remark it is important to note that the trade-off here is not between performance and flexibility (although flexibility and modularity are somehow related). Flexibility within LARA++ is obtained by means of the filter-based composition solution which allows transparent enhancement of existing protocols and extension of node functionality even without much degradation of the system performance. Performance deficiency is only correlated to the amount of active processing that takes place.

7.5 Summary

This chapter has presented the evaluation of the LARA++ active router architecture and its component-based prototype implementation. Section 7.2 started with a discussion on the evaluation approach. A qualitative evaluation has been considered best suitable for the design evaluation of the LARA++ architecture, whereas a quantitative evaluation has been chosen to evaluate the particular implementation of LARA++ described in the thesis.

The qualitative evaluation in section 7.3 has been split into two parts: the first part (section 7.3.1) demonstrated based on a real case study to what extent the LARA++ active router architecture enables rapid implementation and dynamic roll-out of novel network services. This theoretical evaluation has identified two typical problems of wireless access networks, namely access control for public accessible networks and smooth network handoffs between different wireless networks, and demonstrated how both of these challenges can be resolved by means of LARA++ active routers.

The second part of the qualitative evaluation (section 7.3.2) has considered whether or not the LARA++ architecture has fulfilled the functional requirements that have been introduced in chapter 4. Apart from a few requirements, namely L.2 and B.5, which have been partially satisfied so far, only requirement B.1, which demands support for interoperability with other (traditional) active network approaches, could not be satisfied. A justification was given as to why this requirement conflicts with other (more important) requirements imposed on the LARA++ architecture (for example, flexible extensibility and easy usability).

Finally, the evaluation of the LARA++ architecture was complemented by a quantitative evaluation of the prototype implementation (section 7.4). This performance analysis has examined the individual modules and mechanisms that constitute the LARA++ platform. The goal was particularly to measure the most important factors affecting the node, namely the processing time (latency) and load introduced by the individual components. Summarising the performance results, section 7.4.5 tried to estimate the overall system performance by combining the results analytically. The results clearly indicate a trade-off between performance and modularity. However, since LARA++ is primarily designed for use in edge networks, where performance is typically a secondary issue, resolution of this trade-off does not significantly compromise the flexibility and extensibility features of the component architecture. Furthermore, it has been shown that LARA++ can be flexibly configured (on a per-service basis) towards either performance or modularity. For example, service composition based on passive components makes LARA++ very efficient in cases where tight coupling of components through explicit bindings does not unnecessarily limit extensibility.

Chapter 8

Conclusion and Further Work

8.1 Overview

This final chapter recapitulates the work that has been carried out as part of this research effort. It concludes the thesis by summarising the main contributions of the work and drawing the conclusions that could be gained from the design and development of the LARA++ active router architecture.

The first part of this chapter provides an overview of the thesis structure and a summary of each chapter. This is followed by a synopsis of the main contributions. A series of conclusions summarises what has been learnt from this work, and how these experiences contribute to the wider field of research. Finally, a discussion is provided on how this work is being taken further. This describes ongoing projects at Lancaster, which use the LARA++ architecture and/or prototype implementation as an element of their research.

8.2 Thesis Summary

Chapter one of this thesis set the scene by unfolding the evolution of data networks from the early days of the Internet until today. It continued introducing the concepts of active networking and describing the problems of today's computer networks that have led to the establishment of this new research area. It provided a discussion of the research motivation for the field, highlighting the need for network-side processing capabilities and the potential beneficiaries of such a technology. Finally, it presented a summary of the research goals and challenges that are taken on by this work.

Chapter two examined the general background and issues of active and programmable networks. It defined the basic methodology and introduced different architectural approaches towards network programmability, namely active packets and active extensions.

A discussion on the impact of network-side data processing on the current network landscape followed. The chapter then continued with a discussion of various programming models (for example, different code distribution and encoding mechanisms) and other key aspects such as service composition and system integrity.

Chapter three provided a comprehensive overview of the current state-of-the-art in the field of active networking. A large number of different projects were presented, describing their architectural approach, application domain, and the distinction of their design. The variety of projects can be divided into three categories, namely active network and node architectures, enabling technologies, and active applications and services. In the first category, active network solutions following both the integrated and discrete approach, were discussed. The majority of projects presented, however, pursue the discrete approach – like the LARA++ architecture itself. This approach distinguishes itself from the integrated approach in that active programs are downloaded out-of-band or on-demand onto the active nodes (rather than inline with the actual data packets) where they provide "persistent" services to a particular flow or a set of flows. Unlike the integrated approach, whereby conventional packets are replaced by active packets that contain both code and data, this approach is far more practical. For example, it does not restrict programmability in terms of code size and latency that is required for code authentication and execution on every intermediate node. Furthermore, end-to-end flows are not required to be augmented with active code, which enables transparent network programmability or extensibility of functionality. The second category compared related operating systems support and further enabling technologies such as safety and security mechanisms. Finally, the third category concluded this chapter with an overview of recent work in the area of active network applications and services. This overview emphasised the potentials of active networks by presenting a range of novel mechanisms and enhancements, which rely on programmability support inside the network.

Chapter four continued with a requirement analysis for active network systems in general and for LARA++ in particular. The requirements are derived from a thorough study of related work in the field of active networks and past experiences at Lancaster, namely experiences gained during the development of the predecessor platform LARA and the LANode active node architecture. General factors, for example the commercial aspects such as the deployment of active technologies, are also taken into consideration. A differentiation between the absolutely vital requirements and the more long-term requirements for an active network architecture was made. From this multitude of general requirements a subset of requirements, which were considered important for the design of a flexibly extensible active edge node, was drawn. These specialised requirements form the basis for the subsequent LARA++ active router architecture and implementation.

Chapters five and six presented the bulk of the contributions made in this thesis,

namely the LARA++ active router architecture and the prototype implementations of this architecture. Chapter five introduced the design of the LARA++ architecture encompassing the motivation for this novel component-based approach to network programmability and a description of the fundamental building blocks. The chapter also discussed the programming model that is pursued by LARA++ and its novel service composition framework. The composition framework proposed in the thesis enables network programmability through flexible extensibility of network functionality and services. In other words, it allows LARA++ nodes to be augmented with new or enhanced capabilities that are dynamically loadable onto the nodes (on demand), where they can be flexibly and transparently integrated into the data processing chain on the network device. Finally, chapter five provided an overview of the LARA++ safety and security framework. While the former encompasses safety measures for the execution of active code (for example, memory protection and processing resource scheduling), the latter covers policing techniques that allow fine-grained configuration of node security policies (potentially on a per-user basis).

Chapter six outlined how the LARA++ design has been being implemented in an ongoing effort. Described in this section is the development of the core software components of the LARA++ architecture. Due to the considerable extent of this architecture, the development work has focused on validating the core design decisions and key mechanisms (i.e., the packet classification, the safe processing environments, the policy enforcement, etc.) through a 'proof-of-concept' implementation. The closing part of this chapter described the development process (i.e., the debugging and testing) of LARA++ components, and presented a range of example components.

