SELF-CONFIGURING SERVICES FOR EXTENSIBLE NETWORKS – A ROUTING-INTEGRATED APPROACH

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZÜRICH

for the degree of
Doctor of Technical Sciences

presented by

RALPH M. KELLER

Dipl. Informatik-Ing. ETH
born March 14, 1971
citizen of Rorbas (ZH)

accepted on the recommendation of
Prof. Dr. Bernhard Plattner, examiner
Dr. James P. G. Sterbenz, co-examiner
Prof. Dr. Jonathan S. Turner, co-examiner

2004
Abstract

During the last decades the original Internet architecture evolved dramatically with new functionality being added to the network layer to support a wide range of emerging applications. Network services such as firewalls, congestion control, media gateways, and traffic engineering all require a network that not only forwards packets based on the destination address, but also performs packet processing on nodes interior to the network. In an effort to support such application-specific packet handling requirements, router manufacturers have started to embed programmable elements into routers for providing network service functionality in a more flexible way. However, deploying new services in an existing network is usually a manual and time consuming process requiring the installation of code on multiple routers distributed all over the network. Given the complexity of how services can be composed, the only feasible approach is to automate this process. For this reason, it is crucial to have a suitable service infrastructure built on top of the raw processing capabilities to enable programmability of each node.

This thesis presents a service framework that allows router resources to be programmed and coordinated in such a way that the underlying network provides the anticipated services on behalf of applications. We have developed the ANCS (Active Network Control Software), which can be seen as an additional control layer in an active network environment that offers a generic service abstraction and automates the configuration of processing resources to form network services. Our system accepts processing demands from applications, maps their processing requirements onto the available network resources, and configures appropriate resources on network nodes.
In this thesis we focus on all the control mechanisms needed by such a service framework. Firstly, we propose active pipes as a high-level programming interface to the active network. An active pipe models the processing requirements as a sequence of processing steps performed on a data flow, without the application having to know about the underlying topology and location of processing resources. A processing step can be either mandatory or optional, meaning that the execution can depend on the state of the network. Each processing step can have multiple attribute constraints refining the location of processing. Secondly, we describe a resource discovery protocol for the dissemination of information about processing resources. Our approach is based on extending a link-state routing protocol such as OSPF and distributing the processing capabilities as opaque link-state advertisements. Thirdly, we describe an algorithm that maps the processing requirements expressed as an active pipe onto the physically available network resources. This mapping algorithm solves the problem of finding the optimal location of all specified mandatory and optional processing steps including a path transiting the sites, while minimizing network costs. Since our solution optimizes for both link and processing costs, paths can become non-simple, meaning that a given node can be visited repeatedly. The runtime complexity of the algorithm is polynomial, and thus scales to large networks. Fourthly, we have designed a signaling mechanism for the installation of processing code on selected nodes along with the establishment of explicit forwarding state such that traffic gets routed through these nodes as determined by the mapping algorithm.

We have implemented our service framework along with all the necessary control operations and protocols on top of our modular and extensible PromethOS router architecture. We have demonstrated the viability of our approach in a realistic environment using two applications that benefit from network-interior packet processing. On behalf of a video distribution application, we deploy application-specific congestion control modules before congested links. Using our novel video scaling scheme, we show that the perceived video quality improves significantly compared to traditional best-effort packet queuing. In a second application, we implement a security gateway that performs data encryption on routers in a way completely transparent to end systems. Furthermore, our performance evaluation demonstrates that services can be established efficiently with minimal overhead.
Kurzfassung


In der vorliegenden Dissertation wird ein Framework zur Programmierung und Koordination von Routerressourcen vorgestellt, damit das darunter liegende Netzwerk entsprechende Dienste für Applikationen erbringen kann. Im Rahmen dieser These wurde das ANCS (Active Network Control Software) entwickelt, welches eine zusätzliche Kontrollschicht zur Erstellung von Diensten innerhalb eines aktiven Netzwerkes darstellt. Das vorgeschlagene System nimmt Anfragen zur Erstellung von Netzwerkdiensten entgegen, bildet die gestellten Verarbeitungsanforderungen auf die verfügbaren
Netzwerkressourcen ab und konfiguriert die entsprechenden Router, damit schliesslich das Netzwerk den gewünschten Dienst für die Applikation erbringt.


# Table of Contents

Abstract iii  
Kurzfassung v  
Table of Contents vii  
List of Figures xiii  
List of Tables xv  

## Part I: Concepts of Programmable Networks 1

1 Introduction 3  
  1.1 Motivation ................................................. 3  
  1.2 Programmable Networks .................................. 4  
  1.3 Challenges of Making Networks Programmable .......... 5  
  1.4 Problem Statement ........................................ 7  
  1.5 Contributions ............................................ 8  
  1.6 Outline .................................................. 9  

2 Background 11  
  2.1 Trends in Networking ..................................... 11  
  2.2 Demand for Programmable Service Infrastructure .... 13  
  2.3 Mobile Code Paradigm ...................................... 13  
  2.4 Active Networking ......................................... 14  
    2.4.1 Active Node Reference Model ................. 16  
    2.4.2 Packet Processing in Active Router ............ 17  
    2.4.3 Node Operating System ................... 20  
    2.4.4 Execution Environment .................... 20  
    2.4.5 Active Application ....................... 21  
  2.5 Active Services ........................................ 22  
    2.5.1 Service Specification ......................... 24  
    2.5.2 Resource Discovery ........................ 25
3 Related Work

3.1 Classification Criteria .................................................. 32
3.2 Node-Local Services ...................................................... 34
   3.2.1 Chameleon .......................................................... 34
   3.2.2 Bowman/CANes ................................................... 34
   3.2.3 Click .............................................................. 35
3.3 Active Packet-Based, In-Band Deployed Services .................... 35
   3.3.1 ANTS/PAN ......................................................... 36
   3.3.2 Smart Packs ..................................................... 36
   3.3.3 StreamCode ..................................................... 37
   3.3.4 SwitchWare/PLAN .............................................. 37
3.4 Active Packet-Based, Out-of-Band Deployed Services ............... 38
   3.4.1 ANN/DAN ......................................................... 38
3.5 Active Packet-Based, Hybrid Deployed Services .................... 39
   3.5.1 SENCOMM ....................................................... 39
3.6 Active Extension-Based Services .................................... 39
   3.6.1 Darwin .......................................................... 40
   3.6.2 Ninja Paths ...................................................... 42
   3.6.3 End-to-End Media Paths ....................................... 44
   3.6.4 HIGCS .......................................................... 45
3.7 Comparison ...................................................................... 47
   3.7.1 Node-Local Services .............................................. 47
   3.7.2 Active Packet-Based Services ................................... 48
   3.7.3 Active Extension-Based Services ............................... 49
3.8 Implications .................................................................. 51

Part II: Design of Service Framework ................................. 55

4 Active Network Control Software ........................................ 57
   4.1 Design Objectives ...................................................... 57
   4.2 Overall Network View ............................................... 58
      4.2.1 Mapping of Application Processing Requirements ....... 60
      4.2.2 Interactions between Active Nodes ......................... 61
   4.3 Active Node View ...................................................... 62
8.1.1 Setup and Teardown ............................................. 114
8.1.2 Description of Protocol Messages .......................... 116
8.1.3 Establishment of Node State ............................... 118
8.2 Related Signaling Mechanisms ............................... 120
8.3 Summary .......................................................... 122

Part III: Evaluation 125

9 Implementation 127
  9.1 Overall Node Architecture ................................. 127
  9.2 Data Plane: PromethOS ..................................... 129
    9.2.1 Netfilter Packet Classification ....................... 129
    9.2.2 PromethOS Plugin Framework ......................... 131
  9.3 Control Plane: Routing and Signaling Protocols .......... 134
    9.3.1 Extended OSPF Protocol Daemon ....................... 135
    9.3.2 Explicit Path Routing Protocol Daemon ............... 136
    9.3.3 Plugin Loader .......................................... 138
    9.3.4 Local Resource Manager ............................... 138
  9.4 Active Network Control Software ......................... 139
    9.4.1 Service Setup Manager ............................... 139
    9.4.2 Resource Discovery ................................. 140
    9.4.3 Routing ................................................. 140
    9.4.4 Signaling .............................................. 141
  9.5 Summary ........................................................ 141

10 Applications with Network Processing 143
  10.1 Description of Test Network .......................... 144
  10.2 Video Distribution with Congestion Adaptation .......... 145
    10.2.1 Video Encoding Schemes ............................. 146
    10.2.2 Wavelet-Based Video Encoding ....................... 147
    10.2.3 WaveVideo Bandwidth Scaling ....................... 149
    10.2.4 Deployment of Video Adaptation Scenario ............ 151
    10.2.5 Video Quality Measurements ....................... 155
    10.2.6 Fast Reaction to Congestion ....................... 157
  10.3 Security Gateway Application .......................... 160
    10.3.1 Encryption of Data Traffic ......................... 161
    10.3.2 Deployment of Security Gateway Scenario .......... 163
  10.4 Summary ........................................................ 164
## List of Figures

2.1 DARPA active network node reference model .......................... 16  
2.2 Packet flow through an active node (AN working group) ......... 17  
2.3 Different router architectures with processing elements .......... 18  
2.4 Tasks for providing network services ............................... 23  
2.5 Different service graphs ........................................... 25  
2.6 Network graph ...................................................... 26  
2.7 Mapping of multicast service graph onto network graph .......... 27  
3.1 Darwin node architecture ........................................... 41  
3.2 Ninja paths subsystem .............................................. 43  
3.3 HIGCS service deployment phases .................................. 46  
4.1 Distributed network control software for active networks ...... 59  
4.2 Mapping of processing requirements onto physical network .... 60  
4.3 Interactions between active nodes running ANCS ................ 61  
4.4 Active node view ................................................... 62  
4.5 Internal structure of ANCS ......................................... 64  
5.1 Secure data transmission modeled using active pipe ............ 71  
5.2 Video scaling scenario modeled using active pipe ............... 74  
5.3 VPN with data encryption modeled using active pipe ........... 75  
5.4 MPLS traffic engineering modeled using active pipe ............. 77  
5.5 DiffServ architecture modeled using active pipe ................. 78  
6.1 Packet format of OSPF opaque LSA ................................ 86  
6.2 Encoding of processing site attributes as TLV objects ........... 87  
6.3 OSPF API protocol phases ......................................... 89  
7.1 Network with embedded processing sites ........................... 98  
7.2 Graph transformation for required node scenario ................ 100  
7.3 Graph transformation for required link scenario ................. 101  
7.4 Graph transformation for required steps combined ............... 102  
7.5 Graph transformation for conditional nodes scenario .......... 104  
7.6 Graph transformation for conditional links scenario ............. 105  
7.7 Graph transformations for active pipe ............................ 107  
7.8 Hierarchical network structure .................................... 109  
7.9 Divide-and-conquer routing approach .............................. 110  
8.1 Protocol to establish explicitly routed path ....................... 115  
8.2 Path establishment message ...................................... 117  
8.3 Filter table for flow-specific forwarding and processing ....... 119
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Implementation overview of active node</td>
<td>128</td>
</tr>
<tr>
<td>9.2</td>
<td>Internal structure of OSPF daemon with API server</td>
<td>135</td>
</tr>
<tr>
<td>9.3</td>
<td>Internal structure of EPR daemon</td>
<td>137</td>
</tr>
<tr>
<td>9.4</td>
<td>Internal structure of ANCS daemon</td>
<td>139</td>
</tr>
<tr>
<td>10.1</td>
<td>Test network used for evaluation</td>
<td>144</td>
</tr>
<tr>
<td>10.2</td>
<td>Media gateway performing network-interior video scaling</td>
<td>145</td>
</tr>
<tr>
<td>10.3</td>
<td>Hierarchical wavelet decomposition of video images</td>
<td>148</td>
</tr>
<tr>
<td>10.4</td>
<td>Successive scaling of video frames</td>
<td>150</td>
</tr>
<tr>
<td>10.5</td>
<td>Bit layout of network tag describing video segment</td>
<td>151</td>
</tr>
<tr>
<td>10.6</td>
<td>Video distribution scenario with congestion adaptation</td>
<td>152</td>
</tr>
<tr>
<td>10.7</td>
<td>Lossless, plain-queued and actively scaled frames</td>
<td>156</td>
</tr>
<tr>
<td>10.8</td>
<td>Video quality under moderate and heavy congestion</td>
<td>158</td>
</tr>
<tr>
<td>10.9</td>
<td>Reaction to bursty cross traffic</td>
<td>159</td>
</tr>
<tr>
<td>10.10</td>
<td>Frequency distribution for plain- and active dropping</td>
<td>160</td>
</tr>
<tr>
<td>10.11</td>
<td>Security gateway scenario</td>
<td>161</td>
</tr>
<tr>
<td>10.12</td>
<td>Format of AES encrypted packet</td>
<td>162</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Node-local service frameworks .......................... 48
3.2 Network-level active packet-based services ................. 49
3.3 Network-level active extension-based services .............. 50
6.1 Specified attributes .................................. 87
8.1 Comparison of explicit routing mechanisms .................. 122
11.1 Application scenario benchmarks ........................ 168
11.2 Measurements for service establishment ................... 170
11.3 Measurements for routing ............................. 172
11.4 Measurements for signaling on individual nodes .......... 175
Part I:

Concepts of Programmable Networks
Chapter 1

Introduction

1.1 Motivation

In the last decade, the Internet has undergone a fundamental transformation from a small-scale network serving academics and selected technology companies, to a global infrastructure serving people in all parts of the world. Striking as this remarkable growth has been, the Internet is still far away from providing high quality, reliable, easy access to information, communication, and computing services. Today’s Internet is based on a best-effort service model that merely provides means to transport information between participating end users. In this paradigm, routers implement a rather simple forwarding scheme that determines the outgoing interface based on the longest-prefix matching of the datagram’s destination address and perhaps on simple policies set by network administrators.
While the intention of the Internet designers was to provide a simple fixed core service for enabling communication between peers [94], [35], new demands on the network have led to the introduction of more and more functionality and as a consequence much of the simplicity of the original architecture has been lost. In particular, the forwarding scheme has proven to be insufficient for services beyond simple data transport. Emerging services such as firewalls [99], explicit congestion adaptation [74], [14], media gateways [5], [119], network address translation [48], intrusion detection [29], secure communication [10], and flow-specific QoS [118] all require extra processing at the packet level. In this context, services can be defined as application functions provided on network elements as opposed to traditional services implemented on end systems only. To support these emerging services, the packet forwarding scheme implemented by current Internet routers needs to be enhanced. Routers are required to perform additional functions such as inspecting transiting packets, altering the forwarding and queuing behavior, or even modifying the packet content.

Yet, the existing Internet infrastructure is extremely inflexible in supporting new services. This inflexibility arises from the fact that in the Internet paradigm networks are built with a few standard signaling and control protocols (e.g., ICMP [114], RSVP [18], IGMP [52]) that cannot be easily changed. All protocol modifications require a lengthy and arduous standardization process to achieve consensus from all equipment manufacturers and network operators. Since the protocol functionality is generally hard-coded within routers, new services are mostly provided by physically installing dedicated network equipment which then implements the required functions (as part of the system software), making the deployment of new services tedious and expensive. As the transition to IPv6 [43] demonstrates, upgrading the Internet protocol on a global scale is a very time consuming process that requires decades to complete.

1.2 Programmable Networks

To solve the inflexibility of the Internet, the network infrastructure needs to be transformed into an extensible and programmable multiservice network to accommodate in a straightforward way a continuously evolving spectrum of network services. The current Internet architecture needs to be enhanced
to enable the dynamic deployment of \textit{service-specific} functionality interior to the network and the \textit{programmability} of such services. With the possibility of placing service-specific functions into routers, applications are no longer restricted to implement all their functionality on end systems. Rather, applications can \textit{distribute} portions of their program logic among the end systems and intermediate nodes. This allows implementing applications in a highly distributed fashion, with functions being placed at the most appropriate locations. For example, a video streaming application can then install customized modules on routers to adapt the video stream to match the bandwidth of congested links by removing less crucial image information. Such a particular method of video scaling could not easily be implemented on end systems in traditional networks without processing capabilities on routers.

To foster this fundamental transformation, the research community has proposed \textit{active networks} \cite{133}, \cite{134}, \cite{20} as the enabling base technology for the creation of multiservice networks. The active networking paradigm envisions a \textit{programmable network infrastructure that is extensible at runtime} to accommodate the rapid evolution of protocols and services demanded by applications. In this vision, the shared network resources can be tailored specifically for the requirements of applications and as a consequence, applications are no longer restricted to a network that solely forwards packets. While the potential of this new technology is clearly visible, there remain major challenges that need to be addressed to turn this potential into actual benefits for network users.

\section*{1.3 Challenges of Making Networks Programmable}

Transforming the current Internet into a highly flexible network infrastructure is challenging in many ways. Such a network needs to be easily programmable by network administrators as well as end users. Furthermore, significant computational power in routers is needed to process packets at link rates. The development of such a programmable network infrastructure needs to address the following issues:

- \textit{hardware architectures} that incorporate programmable elements in routers that are suitable for the provision of advanced network services
• **software architectures** that provide secure, reliable, high performance, programmable environments for the execution of user-supplied code

• **service frameworks** that manage network resources and provide abstractions such that network services can be easily specified, deployed, and used by applications

Hardware architectures are required to support very high system throughput with a parallel interconnection network moving packets through the switch fabric. Core input and output processing functions such as packet classification (e.g., longest prefix matching), forwarding, checksum handling, and scheduling (e.g., fair queuing) need to be implemented in the appropriate high-performance technology, traditionally in custom hardware. In the last years, advances in processor technology have demonstrated that incorporating general purpose processing capabilities into network routers is becoming feasible both from a technical and economical point of view. In particular, rapid progress in reconfigurable logic creates the opportunity to dynamically reconfigure system resources for high speed processing (e.g., FPX [90], protocol boosters [51]).

While these advances in processor technology have demonstrated that processing at gigabit speed is becoming realistic, there is a need for suitable software architectures of programmable routers. Many research groups have addressed this problem and have come up with a wide range of node architectures with various properties. For example, ANTS [142] employs a Java virtual machine (JVM) [89] in which program instructions are embodied as Java bytecode in each packet (capsule), offering Turing-complete expressiveness for programmers but little performance. SwitchWare [4] and NetScript [126] use restricted languages designed such that the created services have certain desirable properties like termination and preservation of the active node’s safety. The ANN [40] and PromethOS [77] architectures are modular systems that can be dynamically extended by plugin modules installed in the networking kernel, providing a highly efficient data path by preventing data copying and costly context switches. The Dynamically Extensible Router [82] extends the router plugin approach to a multiport gigabit switch, providing high-speed processing capabilities at each input and output port.
Although several node architectures have shown that they are able to execute packet processing functions efficiently, there remains the challenge of designing flexible and easy-to-use service frameworks that allow router resources to be coordinated and shared by different applications. The objective of such a service framework is to provide mechanisms such that applications can program the underlying network according to their specific needs. Specifically, applications need to be able to deploy their own services interior to the network in an automated fashion. Installing code on routers should be a simple task, hiding the topology and internal details from the application whenever possible. Ideally, the application expresses its processing requirements using a high-level abstraction and the service framework then translates these requirements onto the physical network, without the need for the application to know about the underlying network topology and location of processing resources. Another key challenge to enable easy-to-use network services is to automatically discover processing resources and configure application services on behalf of users. In particular, there is a need for network control mechanisms that provide (1) a user–network interface for expressing computational requirements of applications, (2) a dissemination protocol to distribute information about the availability and usage of processing resources, (3) a routing mechanism that can quickly map application requirements onto physically available processing resources, and (4) a signaling mechanism that configures nodes with the appropriate processing functions.