Chapter seven presented an evaluation of the proposed architecture and its prototype implementation. First, the evaluation took the form of a qualitative assessment. Based on a concrete case study and several example applications, it evaluated the design choices that were made during the development of the architecture and discussed to what extent they fulfil the requirements identified in chapter 4. The results were largely positive, since all but one LARA++ specific requirements could be fully satisfied. Requirement L.2, which demands support for active processing speeds close to the line speeds of typical edge networks, could only be partially satisfied, as it depends on the type and amount of active computation that is carried out on the active router. The second part of the evaluation assessed the prototype implementation of the LARA++ architecture in a quantitative manner. A performance analysis examined the individual components and mechanisms that constitute the LARA++ platform. This was concluded with an analytical study, which combined the individual performance results in order to estimate the overall performance of the prototype active router. Again, the results were affirmative.

Chapter eight finally concludes the thesis by bringing together the thread of arguments presented throughout this work. It highlights the contributions that have been made to the field of active networking and presents the conclusions that can be drawn from the design and development of the LARA++ active router architecture. Finally, this thesis finishes with a discussion of further work in this area and in particular with respect to LARA++. This encompasses ongoing research projects at Lancaster, which use the LARA++ prototype and gain from the understanding acquired by this research, confirming its value.

8.3 Contributions

This section summarises the main contributions and achievements of the research carried out as part of this thesis.

The overall goal of this work, namely to design a novel active router architecture that enables flexible extensibility of network functionality, has been successfully fulfilled in the form of the LARA++ architecture. The validation of the architectural design with respect to its feasibility and practicality has been accomplished through prototype implementations of the LARA++ active node architecture.

8.3.1 Extension of Architectural Framework for Active Nodes

During the design phase of the LARA++ architecture, it became clear that there is a strong need for the concept of a safe execution environment for active code in order to provide the level of security and reliability that is expected from routers. As a consequence, a specialisation of the conventional execution environment concept has been suggested, which guarantees resource isolation[1] (i.e., memory and processing resources) for active code executed in different trust zones. These so-called processing environments protect active computations running in different processing environments and the active NodeOS from malicious or erroneous active code (see also section 5.5.2). This extra layer of protection on top of the NodeOS extends the conventional architectural framework for active router as suggested by a DARPA-funded working group.

8.3.2 Component-based Active Router Architecture

The main contribution of this work is the component-based architecture for active nodes, whereby the distributed active programs are components themselves that allow flexible extensibility of the functionality on the target node. This modular approach to active

[1] Resource isolation can be achieved through a range of techniques, such as virtual memory and task scheduling mechanisms, safe code techniques, or a virtual machine.

networking allows complex active services to be "broken up" into many small functional components, which are reassembled on the active nodes by means of a service composition framework. The benefits of such a modular approach, namely flexibility of extensibility, ease of reconfigurability, and reusability of components, have been demonstrated throughout this work.

The remainder of this section highlights further advances and specialities of this architecture:

- *Generic Architecture*

  A fundamental goal of this work has been to design a truly generic active router architecture. A highly flexible architecture, which is by no means tied to a specific application or application domain, was envisioned. This has been accomplished by checking the suitability of the architectural design for a wide-spectrum of active network applications at several stages of the design phase. Furthermore, a platform independent architecture was aimed for. This has also been successfully demonstrated by means of the prototype implementations for both Windows 2000 and Linux, which are currently under way. An implementation for Windows CE is also being considered.

- *Flexible Programmability*

  LARA++ achieves maximum flexibility through support of full data path programmability. This enables the direct integration of new or enhanced functionality into the data processing path of the active routers. The classification-based service composition framework provides the management structures that allow users to flexibly extend the router's processing capabilities.

- *Moderate Performance*

  Satisfactory performance of the active node is accomplished by carefully designing the architecture in a way such that performance intensive operations are moved to infrequently called routines which are only called when absolutely necessary. For example, packet classification, which must be performed on every packet passing an active node, is directly processed in kernel-space (by the active NodeOS); whereas the active computations, which are processed in the more heavy-weight, but protected processing environments, only take place when there is a packet-filter match.

8.3.3 Flexible Service Composition Framework

Extensibility was another key objective of the LARA++ architecture. As a result, a mechanism that enables active nodes to flexibly incorporate new or enhanced func-

tionality in the packet processing chain was desired. The classification-based service composition framework provides such a mechanism. It allows programmers to define where in the overall packet processing chain on an active router their components should be incorporated. This mechanism has several advantages: first, it allows totally independent users to co-operatively take part in the service composition process (without having to know each other). The active components of the various users are simply processed according to the structures provided by the classification graph. Second, it enables dynamic (re)configuration of the service(s). Since composition is based on packet filters, services can be dynamically (re)configured by inserting/removing packet filters (i.e., adding/deleting active components). And third, it supports fine-grained service composition on a per-packet basis. As the bindings of the components are based on packet filters, they depend on the actual data of the packets passing through a node (for example, network protocol, network addresses, and transport ports). The conditional bindings define a service that may differ on a packet-by-packet basis.

8.3.4 Transparent Service Integration

The fact that active components are incorporated into the composition framework (i.e., the classifier) by means of packet filters makes service integration a transparent process. This is especially advantageous as it enables the addition of active computation inside the network in a totally transparent manner, i.e., there is no need to modify the data streams or the end-to-end protocols (for example by adding a special active network tag/header). Furthermore, since LARA++ is designed to provide standard routing/forwarding functionality in addition to the active service framework, conventional edge routers should be directly replaceable by a LARA++ node.

8.3.5 Policy-based Security Model

The LARA++ architecture also contributes a flexibly configurable security model for active network nodes. The fact that LARA++ uses policies to define user and/or component privileges allows active node administrators to flexibly configure the security level on a per-user and/or per-component basis. A set of different policy types is supported to tightly control the level of security; i.e., the installation of active components, the insertion of packet filters into the classification graph, and run-time security. In particular the latter provides a means for fine-grained access control to the system API and resources. Furthermore, support for grouping of individual entities with common characteristics (for example, users, components, and system calls) is provided to simplify the policy specification process through aggregation of common policies.

8.3.6 Scalable Manageability to Support End-user Programmability

A key problem of supporting active programmability for arbitrary end-users is the problem of scaling the management of user credentials, which are required to authorise installation and execution of active code on the network nodes. The LARA++ architecture circumvents this problem by enforcing strong safety measures during the execution of active code, such that even code originating from arbitrary (or potentially malicious) users could not harm the system. Thus, the safe processing environments in conjunction with restrictive 'default policies' for unknown users (i.e., users for which no credentials exist on the node) provide a scalable solution to this management problem.

8.3.7 Commercial Viability

The LARA++ active router architecture has been carefully designed for commercial viability: first, the increasing demand for flexible service deployment capabilities in the network creates a need for network devices that are open and programmable. As demonstrated throughout this thesis, the LARA++ component-based router architecture offers sufficient flexibility and extensibility to meet these requirements. It therefore has the potential to become the design archetype for future commercial routers. Second, the component-based programming model proposed by LARA++ strongly encourages third-party software development and service provisioning. This helps to break up the governing role of current router manufacturers into separate business roles, namely those of network software developers and network software providers. The fact that component developers and providers will then compete on the free market may lead to an even faster evolution of network capabilities and services, and more cost-effective solutions.

8.3.8 Prototype Platform Implementation

Finally, a prototype implementation of the LARA++ architecture has been developed in order to evaluate the novel design choices and mechanisms proposed (i.e., the service composition framework, the safe processing environments, the policy-based security model, etc.). Since it has not been feasible to implement the whole architecture, the focus was laid on the implementation of the critical mechanisms for 'proof-of-concept' (i.e., the packet classifier, the packet channels, the system call control, etc.).

The fact that one of the core design aims of LARA++ was to design and prototype a system that provides sufficient flexibility and extensibility makes it an ideal platform for research into active networking as well as the wider field of computer networks. Its value as a research platform is confirmed by a number of recent projects (further described in the following section) that use LARA++ as their base network platform.

8.4 Further Work

Besides the ongoing development efforts to complete the LARA++ prototype implementations, further work in this area focuses on using and extending the LARA++ architecture and prototype platform in order to build and experiment with novel active network services.

The Mobile-IPv6 Testbed and the ProgNet project are two examples of ongoing research that take advantage of the LARA++ architecture and platform.

8.4.1 Mobile-IPv6 Testbed

The Mobile-IPv6 Testbed [MSR01] is a commercially funded project for research into next generation mobile network protocols, services and applications. The industrial partners behind this new research facility are Cisco (as a network systems vendor), Microsoft (as a software company), and Orange (as a wireless service provider). The overall objectives of this collaboration between Lancaster University and these partners are to draw academia and industry closer in order to exchange their expertise and to develop next generation mobile network and service solutions.

The project strives to support research into many aspects of the general field of mobile computing, including context and location aware ubiquitous services for mobile users, enhanced support for mobile IPv6 networks (such as auto-configuration, ad-hoc networking, handoff optimisation and adaptive applications), and issues arising from QoS and network management in mobile environments.

A key activity of this collaborative project is to provision a real research environment in the form of an operational Mobile IPv6 network that is available to real user communities. The Testbed infrastructure, which is currently being set up, encompasses the University campus and the city centre of Lancaster, and will potentially reach as far as the edges of Lancashire and Cumbria through the CLEO network [For01a]. Remote networks at the partner sides will also be linked to the Testbed network. This environment with its multitude of potential users will provide the keystone for the experimentation and trialling of new services.

The use of active network technologies as proposed by LARA++ has many potential applications in diverse networks such as the Testbed infrastructure. One of the areas where LARA++ will play a major role is security for the access networks. Since large parts of the Testbed infrastructure will be accessible via wireless technologies (i.e., 802.11, Bluetooth, and GPRS), access control is absolutely crucial. The LARA++ active router platform is envisioned to replace the last hop router between the wireless terminals and the core Testbed network. Flexible programmability and extensibility at these edge nodes enable the deployment of network-side support for access control

[SFW+01] and mobile handoff optimisations [SFSS00a]. Both these services are planned to be implemented and deployed as part of this project. The fact that LARA++ enables the dynamic roll-out of new network-side functionality and services greatly facilitates the development and testing of such novel mechanisms.

Future plans also consider these LARA++ edge nodes to provide network-side QoS and accounting support, whereby the real cost is computed from the network resources that a user terminal consumes and from the delivered QoS. Finally, these edge nodes are also envisioned to provide location tracking functionality in order to support high-level context and location aware services.

8.4.2 ProgNet Project

The main aim of this project is to explore the benefits emerging from active and programmable network technologies when applied to mobile environments [PrN01]. One of the key research questions, for example, is to what extent low-performance mobile devices, which typically lack processing and bandwidth resources, can benefit from off-loading computationally expensive and/or bandwidth demanding tasks onto programmable devices inside the network.

The intention is first to develop a framework of low-level mobile network services, providing a platform for the development of generic mobile services and applications. This will be followed by the creation of a set of high-level applications for mobile and nomadic users in order to demonstrate the advantages of such a mobile services framework. The framework is envisioned to provide:

- Mobile network memory and file-system services (that provide the same "view" to the user at any access point on the network)
- Mobile code and remote execution environments (allowing small devices to off-load heavy-weight processing tasks into the network, where sufficient processing power and bandwidth resources are available)
- Support for mobile and transparent communication proxies residing inside the network (that assist mobile devices during handoff and disconnected times)

The LARA++ active router architecture has been chosen as the base platform for the development of this framework. LARA++ is expected to benefit greatly the development of such a distributed network-side system for several reasons: (1) its dynamic and transparent component-based programming model allows the deployment of new services inside the network, when and where needed; (2) the flexibly extensible service composition approach facilitates the introduction of new functionality anywhere in the

forwarding path of a node; (3) support for full programmability of the data path ensures sufficient flexibility to implement this framework; and finally (4) the concept of the processing environments ensures appropriate safety and node reliability for end-user driven network programmability.

The development work and experimentation with the mobile service framework carried out as part of this project is expected to lead to the following outcomes: first, a better understanding of the degree to which active and programmable network technologies can support mobile services and applications; and second, an answer to the question whether the proposed mobile network environment, including its mobile service framework and applications, can provide quantitative and qualitative advances for emerging network systems such as next generation mobile networks.

8.5 Concluding Remarks

Although recent evolutions within the field of data networking indicate a clear need for more flexibility and extensibility inside the network, it is not yet clear what form of open and/or programmable network devices will prevail. Currently, research into active and programmable networks is still for the most part an exercise for testing hypotheses about various programming models and open interfaces.

This thesis has provided evidence that network programmability on edge devices, where low to moderate performance is acceptable, is achievable in a cost-effective manner based on a software router. High performance active processing, by contrast, demands specialised hardware support as for example suggested by the predecessor platform LARA (section 3.2.2.5). And even then, it will remain a challenge to achieve active computations at lines speed of today's core networks. The question whether or not it will be feasible one day to perform line-speed active computation within the core of the network remains open, as both the network speeds and the processor speeds are still rapidly growing.

In terms of the original project goal, namely the design of a flexible and extensible active router architecture, and the implementation of a prototype platform for further network research, LARA++ has been a success. The fact that LARA++ already plays a central role within several recent research projects at Lancaster, especially MSRL [MSR01] and ProgNet [PrN01], underlines this achievement and the need for such a flexible research platform. This also proves that the LARA++ architecture is sufficiently generic for the different types of networking research undertaken by those projects.

Several conclusions about the development of active router architectures can be drawn from this work:

- A component-based active router architecture enables network programmability through extensibility of router functionality and services. The service composition framework determines the degree of extensibility (or in other words how flexibly the active router can be programmed).

- A classification-based service composition approach enables transparent network programmability. New or advanced network functionality can be flexibly integrated into the packet processing chain on the router simply by inserting a packet filters into the classification graph (the representation of the internal processing path on the router). Such a filter-based approach also facilitates dynamic composition of services. Moreover, the fact that services can be transparently composed (without having to know the interfaces of the neighbouring components) facilitates co-operative service composition through independent users. A (dynamic) means to extend the classification graph or in other words the composition model ensures sufficient extensibility for future network protocols and services.

- Active network programmability demands sophisticated safety mechanisms to protect the nodes from malicious or erroneous active code and to provide the reliability familiar in conventional network devices. The concept of the safe processing environments proposed by the LARA++ architecture provides such a mechanism.

- The introduction of default policies for unknown users in conjunction with the concept of flow-filters enables end-user programmability in a controlled and scalable manner. Since these policies allow end-users only to program their own flows[2], no per-user state (i.e., authentication and authorisation information) has to be managed on the active nodes.