### 1.4 Problem Statement

With networks offering embedded processing capabilities, various new questions need to be tackled:

- **How can users define the processing step requirements for a particular application scenario?**
  A uniform programming model is needed that hides network-specific details and the topology from applications, enabling applications to operate independently of the underlying network, without manual configuration from users.

- **How can processing capabilities be described and advertised in a network?**
  Mechanisms are required that distribute information about the avail-
ability and usage of network resources, allowing other nodes discovering suitable processing capabilities.

- **How should different processing steps be distributed over nodes in the network?**
  Efficient algorithms must be found that can translate application requirements onto physical network resources, preferentially while minimizing network costs.

- **How can network state be established such that packets get routed and processed appropriately?**
  A signaling protocol is needed capable of deploying code on remote nodes and allocating network resources such as processing cycles, service program and buffer memory, and link bandwidth.

### 1.5 Contributions

The main contribution of this dissertation is to offer a viable solution that enables the provisioning of network services within active networks. Specifically, the contributions are as follows:

- Design and implementation of a *service framework* that enables the specification, configuration, deployment, and provisioning of services in a programmable network. By putting knowledge about the location and capabilities of processing resources into the network itself, applications are freed from locating suitable resources and mapping their requirements onto physical resources, enabling applications to more easily make use of programmable networks.

- Proposition of *active pipes* as a method for specifying transmission and processing requirements over active networks, while abstracting from the underlying network topology and system details. Application-specific location requirements can be expressed individually for each processing step by various attribute constraints.

- Design and implementation of a *resource discovery mechanism* that distributes information about the availability and usage of processing resources and allows other routers obtaining the underlying network topology, location of processing resources, and their capabilities.
• Design and implementation of a constraint-based routing algorithm that can map an application’s transmission and processing requirements (expressed as an active pipe) onto suitable network resources while minimizing network costs. The routing algorithm solves the problem of finding the optimal location for several required and conditional processing steps along with a path connecting these sites in a predetermined order while minimizing the sum of both link and processing costs. The algorithm has the same linear complexity as Dijkstra’s shortest-path algorithm.

• Design and implementation of an explicit path signaling system that routes traffic along predefined paths and installs processing modules on selected nodes. For reasons of optimality, such paths may be non-simple (looping) where given nodes can be visited repeatedly.

• Qualitative evaluation of the usability of our framework by demonstrating a video distribution application performing bandwidth adaptation on nodes with congested outgoing links and a security gateway encrypting packets within the network.

• Quantitative evaluation of the framework by providing performance measurements and assessing the overhead needed to establish services.

1.6 Outline

This thesis contributes to the design of the control mechanisms needed by next-generation programmable networks and is structured as follows:

• Chapter 2 introduces the active networking paradigm, presents the active node reference model, discusses the concept of services, and defines the terminology used throughout this thesis.

• Chapter 3 reviews related work and describes the architectural key features of existing frameworks providing network services.

• Chapter 4 characterizes the control mechanisms required to manage programmable networks and describes the design rationales of our envisioned service framework.
- **Chapter 5** discusses active pipes as a user-network programming interface that applications use to express their communication and processing demands.

- **Chapter 6** describes mechanisms for resource discovery required to obtain information about processing sites such as their capabilities and availability.

- **Chapter 7** discusses how processing requirements formulated using active pipes can be mapped to the underlying network and proposes an algorithm for the problem of routing packets through processing sites in an active network.

- **Chapter 8** describes a signaling protocol for the installation of processing modules on specific nodes and the establishment of explicitly routed paths transiting these nodes.

- **Chapter 9** describes the overall architectural issues that pertain to the implementation of our network architecture.

- **Chapter 10** evaluates our prototype implementation using a qualitative analysis. We demonstrate the usability of our service framework with a video streaming application that deploys application-specific congestion control within the network and with a security gateway providing data encryption between dislocated subnetworks.

- **Chapter 11** provides a quantitative analysis by presenting experimental performance measurements assessing the time required to map application requirements onto physical resources and to deploy packet processing code in the network.

- **Chapter 12** concludes this thesis with a summary of the main results, reviews the claims, and provides some starting points for further research.
Chapter 2

Background

In this chapter, we discuss the motivation of making networks programmable, introduce the active networking paradigm, present the reference model of a single node proposed by the research community, and state in more detail the inherent challenges involved for providing network-level services on programmable nodes.

2.1 Trends in Networking

Today’s Internet was originally designed in the early 1970’s with the goal of having a simple, packet-switched infrastructure connecting a large number of independently organized networks with routers (or “gateways”) [94], [35]. The network core was designed to be relatively simple, offering a basic com-
communication service that forwards IP packets towards the appropriate destinations. Because the network neither provides reliable connectivity nor flow control, network protocols were designed to implement most of their complexity on end systems, such as retransmission of lost packets, and congestion control based on round-trip time estimates (see the ARPANET design philosophy [35], [94] and end-to-end arguments [121], [13]). These design decisions lead to a robust core network, with no associated interior state to recover in the case of failures since all intelligence is provided by the end systems.

During the last decades the original Internet architecture evolved dramatically. Several additions to the Internet have been proposed to provide better quality and more reliable communication services for network applications. Because today’s Internet does not support the dynamic deployment of new protocols, most of these services were specifically designed in later generations of routers, requiring the replacement of routers with new ones offering the requested functionality. Implementing network services at nodes interior to the network often offers better functionality and performance than adding functionality at end systems solely. This observation is supported by a number of ad-hoc efforts to exploit such functionality:

- Network address translation (NAT) [48] maps a large set of internal addresses to a small set of external ones by keeping state of open connections and rewriting source and destination addresses and port numbers found in packet headers.
- Firewalls [99] evaluate filter rules on packet headers, offering domain-specific access control on behalf of network administrators and users.
- Congestion control mechanisms (such as RED [53]) selectively discard packets from flows that use more bandwidth than their assigned fair share to provide feedback to senders for adapting their sending rate.
- Traffic engineering [11], [72] assigns packets to traffic classes such that packets are routed according to application-specific QoS requirements or administrative policies.
- Web switching [7], [80] inspects incoming HTTP requests and forwards requests to one of several physical server machines, offering load balancing and better scalability of web services.
• Media gateways [5], [119] transcode audio or video streams by adapting their rate and making them compatible with heterogeneous receivers (such as low-end portable devices).

• Reliable multicast routers [109] provide network support for large group communication by minimizing implosion from receiver feedback and exposure from the reception of superfluous packets.

• Mechanisms to detect Denial-of-Service (DoS) attacks [29] analyze transiting packets and counteract by denying flows that match specific traffic patterns (such as SYN flooding).

2.2 Demand for Programmable Service Infrastructure

For the network of the decades ahead, there is a strong demand for services beyond what is required for simple routing and data transport. Emerging services as described above all require flexible processing at the packet level on nodes interior to the network. To deploy such services, network operators and end users require a network infrastructure that is more manageable and can be upgraded to meet changing demands. In our opinion, programmable service platforms are essential to allow for easy deployment of new services. Since it is difficult to predict what exact services will be most useful, this also helps to reduce the risk of installing dedicated hardware that can only serve a single purpose. As an advantage, a service creator is not required to deploy its own suitable infrastructure but can use the underlying programmable network to provide this functionality. Such an infrastructure requires mechanisms to bring application-specific packet handling routines to nodes and to execute those handlers when matching packets transit the system.

2.3 Mobile Code Paradigm

For providing custom handling of packets on nodes, the research community has proposed the mobile code paradigm, which allows for moving code (procedures or programs) into the network and for executing the transferred code at these sites. The two most promising technologies based on this idea are active networks and mobile agents:
• *Active networks* allow modifications of the *network core* by providing customized functions directly on routers, thus replacing the current network infrastructure with a more flexible scheme.

• *Mobile agents* operate on end systems, using the network primarily for connectivity, and implement higher-level communication abstractions such as distributed objects capable of moving their data and execution state using object serialization in conjunction with a class loader.

Although originating from two different research communities, the research concerns of the two mobile code technologies overlap to a great extent, particularly in the areas of discovery, selection, and deployment of network resources. Prominent examples of the mobile code scheme are Java applets [56] downloaded from a web server, or Adobe Postscript [1] files, which are programs interpreted on a printer to generate the page.

### 2.4 Active Networking

The *active networking* [133], [134], [20] paradigm envisions a new networking infrastructure that is flexible and extensible at runtime to accommodate the rapid deployment of networking protocols and provide increasingly sophisticated services demanded by applications. Targeted at *network routers* rather than end systems, the active networking paradigm presumes that it is advantageous for data not merely to be passively relayed from the source to the destination, but actively processed by the network. In an effort to provide better services to users, active networks allow functionality to be deployed internal to the network. Network nodes can be extended by application and service code that then gets executed on nodes and alters the behavior of the network to the favor of users. This allows the extension and modification of the underlying infrastructure, increasing the flexibility and customizability of the network, and accelerate the pace at which network software is deployed. Recently, several applications have emerged that require computations within the network (e.g., firewalls, media gateways, Web proxies, reliable multicast schemes), however no support for these applications can be offered by the existing Internet infrastructure. With active networks, the functions are no longer restricted by the built-in software of the vendors, but can be determined even dynamically by the applications that use the network. For the introduction of new code on routers, there are two distinct
models: In the *active extension* model, new code is deployed explicitly through an out-of-band setup process, whereas in the *active packet* model code is carried within the packet and the installation of code is triggered by the arrival of the packet.

The trend of transforming the current Internet architecture into a more flexible and programmable active network infrastructure is analogous to the transition of the telephone network from the *plain ordinary telephone service* (POTS) into an *intelligent network* (IN) [135]. In POTS, the network consisted of hardwired switching systems which were difficult to upgrade. As new features and services were requested from customers, new switches had to be designed, manufactured, and installed. In addition, different carriers used switches from different vendors, so services were difficult to implement across carrier service areas. In an effort to provide more flexible signaling mechanisms for the telephone system, intelligent networks move the logic for routing calls and establishing connections out of the telephone switching systems into attached nodes called service control points. Service control points are distributed throughout the network and can be programmed with logic to provide new capabilities and services. Features such as local number portability, call waiting and forwarding, speed dialing, or multi-party conference calls can then be rapidly introduced into the network. Essentially, the IN is a service-independent telecommunications infrastructure that provides the means to develop, deploy, and control services more efficiently. While the goals of IN and active networks are similar, the IN protocols are restricted to out-of-band signaling mechanisms for circuit-switched networks (with the very specific MTPL3 and TCAP protocols), and thus are not appropriate for active processing on intermediate nodes in the IP-based Internet.

To the best of our knowledge, the first system based on the active networking concept was SOFTNET [152], in which messages exchanged between packet radio nodes can contain FORTH statements that are then interpreted on each node. A similar idea was published by Wall [138], who proposed an active routing service where “messages act as active agents”. The active networking paradigm has generated much interest because of its appeal as a means of creating new Internet services. Recently, various systems have been proposed and a comparison of them can be found in surveys [24] and [133].
### 2.4.1 Active Node Reference Model

The Active Networks working group funded by the Defense Advanced Research Projects Agency (DARPA) proposed an architectural framework [20], [21] that defines the major components and interfaces that make up an active node, and discusses architectural features common to network services built using such active nodes. The framework serves as a *reference model* for the design and implementation of active networks by describing the basic functionality needed to support such components (Figure 2.1 [129]).

![DARPA active network node reference model](image)

*Figure 2.1: DARPA active network node reference model*

Each node of an active network runs a *node operating system* (nodeOS) and one or possibly several *execution environments* (EE), and optionally an additional *management execution environment* (MEE). The nodeOS is responsible for allocating and scheduling the node’s resources such as CPU cycles, buffer memory, and link bandwidth. Each EE provides an environment for packet processing where application-specific code can be loaded and executed. Users obtain a service from the active node via an *active application* (AA), which is code that programs the EE to provide the requested service.

The model is quite general since it only specifies the basic functionality of an active node, thus it can be implemented by a particular network architecture in its own way (for an example see [130]). In fact, implementations often integrate the EE into the same address space as the nodeOS for performance reasons.
In the following, we describe in more detail each of the parts making up the DARPA reference model. We illustrate how packets are processed in an active router and then describe the functions of the nodeOS, EE, and AA.

### 2.4.2 Packet Processing in Active Router

The general flow of packets through an active router is illustrated in Figure 2.2 [21]. Classification of incoming packets is controlled by filters that can be specified by the execution environment and active applications. Packets are classified based on information in the packet header, such as a specific protocol number, a particular source and destination address and port number, payload content, or a specific ANEP\(^1\) header field.

![Packet flow through an active node (AN working group)](image)

**Figure 2.2: Packet flow through an active node (AN working group)**

Packets that match a given filter are handed off to the corresponding EE for processing. The packet then gets processed by application-specific code which was either previously loaded into the EE or is part of the packet itself.

---

1. The Active Network Encapsulation Protocol (ANEP) [3] suggests a format for encapsulating active packets for transmission over different media. Typically, an ANEP header includes a reference to the code (e.g., plugin identifier [41]) required for handling the active packet.
Also the destination address to which a packet gets routed can be altered during this process. The output side includes protocol processing (e.g., TTL decrement\(^1\)) as well as scheduling before the packet is transmitted on the link. It does not need to be the case that each transmitted packet corresponds exactly to some received packet, since EEs may drop, aggregate for periodic forwarding (pacing), or even generate packets spontaneously.

### Processing Elements

To handle the growing transmission capabilities of links, router architectures are required to be highly scalable in order to support a large number of gigabit links. To keep up with gigabit data rates, routers use dedicated hardware to implement many protocol processing functions in silicon. This functionality is distributed among the line cards that convert bits to and from the physical link medium, and the switch fabric that moves packets from input to output ports, typically a multistage interconnection network or crossbar. Incoming packets are parsed and validated on the input port, where the destination address is looked up in a forwarding table to determine the output port. Then the packet is transferred across the switch fabric to the output port, where finally it is queued and scheduled for transmission. There are different possibilities of how processing capabilities can be added to this architectural framework.

![Figure 2.3: Different router architectures with processing elements](image)

One approach is to add dedicated processing elements to both the input and output ports (Figure 2.3a). Every link interface has embedded programmable processors with memory to store code and application-specific flow state. There is a clear one-to-one mapping between ports and processing ele-

\(^1\) Decrementing the Time-To-Live in the packet header requires the recomputation of the IP checksum. However, adjusting the checksum can be implemented efficiently [91].
ments and the route through the switch fabric determines the processing element where a specific function needs to be executed. An example of this architecture is the Smart Port Card [44].

An alternative approach is to connect dedicated computing modules to the router and use them as a pool for processing capabilities (Figure 2.3b). Packets requiring active processing are routed to a processor from the pool, and from there to the appropriate output port. Thus computation of active flows can be evenly distributed over the processing engines that are available. Scalability of this approach is guaranteed through the ability of configuring any number of processing elements. An example of this architecture is the Multi-Gigabit Router [111]. This approach requires a distributed load sharing algorithm which dynamically distributes active flows over the processing elements.

Network Processors

Network processors [37], [87], [123] are software programmable devices specialized for the networking application domain. A network processor has architectural features such as a fast I/O path and an instruction set specifically tailored for high speed packet processing (for example dedicated instructions for IP prefix matching). To overcome the performance limitations of traditional single processor systems, network processors often include multiple processor cores with integrated cache and memory on a single chip. Such processing clusters exploit the inherent independence among different traffic flows by processing packets concurrently. This parallel utilization of processing cores boosts computational capabilities and allows processing to arbitrarily scale if needed (see [32] and [148] for a discussion of network processor design).

Commercially available programmable network processors have recently appeared on the market. For example, the IBM PowerNP4GS3 [65], [112] includes sixteen processing units (also called picoprocessors) operating at 133 MHz clock frequency, offering aggregate packet processing at speeds of up to 2.5 Gbit/s. Another prominent network processor is the Intel IXP2800 [25], [66] with sixteen 32-bit independent multi-threaded microengines, delivering aggregate packet processing capability at OC-192 (10 Gbit/s) line rates.
2.4.3 Node Operating System

The nodeOS [21] provides an operating system for the active node and offers an API to access system resources to be used by AA/EEs. The nodeOS is responsible for managing all system resources such as CPU cycles, buffer memory, communication channels, and persistent storage. To provide access to these system resources for AAs and EEs, the nodeOS uses higher-level abstractions called thread pools, memory pools, channels, and files. All system resources are assigned to a domain. Domain resources are allocated according to credentials, which are used for admission control, scheduling, and accounting. This resource partitioning allows the isolation of different EEs from each other and the sharing of resources in a fair way. The nodeOS can be implemented as either a dedicated operating system (e.g., ANN [40]) or can be realized as a shim (UNIX process) on top of an existing OS (e.g., Janos [136], Bowman [97]).

2.4.4 Execution Environment

Execution environments provide a context for the evaluation of active packets by hiding user interactions with the active node from the nodeOS. Each EE defines a programming interface to be used by active code. An EE can be implemented in many different ways, ranging from completely general virtual machines which are relatively slow (e.g., ANTS [142]), extension of UNIX-based networking kernels on general-purpose processors (e.g., SPC [44]), to programmable logic that offers packet processing at gigabit line speed (e.g., FPX [90]). Safe execution and limited resource usage can be provided by different means, such as bytecode interpretation within a sandbox (e.g., JVM [89]), language-inherent resource bounds (e.g., CAML [23]), proof-carrying code [106], or only accepting trusted and digitally signed code (e.g., DAN [41]). A single node can support one or multiple EEs, or even provide an additional abstraction layer introducing the concept of a virtual active node (VAN [19]). The management EE can be seen as a special EE that monitors and controls the active node, and has privileged access to system resources (e.g., SENCOMM [69]).
2.4.5 Active Application

An active application refers to program code provided by network users and which gets executed within an EE. Depending on the method and policy of how code is installed, users can be either network administrators or even ordinary end users. If code is directly installed within the kernel of the node, the loadable code is often referred to as a router plugin [40].

In the following subsections, we illustrate a few typical applications that benefit from processing capabilities on routers. Specific for these applications is that they cannot easily be implemented on end systems only, without the possibility of placing packet processing functions on nodes.

Application-Specific Congestion Control

Video applications can respond to network congestion by adapting the quality of the transmitted video to match the available network bandwidth using mechanisms such as higher lossy compression. The installation of video scaling functions directly into the forwarding path of routers enables the adjustment of the transmission rate with finer granularity than an end-to-end mechanism, leading to improved network performance and higher quality video delivery. The video is encoded in different bands of resolution, each subband carried in different packets. This encoding allows application-specific functions in routers to respond to local congestion by selectively discarding packets containing less crucial information from the video stream. Since this mechanism uses only local information, it can react quickly to network congestion, much more responsive than any sender-based feedback. Several video scaling schemes have been proposed in the literature, both for MPEG [14], [63] and wavelet-based [74] encodings.

Data Aggregation

Data aggregation applications combine data flows from multiple sources into a single stream, such that the aggregated outgoing stream has a total lower bandwidth than the sum of all originating streams. A typical application of this mechanism is audio data mixing [147] where multiple senders participate in a video conference. The audio channels from multiple participants can be aggregated into a single audio stream. The benefits are that the
overall data traffic transmitted through the network is reduced and end systems do not need to process audio streams from each participant, thereby increasing the scalability of the system. Another application is reverse multicast [22], providing a many-to-one channel where multiple sources send messages toward one destination, and the network delivers a single aggregated copy to that destination. This mechanism can solve scalability problems that arise when collecting feedback in large multicast applications. Data aggregation is also useful in sensor networks [147], in which a large number of sensors transmit periodic state information. Instead of sending all messages to a single node, intermediate nodes aggregate information and forward a summary of all data such as the average, extreme values, or standard deviation.

### Web Service Balancing

Web switching [7], [80] enables load balancing web requests to multiple physical machines while presenting a single web server to clients outside. A web switch inspects incoming HTTP requests and selects an appropriate server machine to handle that request. Once the server machine has processed the request and returned its result, the web switch generates an HTTP response which is sent back to the client. This load balancing mechanism, which forwards requests to server machines, is completely transparent for clients because the web server is still visible with a single IP address.

### Active Web Caching

Active Web caching [85], [144] has the goal of delivering clients with information from the nearest cache to minimize latency and optimize network utilization. The active network maintains a hierarchy of caches and redirects HTTP requests to locations with the cached content. On the packet level, the active node rewrites IP addresses transparently for the web client.

### 2.5 Active Services

The DARPA reference model focuses on a single active node that offers custom packet handling capabilities in form of code that come as “active applications”. However, this view is too restricted since typical application sce-
scenarios require multiple processing functions to be distributed within the network. We refer to the application functions coordinated to work together and implemented on network elements as an active service. Since the terminology in the context of services is not used consistently in the literature, we define the following terms:

- A component is an entity that has a well-defined interface and behavior, as defined by the component developer. A component has both an interface specification and an actual implementation. Thus, each component includes code that gets dynamically loaded into the execution environment on a node. Depending on the execution environment, a component can be implemented in either software or hardware. If a component is installed in the node as a loadable kernel module, it is often referred to as a router plugin.

- An active service is a collection of individual components that have to be identified and placed appropriately in the network, and are interconnected through communication channels. Like a component, a service has a well-defined interface and behavior. However, the important distinction between a service and component is that a service can be composed of multiple (and usually distributed) components.

Thus, components provide the basic building blocks to build services through composition. For example, a service providing secure data transmission consists of an encryption and decryption component. The deployment of these components within the network along with the establishment of interconnected channels then leads to the requested service.

Providing a new service within a programmable network involves several tasks (Figure 2.4):

![Figure 2.4: Tasks for providing network services](image)

The service specification defines the components required by a service and describes how components are interrelated. Resource discovery identi-
fies the location and capabilities of processing resources embedded within the network, and builds an annotated network graph describing the physical topology including the processing capabilities. Service mapping translates the service specification onto the physical network graph while taking into account all service-specific constraints. The service allocation task reserves and configures appropriate physical resources as determined by the service mapping process. Once the service has been deployed, service provisioning is the final task that executes all required components to provide an operational service for users.

Given the size of current networks, the variety of processing capabilities as well as the complexity of how services can be composed, this process clearly needs to be automated to be feasible. For that reason, it is crucial to have a suitable service infrastructure built on top of raw processing modules to efficiently coordinate and enable programmability of each node.

In the following, we describe each task involved in providing a network-interior service in more detail.

2.5.1 Service Specification

The service specification describes the logical properties that make up a service. This includes the communication and processing requirements from the network and how they are interrelated. There are several forms of how a service can be expressed, but typically a service can be expressed using a graph as illustrated in Figure 2.5. We refer to the graph describing the composition of a service as the service graph. The sources $s_i$, destinations $t_j$, and intermediate processing steps $p_k$ are modeled by nodes, while the edges correspond to communication channels.

The first service graph is a path that shows an example of a unicast service with a source, two intermediate processing steps, and an end node. The second example is a tree that refers to a multicast session with processing steps and several receivers. The last example illustrates a service graph that involves multiple end nodes, and several processing and data flows. In general, a service graph forms a directed graph with an arbitrary topology. While such arbitrary graphs are possible, we focus our attention primarily on paths and trees since many current application scenarios belong to these groups.
2.5.2 Resource Discovery

The resource discovery task involves locating network-interior processing resources and determining the underlying network topology. As its result, resource discovery produces an annotated network graph describing the network (for which the service deployment occurs) with end systems, routers, and physical links (Figure 2.6), with some nodes in the network graph including special attributes representing programmable routers.

The network graph includes various attributes attached to both nodes and links. Attribute-value pairs \(<a_i, v_i>\) describe capabilities such as link properties or the type of processing a node can handle. This categorization is determined by the system specification of routers, in which properties like supported execution environments, processing power, memory buffer size, or software capabilities are considered. Node attributes describe hardware properties (CPU capabilities, buffer space), software properties (implemented execution environments, supported routing and signaling protocols), and location properties (network address, border gateway router). Link attributes describe properties such as the underlying physical link layer technology.
(Ethernet, ATM, wireless), link bandwidth, bit error probability, delay, and jitter.

Attributes can be either static or dynamic, describing whether the attribute value remains fixed or might change over time. Static attributes are usually defined by network administrators and once defined are immutable. Dynamic attributes are periodically regenerated to reflect changes in the network (such as the available link bandwidth or residual processing cycles on a network processor).

2.5.3 Service Mapping

As we have described the service and network graphs, the service establishment process includes the task of mapping the service specification onto the physical network graph. As depicted in Figure 2.7, this task involves the mapping of the service specification’s end nodes to the corresponding end nodes of the physical network, processing steps to nodes that are capable of the appropriate processing, and establishing connections between physical nodes as they correspond in the service graph.

The mapping of a graph $G$ onto another graph $G'$ is known to be the graph embedding problem [98]. Unfortunately, the embedding problem for an arbitrary graph $G$ is known to be NP-complete [55]. Even if the service graph represents a tree, the optimal mapping remains NP-complete (see Steiner Tree problem [64], [145], and multipoint routing [140]). Nevertheless, there are service patterns, to which most application scenarios belong.
in which the optimal mapping can be found more efficiently. As we will demonstrate, the mapping problem can be solved in polynomial time when the service graph consists of a path.

2.5.4 Service Allocation

The service allocation task configures network resources (as determined by the service mapping algorithm) such that the network provides the expected service. For signaling the deployment of new code on routers, there are two distinct models:

- In the *active packets* model, the installation of code is triggered implicitly by the arrival of the packet. An active packet is marked that it requires special handling on the node. Packets either carry small amounts of program code or contain a reference to external library code to be executed.

- In the *active extensions* model, the installation of code is triggered explicitly through a setup process (or manually by an administrator). This also includes the establishment of forwarding state on routers such that packets are routed through processing nodes and will be recognized to be actively processed.

Furthermore, for the active packets model, there are two different schemes how the actual code is transferred to the node:

- *In-band* deployment refers to a system where the service logic is distributed in the same way as the payload data. In this model, packets
carry small amounts of program code which is transported within the ordinary data stream. The code within a packet is self-contained and may be executed on every node along the path a packet travels. For security reasons, active packets often use a virtual machine that safely interprets the code within packets. However, interpretation of code may cause significant performance degradation.

- **Out-of-band** deployment refers to an architecture where service deployment and payload data use logically and/or physically distinct communication channels. In this model, packets include only a reference to the code. If the code is not already present on the node, it is retrieved from a remote code server. The integrity of the new code is verified before it is loaded into the execution environment. Once the code has been approved (based on a digital signature [31] or proof-carrying code [106]), the new code is fully trusted and for that reason can generally be executed as fast as native code on the router.

### 2.5.5 Service Provisioning

Once the underlying programmable network has been configured to provide a given service, the focus of service provisioning is the coordination and execution of all components that make up the service. In particular, this includes the execution of previously installed and configured code on routers. Data traffic entering a router first gets classified, and if according to filter rules packets require active processing, packets will be processed by an appropriate active application running within an execution environment. This execution of packet processing code involves the allocation and scheduling of the node’s resources such as CPU cycles, buffer memory, and link bandwidth to guarantee that packets can be processed at the required rate. Since multiple flows are competing for resources at the same time, the node resources must be partitioned in some fair manner. This assignment of node resources to different flows is controlled by the CPU scheduling policy. Each of the packet processing components runs independently for the provision of services, and their interaction will be policed and controlled to ensure that an abnormal execution of one component will not negatively impact on other components’ execution.
2.6 Summary

In this chapter, we provided background information on programmable networks to better understand important design and implementation issues relevant for the following chapters. We discussed the active networking paradigm that envisions a highly flexible network infrastructure to be used for building advanced network services. In particular, the DARPA reference model specifies the architectural features common to network services and defines the major components and interfaces that make up an active node. While the reference model considers only a single node, typical application scenarios require multiple processing steps to work together to provide the demanded service. How a service should be composed can be formulated using a service graph which can be a path, tree, or arbitrarily complex. To deploy a service, the service graph needs to be mapped onto the physical network graph and deployment mechanisms are required to install and configure processing code on appropriate nodes.

Before we elaborate on our proposed infrastructure for building advanced network services in Chapter 4, we look at related work that has been proposed in the literature.
Chapter 3

Related Work

Several research groups have worked on service frameworks with the purpose of making networks programmable with user-supplied functionality. Although the focus of these efforts varies widely between services executing on a single node to composite services spanning multiple nodes, they all share the overall goal of transforming the current Internet service model into a more sophisticated infrastructure where envisioned services comprise not only communication facilities, but also computational resources.

In this chapter we examine several service frameworks that have been proposed in the literature. In the following discussion, it is important to note that we focus on service deployment frameworks and not node architectures per se. To differentiate, a node architecture is mainly concerned with the execution of code, while service deployment is concerned with the selection of
appropriate active nodes and the installation of programmable code into an execution environment. Also, design issues of a node architecture have a node-local scope while service deployment considers functionality to be distributed on multiple nodes in a network-wide scope.

First, we describe our classification scheme and then group existing service frameworks according to these criteria. Related mechanisms and algorithms that specifically perform discovery, routing and signaling will also be addressed in Section 6.5, Section 7.4 and Section 8.2, respectively.

3.1 Classification Criteria

Service frameworks can be designed with different objectives in mind and be subject to various architectural constraints. In the following, we explore the design parameters and discuss existing approaches proposed in the literature. We base our evaluation primarily on the following design criteria (also see [17]):

- **Node-level vs. network-level scope**
  The scope of a service framework can be divided into node-level where code must be installed within the environment of a single node or network-level where code should be installed and coordinated on multiple distributed nodes.

- **Active packets vs. active extensions model**
  Depending on the model used for the introduction of new code, systems based on the active packets model trigger the installation of code implicitly by the arrival of a packet, while systems based on the active extensions model install service logic explicitly using a signaling protocol.

- **Service specification**
  The service specification describes how application requirements are specified and how services can be structured from individual components.

- **Resource discovery mechanism**
  Depending on how information about processing resources is discovered, central information processing stores all service-related informa-
tion at one place while a distributed approach spreads information within the network.

- **Resource mapping algorithm**
  The resource selection mechanism describes how application requirements are mapped to physical network resources by determining appropriate processing resources. Important design issues are the optimality and the computational complexity of the mapping algorithm.

- **In-band vs. out-of-band allocation mechanism**
  The code to be deployed on a node can be either transported in-band\(^1\) as part of the data stream or can be retrieved out-of-band from a remote code server.

- **Service provisioning**
  Services can be implemented using various programming languages with either Turing-complete or resource-bounded expressiveness, evaluated using bytecode interpretation or executed in native format, and implemented in either user or kernel space.

The works presented in this chapter are grouped as follows: First in Section 3.2, we look at architectures that offer services on the local node only. While these frameworks are limited to one node, they represent the first step in transforming the Internet into a programmable infrastructure. Then we investigate frameworks that allow the deployment of functions on multiple nodes within the network. We first look at approaches based on the active packets model and group them according to their code deployment model, which can be an in-band (Section 3.3), out-of-band (Section 3.4), or hybrid (Section 3.5) approach. Section 3.6 then discusses approaches belonging to the active extensions model. In Section 3.7, we compare all the presented approaches such that we can draw implications in Section 3.8 about our envisioned service framework.

---

1. Unlike in the context of telephony networks, in which the terms in-band and out-of-band refer to the physical infrastructure, this refers to whether the signaling is combined or separated with respect to a particular information flow.
3.2 Node-Local Services

Service deployment at the node level consists of installing and configuring software within the local node environment. Although the installation is restricted to a single node, these frameworks provide a service specification mechanism and some simple resource mapping by selecting appropriate service logic that matches the node’s capabilities.

3.2.1 Chameleon

Chameleon [16] facilitates the installation of a service on a single node. A service is expressed using a node-independent description, allowing to send the same service specification to different types of active nodes. Each node that needs to install a service resolves the service specification depending on its locally available processing capabilities, thus allowing a service to exploit the particular functionality of an active node. This way, the service installation scheme can support heterogeneous active nodes and still take advantage of node-specific performance features.

A service is modeled as an arbitrary tree of containers and connectors. A container can either be an implementation of a specific processing function or can refer to an already existing service. Thus, a service can be structured in a hierarchical and recursive way, allowing services to be built from simpler services and composed to more complex ones. An implementation, however, is not composed and represents the basic unit for the composition scheme. The mapping of the service specification to the node environment uses local information from the node information base (listing all available processing resources) and local policies (mapping rules).

3.2.2 Bowman/CANEs

The Bowman/CANEs [96] node architecture is divided into two components: an underlying nodeOS that defines a set of basic functions to access and manage the available resources on a node, and an execution environment (EE) that enables users to control the active node with application-specific functionality. The Bowman nodeOS layers all the active network-specific functionality on top of a standard host OS and provides three basic resources
offered by the active node: a channel communication abstraction, an a-flow computation abstraction, and a state-store memory abstraction (for persistent state). Built on top of Bowman, the CANEs execution environment offers a composition framework for active services. CANEs introduces the slot processing model which comprises two parts: a fixed part (underlying program) that represents uniform processing applied on every packet, and a variable part (user-injected program) representing application-specific functionality on the packet. The specific points in the underlying program where the injected program can be executed are called slots. Composition of services in CANEs is then achieved in two steps. First, an underlying program that provides a basic service (such as forwarding) is selected from amongst those offered by an active node. Second, one or multiple injected programs are selected that customize the underlying program. These injected programs can be available at the active node, or can potentially be downloaded from a remote site. Each injected program is bound to one or more processing slots.

3.2.3 Click

Click [81] is an architecture for building flexible and configurable routers, assembled from packet processing modules called elements. Individual elements implement simple router functions like packet classification, queuing, scheduling, and interfacing with network devices. A router configuration is a directed graph with elements at the vertices and packets flow along the edges of the graph. Click configurations are modular and easy to extend. A standards-compliant IP router can be composed from just sixteen elements. Extending the IP router to support dropping policies, fairness among flows, or Differentiated Services requires adding a few elements at the right place in the forwarding path. However, a Click configuration is static since it cannot be extended once it has been compiled and installed on a node.

3.3 Active Packet-Based, In-Band Deployed Services

In the following, we look at the deployment of services on the network-level where the installation of a service is implicitly triggered by the arrival of an active packet. The code to be executed on nodes is transported in-band as part of the data stream.
### 3.3.1 ANTS/PAN

Tennenhouse et al. introduced the “capsule” model [134], where datagrams carry (in-band) small fragments of code to be executed on each node along a packet’s path. An implementation of the pure capsule model was first proposed in the form of an *ACTIVE IP option* [141], with a TCL interpreter used to provide safe code execution. The TCL-based implementation was mainly a proof-of-concept for the capsule idea. However, the approach proved to be impractical from a performance standpoint since the complete program needs to be delivered with each packet, which seems overly wasteful.

The *ANTS* [142] toolkit is a refinement of the traditional capsule idea, introducing an optimization. Instead, a packet can include a *program identifier*, referring to a Java program (or “code group”) required for the processing of the packet. If the program that a packet references has not been loaded, the program is retrieved from the *previous hop*¹ a packet follows. Since most applications require the same code to be executed repeatedly on the node, this approach diminishes space and time overheads of carrying the complete code. The program is retrieved and converted into an efficient executable format only once.

To address the performance issues inherent with the interpretation of virtual machine code, *PAN* [108] uses native, microprocessor-specific object code instead of Java bytecode. Obviously, this results in better performance, but opens up concerns about interoperability and safe program execution.

### 3.3.2 Smart Packets

*Smart Packets* [122] are another active packet-based approach targeted at network management and monitoring. Smart Packets are programmed in *Sprocket*, a high-level language much like C, but with security-threatening constructs such as pointers removed, and network management features added such as built-in types for MIB access. Sprocket programs are then *compiled* into *Spanner* code, which is a CISC assembly-like representation of the program. Spanner uses a machine-independent binary encoding designed to

---

¹ We consider the retrieval of code from the previous hop as in-band deployment, since code follows exactly the same path as the data stream.
yield very small programs. The Spanner code contained in active packets is then safely executed using virtual machine interpretation.

One of the design decisions of Smart Packets is that there is no persistent state across packets since this would open up problems of management and consistency. Consequently, programs sent in Smart Packets must be completely self-contained. This implies that the practical limit of meaningful programs must fit into a typical Ethernet frame less headers, denoting that the complexity of programs that can be implemented using Smart Packets is essentially limited.

3.3.3 StreamCode

StreamCode [47] is heavily influenced by the capsule approach, with service logic located in each packet, and a packet execution unit provided at each node. StreamCode uses a hardware-based layer-3 active networking system, which can be programmed with machine language level binary object code. The user includes layer-3 actions as StreamCode fragments in each packet, which are then executed by the StreamCode processor as the packet traverses from hop to hop. The advantage of this hardware-based native execution model is that it allows for high-performance execution in the data path at gigabit rates. Since the complete instructions to process a packet must be embedded in each packet, the functionality and complexity of StreamCode programs are limited, thus elaborate services cannot be expressed. In fact, the instruction set is very restricted, with no possibility to express iterations (no program loops). To compensate for this limitation, StreamCode programs can interact with an ordinary execution environment, where long lived programs such as routing daemons reside.

3.3.4 SwitchWare/PLAN

The SwitchWare [4] architecture consists of three crucial components, namely active packets, switchlets, and a secure router core. Network services are implemented as a combination of active packets with active extensions. Active packets are written in a simple, resource-bounded scripting language called Programming Language for Active Networks [62]. PLAN programs use strong typing and are statically type-checked to provide safety, eliminat-
ing the possibility of type errors at runtime. To compensate for the limited expressiveness of PLAN programs, active packets can directly invoke node-resident service routines of active extensions called switchlets. Switchlets are programmed in (CAML) [23], a more expressive language which supports formal methodologies to prove security properties of the modules at compile time and no interpretation is needed. These code segments are authenticated by the developer and explicitly (not on demand) loaded into the active router.