From the LARA++ prototype implementations, one can conclude:

- Reuse of the standard 'process technology' of today's operating systems as safe execution environments for active code has proven to be very practical. The requirements for system protection from typical user programs on end nodes and active code in the case of active network nodes are similar.

- Implementation of a NodeOS that provides resource control and safety features requires the NodeOS to be tightly coupled with the underlying (host) operating system. For example, appropriate NodeOS support is required to control the system interface and to pass the network data into the appropriate user-space memory.

[2]Note that the problem of connecting network users to their data flows is not trivial. An authentication mechanism based on a token exchange between the active router and the end-terminal is considered.

- Therefore, a split implementation across both kernel and user space of the underlying system appears to be a good choice. This approach takes advantage of the highly optimised and sophisticated protection and safety mechanisms of today's operating systems.

- The virtual memory mapping mechanism developed for the LARA++ packet channel implementation has proven vital for active nodes that process active computations in user-space, when reasonable performance is required. Note that standard user-space implementations for active networks typically suffer largely from the performance hit resulting from the copy operations required to pass the network traffic "up" into user-space and back "down" again.

In summary, this thesis has demonstrated that active networking provides the necessary technology for vastly increasing the flexibility of current network systems. However, the results of this work – like most other work in the field to this day – provide only a piece of the overall mosaic. Further development will be required to bring together the various research contributions. Current technologies will not only need to be improved, but also combined to form complete systems in order to progress active networks from individual laboratories into commercial use.

The extra level of complexity that is required to enable programmability inside the network and the new potential security holes that result from this added flexibility has made for a sceptical audience. Active networking still has a long way to go before a definitive conclusion can be reached about its place in next generation communication networks. Furthermore, before active networks can be widely deployed on a large scale, such as the global Internet, more hurdles will have to be overcome: a standard will need to be defined. The greatest challenge of active networking might be to overcome this initial standardisation process that active networks themselves are supposed to circumvent when new functionality and services are introduced into the network. However, if this challenge can be overcome, there seems no immediate reason why active networks should not be the key technology for the future network infrastructure. In fact, active networking, in addition to the broader field of programmable networks, has many attractive features which make it suitable for adoption.

Appendix A

Common Packet Filter Properties

LARA++ packet filters a are defined by a set of properties. The following list describes the common properties of both *active component filters* and *graph filters*:

Type: The type property indicates the type of the filter (for example, *type*: **AC_FILTER** or **GRAPH_FILTER**).

Filter Pattern: The actual filters are expressed as lists of tuples of the form {**frame offset, bit pattern, bit mask**}. In order to facilitate filter specification, semantics on well-known packet structures and header fields can be used. For example, the **frame offset** can be defined relative to well-known reference points and pre-defined tags can be used to express fix values (i.e., offsets, patterns, etc.). The following example illustrates the use of the *filter pattern*: {**TP_HEADER,TCP_PROT_ID, TP_MASK**}. This defines a matching filter for all TCP packets.

Input Node: The input node defines at which point in the classification graph structure the filter should be insert. For example, a filter that wants to intercept all TCP packets with a particular transport protocol port (for example, TCP port 80) could define the filter pattern: {**TP_HEADER+TCP_DSTPORT,80,PORT_MASK**} and insert the filter at the *input node*: '**tcp**'.

Output Node: The output node defines the point in the classification graph where the packet classifier should continue the classification process upon successful matching this filter. This parameter is used in different ways for the distinct filter types. While AC filters may use the *output node* to define the next point in the classification process (for example, to define a *cut-through path* through the classification graph), graph filters use this parameter to define a new branch in the classification graph.

Active component filters require additional properties to indicate their association with a particular AC instantiation and the relevant authentication information:

Active Component Reference: AC filters also include a reference to the active component instance that "owns" the packet filter. The reference is required to enable the classifier to deliver the packet to the appropriate component.

Operation: The operation property expresses the access permissions required by the active component. Initially, only three operations are defined, namely **ro**, **rw** and **wo**. The first operation indicates that the AC only wants to observe (read-only) the filtered packets, whereas the second option is required when the AC wants to modify (read-write) the packet data as well. Finally, the third operation is used, when the AC does only want to send packets.

Principal: The principal indicates the network user who installs the filter and/or the code producer providing the active component. The principal's credentials in conjunction with the operation properties permit the classifier to check whether or not a filter confirms with the the node-local security policies and the privileges associated with that principal.

Appendix B

Policy Specification

Policy specification within LARA++ is divided into three parts. Each part governs the policies for one of the following protection realms: (1) instantiation of active components, (2) installation of packet filters, and (3) run-time control of active components. To optimise policy specification for these protection realms, LARA++ uses different notations. The remainder of this section describes the various policy types and illustrates how they are used.

B.1 Installation of Active Components

The LARA++ policies to safeguard the installation of active components have the following structure:

```
<users>[,<code producers>]   (allow|deny)   <components>
```

This rule allows node administrators to control active component installation based on the identity of the network user trying to install the component and the code producer of the component.

The following example illustrates how policies of this type can be used to safeguard active nodes from malicious users:

```
system administrators                allow   all
authorised users,trusted producers   allow   untrusted components
steve.jobs,microsoft                 deny    untrusted components
all,microsoft                        allow   untrusted components
researchers,cisco                    allow   all
```

The policy enforcement component applies these policies before a new component (active or passive) is installed. It processes the policy rule base from top to bottom

until any of the policies matches. Depending on the `allow` or `deny`, it either accepts or rejects the component. If none of the rules match, the default behaviour is to reject the component. As a consequence of this procedure, it is clear that the order of the policies in the database matters. For example, if the third rule would appear after the forth rule, the user Steve Jobs would not be precluded from installing untrusted Microsoft components.

B.2 Installation of Packet Filter

Policy rules to control the installation of packet filters in the classification graph have the following general form:

```
<users>[,<code producers>]  <type>  <operation>  <input node>[,<output node>]
```

These policies also use the network user who installed the active component and the code producer as criteria to whether or not the component is allowed to insert a certain packet filter into the classification graph. The `type` defines the type of packet filters (i.e., general or flow filter) addressed by this policy rule. The `operation` defines whether access to data packets is prohibited `[no]`, or whether data can be read `[ro]`, read and written `[rw]`, or only written `[wo]`. The `input node` is required to define where in the classification graph a packet filter can be installed. The optional `output node` can be used to allow components to "shortcut" the classification graph (see section 5.6.3). It defines the node in the classification graph where the matched packets will be reinserted after the respective component has completed the active processing. By default packets are reinserted at the point in the classification where they have been extracted to ensure that all components get the chance to process the packets.

The following example policies illustrate how this policy rule is used:

```
system administrators                 generic   rw   *
all,cisco                             generic   rw   /ipv6
lancaster researchers                 generic   rw   /ipv7,/netout
authorised users,trusted producers    flow      rw   *
all,trusted producers                 flow      ro   *
```

Note, the `*` is used to indicate all nodes within the classification graph.