3.4 Active Packet-Based, Out-of-Band Deployed Services

The following systems trigger the installation of a service by the arrival of an active packet, however the active packet contains only a reference to the code (rather than the complete program). The code is then retrieved out-of-band from a remote code server using a channel different from the data stream. The advantage of this approach is that the code does not need to fit into a single packet, and code is loaded only once (on-demand) on the first occurrence of the code reference.

3.4.1 ANN/DAN

The Active Network Node [40] is a modular and extensible architecture that allows code modules called router plugins to be dynamically installed into the node. Router plugins, which contain the service logic, are loadable kernel modules executed like native code on the router, and thus, run as fast as any other router code. The deployment of a new router plugin is initiated on the arrival of an active packet using the Distributed Code Caching for Active Networks (DAN) [41] scheme. An active packet includes an ANEP [3] header which references one or multiple router plugins required for the execution of the packet. If a packet carries a reference to an unknown code fragment, the code is retrieved on-the-fly from a code server and loaded into the runtime system. The service installation is triggered by an active packet received as part of the data stream, however the code is deployed out-of-band since the code is not part of the packet and the download process uses a different logical communication channel than the data stream.
3.5 Active Packet-Based, Hybrid Deployed Services

As a variant, the in-band and out-of-band deployment schemes can be combined into hybrid model. Code that fits into a single packet can be delivered in-band within a probe, but more complex functionality such as libraries are installed out-of-band.

3.5.1 SENCOMM

SENCOMM [69] comprises a management execution environment (SMEE) for node management. The SMEE runs on top of the nodeOS and provides an environment for the execution, monitoring, and control of active applications for management. Management applications, executing within the SMEE, consist of smart probes and loadable libraries. Smart probes are typically injected into the network from a management workstation, or may be created automatically by individual SMEEs. Typically, probes reside in the SMEEs for a period of time, and can leave persistent data to be used by probes currently running on the system or for probes that follow it. In addition to probes, SENCOMM supports loadable management libraries that are designed to provide code and data structures to be used by one or more smart probes. This allows common functionality to be shared among multiple smart probes, reducing the size and complexity of the probes. Libraries get loaded when the SMEE is bootstrapped or as needed by smart probes and other libraries. Any required libraries that are not currently loaded are retrieved from a code server before executing the probe.

The intention of SENCOMM is similar to Smart Packets [122], with the difference that SENCOMM offers persistent node state and proposes a hybrid code deployment scheme using both active probes (in-band) and loadable libraries (out-of-band).

3.6 Active Extension-Based Services

In the following, we look at systems where the service deployment is triggered by an explicit signaling protocol request rather than implicitly by the arrival of an active packet as in the previous approaches. In this scheme, the
network is configured before the actual data transmission begins. For that reason, the service establishment scheme can be much more elaborate. All the service frameworks based on explicit signaling provide (at least some) resource discovery, resource selection, and allocation mechanisms on behalf of applications. Thus, applications are not required to program their own deployment mechanisms (and place such functions into the active packet), but rather can request this functionality from the service framework. As a result, most of these systems can be seen as integrated approaches for resource management.

### 3.6.1 Darwin

*Darwin [27] comprises a set of integrated resource management components that support value-added services. The key property of all these components is that they can be customized according to service-specific needs. Services are structured in a hierarchical fashion, implementing value-added services in terms of lower-level services. The motivation is to provide applications with a high-level interface abstracting from low-level details and to reuse functionality rather than duplicating efforts to implement these services. In Darwin, resource requests are formulated as a virtual mesh, which is a network graph that includes a set of resources to be allocated and managed in an integrated way to meet the requirements of service providers and applications. A virtual mesh does not contain physical resources, but rather describes the communicating entities and the abstract resource requirements that applications have.*

As shown in Figure 3.1, Darwin’s architecture consists of the following four components:

- **High-level resource selection**
  For high-level resource selection, the resource broker called Xena performs global resource determination as specified by a virtual mesh. Xena’s tasks include balancing between services (e.g., trade-off between computation and communication resources when performing data compression), coordinating allocations for interdependent resources (e.g., required processor cycles for a transcoder depends on bandwidth of transiting video flow), and interconnecting incompatible services and endpoints (e.g., adding transcoding function to make data
formats compatible). Xena translates the virtual mesh onto network resources by expressing it as a boolean optimization problem. Since this problem is generally NP-hard, this approach is only appropriate for small-sized networks.

- **Customizable runtime resource management**
  For customized management of data flows, Darwin proposes *delegates* which are code segments that are installed in the router's control plane. Delegates have the ability to control and monitor the router through the *Router Control Interface (RCI)* to allow customized traffic management for specific applications. The RCI offers a set of functions that the delegate can use to monitor traffic, detect traffic congestion, mark packets for discard, and reroute flows.

  The current implementation of delegates uses the JVM [89] for the execution of bytecodes, making use of the portability and safety features inherited from the Java language. Each delegate runs as a separate thread inside the virtual machine sandbox.

- **Hierarchical scheduling**
  To provide isolation and controlled sharing of individual resources among user applications, Darwin uses the *Hierarchical Fair Service Curve (HFSC)* [132] scheduler that offers hierarchical service classes, real-time scheduling guarantees, and supports service curves that de-
couple bandwidth and delay. Besides the scheduler, a packet classifier examines incoming data packets and assigns them to a specific flow for application-specific treatment. Classification is done by matching the packet header against a set of known filters, specified by applications. Both the scheduler and classifier export a control interface to the RCI which is available to delegates and signaling protocols.

- **Low-level resource allocation signaling protocol**
  To allocate low-level network resources (such as bandwidth, buffers, processing cycles, memory), Darwin uses a signaling protocol called Beagle [28]. Beagle interfaces with Xena to obtain a virtual mesh, which includes a list of flows and delegates that need to be established. Beagle then issues a sequence of flow setup messages, each message containing the resources required on different nodes. These requests are passed to the local resource manager which then allocates the flow’s resources using the interface to the classifier and packet scheduler. Beagle also establishes delegates by setting up flow reservation state, instantiating new delegates, and binding them to flows. Beagle is based on RSVP [18] and introduces a route constraint object to support customized routing.

### 3.6.2 Ninja Paths

*Ninja Paths* [57] is an architecture for creating application-level services from simpler ones through composition. Each existing service registers itself with the service discovery service (SDS), which is a central database that includes each service's capabilities and structural information such as input and output data types. More complex services can be composed through operators and connectors. An operator is a service itself that accepts some data types (via a streaming protocol) and outputs data in a possibly different format. A connector provides a connection between two operators, but has no understanding of the structure of the data it is transmitting, and thus does not alter the data format.

Central to the Ninja architecture is the path concept. A path is a sequence of operators and connectors between the source and destination that, when composed, results in the desired service. When composing a path, Ninja uses a type system to determine the validity of a path by verifying that the data
types match for each operator. The concept of paths is refined into *logical* and *physical* paths. A logical path is composed of a sequence of compatible operators and connectors, with a set of transmission requirements that need to be satisfied, *without* information about specific nodes on which these operators run. A physical path is an instantiation of a logical path *with* information about real nodes used to execute those operators.

The key components of the Ninja path architecture include the *Automatic Path Creator*, *Path Instantiator*, and *Path Implementor* (Figure 3.2). At the highest abstraction level, the Automatic Path Creator generates a logical path that consists of the operators, connectors, and endpoints and makes sure that adjacent operators have compatible input and output types. This logical path is then passed to the Path Instantiator which creates a physical path by determining nodes that can execute the operators and download code for the operator. Once a physical path has been instantiated, it is passed to the Path Implementor which creates a new thread for the operator that is responsible for processing the data stream.

In the current implementation, Ninja selects processing sites for operators *randomly* and then runs a shortest-path algorithm to connect the chosen sites through connectors. However, such a path finding algorithm can produce
highly inefficient paths, in particular if the path includes many operators, thus this approach is only applicable for small networks.

In Ninja, all packet manipulations of data streams are performed on end systems, thus this approach is targeted at the application-layer and not really intends to move application code into network routers.

### 3.6.3 End-to-End Media Paths

*End-to-End Media Paths* [103] from Princeton University is a Java-based framework to build multimedia applications from components. Based on user demands and resource requirements from media objects, the system first determines an end-to-end path connecting the source with the destination that has sufficient resources. In a second step, the system then configures individual nodes along the selected path with modules that implement the service. These end-to-end paths can be seen as a natural extension of node-local Scout [100] paths to a networked environment.

Creating an end-to-end path across a set of nodes involves two steps: at the global level, an end-to-end path needs to be mapped onto network nodes. This involves resource discovery throughout the network and selecting a route for the end-to-end path that satisfies the requirements of the user. Then at the node level, a local subsegment of the end-to-end path needs to be instantiated, and loading and binding code to particular packet flows.

The system architecture consists of various components: (1) a *user control point* (UCP) from which users indicate what objects they want to access, (2) a *collection of objects* that record certain facts about the environment and services that can be established and (3) a *path manager* running on each node instantiating the local segment of the end-to-end path. The end-to-end media framework uses information from different sources to make intelligent choices where to place functions in the network:

- *Media objects* describe the available media devices, such as input and output MIME [54] types, network capabilities (delay, loss rate), and device capabilities (e.g., number of audio channels, output video resolution)
- *User preferences* for devices such as desired video resolution
3.6. Active Extension-Based Services

- **Information about node resources** (such as available CPU cycles, bandwidth of links) and end system devices (device-specific attributes like display resolution). Each node in the network monitors its own state and maintains an up-to-date node object.

- A set of rules defines the end-to-end paths that can be constructed for a given application. A pattern matching algorithm then tries to map the path onto network resources.

The first step in instantiating a path involves selecting viable source and sink nodes that support the given media type as defined by the preferences of the user. Then the system determines the possible routes between the selected source and sink nodes. This is performed by a $k$-shortest path algorithm. Finally, the UCP attempts to match one or several templates, which define valid compositions expressed as regular expressions, against the nodes that lie on the $k$-shortest routes. This pattern matching is implemented by a deterministic finite automata. Once the UCP knows all the nodes that satisfy the requirements, it configures the individual nodes along the end-to-end path by downloading code modules. The current implementation is based on a path resolution mechanism with a centralized network state database, thus is not scalable for larger networks.

In the End-to-End Media Paths model, various end system resources are coordinated to build distributed multimedia applications. However, routers are not really involved in the manipulation of the packets but are rather used for connectivity only.

### 3.6.4 HIGCS

The *Hierarchical Iterative Gather-Compute-Scatter* (HIGCS) [60] algorithm provides a mechanism for the automated installation of services within large, hierarchically organized networks. HIGCS is based on the IGCS computation model proposed in [26] that enables applications to obtain various network status information, and to identify nodes or links with specific properties when installing a service. HIGCS extends the flat network model to a hierarchy with aggregation of network state to increase scalability.

Each service to be deployed can specify its specific needs from underlying network nodes. Network elements present their capabilities in XML
Figure 3.3: HIGCS service deployment phases

[149], uniformly describing the interface to access, configure, and operate node resources. To guarantee scalability for large networks, the framework uses a service deployment hierarchy to collect and aggregate information and then installs services in the network according to application-specific allocation policies. The service deployment mechanism is divided into the following steps (Figure 3.3):

1. Solicitation. This phase selects appropriate service nodes based on a high-level view of the network. For example, a service that needs functions between a source and an end node selects nodes along the shortest path. This process starts at the highest hierarchy level and propagates the selection process to lower levels until the physical nodes are reached.
2. **Summarization.** Each solicited node then computes a metric for a physical node or summarized metric for a logical node and propagates this information up the hierarchy.

3. **Dissemination.** This phase inspects the summarized metrics at the top hierarchy level and determines the nodes required for the requested service. Deployment requests are propagated down the hierarchy and forwarded to only those nodes where the service indeed needs to be installed, rather than based on a complete flooding.

4. **Installation.** On all physical nodes that received a deployment request, the service is installed into the node’s execution environment.

5. **Advertisement.** Finally, nodes with a new service perform autoconfiguration (for example a new routing protocol discovers its neighbors).

Executing all five steps leads to an *automatic deployment* with *custom metrics and automatic configuration* of the service.

### 3.7 Comparison

In the following, we provide a comparison of all the discussed service mechanisms.

#### 3.7.1 Node-Local Services

Table 3.1 lists service frameworks for node-local deployment. Chameleon is based on a node-independent service description, which is resolved according to the locally available processing capabilities, thus supports heterogeneous node architectures. CANEs introduces the slot processing model which comprises a fixed part (uniform processing applied on every packet), and a variable part for application-specific functionality. Click assembles an IP router from a set of packet processing modules called elements, however the configuration must be described at compile time and cannot be changed once the router is operating. Node-level mechanisms as illustrated in Table 3.1 are required for a programmable service infrastructure, but are *not sufficient*, since they ignore the fact that services can be distributed and span mul-
tiple nodes. Still, these systems provide node-resident composition by building advanced services based on simpler ones.

Table 3.1: Node-local service frameworks

<table>
<thead>
<tr>
<th></th>
<th>Chameleon</th>
<th>Bowman/CANEs</th>
<th>Click</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus</td>
<td>service installation on heterogeneous nodes</td>
<td>active node</td>
<td>IP router configuration from elements</td>
</tr>
<tr>
<td>Deployment scope</td>
<td>node-level</td>
<td>node-level</td>
<td>node-level</td>
</tr>
<tr>
<td>Service specification</td>
<td>service tree, architecture-independent</td>
<td>basic service selected and customized through slots</td>
<td>directed graph with elements</td>
</tr>
<tr>
<td>Resource discovery</td>
<td>node-local resource database</td>
<td>node-local</td>
<td>available resources statically defined</td>
</tr>
<tr>
<td>Resource mapping</td>
<td>recursive resolution to containers or implementation</td>
<td>not-supported</td>
<td>not supported</td>
</tr>
<tr>
<td>Resource allocation</td>
<td>out-of-band (code fetcher)</td>
<td>out-of-band (possible)</td>
<td>static (compile time)</td>
</tr>
<tr>
<td>Service provisioning</td>
<td>Java</td>
<td>Java</td>
<td>C++</td>
</tr>
</tbody>
</table>

3.7.2 Active Packet-Based Services

Table 3.2 illustrates service frameworks where the installation is triggered by the arrival of an active packet. ANTS, Smart Packets, StreamCode, and SwitchWare are inspired by the capsule model with in-band deployment of code. Since putting the complete service logic into each packet seems overly wasteful, ANN/DAN supports a variant where the packet contains a reference to the code instead of the complete program. SENCOMM uses a hybrid code deployment scheme, that is, self-contained functions can be delivered in-band, and more complex library functions are deployed out-of-band. However, all these service frameworks clearly lack mechanisms to locate processing resources within a network on behalf of users. As a consequence, the mapping of application requirements to available processing resources is
assigned to the application itself and must be implemented by the code contained in the active packet.

3.7.3 Active Extension-Based Services

Table 3.3 summarizes frameworks for services spanning multiple nodes with code installed based on explicit signaling. These frameworks offer an integrated resource management, allowing applications to express resource

---

**Table 3.2: Network-level active packet-based services**

<table>
<thead>
<tr>
<th></th>
<th>ANTS</th>
<th>Smart Packets</th>
<th>StreamCode</th>
<th>SwitchWare/PLAN</th>
<th>ANN/DAN</th>
<th>SENCOMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus</td>
<td>proof-of-concept of capsule model</td>
<td>network management and monitoring</td>
<td>high-performance active packets</td>
<td>secure switch extensions</td>
<td>high-performance kernel node OS</td>
<td>network management and monitoring</td>
</tr>
<tr>
<td>Scope</td>
<td>network-level</td>
<td>network-level</td>
<td>network-level</td>
<td>network-level</td>
<td>network-level</td>
<td>network-level</td>
</tr>
<tr>
<td>Service specification</td>
<td>reference to code group</td>
<td>Sprocket code in packet</td>
<td>complete code in each packet</td>
<td>code in active packet</td>
<td>plugin references in ANEP header</td>
<td>probe program</td>
</tr>
<tr>
<td>Resource discovery</td>
<td>not supported</td>
<td>not supported</td>
<td>not supported</td>
<td>not supported</td>
<td>not supported</td>
<td>not supported</td>
</tr>
<tr>
<td>Resource mapping</td>
<td>not supported</td>
<td>not supported</td>
<td>not supported</td>
<td>not supported</td>
<td>not supported</td>
<td>not supported</td>
</tr>
<tr>
<td>Resource allocation</td>
<td>in-band(^a), persistent state</td>
<td>in-band, no persistent state (MIB changes)</td>
<td>in-band(^b), persistent state</td>
<td>out-of-band, persistent state, digital signatures</td>
<td>hybrid, persistent state</td>
<td></td>
</tr>
<tr>
<td>Service provisioning</td>
<td>Java bytecode executed in sandbox</td>
<td>Sprocket code virtual machine</td>
<td>restricted machine language(^c)</td>
<td>resource-bounded instruction set(^d)</td>
<td>kernel plugins (compiled C code)</td>
<td>Java, ASP EE</td>
</tr>
</tbody>
</table>

a. Code retrieved from previous hop.
b. Switchlets can be explicitly loaded (but not on-demand).
c. StreamCode does only support forward branches, thus program loops are impossible. In addition, the maximum program size is limited to the MTU.
d. PLAN guarantees program termination and the number of packets a program can generate is limited.
requirements which are then translated and allocated by the framework. In
particular, Darwin allows applications to specify resource and communica-
tion requirements using a virtual mesh, which is then mapped to the underly-
ing physical network and deployed using the Beagle signaling protocol.
Ninja Paths is based on the path concept, where service requirements are ex-
pressed as operators and connectors, and mapped with a \( k \)-shortest path algo-

\[ \text{Table 3.3: Network-level active extension-based services} \]

<table>
<thead>
<tr>
<th></th>
<th>\textit{Darwin}</th>
<th>\textit{Ninja Paths}</th>
<th>\textit{End-to-End Media Paths}</th>
<th>\textit{HIGCS}</th>
<th>\textit{ANCS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus</td>
<td>integrated</td>
<td>service creation</td>
<td>inter-connecting</td>
<td>large-scale,</td>
<td>enabling</td>
</tr>
<tr>
<td></td>
<td>resource</td>
<td>through composition</td>
<td>multimedia devices</td>
<td>hierarchical</td>
<td>services for</td>
</tr>
<tr>
<td></td>
<td>management</td>
<td></td>
<td></td>
<td>service</td>
<td>programmable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>deployment</td>
<td>networks</td>
</tr>
<tr>
<td>Deployment scope</td>
<td>network-level</td>
<td>network-level</td>
<td>network-level</td>
<td>network-level</td>
<td>network-level</td>
</tr>
<tr>
<td>Service specification</td>
<td>virtual mesh (arbitrary graph)</td>
<td>path with operators, connectors</td>
<td>path templates</td>
<td>compute</td>
<td>active pipe,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>functions</td>
<td>required and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>defined by</td>
<td>conditional</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>agents</td>
<td>processing</td>
</tr>
<tr>
<td>Resource discovery</td>
<td>central (Xena)</td>
<td>central (SDS registry)</td>
<td>central</td>
<td>distributed,</td>
<td>distributed,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>custom</td>
<td>based on</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>aggregation</td>
<td>OSPF LSAs</td>
</tr>
</tbody>
</table>
| Resource mapping | non-optimal      | non-optimal          | non-optimal                    | defined by     | optimal (pol-
|          | (NP-complete     | (random selection     | (pattern        | agents using  |ynomial  |
|          | problem), link   | of sites on path),   | matching of sites on path),   | compute        | complexity), |
|          | costs only       | link costs only      | link costs only                | functions      | costs for link |
|          |                  |                      |                                |                | and processing|
| Resource allocation | out-of-band,    | out-of-band,          | out-of-band,                   | defined by     | out-of-band, |
|          | network-layer    | application-layer    | application-layer              | agents         | network layer, |
|          | (Beagle\textsuperscript{a}) |                        |                               |                | (EPR\textsuperscript{b}) |
| Service provisioning | Java           | Java                  | Scout                          | simulated only | PromethOS    |
|          |                  |                      |                                |                | plugins      |
3.8 Implications

All proposed service frameworks presented in this chapter provide some support for network services, however the schemes vary widely between deployment model, service composition scheme, expressiveness of active code, and scalability issues regarding network size.