The example demonstrates the flexibility and the power of this policy type. It allows node administrators to precisely define who can insert packet filters, what type of filters and where in the classification graph they can be inserted. The second rule, for example, allows components (produced by Cisco and installed by arbitrary users) to

insert a generic packet filter with read/write permissions at the `/ipv6` input node of the classification graph. The third rule, by comparison, enables the user group `lancaster researchers` to insert a packet filter for a hypothetical IPv7 protocol. This policy, for example, allows the group to safely develop a new network protocol without interfering with standard IPv4/v6 traffic. Note that the component inserting the generic packet filter can directly re-insert the processed data packets at the `/netout` node in the classification graph.

B.3 Run-time Security

Policy rules to control access to the system API and the node-local resources at run-time have the following form:

```
(<users>[,<code producers>] | <components>)   (allow|deny)    <system calls>
(<users>[,<code producers>] | <components>)   (grant|limit)   <resource:value>
```

Similarly to the policy rules defined previously, run-time policies can also use the network user and code producer as the basis to either allow or deny access to system calls or resources. The component type (i.e., system or user) and the properties (i.e., trusted, privileged, authenticated, etc.) can be used alternatively.

The following example shows how these policy types can be used to control access to the system API and resource usage:

```
all,trusted producers   allow   syscalls
privileged users        allow   disk syscalls
all                     deny    disk syscalls
system components       grant   processing:high-priority
all                     limit   processing:round-robin
user components         limit   bandwidth:100kbps
```

The first rule enables trusted components full access to the system API. The next two rules restrict access to disk related system calls only to privileged users. Rule four enables system components to request high-priority process scheduling, whereas all other components are constrained to round-robin scheduling. Finally, the last rule limits all user components to a maximum data rate of up to 100 Kbps.

References

[A+97] D.S. Alexander et al. Active Network Encapsulation Protocol (ANEP). Internet draft, IETF, July 1997. Work in progress.

[AAA+99] D.S. Alexander, K.G. Anagnostakis, W.A. Arbaugh, A.D. Keromytis, and J.M. Smith. The Price of Safety in an Active Network. In *Proceedings of ACM SIGCOMM*, 1999.

[AAH+98] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J.M. Smith. The switchware active network architecture. In *Proceedings of IEEE Network*, May/June 1998.

[AAKS98] D.S. Alexander, W.A. Arbaugh, A.D. Keromytis, and J.M. Smith. A secure active network environment architecture: Realization in SwitchWare. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):37–45, May/June 1998.

[AMZ95] Elan Amir, Steve McCanne, and Hui Zhang. An application level video gateway. *Proceedings of ACM Multimedia '95, San Francisco, CA*, 1995.

[ANW98a] Active Networks Working Group. E. Zegura (Ed.). Composable Services for Active Networks. Draft, September 1998.

[ANW98b] Active Networks Working Group. K.L. Calvert (Ed.). Architectural Framework for Active Networks. Draft, August 1998.

[ANW99] Active Networks Working Group. L. Peterson (Ed.). NodeOS Interface Specification. Draft, July 1999.

[ANW01] Active Networks Working Group. S. Murphy (Ed.). Secutiry Architecture for Active Networks. Draft, November 2001.

[AR94] F.M. Avolio and M.J. Ranum. A network perimeter privacy, with secure external access. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, Glenwood, ML, February 1994.

[ART94] System Management ARTS. SMARTS, 1994.

[ASNS97] D.S. Alexander, M. Shaw, S. Nettles, and J.M. Smith. Active Bridging. In *Proceedings of ACM SIGCOMM*, pages 101–111, 1997.

[ATLLW96] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, Philadelphia, PA, 1996.

[B+97] A. Banchs et al. Multicasting Multimedia Streams with Active Networks. Technical report 97-050, ICSI, 1997.

[B+98] J. Biswas et al. The IEEE P.1520 Standards Initiative for Programmable Network Interfaces. *IEEE Communications*, 36(10):64–70, 1998.

[B+01] B. Braden et al. Active Reservation Protocol (ARP) – Lessons Learned, December 2001.

[Ban01] M. Banfield. *Service creation combining programmable networks and open signalling technologies*. PhD thesis, Lancaster University, United Kingdom, 2001.

[BBC+98] D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, IETF, December 1998.

[BCE+94] B.N. Bershad, C. Chambers, S.J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E.G. Sirer. SPIN – An extensible microkernel for application-specific operating system services. In *Proceedings of 4th ACM SIGOPS European Workshop*, pages 68–71, 1994.

[BCF+01] B. Braden, A. Cerpa, T. Faber, B. Lindell, G. Phillips, J. Kann, and V. Shenoy. Introduction to the ASP Execution Environment (Release 1.5). Technical report, USC – Information Science Institute, http://www.isi.edu/active-signal/ARP/DOCUMENTS/ASP_EE.ps, November 2001.

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, IETF, June 1994.

[BCZ96] S. Bhattacharjee, K.L. Calvert, and E.W. Zegura. On Active Networking and Congestion. Technical report git-cc-96/02, 1996.

[BCZ98] S. Bhattacharjee, K.L. Calvert, and E.W. Zegura. Self-organizing wide-area network caches. In *Proceedings of IEEE INFOCOM (2)*, pages 600–608, San Francisco, CA, March 1998.

[Ber00] S. Berson. A Gentle Introduction to the ABone, October 2000.

[BFK98] M. Blaze, J. Feigenbaum, and A.D. Keromytis. Keynote: Trust management for public-key infrastructures (position paper). In *Proceedings of the Security Protocols International Workshop*, pages 59–63, Cambridge, U.K., 1998.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 17th Symposium on Security and Privacy. IEEE Computer Society Press*, pages 164–173, 1996.

[BH99] G. Back and W. Hsieh. Drawing the Red Line in Java. In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.

[BLFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L Masinter. Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396, IETF, August 1998.

[Bor94] N. Borenstein. Email with a Mind of its Own: The Safe-Tcl Language for enabled Mail. In *Proceedings of IFIP International Conference*, Barcelona, Spain, 1994.

[BR99] B. Braden and L. Ricciulli. A Plan for a Scalable ABone – A modest proposal. Technical report, USC – Information Science Institute, ftp://ftp.-isi.edu/pub/braden/ActiveNets/ABone.whpaper.ps, January 1999.

[Bro81] L. Brodie. *Sarting FORTH*. Prentice Hall, 1981.

[BTS+98] G. Back, P. Tullmann, L. Stoller, W.C. Hsieh, and J. Lepreau. Java Operating Systems: Design and Implementation. Technical Report UUCS-98-015, University of Utah, August 1998.

[BZB+97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification. RFC 2205, IETF, September 1997.

[C+96] R. Caceres et al. Fast and Scalable Handoffs for Wireless Internetworks. In *Proceedings of ACM Mobicom*, pages 56–66, 1996.

[C+98] P. Chandra et al. Darwin: Customizable resource management for value-added network services. In *Proceedings of Sixth IEEE International Conference on Network Protocols (ICNP)*, Austin, TX, October 1998.

[C+00] A. Cambell et al. Cellular IP. Internet Draft draft-ietf-mobileip-cellularip-00.txt, IETF, 2000. Work in progress.

[CAM] The Caml Language. Online reference, Institut National de Recherche en Informatique et en Automatique, http://caml.inria.fr/.