Clearly, our focus is on services that can be distributed and span multiple nodes. For that reason, we do not further concentrate on the node-local approaches discussed in Table 3.1.

The systems discussed in Table 3.2 are based on the implicitly signaled active packets scheme, where code is either self-contained within the packet (in-band) or retrieved from a code server (out-of-band). However, the research community is starting to recognize that this model has too many shortcomings to allow it to serve as the primary vehicle for the delivery of active services and has little chance of broad acceptance (for an in-depth discussion see [143]). First of all, active packets do not seem practical for many advanced applications, especially applications that are long-lived, require persistent state, have rather complex program logic, operate with multiple concurrent traffic streams (e.g., firewalls), or provide elaborate control functionality (e.g., routing daemons). In particular, pure active packets with in-band deployment of code have shown to be too limited for practical use. In fact, few really useful applications based on this model have been demonstrated. Examples include active ping [42], trace route [83], an auction service [84], web cache routing [144], and multicast tree mapping [146]. Moreover, the expressiveness of active packet programming languages is often restricted due to security or performance concerns. Since putting code into each packet is overly wasteful, out-of-band deployment seems more appropriate because most applications require the same code to be executed repeatedly on the node. An example of this model is the deployment of congestion control for a video distribution application [74]. Still, the functionality of active packets remains limited. Active packets allow putting some rather simplistic processing functions into the network layer, but are
not intended to really migrate elaborate application code to locations within the network. Second, the active packets model has numerous implicit implementation assumptions since most systems focus on single processor, software-based router architectures with virtual machine interpretation of active code. However, it is not well-suited to the requirements of high-end router architectures of today and even less of tomorrow. Future high-speed routers will be offering many 10 Gbit/s links, with aggregated bandwidth in the range of multiple terabit per second [32], [128]. Even if we assume that not all packets need active processing, it seems obvious that architectures based on virtual machines are not well suited to a multi-gigabit scenario. Typical virtual machine-based routers can handle a few tens of thousands packets per second, or roughly 1 to 2% of the rate to saturate a single OC-192 link (see [83] for more performance-related issues). To overcome these performance limitations of virtual machine-based interpretation of active code, there has also been approaches (e.g., [47]) to use dedicated processors executing active code in hardware. Although performance seems promising, still the functionality and expressiveness of these programs proved to be very restricted, and far too limited to implement a useful network service.

In our opinion the introduction of new code into routers needs to be performed in a more structured way, with network service providers in control of the process. We favor an operating environment in which application services are explicitly signaled when the application starts up. The configuration of an active service includes the selection of intermediate processing nodes and the network links used for communication among the various components of the application. In our view, a session-oriented approach is needed to enable efficient allocation of network resources among competing applications. This is especially important for applications that require a certain level of resources in order to achieve an acceptable QoS.

Clearly, deploying code should be a simple task for the application, hiding the internal details of the network whenever possible. Also, optimal resource allocation should be delegated to the network control layer, thus freeing applications from this task. For these reasons, we consider an integrated approach based on explicit signaling to be more adequate for the introduction of new code. In such a scheme, the service framework locates available resources, maps application demands onto physical resources, and allocates all required resources before data transmission begins. Then the ap-
Implication is not required to know about the exact location of processing resources and the topology, since the network offers resource discovery and mapping on behalf of applications.

For these reasons, the systems offering an integrated resource management described in Table 3.3 seem to be more appropriate for our objectives. However, all proposed systems lack architectural features which make them difficult to use. Most systems are based on a central resource database, which stores the network topology and location along with the capabilities of processing resources. A central approach opens up redundancy and scalability issues. In addition, all proposed resource mapping mechanisms are clearly limited with respect to optimality and scalability. In Darwin, the resource selection problem is expressed as a binary optimization problem, which has NP-complete complexity, thus the size of the network is limited to a few nodes. Ninja selects processing sites randomly and then runs a shortest-path algorithm to connect the sites. In larger networks, choosing processing sites randomly can potentially produce highly inefficient paths, introducing high delays and wasting link bandwidth. End-to-End Media Paths approach pre-computes k-shortest paths, and then selects processing sites on this path using pattern matching. Trying out all possible pattern combinations can be computationally excessive for large networks with many rules.

When computing paths, all of the existing approaches select paths solely on link costs and completely ignore the fact that processing has an associated cost as well. In a network with programmable routers, it might be advantageous to take a somewhat longer path to reach less expensive processing sites rather than always process on the nearest site. This observation illustrates that there is a trade-off between link and processing costs, which needs to be considered when allocating network resources. The cost for processing cycles and memory buffers must be taken into account and brought into relation with the bandwidth costs of traversing links, such that paths are optimal with respect to both link and processing costs. As we will illustrate, optimal paths can be non-simple, meaning that some of the nodes are visited repeatedly.

Moreover, the presented service frameworks have different composition methods, that is, the semantics for creating services from individual components. All the already existing service frameworks support required process-
ing steps, however none of them allows expressing conditional processing (such as bandwidth adaptation whenever congested links are encountered). Since many practical scenarios depend on conditional processing, this is a crucial limitation. Clearly, Darwin supports the most complex composition scheme where components can be structured according to a virtual mesh (but still lacks conditional processing). All other schemes are path-based, that is, components are placed along an end-to-end path. While the virtual mesh supports application scenarios with multiple flows, the problem of mapping the virtual mesh onto the underlying network is generally NP-hard and therefore does not scale to a network with more than a few nodes. Since our focus is on providing a viable solution that enables the provision of services in active networks, scalability is a crucial factor and must be guaranteed. Because arbitrary complex application scenarios can be composed from paths when some of the nodes are fixed, we believe that providing useful building blocks for creating services is more appropriate than mechanisms that do not scale for larger networks. Our focus is on providing a service framework with a minimal set of mechanisms that serves a large fraction of applications, while avoiding unnecessary complexity. In our opinion this trade-off is necessary since optimal mapping of complex service graphs must generally be sacrificed for scalability reasons.

The discussion of related work indicates that no service framework exists that is able to provide the full spectrum of functionality and capabilities required, while guaranteeing scalability and optimal resource selection. It is the claim of the proposed ANCS to provide in an integrated fashion all the necessary service framework functions for discovering processing resources, optimally mapping service requirements onto network resources (with polynomial runtime complexity), and deploying processing functions within the network at appropriate locations.
Part II:

Design of Service Framework
Chapter 4

Active Network Control Software

4.1 Design Objectives

The key to the success of practical programmable networks will be the creation of effective protocols and algorithms for managing these networks and to facilitate the deployment of services that applications use. In the following we present the design rationales of our distributed programming framework that supports the concept of services and offers the ability to allocate resources on behalf of applications. In particular, the design goals include:
• Provide users with mechanisms to effectively express processing and communication demands of applications

• Introduce abstractions such that low-level network details like the physical network topology, system properties of routers (e.g., nodeOS and EE), network processor capabilities (e.g., instruction set), and software running on routers (e.g., supported routing protocols) can be hidden from applications

• Provide efficient mechanisms for the selection of network resources that satisfy application demands, while preferentially minimizing allocated processing cycles and link bandwidth

• Provide mechanisms for the allocation and configuration of appropriate processing resources such that the network provides the expected service to users

• Gracefully recover from link and node failures by automatically adapting to the altered topology and reestablishing service state in a manner completely transparent for applications

• Enable incremental deployment of active applications in networks where only a subset of nodes is programmable

To describe our framework in this part of the dissertation, we first focus on the overall network by explaining the high level view of how services are established and how individual active nodes interact. In Chapter 5, we then discuss in more detail how processing and communication requirements are expressed by an application. Chapter 6 describes how the location and capabilities of remote processing resources can be discovered. In Chapter 7, we look at how application requirements are mapped effectively onto available network resources. Chapter 8 describes how resources internal to the network are allocated and how packets are routed to transit these nodes.

4.2 Overall Network View

To enable flexible services within a programmable network, there is a need for an additional control layer built on top of raw processing capabilities that facilitates the deployment and configuration of user-supplied code. The active network control software (ANCS) [79] can be seen as a distribut-
ed system that automates the configuration of network resources to form services that applications use. The system accepts processing demands from applications using a network programming interface (NPI), discovers available processing resources, maps those processing requirements onto network resources, and configures the appropriate state on network nodes. The fact that nodes can perform this mapping autonomously on behalf of applications is a distinct feature of ANCS’s design.

Figure 4.1 illustrates our envisioned system where nodes participate in the establishment of network services. Each node continuously monitors the state of its locally available processing resources and distributes any information it learned to its neighbors. Based on this scheme, each router learns about the location and capabilities of processing resources of other routers and propagates this information throughout the network.

To set up network services, applications send a request to the network control software running on the active nodes. In particular, the router that receives the request then tries to determine the network resources that are needed to satisfy the request. More specifically, the router computes the location for placing processing resources and communication channels connecting the peers with the processing nodes. If a feasible configuration can be computed, a signaling protocol then deploys code on the selected nodes and establishes communication channels in between such that packets are routed...

---

Figure 4.1: Distributed network control software for active networks
through the processing functions. Packet processing code comes in form of
kernel modules that are loaded into the execution environment of the node. If
processing code has not been loaded previously, it is first retrieved from a
code server.

### 4.2.1 Mapping of Application Processing Requirements

The ANCS provides applications with an environment that makes the use
of an active network as simple as possible. Depending on the level of ab-
straction, we distinguish between the following views (Figure 4.2):

- **Application view.** From an application’s point of view, solely the com-
munication relationship between participating peers and intermediate
processing steps that operate on the data stream is relevant rather than
the exact underlying topology and system details. Thus, users preferen-
tially formulate their requirements by completely abstracting from the physical network topology and system architecture.

- **Active network control view.** From the view of the network control layer, the network is represented as a directed graph describing the connectivity between nodes, but abstracting from system details such as link layer technology. Properties such as whether a node is a processing site are expressed using specific attributes.

- **Physical network view.** The physical network is implemented using a specific link layer technology with routers incorporating different network processors.

### 4.2.2 Interactions between Active Nodes

The ANCS operates in a distributed manner, with interactions between end systems and the network (user–network interface) and between active nodes. The following interactions are involved (Figure 4.3):

![Figure 4.3: Interactions between active nodes running ANCS](image)

1. **User–network programming interface (NPI).** To request the establishment of a network service, the application assembles an appropriate request and sends it to the ANCS. Such a request includes the source and destination addresses of the participating peers as well as network-interior processing requirements.

2. **Resource discovery protocol.** Every active node distributes information about the availability and usage of its network resources, allowing other nodes to discover suitable processing capabilities located within the network.

3. **Resource allocation protocol.** Active nodes participate in a resource al-
location protocol that reserves and configures the resources making up a service. Active nodes that receive signaling requests then install code within their local environment and establish forwarding state such that data traffic will be directed accordingly.

4. **Data transmission and processing.** Once all appropriate network resources have been configured, data traffic from the application is then routed along the preconfigured path and processed by the previously deployed processing code.

### 4.3 Active Node View

Figure 4.4 illustrates an individual node of our programmable network infrastructure. Each active node offers programmable processing elements in which application-specific code can be installed. From bottom to top, the architecture can be partitioned into *data plane*, *control plane* and *active network control plane*. In the data plane occur per-packet activities such as forwarding and customized packet handling, the control plane is responsible for the configuration of the data plane (such as maintaining the forwarding table), and the active network control plane deals with establishing network services with processing needs.

![Figure 4.4: Active node view](image-url)
More specifically, the *data plane* is responsible for moving packets from input to output ports and processing them if requested. Arriving packets are first classified based on information in the packet header and matched against the forwarding table. Depending on the result of this classification, the packet either gets directly forwarded to an output port or is passed to the CPU scheduler if it needs active processing. The CPU scheduler manages a queue of arriving packets and decides (based on an algorithm such as [110]) which packet needs to be handled next by the processing elements. Finally, the outgoing packet is modified to reflect the decremented TTL and scheduled for transmission by the packet scheduler running on the output port.

The *control plane* configures the data plane such that packets arriving at a port can be handled efficiently. The local resource manager (LRM) controls the usage of all node resources such as the available link bandwidth, internal buffers, and network processor memory and remaining cycles. The LRM keeps track of how much can be further allocated by reservation requests or whether incoming requests need to be denied. The control plane executes various control protocols such as standard routing and signaling protocols, as well as resource discovery and allocation protocols specifically needed for ANCS’s operation. Each routing protocol contributes route prefixes to a common routing table, and the prefixes are then propagated to the forwarding table part of the data plane. Signaling protocols allow applications to make bandwidth reservations for guaranteed end-to-end flows. Specifically for the ANCS, the control plane runs two other protocols to manage processing resources. An *active resource discovery protocol* disseminates the capabilities, usage, availability, and location of processing elements to other routers. Using this discovery protocol, the ANCS learns about network-interior processing resources. An *active resource allocation protocol* is used by the ANCS to allocate and configure processing resources distributed within the network. Code for application-specific processing gets loaded into the node by the plugin loader from a plugin database.

The *active network control plane* can be seen as a new layer of abstraction implementing the functionality needed to set up services, thus freeing applications from knowing about the topology and details of low-level processing elements. The ANCS collects network resource information from various sources and stores all relevant information in its internal network database. To obtain information about other active nodes, the ANCS accesses
the database maintained by the active resource discovery protocol running in
the node’s control plane. Properties about local processing elements along
with their CPU and memory usage can be gathered from the LRM. In order
to inform other routers about the node’s available processing resources, these
properties are also disseminated by the discovery protocol throughout the
network. Once the ANCS has determined the network resources required for
a service, it uses the active resource allocation protocol provided by the con-
trol plane.

4.4 Internal Structure

The ANCS automates distributed service establishment by reserving and
allocating resources on behalf of applications with specific processing re-
quirements. The ANCS hides network implementation details from applica-
tions, offering a high-level network programming interface to applications.
Internally, the ANCS provides the following functionality (Figure 4.5):

- A service setup manager accepts requests from applications using the
  user–network programming interface. To simplify the deployment of
  services, this interface needs to be designed in a way applications can
  abstract from the topology and low-level system details. Once a service
  has been established, the ANCS keeps all application parameters de-
  scribing the service. The reason for keeping this information is that in
case the network topology changes due to link failures, the ANCS can
recompute the configuration of all affected services and reroute servic-
es automatically around the failed network area. This rerouting happens completely transparently and is not visible for the application.

- A network information database (NIB) keeps topology information about the underlying network annotated with specific attributes describing the capabilities (e.g., processing element type), availability (e.g., reservable processing cycles), and cost of attached network resources. Each node runs the active resource discovery protocol to detect neighboring active routers and to announce its locally available processing capabilities to its neighbors. Using this mechanism, each router learns and continuously gets updated about the processing capabilities of other active routers.

- A service mapping mechanism translates high-level application requirements onto the network topology graph based on the service constraints formulated by the application. This step first prunes nodes and links from the network graph that cannot satisfy the service requirements (e.g., insufficient cycles or bandwidth) and then determines the optimal location of processing sites and interconnecting communication channels, while making sure all service constraints are satisfied.

- A service deployment mechanism establishes state across the network as computed by the mapping process. This includes a preconfigured sequence of routers that must be included as part of the service path and forwarding state that must be configured such that traffic will be directed through the nodes with the installed processing code.

As depicted in Figure 4.4, the ANCS interacts with a number of underlying components present on the node. In particular, to obtain the state of the locally available processing resources, it interacts with the local resource manager, and to disseminate network state information, it uses an underlying resource discovery protocol. This way, each node learns about the topology and availability of resources embedded in the network. For the deployment of code and the establishment of network state across the network, the ANCS uses a suitable resource allocation protocol.
4.5 Summary and Outlook

In this chapter, we have described the requirements and design of our proposed network control software consisting of four core functions, namely the network programming interface, resource discovery protocol, resource mapping algorithm, and a signaling mechanism. In the following chapters, we focus on each of these parts. Chapter 5 introduces a method for expressing communication and computational application requirements using a high level abstraction. Chapter 6 illustrates a mechanism for distributing information about the availability of processing capabilities. Chapter 7 describes mechanisms that map application requirements onto the network, while Chapter 8 proposes an allocation protocol that configures network resources appropriately.
Chapter 5

Service Specification using Active Pipes

In this chapter, we describe an intuitive formalism that applications can use to express their processing needs. Clearly, one of our objectives is to make the use of advanced services as simple as possible for users. However, most active networking environments proposed in the literature require the application to explicitly specify the location of code modules by a network address. Thus, an understanding of the underlying network infrastructure and system architecture is necessary. This burdens end users and makes large-scale deployment of active network services impractical. In our opinion, deploying application-specific code should be a simple task for users, hiding the internal details of the network from the application whenever possible. Also, optimal resource allocation should be delegated to the network itself;
freeing the application from this tedious task. Thus, it is necessary to have a general scheme of specifying application requirements that is expressive enough to describe typical application scenarios while simple enough to be used effectively.

To establish services for active networks, a network programming interface (NPI) is required through which individual users request and utilize resources. Users should be able to formulate communication and processing requirements in an intuitive way, abstracting from low-level system details and the exact network topology. The design of the NPI is crucial since it influences whether active networks can be used effectively. As IP networks have demonstrated, their success can be at least partially attributed to the simple BSD socket interface [131] which enables users to access the network without having to know about the underlying link layer details and network routes. In our opinion, a similar mechanism is required to make active networks a real success.

5.1 Active Pipes

To provide an easy-to-use programming abstraction, we propose active pipes [76] that allow specification of communication and processing requirements by applications. An active pipe provides an interface between the user and the active network and significantly simplifies the use of active networks regardless of underlying implementation details, providing a crucial component missing in current active network frameworks. The idea of specifying transmission and processing requirements is to model a flow as a sequence of functions that have to be performed on the data stream. This concept is analogous to pipes in UNIX where data can be sent through a sequence of programs. In the context of active networks, these functions can be distributed on several nodes (unlike UNIX pipes were all processes execute on the same computer). Each function corresponds to a code module that has to be instantiated on a router along the path of the connection. However, an active pipe is a more general definition of the execution sequence since processing requirements can be conditional, thus can depend on the state of the network.

From a logical point of view, code modules can be deployed on either a node or along a link:
• **Deployment on a node** means that processing will be performed when the appropriate node is transited by the data traffic. An example for this type of processing is transcoding, where packets that go through the node need to be processed, regardless of the link the packet enters or leaves the node.