[CDK+99] A. Campbell, H.G. De Meer, M.E. Kounavis, K. Miki, J. Vieente, and D. Villela. The Genesis Kernel: A virtual network operating system for spawning network architectures. In *Proceedings of the 2nd Workshop on Open Architectures and Network Programming (OpenArch)*, NewYork, NY, March 1999.

[CDM+00] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Developing a context-aware electronic tourist guide: some issues and experiences. In *Proceedings of CHI*, pages 17–24, Netherlands, April 2000.

[CFSS99] R. Cardoe, J. Finney, A.C. Scott, and W.D. Shepherd. LARA: A prototype system for supporting high performance active networking. In *Proceedings of the First International Working Conference on Active Networks (IWAN)*, volume LNCS 1653, pages 117–131, Berlin, Germany, 1999. Springer-Verlag.

[CGK+00] A. Campbell, J. Gomez, S. Kim, A. Valko, C. Wan, and Z. Turanyi. Design, implementation, and evaluation of Cellular IP. *IEEE Personal Communicaitons*, 7(4), August 2000.

[CHLL96] M. C. Chan, J.-F. Huard, A.A. Lazar, and K.-S. Lim. On realizing a broadband kernel for multimedia networks. *Lecture Notes in Computer Science*, 1185:56–64, 1996.

[COM] Information and Resources for the Component Object Model-based Technologies. Online reference, Microsoft Corporation, http://msdn.microsoft.-com/com/.

[CVP99] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 140–153, 1999.

[DAR98] DARPA Active Network Research Program. Active Networks CBD Reference (BAA #98-03), 1998.

[DCMF99] N. Davies, K. Cheverst, K. Mitchell, and A. Friday. Caches in the Air: Disseminating Information in the Guide System. In *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, New Orleans, LU, February 1999.

[DDLS01] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Proceedings of Policy Worshop*, pages 18–38, Bristol, U.K., 2001.

[DDPP98] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plug-ins: A software architecture for next generation routers. In *Proceedings of ACM SIGCOMM*, pages 229–240, September 1998.

[DP98] D. Decasper and B. Plattner. DAN – Distributed Code Caching for Active Networks. In *Proceedings of IEEE INFOCOM (2)*, San Francisco, CA, March 1998.

[DPP99] D. Decasper, G. Parulkar, and B. Plattner. A Scalable, High Performance Active Network Node. *IEEE Network*, January 1999.

[Dro97] R. Droms. Dynamic Host Configuration Protocol. RFC 2131, IETF, March 1997.

[dS98] S. da Silva. Netscript Tutorial (Version 0.10). Online reference, Columbia University Department of Computer Science, October 1998.

[dSFY98] S. da Silva, D. Florissi, and Y. Yemini. Composing Active Services in NetScript, March 1998.

[DVW92] W. Diffie, P.C. Van Oorschot, and M.J. Wiener. Autentication and Authenticated Key Exchanges. *In Designs, Codes, And Cryptography*, 2(2):107–125, 1992.

[EF94] K. Egevang and P. Francis. The IP Network Address Translater (NAT). RFC 1631, IETF, May 1994.

[EHK96] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 131–144, 1996.

[FG99] M. Fry and A. Ghosh. Application Level Active Networking. *Computer Networks*, 31(7):655–667, 1999.

[FKFH00] A. Fernando, B. Kummerfeld, A. Fekete, and M. Hitchens. A new dynamic architecture for an active network. In *Proceedings of the 3rd Workshop on Open Architectures and Network Programming (OpenArch)*, Israel, 2000.

[FMS+98] D. Feldmeier, A. McAuley, J.M. Smith, D. Bakin, W. Marcus, and T. Raleigh. Protocol Boosters. *IEEE Journal on Selected Areas in Communications (Special Issue on Protocol Architectures for 21st Century Applications*, 16(3):437–444, April 1998.

[For01a] B. Forde. CLEO – Cumbria and Lancashire Education Online. Presentation at BECTa Wireless Networking for Education, Lancaster University, http://www.becta.org.uk/technology/techseminars/250101/cleo.pdf, January 2001.

[For01b] Bluetooth Forum. Specification of the Bluetooth System – Core. Specification volume 1, http://www.bluetooth.com/pdf/Bluetooth_11_-Specifications_Book.pdf, February 2001.

[FZ81] R. Forchheimer and J. Zander. Softnet – Packet Radio in Sweden. In *Proceedings of AMRAD Conference*, 1981.

[G+00] B. Gleeson et al. A Framework for IP Based Virtual Private Networks. RFC 2764, IETF, February 2000.

[GBB+01] D. Glynos, C. Boukouvalas, P. Bosdogianni, K. Ahola, A. Juhola, and P.A. Aranda-Gutierrez. Mobile Active Mail. In *Proceedings of the Third International Working Conference on Active Networks (IWAN)*, volume LNCS 2207, Philadelphia, USA, September 2001. Springer-Verlag.

[GFC00] A. Ghosh, M. Fry, and J. Crowcroft. An Architecture for Application Layer Routing. In *Proceedings of the Second International Working Conference on Active Networks (IWAN)*, volume LNCS 1942, pages 71–86. Springer-Verlag, October 2000.

[Gho00] A. Ghosh. FunnelWeb v2.0.1. Online reference, http://dmir.socs.uts.edu.-au/projects/alan/, 2000.

[GII01] GUIDE II: Services for Citizens. Research Project, EPSRC Grant GR/M82394, Lancaster University, 2001.

[GM95] J. Gosling and H. McGilton. The Java Language Environment, 1995.

[Gos95] J. Gosling. Java Intermediate Bytecodes. In *Proceedings of SIGPLAN Workshop on Intermediate Representations (IR95)*, San Francisco, CA, 1995.

[GR93] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1993.

[GY98] G. Goldszmidt and Y. Yemini. Delegated Agents for Network Management. *IEEE Communications*, 36(3):66–70, March 1998.

[HBB+99] J.J. Hartman, P.A. Bigot, P. Bridges, B. Montz, R. Piltz, O. Spatscheck, T.A. Proebsting, L.L. Peterson, and A. Bavier. Joust: A Platform for Liquid Software. *IEEE Computer*, 32(4):50–56, April 1999.

[HCC+98] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, 1998.

[Hj00] G. Hjlmtysson. The Pronto Platform - A flexible toolkit for programming networks using a commodity operating system. In *Proceedings of the 3rd Workshop on Open Architectures and Network Programming (OpenArch)*, Israel, 2000.

[HK99] M. Hicks and A. D. Keromytis. A Secure PLAN. In *Proceedings of the First International Working Conference on Active Networks (IWAN)*, volume LNCS 1653, pages 307–314, Berlin, Germany, 1999. Springer-Verlag.

[HKM+98] M.W. Hicks, P. Kakkar, J.T. Moore, C.A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of Third ACM SIGPLAN International Conference on Functional Programming*, pages 86–93, 1998.

[HL98] K. Hafner and M. Lyon. *Where Wizards Stay Up Late: The Origins of the Internet*. Simon and Schuster, 1998.

[HMA+99] M.W. Hicks, J.T. Moore, D.S. Alexander, C.A. Gunter, and S. Nettles. PLANet: An Active Internetwork. In *Proceedings of IEEE INFOCOM (3)*, pages 1124–1133, 1999.