• **Deployment on a link** means that processing needs to happen when a given link is traversed. Here, the link that the data traffic enters or leaves a node is crucial for the application. An example is congestion control where processing needs to be performed when traffic is sent to a congested link.

That is, for each processing step the application must specify whether processing should be performed when traffic either goes through a node or when traversing a link.

Moreover, we distinguish between the following two types of code modules used in an active pipe:

• A *required module* provides a processing function that *must* be performed *exactly once* in the network. This functionality is essential for the correct operation of the application and cannot be omitted. Such a processing function typically changes the format of the data stream (e.g., encryption, media transcoding).

• A *conditional module* provides a functionality that is not necessary for the correct operation, but can significantly improve the quality of the connection. This type is installed on all nodes that fulfill a given condition. As a result the code module can be deployed *multiple times* along the path. If the condition is not satisfied at all, then no modules need to be installed. These code modules typically do not change the format of the data stream (e.g., monitoring, pacing, congestion adaptation, traffic metering).

As depicted in Figure 2.6, our network model has *attributes* attached to all nodes and links to define specific properties. Attributes can be either *static* or *dynamic* and are usually configured by network management but can also be defined by end users depending on the context. For example, the “address” of a node is a static attribute whereas the “available bandwidth” on a link is dynamic. Formally, each attribute consists of a two-tuple
Each node or link has zero or more attributes defined as an 
attribute set \( A = \{a_1, a_2, a_3...a_n\} \), where each \( a_i \) is an individual attribute. We refer to the value of attribute \( name \) on node \( v \) as \( v.name \) and on link \( e \) as \( e.name \).

In addition to defining a sequence of required and conditional modules, the programmer can use (individually for each module) installation conditions that describe the circumstances under which an active code module should be deployed. Since applications can have stringent requirements on the location of processing modules, each processing step can have multiple installation conditions that a node must satisfy in order to be considered for the execution of a given function. Constraints therefore restrict the location where processing can be performed. Each constraint is defined as a tuple \(<\text{attr}, \text{rel}, \text{value}>\), expressing that attribute \( \text{attr} \) must fulfill \( \text{value} \) by the relation \( \text{rel} \). If no constraints are defined for a processing step, then from an application’s point of view all nodes are considered to be valuable processing sites.

More formally, we define the installation condition \( C \) as a function that maps nodes or links to a boolean value \( C: V \rightarrow \{true, false\} \) or \( C: E \rightarrow \{true, false\} \), respectively. For the evaluation of this function, node and link attributes are used. Based on this function we can define the candidate set of links and nodes, which is the set of all locations where the installation condition is satisfied.

An application can express multiple location constraints at once, such as when it requires sufficient processing capabilities from an active node to guarantee proper execution and it desires to restrict the installation of the processing module within a given address range. To be considered for deployment, a node or link then needs to meet all of the given constraints.

Figure 5.1 depicts an active pipe for a scenario where a connection for sensitive data transmission should be established between two end systems, located in different domains, and congestion control mechanisms desired on parts of the network where the stream is encrypted. The end system domains are assumed to be secure but since traffic transits untrusted nodes, encryption and decryption steps are needed somewhere within the source and destination domains, respectively.
More specifically, the active pipe consists of:

- **source and end nodes** denoting the beginning and end points of the active pipe, specified using an IP address of either an end system or router

- **required processing functions** for encryption and decryption that *must* be installed within specific security boundaries of the network

- **connectors** linking the source, required processing steps, and the destination node. Connectors themselves can have various constraints, restricting the path to nodes that satisfy certain criteria (such as available bandwidth, located within a specific address range).

- **conditional processing functions** for congestion control attached to connectors are optional but desired along links located between the encryption and decryption steps

- **installation conditions** associated with each processing step, expressed as a set of attribute value constraints

- **flow specification** describing the traffic that requires active processing, specified using a 5-tuple filter `<source addr/mask, dest addr/mask, source port, dest port, protocol>`

An active pipe provides a method for specifying the transmission and processing requirements for connections over active networks. In Chapter 7, we will illustrate how we can map such a requested service path onto physical network resources while minimizing network costs.
5.2 Deployment Categories

As we have illustrated, processing can be either required or conditional, and processing functions deployed on either a node or along a link. Combining these cases results in the following four deployment schemes:

- **Required processing on node**
  In this scenario, processing *on exactly one node* is required. The path of the connection has to be set up such that it traverses at least one node that then performs the required processing. An example for this type of processing is data transcoding, where several nodes could potentially execute this function, but it must be performed exactly once.

- **Required processing on link**
  Similar to processing on one node, we can require processing to happen on exactly one link. In reality, the processing will be performed at either the node proceeding or succeeding the link. An example of this type of processing could be encountered in conjunction with DiffServ where the marking and shaping operations are implemented at network boundaries. A traffic shaping module needs to be placed along a link before entering the DiffServ domain.

- **Conditional processing on nodes**
  When processing is conditional, it can happen arbitrarily many times (including not at all). The code module will be installed on each node that qualifies for processing under the given condition along the traversed path. One example for this type is traffic monitoring on all routers along the path that are able to install the monitoring functionality.

- **Conditional processing on links**
  Similar to conditional processing on nodes, processing can also be done along links. An example for such a scenario is application-specific congestion control. If the data stream is routed along several congested links, the congestion control modules should be installed at each of these links. If the stream is not routed through any congested links, no congestion control modules need to be installed.
5.3 Application Scenarios

In the following, we describe several application scenarios that require special packet handling within the network and demonstrate how they can be expressed using active pipes. While the illustrated scenarios are at least partially deployed in the existing Internet architecture, their realization requires dedicated router hardware and manual configuration. Assuming a programmable network infrastructure, we illustrate how these applications can be dynamically configured in a more generic approach.

5.3.1 Application-Specific Congestion Control

*Application-specific congestion control* [14], [74], [53] is often cited as a good example that benefits from active networking. The idea is that customized modules modify an application’s data stream dynamically as a reaction to network congestion. The adaptation of the data stream should occur in a manner that minimizes the impact on the application. For example, when a video transmission encounters congestion in the network, it is desirable that less important image information (such as higher-frequency subbands) is removed first rather than information that has a higher influence on the perceived video. With traditional queuing, it is not possible for a router to make any distinction among packets. While the processing for this kind of application-specific queuing is quite simple, it has a considerable effect on the application. Under congestion the video at the receiver has a significantly higher subjective quality than under a random dropping policy. In addition, the quality degrades gracefully under increasing congestion. It should be noted that this benefit could not be obtained by implementation on end systems in traditional networks without processing capabilities on routers.

Figure 5.2 describes an application where modules should be installed at links that are likely to be subject to congestion, and illustrates how this scenario can be modeled using an active pipe. We assume that all congested links are marked with an appropriate link load attribute. If the data stream transits several congested links, congestion control modules will need to be installed before each of those links. If the data stream is not routed through any congested link, congestion control modules can be omitted. We have im-
implemented this application scenario for our test network which will be described in Section 10.2.

5.3.2 Security Gateway

A Virtual Private Network (VPN) provides secure communication between dislocated computer networks, which are interconnected through an insecure network such as the public Internet. At each of the VPN tunnel ends, security gateways encrypt and decrypt data traffic. VPNs are especially important for large enterprises which need connectivity between geographically divided branch offices and business partners. To ensure privacy, technologies such as IPSec [10] authenticate participating peers and encrypt transmitted data.

Figure 5.3 depicts a scenario where two networks are interconnected through a public network, modeled using an active pipe. We assume that all traffic enters the source access router (SAR) being located in the traffic originating domain. Within the source domain there are several active nodes that could be used for encryption. The destination domain is reached through a public network which is considered to be insecure and therefore cannot be trusted. Within the destination domain, again some of the nodes are active and could be used for the decryption task. All traffic is directed towards the destination access router (DAR) which has several attached end systems. Al-
though peering end systems could perform encryption and decryption individually, this setup simplifies the administration and maintenance of the network significantly. Network administrators can make sure that all traffic applies to administrative policies guaranteeing that no unencrypted traffic can enter the public network.

In a traditional approach, a network administrator would need to determine the location of the encryption and decryption steps and a path interconnecting these modules to set up this scenario. This requires the network administrator to have detailed knowledge about the underlying network, that is, the location of active routers and their specific capabilities. Since each router potentially has a different system software and a limited number of available cycles, a specific processing task can be executed only on a subset of active nodes, making the service deployment more complicated.

Using the active pipe model, this secure data communication scenario can be expressed quite easily (Figure 5.3). The source and destination of the active pipe are being located on the source and destination access routers. The data traffic interconnecting the domains is specified by the flow specification
All traffic directed towards the destination domain that matches the flow specification gets routed and processed along the service path. To guarantee that the encryption and decryption tasks are being placed within the trusted areas of the network, each processing step has location constraints, requiring the active node's address to be within the address range of the trusted domains. In Section 10.3, we describe how we implemented this scenario on our test network.

### 5.3.3 Traffic Engineering

*Traffic Engineering* (TE) routes traffic flows across a network based on the resources that the traffic flow requires and the resources available in the network. Traffic engineering is essential for Internet service providers that operate backbones in order to support a high use of transmission capacity and minimize congestion in the network. *Multiprotocol Label Switching* (MPLS) [118] enables IP-based networks to replicate and expand upon the traffic engineering capabilities of underlying layer 2 ATM and Frame Relay networks. MPLS traffic engineering automatically builds label-switched paths (LSP) across the backbone. An LSP is a traffic trunk where all packets follow the same explicit route. At the ingress node of the TE tunnel, a packet classifier assigns packets to a label-switched path according to filter rules and attaches a label to each packet describing the LSP. Within the MPLS domain, each node determines a new label based on a lookup table, replaces it in the packet header, and sends the packet out on the outgoing interface. Label swapping is more efficient than IP prefix matching, leading to faster packet forwarding. The egress node removes the MPLS label from the packet header and forwards the packet based upon a traditional IP lookup.

The MPLS scenario can be expressed using an active pipe as shown in Figure 5.4. The source and destination of the active pipe match the ingress and egress routers of the MPLS tunnel, respectively. Packets that enter the MPLS tunnel first get classified and marked with a corresponding label. This label assignment step must happen on the ingress node as well, and for that reason, an address constraint to match the ingress node is given. In this scenario, the source of the active pipe and the first processing step are located on the same physical node. This is not a problem for an active pipe, since the model describes a *logical dataflow* through a network with a sequence of
Figure 5.4: MPLS traffic engineering modeled using active pipe processing steps. These processing steps can be executed on the same node\(^1\) as long as the sequence is not violated.

Interior to the MPLS domain (not including the ingress and egress routers), each node performs label swapping and forwards packets on the LSP. That is, label swapping is needed along the path between the ingress and egress routers. To accomplish this, the active pipe includes label swapping steps on all nodes traversed. Since label swapping is not required on the ingress and egress routers, these nodes are excluded by two location constraints. The egress node finally removes the MPLS tag from the packet header. The destination of the active pipe is identical with the egress router, which is expressed by a location constraint.

The MPLS specification [118] provides a signaling mechanism (based on RSVP-TE or CR-LDP) for establishing explicitly routed paths, but path selection based on TE requirements is not part of this specification. The speci-

\(^1\) We assume that active nodes support chaining of processing modules.
fication envisions that LSPs are determined with some traffic engineering path calculation tool run by network administrators. In fact, our envisioned ANCS can be seen as a tool for computing and deploying the required functions along such traffic-engineered paths.

5.3.4 Differentiated Services

The Differentiated Services (DS) [15] architecture is based on a model where traffic entering a network is classified and possibly conditioned at the boundaries of the network, and assigned to different per-hop behavior (PHB). Interior to the network, packets get forwarded according to the PHB associated with the packet. Traffic conditioning performs metering, shaping, policing, and possibly remarking to ensure that the traffic entering the DS domain conforms to the traffic agreement.

This establishment of DS paths can be formulated using an active pipe as illustrated in Figure 5.5. The source and destination of the active pipe are lo-
located on the egress router of domain A and ingress router of domain C, respectively. The ingress router of domain B conditions incoming traffic to make sure that the traffic stream is in compliance with the agreed traffic profile. Conditioning includes measuring the packet arrival rate (metering), marking packets with a PHB, possibly delaying some of the packets (shaping), and discarding packets violating the traffic agreement (dropping). Within the DS domain, routers queue packets according to the PHB, and schedule outgoing packets accordingly. That is, DS schedulers are needed within domain B on all nodes along the DS path.

5.4 Summary

Active pipes provide an interface between the user and the active network and allow for the specification of communication and processing requirements. The idea is to model an application’s requirements as a sequence of functions that need to be performed on the data stream, similar to the pipe concept in UNIX, however with processing functions distributed on several nodes. Since active pipes abstract from the underlying network topology and implementation details, they significantly simplify the use of active networks, providing a crucial component missing in today’s service platforms. With active pipes, processing steps can be either required or conditional, and processing can happen when either going through a node or transiting a link. Using the four basic deployment schemes, we have demonstrated that we can formulate various application scenarios such as application-specific congestion control, security gateway between dislocated networks, traffic engineering, and differentiated services. This is an indicator that active pipes are simple enough to be used effectively while powerful enough to express even complex scenarios with many processing functions.
Chapter 6

Discovery of Topology and Processing Capabilities

The objective of our proposed service framework is to facilitate the development and deployment of network services by hiding the topology and network-specific details from users. This requires mechanisms that can distribute information about the location, capabilities, and availability of network-embedded processing resources.

This chapter deals with the resource discovery mechanisms needed to localize active routers and to obtain information about the specific properties of attached processing elements. First, we describe what information needs to be collected and how it could be obtained. Then, we propose how existing
link-state routing protocols can be extended such that they disseminate information about processing properties of active routers.

6.1 Resource Discovery Process

The resource discovery process includes the following tasks:

- **Monitoring local resources.** Each active router continuously keeps track of its locally available resources. This includes the supported execution environments, available processing cycles, and bandwidth reservations on links.

- **Announcing local resources.** Each node periodically distributes information about its locally available resources to its adjacent neighbors.

- **Gathering information.** By receiving advertisements from a neighbor, each router learns about the capabilities of other routers. All this information gets stored in the node’s network information base.

- **Propagating information.** All information that a router learns from other routers is also propagated throughout the network. When a router receives a state advertisement from a neighbor, it sends the information to all of its directly attached neighbors (except to the neighbor that it was received from). The state distribution algorithm should be reliable, ensuring that all routers have the same collection of state advertisements.

All of these tasks execute in parallel. As a result, each node learns about the processing resources from other nodes and can later use this information to make suitable decisions when establishing services.

6.2 Building Network Information Base

The network information base (NIB) is a distributed database containing all relevant information about network nodes, their processing capabilities, and how they are interconnected. The NIB is composed of a collection of link-state advertisements (LSAs). Each LSA describes a particular router’s local state. This includes the router’s outgoing links with associated costs.
(link weights) to each reachable neighbor, and specific properties such as the supported EE, processing capabilities, processing costs, and current CPU load. LSAs are propagated within the network and each router maintains an identical database describing the current state of the underlying network. As routers announce their changes, the NIB continuously gets updated to reflect the most current view of the network. Since the NIB contains the connectivity between all nodes, the NIB allows constructing a topology graph annotated with attributes describing processing capabilities.

To build the internal database that describes the underlying network topology and available resources, the NIB collects information from multiple sources.

- **Local resources** are continuously monitored by the local resource manager (LRM). The LRM keeps track of all locally available resources such as the load of network processors, reserved bandwidth on links, and memory buffers.

- **Link-state protocols** keep up-to-date topological information required for the calculation of shortest routes.

- **Opaque capabilities of link-state protocols** enable the distribution of information in a way completely transparent for the routing process but can be used to announce the capabilities of processing sites.

In the following, we describe each of the information sources in more detail.

### 6.2.1 Local Resources

The local resource manager controls the usage of all node-resident resources such as network processors, link bandwidth, internal memory buffers, and keeps track of all reservation demands from various protocols that request node resources. For example, if the RSVP protocol daemon receives a reservation request for guarantee of flow bandwidth, it determines the required resources (QoS mapping), and then requests the appropriate local resources from the LRM. Since the LRM knows which resources can further be allocated, it either grants or denies such a request, making sure that a node’s resources are not over-allocated. The same applies to processing re-
sources. Whenever a new processing module gets loaded into a node’s execution environment, the LRM deducts the required CPU cycles from the processing element’s pool of remaining cycles.

6.2.2 Network Topology

For the selection of appropriate processing sites, the network control software needs to know about the topological structure of the network and the location of network-embedded processing sites. In general, the more information is available for this selection process, the better decisions can be made.

Routing protocols such as OSPF [101], IS-IS [67], and PNNI [8] build an internal topology database describing the connectivity between routers. This database is referred to as the link-state database (LSDB) which is identical for each participating router. Each individual piece of this database represents a particular router’s local state. To discover its neighbors, each router periodically emits hello packets. Adjacent routers acknowledge hello messages and automatically learn about their directly attached neighbors, along with their capabilities. Neighbor information is then propagated throughout the network such that each router learns about other routers. Based on this link-state database, a router can compute a graph consisting of all reachable routers.

Since routing protocols already keep track of the network topology, our approach is to make use of this information by accessing a routing protocol’s internal link-state database. We describe our method for the OSPF protocol daemon in Section 6.3.

6.2.3 Processing Sites and Capabilities

To obtain information about processing resources embedded within the network, we require a mechanism that can distribute information about their location, capabilities, and availability. Our approach is to make use of existing mechanisms whenever possible. Recently, link-state routing protocols

---

1. If the network is represented in a hierarchical manner, the content of the link-state database can differ since network state from routers further apart is aggregated.
such as OSPF, IS-IS, and PNNI have been extended with capabilities to carry opaque information. This information is completely transparent for the routing process but can be interpreted by external processes for their specific needs.

For example, the OSPF protocol has been extended with opaque LSAs [36] to provide a mechanism for the distribution of application-specific information. An opaque LSA includes arbitrary information, typically encoded as a collection of type-length-value (TLV) objects, where each object carries information related to a particular attribute. With the IS-IS protocol, this information is carried by opaque Link-State Packets, with attributes encoded again as TLV objects. PNNI Augmented Routing (PAR) [9] is an extension of PNNI to allow flooding of non-ATM-specific information. PAR specifies the format of messages, but the information is completely transparent to PNNI routing. Non-PAR-capable switches simply flood the information according to the PNNI scoping rules but do not interpret these messages.

Opaque capabilities have been used by traffic engineering applications to distribute information about link properties. For example, OSPF-TE [72] distributes link information about the services classes of each link and the available bandwidth within each class. MPLS-TE [11] uses the opaque capabilities of the routing protocol as a mechanism for the distribution of MPLS labels needed for tag switching.

6.3 Resource Discovery using OSPF

In the following we describe how the OSPF protocol can be extended such that our network control software can use it to discover the underlying topology. We also describe how the capabilities of processing elements can be distributed based on opaque LSAs.

6.3.1 OSPF Protocol

The Open Shortest Path First (OSPF) [101] protocol is a link-state routing protocol which is designed to be run internal to a single Autonomous System (AS). Each OSPF router maintains an identical database describing the Autonomous System’s topology. From this link-state database, a routing ta-
Discovery of Topology and Processing Capabilities

Chapter 6

86

OSPF is a dynamic routing protocol since in the face of topological changes (such as router interface failures), OSPF recalculates routes quickly and determines new loop-free routes after a period of convergence. This period of convergence is short and involves a minimum of routing traffic by making sure that the protocol responds immediately to topology changes. All routers run the same algorithm, in parallel. To increase the scalability of the protocol, an area routing capability is provided, enabling an additional level of routing protection and a reduction in routing protocol traffic.