[HMPP96] J.J. Hartman, U. Manber, L.L. Peterson, and T. Proebsting. Liquid Software: A new Paradigm for Networked Systems. Technical Report 96-11, Department of Computer Science, University of Arizona, June 1996.

[HN00] M.W. Hicks and S. Nettles. Active Networking means Evolution (or Enhanced Extensibility required). In *Proceedings of the Second International Working Conference on Active Networks (IWAN)*, volume LNCS 1942, pages 16–32. Springer-Verlag, October 2000.

[Hor00] L. Hornof. Self-specializing mobile code for adaptive network services. In *Proceedings of the Second International Working Conference on Active Networks (IWAN)*, volume LNCS 1942, pages 102–113. Springer-Verlag, October 2000.

[IDD] NDIS Intermediate Drivers. Online reference, developer resources, msdn - library, Microsoft Corporation, http://msdn.microsoft.com/library/.

[ISO84] ISO. Information Processing Systems: Open System Interconnection – Basic Reference Model. Technical report, ISO/IEC, 1984.

[JAV] Java 2 SDK, Standard Edition Documentation v1.3. Online reference, Sun Microsystems, http://java.sun.com/j2se/1.3/docs/index.html.

[JGS93] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[JOU] Joust. Online reference, University of Arizona Computer Science Department, http://www.cs.arizona.edu/scout/joust.html.

[KA98] S. Kent and R. Atkinson. Security Architecture for Internet Protocol. RFC 2401, IETF, November 1998.

[Kat97] D. Katz. IP Router Alert Option. RFC 2113, IETF, February 1997.

[LAN99] LAN MAN Standards Committee of the IEEE Computer Society. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE standard 802.11, 1999.

[Lan01] LandMARC – Lancaster and Microsoft Active Research Collaboration. Research Project funded by Microsoft Research (Cambridge), Lancaster University, http://www.landmarc.net/, 1999–2001.

[LG98] Ulana Legedza and John Guttag. Using network-level support to improve cache routing. *Computer Networks and ISDN Systems*, 30(22–23):2193–2201, 1998.

[LGT98] L.H. Lehman, S.J. Garland, and D.L. Tennenhouse. Active Reliable Multicast. In *Proceedings of IEEE INFOCOM (2)*, pages 581–589, 1998.

[Lin00] B. Lindell. Active Signaling Protocol (ASP) Execution Environment, October 2000.

[LL94] M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, Orlande, FL, June 1994.

[LWG98] U. Legedza, D. Wetherall, and J. Guttag. Improving the performance of distributed applications using active networks. In *Proceedings of IEEE INFOCOM (2)*, San Francisco, CA, April 1998.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing, 1996.

[Mar94] R.T. Marshall. *The Simple Book: An Introduction to Internet Management*. Prentice Hall, 1994. 2nd Edition.

[MBC+99] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and CANEs: Implementation of an Active Network. In *Proceedings of the 37th Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 1999.

[MBZC00] S. Merugu, S. Bhattacharjee, E.W. Zegura, and K.L. Calvert. Bowman: A Node OS for Active Networks. In *Proceedings of IEEE INFOCOM*, pages 1127–1136, 2000.

[MCH01] Laurent Mathy, Roberto Canonico, and David Hutchison. An Overlay Tree Building Control Protocol. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication (NGC 2001)*, volume LNCS 2233, pages 78–87, London, UK, November 2001. Springer-Verlag.

[MDCG99] G. Morrisett, D.Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[Men99] P. Menage. RCANE: A resource controlled framework for active network services. In *Proceedings of the First International Working Conference on Active Networks (IWAN)*, volume LNCS 1653, pages 25–36, Berlin, Germany, 1999. Springer-Verlag.

[MF01] G. MacLarty and M. Fry. Active Content Distribution Networks, September 2001.

[MHN01] J.T. Moore, M. Hicks, and S. Nettles. Practical Programmable Packets. In *Proceedings of IEEE INFOCOM*, pages 41–50, April 2001.

[MHWZ99] B. Metzler, T. Harbaum, R. Wittmann, and M. Zitterbart. AMnet: Heterogeneous multicast services based on active networking. In *Proceedings of the 2nd Workshop on Open Architectures and Network Programming (OpenArch)*, NewYork, NY, March 1999.

[MKJK99] R. Morris, E. Kohler, J. Jannotti, and M.F. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 217–231, December 1999.

[MMO+94] A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, T.A. Proebsting, and J.H. Hartman. Scout: A communications-oriented operating system. In *Operating Systems Design and Implementation*, page 200, 1994.

[MRPH01] L. Mathy M. Rennhard, S. Rafaeli, B. Plattner, and D. Hutchison. An Architecture for an Anonymity Network. In *Proceedings of 10th IEEE Intl. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE)*, Boston, MA, June 2001.

[MS00] K. El Malki and H. Soliman. Hierarchical Mobile IPv6 and Fast Handoffs. Internet Draft draft-elmalki-soliman-hmipv4v6-00.txt, IETF, 2000. Work in progress.

[MSR01] MSRL – Mobile-IPv6 Systems Research Lab. Research Project funded by Cisco Systems, Microsoft Research (Cambridge), and Orange Ltd., Lancaster University, http://www.mobileipv6.net/, 2001.

[NDI] The Network Driver Interface Specification (NDIS) Interface. Online reference, developer resources, msdn - library, Microsoft Corporation, http://msdn.microsoft.com/library/.

[Nec97] G.C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 1997.

[Nec98] G.C. Necula. *Compiling with Proofs*. PhD thesis, 1998.

[Nel91] G. Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[NET] Netfilter. Online reference, http://www.netfilter.org/.

[NGK99] E. Nygren, S. Garland, and M.F. Kaashoek. PAN: A high-performance active network nodei supportingi multiple mobile code systems. In *Proceedings of the 2nd Workshop on Open Architectures and Network Programming (OpenArch)*, pages 78–89, NewYork, NY, March 1999.

[NK98] S. Nilsson and G. Karlsson. Fast Address Look-up for Internet Routers. In *Proceedings of IEEE Broadband Communications 98*, April 1998.

[NL96] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, October 1996.

[Nok98] Nokia. General Packet Radio Service (GPRS) – nokia's vision for a service platform supporting packet switched applications. White paper, 1998.

[OCA] The Objective Caml System Release 3.04 – Documentation and user's manual. Online reference, Institut National de Recherche en Informatique et en Automatique, http://caml.inria.fr/ocaml/htmlman/.

[OLW98] J.K. Ousterhout, J.Y. Levy, and B.B. Welch. The Safe-Tcl Security Model. *Lecture Notes in Computer Science*, 1419:217–235, 1998.

[One99] W. Oney. *Programming the Microsoft Windows Driver Model*. Microsoft Press, 1999. ISBN 0-7356-0588-2.

[OSK] OSkit. Online reference, University of Utah Computer Science Department, http://www.cs.utah.edu/projects/flux/janos/summary.html.

[oST94] National Institute of Standards and Technology. Digital Signature Standard. Technical Report FIPS-186, U.S. Department of Commerce, May 1994.

[Per96] C. Perkins. IP Mobility Support. RFC 2002, IETF, October 1996.

[Per01] C. Perkins. Mobility Support within IPv6. Internet Draft draft-ietf-mobileip-ipv6-15.txt, IETF, 2001. Work in progress.