6.3.2 Opaque Link-State Advertisements

Opaque link-state advertisements (opaque LSAs) [36] provide a generalized mechanism to allow for the distribution of arbitrary information using the OSPF protocol. As depicted in Figure 6.1, an opaque LSA consists of the standard LSA header followed by some application-specific information. The standard OSPF link-state database flooding mechanism is used to transparently distribute opaque LSAs to all or a limited portion of the OSPF topology. The opaque information field can be interpreted by an OSPF daemon-internal module (e.g., traffic engineering such as OSPF-TE) or by an external application wishing to distribute information throughout the OSPF domain. The flooding scope of an opaque LSA is identified by its link-state type, with type 9 denoting link-local scope (directly attached to the local network), type 10 within area-local scope (associated logical routing area), and type 11 flooded throughout the Autonomous System.

<table>
<thead>
<tr>
<th>LSA Age</th>
<th>Options</th>
<th>LSA Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opaque Type</td>
<td>Opaque ID</td>
<td></td>
</tr>
<tr>
<td>Advertising Router</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS Sequence Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS Checksum</td>
<td>Length</td>
<td></td>
</tr>
<tr>
<td>&lt;cpuavail, 200&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;cpu-cost, 10&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;linkload, 10.0.1.5, 30%&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: Packet format of OSPF opaque LSA
An opaque LSA contains various fields in its header. The opaque type field denotes what kind of information is contained in the opaque LSA. Opaque type values in the range of 0–127 are allocated by IANA [104] (for example opaque type 1 has been assigned to traffic engineering), and values in the range of 128–255 are reserved for private and experimental use. The opaque ID is selected by the originating router to uniquely identify the opaque LSA and can be chosen arbitrarily.

### 6.3.3 Attribute Encoding

Each opaque LSA carries one or several attributes describing specific properties or capabilities of the node. Attributes are encoded as a collection of type-length-value (TLV) objects as illustrated in Figure 6.2.

![Figure 6.2: Encoding of processing site attributes as TLV objects](image)

The type field uniquely identifies the TLV with a constant. The length field specifies the length of the value field in bytes. If the length field is zero, no value is present in the TLV. The value has a variable length. To propagate information about processing resources, the following attributes have been specified in the context of our service framework:

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUCOST</td>
<td>2</td>
<td>cost per processing unit</td>
</tr>
<tr>
<td>CPUAVAIL</td>
<td>2</td>
<td>available cycles on processing unit</td>
</tr>
<tr>
<td>ADDRANGE</td>
<td>5</td>
<td>&lt;address/mask&gt; describing logical address domain</td>
</tr>
<tr>
<td>CONGESTION</td>
<td>2</td>
<td>congestion level of link</td>
</tr>
</tbody>
</table>

### 6.3.4 OSPF API Protocol

Our resource discovery approach is based on extending link-state routing protocols such that the internal link-state database can be accessed and the
protocol can distribute properties about node resources. To gain access to the OSPF routing daemon, there is a need for an API that provides external applications with the following functionality:

- **Retrieval** of the full or partial link-state database of the OSPF daemon. Whenever an LSA update arrives at the OSPF daemon, the external application should be informed and a copy of the LSA sent to the application. This way the external application is always kept synchronized with the OSPF daemon’s internal database.

- **Origination** of application-specific opaque LSAs (of type 9, 10, or 11) which are then distributed transparently (according to their flooding scope) to other routers and received by other applications through the API.

In the following, we describe our newly developed protocol between the OSPF daemon and external applications for synchronizing with the link-state database and distributing opaque LSAs. The protocol phases are illustrated in Figure 6.3.

- **Connection initiation with OSPF daemon.** The communication between an application and the OSPF daemon is based on two channels for synchronous and asynchronous messages. Requests operate synchronously using a two-way scheme, with response messages flowing in the opposite direction containing status codes denoting whether the request succeeded or failed. Indications and notifications operate asynchronously by sending messages one-way from either the application or OSPF daemon, respectively. These notifications are used to report a new, updated, or deleted LSA via the network, changes of the neighbor connectivity and failures of interfaces.

- **Link-state database synchronization.** Once the communication has been established between the application and the OSPF daemon, the application needs to be synchronized with the OSPF daemon and initiates a link-state database synchronization request. The OSPF daemon then dumps its internal link-state database by sending a sequence of LSA update notifications.

- **Opaque type registration.** Once the application has retrieved the complete LSDB, it can register the opaque types for which it wants to originate opaque LSAs. A given opaque type can be registered only once to
prevent contention between competing client applications. As long as opaque types are unique, an application can register several opaque types.
• **Origination of own opaque LSAs.** Once the OSPF daemon has learned that an opaque-capable neighbor’s state is full, the OSPF daemon will notify for each registered application that it is ready to originate application-specific opaque LSAs.

Note that there must be *at least one* opaque-capable neighbor before a router can originate opaque LSAs. A router distributing a new LSA must first be synchronized with its neighbors (state full). The reason for this restriction is that a neighbor router might still have stalled opaque LSAs (with identical LSA- and opaque types) which could not be flushed due to a router crash. If the router comes up again and starts originating new opaque LSAs (which again have the initial sequence number), the stalled LSAs are considered to be newer (higher sequence number) and the new originated LSA will be simply ignored. However, if a router first synchronizes its database with its neighbors before originating new opaque LSAs, the router will detect stalled opaque LSAs and can flush them first.

Once the application has been notified that the OSPF daemon is ready to accept opaque LSAs, the application can begin originating its own LSAs. The OSPF daemon then completes the LSA header, installs the LSA into the link-state database, and floods it throughout the network.

• **Update own opaque LSA.** Applications can update the content of an opaque LSA by re-originating the opaque LSA with a modified content. When distributing updates, the router increments the opaque LSA’s sequence number such that all routers consider the update as more recent and discard older LSA instances.

The OSPF protocol requires all LSAs to be periodically refreshed (with an identical LSA content but incremented sequence number), otherwise distributed LSAs expire\(^\text{1}\). Once an application has distributed an opaque LSA, the OSPF daemon refreshes LSAs, that is, the application does not need to take care of refreshing. Whenever an LSA gets refreshed, the application receives an LSA update notification.

• **LSA updates from neighbors.** Whenever an LSA of any type is received over the network, the OSPF daemon immediately notifies all applications. Using this scheme, all applications have a synchronized copy of the LSDB all the time.

---

\(^1\) If not refreshed earlier, LSAs expire after 3600 seconds (MaxAge) [101].
6.4 Discussion

Using the underlying routing protocol as a mechanism for the discovery of processing capabilities is an effective method. The link-state database built by the routing protocol provides complete connectivity information from which a network topology graph can be generated quite easily. Application-specific information about processing capabilities can be disseminated using opaque LSAs.

The use of our extended OSPF daemon is not limited to the ANCS. In fact, the design has been kept very general so that it can be used by other applications. For example, the Service Routing Redundancy (SRR) [59] daemon uses the extended OSPF daemon to distribute the inter-dependencies and current state of application-level services.

Since routers use a flooding-based dissemination protocol, the approach is only applicable for medium-sized networks. A study [124] has shown that OSPF works well in an enterprise network covering about 250 routers organized in eight areas. That is, for service providers that use the public Internet as a lower layer infrastructure for the construction of overlay networks our OSPF-based resource discovery mechanism might still be sufficient since the size of the overlay network usually remains small. Using overlays, network-enabled services are provided within the corporate-specific virtual network, with designated active nodes used for packet processing.
However, the OSPF’s flooding-based update mechanism runs into scalability problems for very large networks. With \( n \) OSPF routers in a network, a network topology update can theoretically generate on the order of \( n^2 \) LSA packets. This phenomenon, known as the LSA \( n \)-squared problem [2], severely degrades network performance and scalability. To reduce the number of LSA updates, hierarchical network architectures are needed to summarize the network topology graph for parts of the network further apart. Several hierarchy building methods have been proposed in the literature, ranging from a rather simple two-level hierarchy (OSPF areas [101]) to an arbitrary number of hierarchies (PNNI [8], HIGCS [60]). Note that an optimal hierarchical network reduces the number of LSA updates from \( O(n^2) \) to \( O(\sqrt[3]{n} \cdot n) \) [2].

To make our approach scalable for very large networks, additional hierarchies would need to be introduced on the ANCS level (but keeping the current OSPF-based dissemination mechanism), with an inter-ANCS protocol used for exchanging aggregated topology and processing site information. Approaches such as PNNI/PAR [9] have demonstrated that this is indeed feasible, but an implementation of such an approach would exceed the scope of this thesis.

### 6.5 Related Resource Discovery Mechanisms

In \textit{AMnet} [127], available service sessions are announced based on a multicast tree. A session announcement contains a description of the session and information such as bandwidth and delay requirements. A session consists of one or multiple service level groups, each transmitted using its distinct multicast group. Based on this session description, receivers choose their service level by joining the appropriate multicast groups. To select a node for a given processing function, a receiver sends out an evaluation packet that contains a program to be executed on each node along the path towards the root of the session. Each node along the path determines whether it can satisfy the qualifications for providing the receiver-requested service. If it represents a better match than what is marked in the packet, the node writes its result into the packet. Once the packet reaches the root, it contains the address of the best-matching node and the receiver can then request to install the service on that node. Services can only be placed on the reverse path and AMnet does not provide a the topology graph.
In Jini [137], a lookup service maps interface service descriptions (indicating the functionality offered by a service) to a set of objects that implement the service. Associated with the service are descriptive entries that allow for the selection of services based on properties. Discovery is referred to the process of finding a suitable lookup service. A lookup service generally announces its presence on the network via a broadcast message. Once a lookup service has been found, an application may request a service with certain capabilities. The lookup service then provides one or more objects implementing the service. The use of a resource is granted for a specific lease time which must be renewed periodically. This allows for the expiration and cleanup of services that are no longer required. Jini allows finding an application layer service with a specific interface and certain properties, however the network topology cannot be retrieved.

TSpaces [86] is a communication middleware with a set of network communication buffers called tuple spaces and a set of classes that implement an API for accessing those buffers. TSpaces allows heterogeneous, Java-enabled devices to exchange data by mapping all their system-specific services (e.g., printing service) to a standard tuple representation. In the TSpaces model, server applications generate tuples and send them to a TSpaces server. Then, client applications request for specific tuples and retrieve them for processing. This creates a single interface for talking to various different devices since all services use tuples as the common language. While Jini and TSpaces offer overlapping functionality, they have different purposes. Jini is oriented towards discovery of a single device offering a specific capability, whereas TSpaces is oriented towards coordination of several devices to allow them to intercommunicate. Again, the network topology is hidden and cannot be obtained.

6.6 Summary

To facilitate the automatic deployment of new services within active networks, there needs to be mechanisms for locating network-embedded processors and determining their capabilities and work load. We propose a resource discovery mechanism that extends existing routing protocols such that these protocols can also distribute properties about network resources. Specifically, we have described how the OSPF protocol can be used for the
dissemination of application-specific content based on opaque LSAs and how external processes can be given access to the routing protocol’s internal link-state database. The information stored in the link-state database facilitates building a topology graph of the underlying network, annotated with attributes describing processing sites. Obtaining this network graph is a fundamental requirement for automating the service deployment process.

In the following chapter, we describe an algorithm that can map the processing requirements of an application (expressed as an active pipe) onto the underlying network graph.
In Chapter 5, we introduced the active pipe formalism for specifying transmission and processing requirements of applications and Chapter 6 proposed a resource discovery mechanism that distributes information about the location and capabilities of processing resources. In this chapter, we focus on mechanisms that can translate application demands (expressed as active pipes) onto physically available processing resources embedded in the network. This opens up new challenges in finding feasible algorithms that can determine the location of processing nodes and paths transiting these sites in the required order, preferentially while minimizing network costs.
In the following, we first introduce the general concept of constraint-based routing and then look how constraints are defined in the context of active networks. We then present an algorithm for mapping an active pipe expression onto the available network resources. For each of the deployment cases of a processing step that can be expressed in an active pipe, we demonstrate how the problem of finding optimal processing sites can be mapped onto a generic shortest-path problem based on various graph transformations.

7.1 Concept of Constraint-Based Routing

Finding paths that are subject to a variety of constraints is called constraint-based routing [30]. The objective is to find a path from a given source node to a destination such that the path is optimal with respect to some metric while not violating any constraints. Various types of constraints can be formulated: One constraint type would be the ability to find paths that satisfy certain performance characteristics. For example, link constraints specify restrictions on the use of links such as a certain amount of residual bandwidth must be available. Path constraints define end-to-end requirements on a path such that the latency from the sender to the receiver does not exceed an upper bound. Another type of constraints would be administrative as is common in the context of traffic engineering, as for example, when a network administrator wants to direct certain traffic trunks (e.g., all web traffic) through certain links. Constraint-based routing may also include a combination of performance and administrative constraints. For example, computing a path that provides a certain available bandwidth and at the same time excludes certain links.

To make the routing problem even more challenging, routing algorithms need to determine not only feasible paths that satisfy all constraints but also need to consider the optimization of network resources, achieving global efficiency if possible. The utilization of network resources can be measured by an abstract cost metric. For example, the cost of using a link can be defined as a function of the bandwidth utilization. Or from a latency point of view, the cost of a link can be a function of the delay that affects packets. Other common metrics are jitter, loss rate, and packet error probability. Hence, the cost of a path can have multiple metrics, each expressed in its own unit.
Multiple constraints often make the routing problem intractable, meaning that the complexity of the path computation algorithm grows exponentially with the size of the network. For example, the problem of finding a path with at least 4 Mbit/s and a maximum delay of 50 ms has two independent path constraints and is proven to be NP-complete [139], thus no algorithm for this problem finding the optimal solution scales to networks with more than a few nodes. Practical solutions should be of the same computational complexity as Dijkstra’s [46] shortest path algorithm.

Approximation algorithms that produce feasible paths but not necessary optimal solutions are often good enough. We believe this trade-off is acceptable since optimal routing must generally be sacrificed for scalability reasons. A common method to tackle NP-complete routing problems is called sequential filtering, under which a combination of metrics is ordered in some fashion, reflecting the importance of different metrics (for example bandwidth followed by delay). Paths based on the primary metric are computed first and a subset of them are eliminated based on the secondary metric and so forth until a single path remains. This scheme implements a trade-off between optimality and computational complexity.

Note that constraint-based routing cannot be supported by the current IP routing model. One of the main reasons is that constraint-based routing requires path calculation at the source. Various sources may have different constraints for a path to the same destination, known only to the source router, but not to any other router in the network. Remember that conventional IP routing performs path computation in a distributed manner, not taking into account constraints specified by sources. Another reason is that the traditional IP forwarding paradigm cannot support routing along paths other than the IP default path, thus explicit routing capabilities are required as will be discussed in Chapter 8.

7.2 Constraint-Based Routing with Processing

The routing problem for active networks can be seen in the context of constraint-based routing. However, constraints are defined somewhat differently since feasible solutions need to transit processing functions installed in the network. In particular, processing constraints define requirements on the
order and location of processing functions to be placed along the end-to-end path. Furthermore, utilizing a processing resource has an associated cost which is determined by the profile of the active module (e.g., required processing cycles, memory usage). Thus, we need to focus on solutions that optimize for both link and processing costs. As we noted, the time complexity of routing problems with multiple independent cost metrics such as link and processing costs is NP-complete. Because such algorithms are not scalable, we assume that processing costs are scaled to match the link cost metric. This is a convenience that will allow us to transform the routing problem into an ordinary shortest path problem with just link costs. Clearly, we are interested in solutions that are solvable in polynomial time such that they can also be applied to large networks. Specifically, we are looking for ways of mapping our constraint-based routing problem to an ordinary shortest-path problem.

In the following, we first illustrate how the routing is performed for each of the deployment cases described in Section 5.2. Specifically, we look at how the routing can be done in the case of a required or conditional module, with the processing step deployed on nodes or along links. Then we describe how the routing works for a sequence of active modules in an active pipe. The main idea is to transform all the deployment cases into specific shortest path problems and use the methods incrementally to generate a shortest path problem for the complete active pipe. Our method is based on the layered graph method which has been described in [33], [75], [76].

To state the routing problem for each of the deployment scenarios more formally, we use the following notation. We are given a directed graph, \( G = (V, E) \), with a transmission cost \( c(e) \), for each link \( e \in E \), and a processing cost \( c(v) \) for the module, for each node \( v \in V \). Let the source be defined by \( s \) and the destination by \( t \). As an example network, we use the graph

![Network with embedded processing sites](image-url)
shown in Figure 7.1. Transmission costs are denoted on the links. If links are not shown directed, the link has the same weight in both directions. The cost for processing a module is shown within the node.

7.2.1 Required Processing

For required processing, the routing algorithm’s objective is to determine the location where processing needs to be performed such that the cost of the path (sum of transmission costs on links plus the processing cost for the module is minimized). As discussed in Section 5.2, there are two cases for required processing, namely either on a node or along a link. We start with describing the case of a single required processing step and then explain how our scheme can be extended to a combination of processing steps.

Required Processing on Node

In this case, given a candidate set of nodes, $N \subseteq V$, we need to determine the optimal node $n \in N$, where processing should be done. We can solve this problem by transforming it to a generic shortest path problem by introducing layers. As illustrated in Figure 7.2, we modify the graph $G$ by making two copies which we identify as layer 1 and layer 2. For each vertex $v$ in the initial graph, let $v_1$ denote the vertex in layer 1 of the target graph while $v_2$ denotes the vertex copy in layer 2. To model the processing of modules, we add edges between the two layers. For every node $n \in N$, where processing may occur, we add an edge $(n_1, n_2)$ in the target graph and let the link cost of $(n_1, n_2)$ be the processing cost on node $n$, $c(n)$.

The source node in layer 1, $s_1$, is the source for this new graph and the destination node in layer 2, $t_2$, is the destination node for the new graph. This ensures that the path from the source to the destination is forced to go through exactly one processing site. To solve the routing problem with one mandatory processing site, we find a least-cost path in the target graph using a shortest path algorithm. The path can then be mapped back to the original graph by projecting the two layers onto a single layer and the processing is
optimally performed where the path crosses the two layers. A proof for this method is presented in [33].

**Figure 7.2: Graph transformation for required node scenario**

**Required Processing on Link**

Given a candidate set of directed links, \( L \subseteq E \), the application wants the processing to be done on the node adjacent to exactly one link among the set. In this case, the procedure of determining the optimal location is similar to the previous case, except for one small variation. Again, we transform the graph \( G \) by making two copies as illustrated in Figure 7.3. For each vertex \( v \) in the initial graph, \( v_1 \) denotes the vertex in layer 1 of the target graph while \( v_2 \) corresponds to the vertex copy in layer 2. Now for every edge \( e \in L \), which connects nodes \( i \) and \( j \), we add a new diagonal edge \((i_1, j_2)\) in the target graph between the two layers. The weight of this new edge is the sum of the processing cost at the node and the transmission cost of the link. This is given by the expression \( c(k) + c(e) \), with \( k = i \) when processing should be
done at the node preceding the link, or $k = j$ for processing at the succeeding node, respectively. The shortest path from $s_1$ to $t_2$ gives an optimal path that transits the processing site, with the link crossing the two layers denoting the optimal location to do processing.