[Pos81a] J. Postel. Internet Control Message Protocol. RFC 792, IETF, September 1981.

[Pos81b] J. Postel. Internet Protocol. RFC 791, IETF, September 1981.

[PrN01] ProgNet: Programmable Network Support for Mobile Services. Research Project, EPSRC Grant GR/R31461/01, Lancaster University, 2001.

[Rac00] N.J.P. Race. *Support for Video Distribution through Multimedia Caching.* PhD thesis, Lancaster University, United Kingdom, September 2000.

[Rie99] E. Riedel. *Active Disks – Remote Execution for Network-Attached Storage.* PhD thesis, Carnegie Mellon University, November 1999.

[RWS00] N.J. Race, D.G. Waddington, and D. Shepherd. A Dynamic RAM Cache for High Quality Distributed Video. In *Proceedings of Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS)*, volume LNCS 1905, Enschede, The Netherlands, October 2000. Springer-Verlag.

[S+98] B. Schwartz et al. Smart Packets for Active Networks. Technical report, BBN Technologies, http://www.net-tech.bbn.com/smtpkts/-smart.ps.gz, 1998.

[S+01] S. Schmid et al. An Access Control Architecture for Metropolitan Area Wireless Networks. In *Proceedings of Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS)*, volume LNCS 2158, pages 29–37, Lancaster, U.K., September 2001. Springer-Verlag.

[SA95] Z. Shao and A.W. Appel. A type-based compiler for standard ML. In *Proceedings of ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, CA, 1995.

[SBSH01] S. Simpson, M. Banfield, P. Smith, and D. Hutchison. Component Selection for Heterogeneous Active Networking. In *Proceedings of the Third International Working Conference on Active Networks (IWAN)*, volume LNCS 2207, pages 84–100, Philadelphia, USA, September 2001. Springer-Verlag.

[SFG+96] J.M. Smith, D.J. Farber, C.A. Gunter, et al. SwitchWare: Accelerating network evolution. Technical Report MS-CIS-96-38, University of Pennsylvania, 1996.

[SFSS00a] S. Schmid, J. Finney, A. Scott, and D. Shepherd. Active Component Driven Network Handoff for Mobile Multimedia Systems. In *Proceedings of Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS)*, volume LNCS 1905, pages 266–278, Enschede, The Netherlands, October 2000. Springer-Verlag.

[SFSS00b] S. Schmid, J. Finney, A. Scott, and D. Shepherd. Component-based Active Networks for Mobile Multimedia Systems. In *Proceedings of NOSSDAV*, Chapel Hill, NC, June 2000.

[SFW+01] S. Schmid, J. Finney, M. Wu, A. Friday, A.C. Scott, and W.D. Shepherd. An Access Control Architecture for Microcellular Wireless IPv6 Networks. In *Proceedings of the 26th IEEE Conference on Local Computer Networks (LCN)*, pages 454–463, Tampa, FL, November 2001.

[Sha99] Z. Shao. Typed cross-module compilation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 34(1), pages 141–152, 1999.

[SJS+00] B. Schwartz, A.W. Jackson, W.T. Strayer, W. Zhou, R.D. Rockwell, and C. Partbridge. Smart Packets: Applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1):67–88, 2000.

[SL99] M. Sloman and E. Lupu. Policy Specification for Programmable Networks. In *Proceedings of the First International Working Conference on Active Networks (IWAN)*, volume LNCS 1653, pages 73–84, Berlin, Germany, 1999. Springer-Verlag.

[SMSF97] J.S. Shapiro, S.J. Muir, J.M. Smith, and D.J. Farber. Operating system support for active networks. Technical report, 1997.

[SRC84] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SS00] S. Schmid and A. Scott. QoS Support within Active LARA++ Routers. In *Proceedings of GEMISIS*, Manchester, U.K., March 2000.

[SSL] Secure Socket Layer SSL. Online reference, Netscape Corporation, http://www.netscape.com/security/techbriefs/ssl.html.

[SSV99] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification using Tuple Space Search. In *Proceedings of ACM SIGCOMM*, pages 135–146, 1999.

[Ste94] W.R. Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley, 1994. ISBN 0–201–63346–9.

[SVSW98] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast Scalable Level Four Switching. In *Proceedings of ACM SIGCOMM*, 1998.

[TCM98] S. Thibault, C. Consel, and G. Muller. Safe and Efficient Active Network Programming. In *Proceedings of Symposium on Reliable Distributed Systems*, pages 135–143, 1998.

[THL01] P. Tullmann, M. Hibler, and J. Lepreau. Janos: A Java-oriented OS for Active Networks. *IEEE Journal on Selected Areas of Communication*, 19(3), March 2001.

[TL98] P. Tullmann and J. Lepreau. Nested Java Processes: OS Structure for Mobile Code. In *Proceedings of 8th ACM SIGOPS European Workshop*, Sintra, Portugal, 1998.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[TMM99] S. Thibault, J. Marant, and G. Muller. Adapting Distributed Applications using Extensible Networks. In *Proceedings of International Conference on Distributed Computing Systems*, pages 234–243, 1999.

[TS97] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997. New York: ACM.

[Tsc99] C. Tschudin. An Active Networks Overlay Network (ANON). In *Proceedings of the First International Working Conference on Active Networks (IWAN)*, volume LNCS 1653, pages 156–164, Berlin, Germany, 1999. Springer-Verlag.

[TSS+97] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, 1997.

[TW96] D.L. Tennenhouse and D.J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), 1996.

[Van97] V.C. Van. A Defence against Address Spoofing using Active Networks. M.eng thesis, Massachusetts Institute of Technology, June 1997.

[Ver02] D.C. Verma. *Content Distribution Networks – An Engineering Approach*. Wiley, 2002. ISBN 0–471–44341–7.

[VRLC97] J. Van der Merwe, S. Rooney, I. Leslie, and S. Crosby. The Tempest – A practical framework for network programmability. In *Proceedings of IEEE Network*, November 1997.

[Wet95] D. Wetherall. Safety Mechanisms for Mobile Code. Internal, Examination Paper, Telemedia networks and Systems Group, Laboratory for Computer Science, MIT, 1995.

[WGT98] D.J. Wetherall, J. Guttag, and D.L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of IEEE OPENARCH*, April 1998.

[WJGO98] I. Wakeman, A. Jeffrey, R. Graves, and T. Owen. Designing a Programming Language for Active Networks, June 1998.

[WLP98] P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and modal-ML. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–235, 1998.

[WT96] D.J. Wetherall and D.L. Tennenhouse. The ACTIVE IP option. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[WVTP97] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proceedings of ACM SIGCOMM*, pages 25–36, September 1997.

[YCN99] J.-H. Yeh, R. Chow, and R. Newman. A dynamic interdomain communication path setup in active network. In *Proceedings of the First International Working Conference on Active Networks (IWAN)*, volume LNCS 1653, Berlin, Germany, 1999. Springer-Verlag.

[YdS96] Y. Yemini and S. da Silva. Towards Programmable Networks. In *Proceedings of IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Aquila, Italy, October 1996.

[ZF83] J. Zander and R. Forchheimer. Softnet – An approach to high-level packet communication. In *Proceedings of AMRAD Conference*, San Francisco, CA, 1983.