The proof of this transformation is straight-forward. To reach the destination $t_2$ from $s_1$, we need to traverse one of the edges connecting the two layers. The edge weights of these links ensure that when crossing a layer, we take into account the cost of installing the active module. Since the shortest path algorithm selects a route with minimum costs, the edge that crosses the layers is the optimal location to install the active module.

Note that in the case of required processing on a node or link, all paths from any node in layer 1 to any node in layer 2 satisfy the constraint that an active module must be installed on exactly one node of the candidate set. In other words, to reach any node in layer 2, we have to traverse one of the crossing links connecting the two layers. No matter how we modify layer 2
to do other processing, any path from layer 1 must use one of the links connecting the two layers. This shows the correctness of the shortest path problem that is constructed for the \textit{required} module scenario.

**Combination of Required Steps**

The layering model can be extended to include \textit{several} required computational steps, with each processing step having its own location constraints to be considered. For example, a secure data transmission application requires deploying an encryption and decryption step in the source and destination domains, respectively, meaning that the location constraints of processing steps are defined individually.

For each \textit{required processing step}, we extend the graph by adding a \textit{new layer} according to the transformations defined previously. Figure 7.4 illustrates an example graph transformation for $k = 2$ required processing steps. The target graph $G$ has $k + 1$ layers, each layer representing a copy of the
original graph. We let \( v_i \) denote the copy of node \( v \) in layer \( i \). Let \( N_i \) be the set of valid processing nodes that satisfy the constraints for processing step \( p_i \). For each \( n \in N_i \) we add a vertical edge \((n_i, n_{i+1})\) in the target graph and set the cost to \( c_i(n) \) from the original graph. Similarly, for each link required step, let \( L_i \subseteq E \) be the candidate set of valid processing links for processing \( p_i \). For each \( e \in L_i \) we add a diagonal edge in the target graph between two layers as described previously.

As shown in Figure 7.4, two processing steps \( p_1 \) and \( p_2 \) are needed. The first required step \( p_1 \) on a node is done between the top and the middle layer and the second required step \( p_2 \) along a link is performed between the middle and the bottom layer. Running a shortest-path algorithm on this layered graph and projecting the layers onto a single layer gives the optimal location for \( p_1 \) and \( p_2 \) and a corresponding transiting path. As can be seen from the example, both processing steps can be executed on the same node. This is not a problem as long as on the selected node \( p_1 \) is executed before \( p_2 \).

### 7.2.2 Conditional Processing

In the conditional processing case, active modules should be installed at locations that satisfy a given condition along the path (or some subsegment thereof). Since the installation of a conditional module has a cost that needs to be considered, the decision on whether to process can affect the path routing. This means that if the cost for installing a conditional processing module is high, it might be advantageous to route around the conditional module (a scenario that demonstrates this case is illustrated in Section 10.2.4).

In the following subsections, we look at the cases for conditional processing on nodes and links (which we have presented in Section 5.2) and describe how the locations for placing these modules can be determined.

**Conditional Processing on Nodes**

In this case, we are given a candidate set of nodes, \( N \subseteq V \), and we need to determine the set of nodes where the active module must be installed such that the path cost is optimal. More formally, suppose there exists a path \(<s, v_1, v_2, ..., t>\) from the source \( s \) to the destination \( t \), and if \( v_i \in N \), then an active module must be installed on that node \( v_i \). This can be solved as fol-
For every node \( v \in N \), let \( \{e_1, e_2, e_3, ..., e_n\} \) be the set of outgoing edges. As shown in Figure 7.5, the transformation on the graph is to increase the edge weights of the outgoing links by the processing cost of the node (suitable nodes are indicated with arrows). That is, \( c_{new}(e_i) = c(e_i) + c(v) \). The source and destination nodes remain the same.

![Graph transformation for conditional nodes scenario](image)

Figure 7.5: Graph transformation for conditional nodes scenario

Proof of the correctness of this method is, again, straight-forward. When a path transits a node, the processing costs at the node are taken into account by the increased link weight of the outgoing links. Thus, when a path goes through a node, an active module can then be deployed on that node since its cost has been considered. Since the shortest path algorithm optimizes total costs, both link and processing costs are minimized.

### Conditional Processing on Links

In this scenario, the application wants modules to be installed at all links in the path that belong to a candidate set of directed links, \( L \subseteq E \). This case is similar to the node case, except that the link weights are increased corresponding to the links in the candidate set only. Thus, the transformation as illustrated in Figure 7.6 on the graph is, for every edge \( e \in L \), to increase the edge weight by the processing cost of the node adjacent to that link. That is, \( c_{new}(e) = c(e) + c(v) \), where \( v \) is either the node preceding or succeeding the link \( e \).

The proof directly follows from the proof of the previous case with the difference that the active module needs to be installed if we go through the particular links only. Thus, the increase in the edge weights of these links en-
7.2. Constraint-Based Routing with Processing

sures that if any of the links is taken, then the cost includes the processing cost as well.

![Graph Transformation for Conditional Links Scenario](image)

*Figure 7.6: Graph transformation for conditional links scenario*

Note that in both conditional cases, *all* paths from *any* source to *any* destination satisfy the given condition that if the path transits any node (or link) that belongs to the candidate set, then an active module is installed there. This is true because the link weights are increased in a way that the processing cost of the active modules is included in the link costs. Thus, the shortest path problem includes the cost of installing the active modules.

**Combination of Conditional Processing**

The graph transformations can also be applied when a combination of conditional modules needs to be installed within the same network. In this case, the graph $G$ is transformed using the methods described for the conditional module cases individually for each of the modules on the same graph. This results in a *superposition* of graph transformations, with link weights modified for each of the conditional modules. All paths between any two nodes in the graph satisfy the required constraint that whenever going through a selected node or link, the corresponding active module needs to be installed. This follows directly from the observation mentioned about the conditional module transformations above. Also, if the installation conditions for two different modules overlap (i.e., some node or link is a candidate for installing the active module for both cases), then the flow has to be processed by *both* modules at the same location. We assume that active nodes support chaining of active modules for a single flow, that is, a flow can be processed by multiple code modules in a sequence on the same node.
7.2.3 Transformations for Active Pipe

We shall now define an algorithm to perform routing for all processing steps in an active pipe that includes a combination of required and conditional steps. Our approach is to combine the transformations described in Section 7.2.1 and Section 7.2.2 and produce a target graph that represents the solution for the active pipe.

As described in Section 5.1, an active pipe includes a source \( s \), destination \( t \), a set \( P \) of required processing steps, and set \( Q \) of conditional processing steps:

- For each required step \( p_i \in P \) we extend the graph by adding a new layer according to the transformations defined previously. With \( k \) the number of required processing steps, the target graph \( G \) has \( k + 1 \) layers, each layer representing a copy of the original graph. Between layers, we add edges in the target graph as described for the required processing transformations.

- Each conditional step \( q_j \in Q \) results in modification of links weights of the current layer \( c \). Initially, \( c = 1 \), and whenever a new layer is being added, \( c \) is incremented by 1 to reflect the most recently added layer. Each conditional processing step performs modifications of the current layer only. As illustrated previously, a number of conditional modules can be installed together on the same layer, resulting in modifying link weights for each of the modules.

Figure 7.7 describes the graph transformations for an active pipe containing two required steps \( p_1, p_2 \) and a conditional processing step \( q_1 \). The target graph includes three layers. The nodes suitable for a processing step on a given layer are indicated with arrows. The inter-layer edges represent the required steps \( p_1 \) and \( p_2 \), and changes of the link weights on the middle layer represent conditional processing \( q_1 \). The resulting graph satisfies all processing requirements given by the active pipe because for each step a corresponding graph transformation has been performed. Solving the shortest path for the resulting graph returns the optimal places for \( p_1, p_2 \), and in addition all the locations for \( q_1 \).
Note that to compute a path for $k + 1$ layers, the running time for a solution of the layered graph is essentially $k + 1$ times the cost of computing a shortest path in the original graph, thus the total computational still remains polynomial.
7.2.4 Non-Simple Path Solutions

The solution of the layering model produces an optimal path from the source to the destination which includes $k$ required processing steps. However, it is important to note that this solution can produce non-simple paths\(^1\) as illustrated in Figure 7.7, where all of its vertices are no longer distinct and duplicated vertices may exist. That is, a given vertex can be visited multiple times and a node can be used more than once as a processing site. The layering model guarantees that processing steps are performed in a strict order but does not prevent solutions where the same node is visited repeatedly.

Such solutions are generally acceptable (or may be even desired) since applications want to deploy functions in the network that are executed in a given order, regardless of the exact location within the network. It is assumed that active nodes support chaining of active modules, that is, a flow can be processed by multiple modules on the same node. If such solutions are not preferred, applications can prevent processing steps from being placed at the same site by defining disjunctive sets of location constraints.

Since traditional IP does not support forwarding along such paths\(^2\), we describe our own explicit path signaling protocol in Chapter 8.

7.3 Extension to Hierarchical Routing

So far, we explained how we can determine processing sites assuming we have a complete view over the network. However, this assumption is not realistic for larger networks consisting of thousands of nodes. In this section, we address scalability issues and show how our routing scheme can be extended to very large networks.

To make our approach scalable to very large networks, we aggregate information about sections of the network, similar to the hierarchical PNNI [8] scheme for routing in ATM networks. Nodes in the network are partitioned into groups of interconnected nodes called peer groups. A peer group ap-

\(^1\) Also referred to as “walk”.
\(^2\) The IP source option limits the number of hops to 8, thus we do not consider this as a valuable alternative.
pears as a single logical node at the next level of the hierarchy. Figure 7.8 shows a simplified example of a physical network partitioned into three peer groups and a logical hierarchy level which has been built on top.

![Hierarchical network structure](image)

**Figure 7.8: Hierarchical network structure**

Each peer group has an assigned peer group leader that acts as a logical group node for the next hierarchy level and which aggregates and distributes topology and state information to maintain the hierarchy. Apart from its specific role in summarizing and distributing peer group information, it acts like any other node. Each node only maintains a partial global state with groups of nodes aggregated into logical nodes. The size of the aggregated state is logarithmic in size compared with the complete global state information. Several approaches exist for aggregating state information [8], [60]. As network state is aggregated, additional imprecision is introduced, which is an issue of concern since it can have a significant negative impact on the selection of routes [58].

With the introduction of network hierarchies, the routing of an active pipe needs to be performed in a hierarchical way, that is applying the constraint-based routing algorithm first on a higher level (using aggregated information) and then executing the same algorithm on individual peer groups at lower levels (which have more accurate topology information). When an active pipe request arrives at a router, that node is responsible for determining the hierarchy level that is needed to process the setup by examining the network addresses of the source, destination, and intermediate processing sites. If not all nodes belong to the peer group of the current hierarchy level, the
routing request is delegated to the peer group leader which acts as a logically higher peer group node. This process is repeated until a peer group leader is found whose scope of the network includes both the source, destination, and intermediate processing.

Figure 7.9 illustrates this process where the routing has been delegated to a higher-level peer group that includes logical group nodes $A$, $B$, and $C$. The peer group leader executes the routing algorithm based on summarized information and selects a path and processing sites that seem to be capable of providing the requested computation. In the example, it selects the path $<s, A, B, C, t>$ with processing steps $p_1$ and $p_2$ within $A$ and computation $p_3$ in $C$. Since the final path needs to be a path on the physical topology, the routing system uses a divide-and-conquer approach. When a path segment transits a logical node, it delegates the routing through that particular peer group to the lower hierarchy level.

The routing algorithm is recursively applied to a more detailed subsection of the initial network. In the example given, peer group $A$ performs path and processing site selection through $A$ itself and chooses the path $A.1, A.2, A.4$ with processing $p_1$ on $A.2$ and $p_2$ on $A.4$. The routing algorithm also selects an external path to peer group $B$ that seems to be a good candidate to reach $B$ based on $A$'s topology and state information.
Once the routing system has determined a path through peer group $A$, it initiates path determination through peer group $B$. As chosen by $A$, peer group $B$ will be reached from node $A.4$. Peer group $B$ runs a simple shortest-path algorithm (since no processing is required in peer group $B$) to reach $C$ and selects the path $B.1, B.2$. Finally, the routing system initiates routing through $C$. Again, peer group $C$ runs the routing algorithm within its own peer group and chooses the path $C.2, C.3, C.4$ with processing $p_3$ on $C.3$. Finally, we have found in a distributed manner a complete path from the source to the destination that includes the required processing sites.

Note that the selected path may not be globally optimal, since $A$ does not have complete information about the neighboring peer group $B$. However, we believe this trade-off is acceptable since optimal routing must generally be sacrificed for scalability reasons.

7.4 Related Constraint-Based Routing Algorithms

In [150], the problem of finding service paths in a media service proxy network is discussed. A service path connects a pair of communication end points, and a chain of service proxies that process the media stream in different ways (such as transcoding). The authors argue that media streams should follow a safest service path. As opposed to the shortest path, the safest path can be seen as shortest-widest path for both computing as well as bandwidth resources with respect to end-to-end service availability.

End-to-End Media Paths [103] uses a centralized pattern-matching method to perform service path resolution. The system determines the possible routes between the source and sink by computing the $k$-shortest paths. Then the systems attempts to match one or several templates, which define valid compositions expressed as regular expressions, against the nodes on the $k$-shortest paths. However, this method does not guarantee an optimal service path.

Ninja [57] proposes logical paths that consist of operators and connectors that perform a given service. A logical path is instantiated into a physical path by assigning operators to intermediate nodes. Ninja selects processing nodes for operators randomly and the runs a shortest-path algorithm to con-
nect the chosen nodes through connectors. This can produce highly inefficient paths, specifically if the path includes many operators.

In Darwin [27], high-level resource selection is performed by Xena that determines the resources based on the virtual mesh. Xena translates the virtual mesh onto network resources by expressing it as a boolean optimization problem, which is generally NP-hard, thus this approach is not scalable for larger networks.

In [34], an extension of the layered graph method is proposed to explicitly model limits on link bandwidth or processing capacity. Even when a suitable path solution within the layered graph can be computed, it is possible that when projected to a single layer (representing the physical network) some of the links are over-used. Since the problem of finding an optimal path when capacity is constrained is known to be NP-hard, an efficient heuristics has been proposed for which the network performance closely approximates the performance that can be achieved with optimal path selection. Specifically, the Dijkstra SPF algorithm has been modified to include an additional check for over-used resources. This link capacity tracking increases the runtime of the complexity of the algorithm only minimally, however it does not guarantee to always find a valid path, even when a path exists.

7.5 Summary

With the proliferation of processing capabilities within networks, there is a need for effective mechanisms that map application demands to the underlying network and that assign appropriate processing resources to applications. In this chapter, we presented an algorithm for translating an active pipe expression onto network resources. Our approach is based on a set of graph transformations that map the routing problem with processing constraints onto a generic shortest-path problem. In particular, we described how the four basic deployment cases can be expressed as graph transformations, as well as how the scenarios can be combined to a multi-layered graph. In addition, we explained how the routing algorithm can be extended to operate on a hierarchically organized networks.
Chapter 8

Explicit Path Signaling and Code Deployment

In the context of programmable networks, manual deployment of network services on routers seems impractical, especially on a network-wide scope. Since the effort of manually setting up a network service is very high, there is a need for mechanisms to automate the configuration of network state. This chapter presents a signaling protocol that installs and configures processing modules on selected nodes (as computed by the routing algorithm) and establishes associated state across the network. Such explicitly routed paths differ from the conventional path calculated by the interior gateway protocol. An explicit route consists of a sequence of active routers to be visited and for each router a (possibly empty) set of plugins to be installed. As we have illustrate in Section 7.2.4, optimal service paths can be non-simple,
where nodes are visited repeatedly, which must be taken into account. Clearly, the signaling protocol plays a crucial role in automating the setup process. A successful signaling mechanism is required to perform a number of important tasks:

- *Install and configure* plugins in the networking subsystem, and bind them to the application's traffic flow.
- *Establish forwarding state* in network nodes such that packets are routed along the predefined path and processed by intermediate plugins.
- *Maintain state* of an explicitly routed path by periodically refreshing the state to deal with inherent network failures.

### 8.1 Explicit Path Routing Protocol

Our proposed *Explicit Path Routing* (EPR) protocol [77] supports per-flow-based explicit path establishment for *one-way, unicast flows* routed through a predefined list of hops and the installation and configuration of processing modules along such paths.

In the following we describe the design and message formats used by the EPR protocol. The actual implementation of the EPR daemon will be described in Section 9.3.2.

#### 8.1.1 Setup and Teardown

As illustrated in Figure 8.1, the EPR protocol establishes a service path between a source router (A) and destination router (H). An application that requires a service path then sends a corresponding request to the source router (A). The path establishment is then based on *two phases*:

- In the first phase, the protocol verifies whether sufficient resources are available along the *downstream path*. Beginning at the source, each node checks whether the required resources are locally available and if true, reserves (but does not allocate yet) resources, and forwards the reservation request to the next node along the downstream path. This process is repeated until the destination node is reached. Once the first
phase of the setup process has been completed, it is assured that sufficient resources are available.

- In the second phase, the actual allocation of network resources takes place. This happens along the reverse path, that is, on all routers from the destination towards the source. This includes the installation of flow-specific filters such that packets matching the filter are forwarded on the corresponding outgoing interface, and the installation and configuration of plugins and binding them to the filter.

Once all state has been established along the path, the application is informed and can begin transmitting its traffic.

If during the first phase a reservation request is refused due to limited resources, the path setup process cannot continue and is aborted. The node then sends a reservation release message along the reverse path so that nodes that have already reserved resources can free them. If the establishment fails, the application is notified that the path could not be set up.

The EPR protocol uses TCP as the transport mechanism between EPR peers (hop-by-hop) for sending control messages to establish paths, deploy plugins, and release resources. This guarantees reliable distribution of control messages.

However, for both path forwarding and plugin state stored on nodes, EPR uses soft-state to take into account that nodes and links are inherently unreliable and can fail. For that reason, an application that sets up a path is required to refresh the path (by sending the path establishment request

![Figure 8.1: Protocol to establish explicitly routed path](image-url)
periodically), otherwise nodes will purge path and plugin state once the time-out expires. Path tear down works analogous to the path setup process, with a release request used instead.

### 8.1.2 Description of Protocol Messages

The EPR protocol defines the following messages which are used between the application and the source router, and between routers along the service path.

#### Create Path Message

The CREATE_PATH request is transmitted by path-initiating application to the source router, and then towards the destination router for establishing an explicitly routed path. The message contains the following different sub-objects (Figure 8.2):

- **FLOWSPEC**
  The flow specification describes the format of packets that follow the explicit path using the tuple:
  \(<\text{source addr}/\text{mask}, \text{dest addr}/\text{mask}, \text{source port}, \text{dest port}, \text{protocol}>\)
  Any field can be wildcarded by setting the appropriate value to zero. Network addresses can be partially wildcarded with a prefix mask.

- **EXPLICIT_ROUTE**
  An explicit route object is encoded as a series of nodes to be traversed. Each hop is specified with its IP address and a list of optional plugins to be deployed on that node. In the current EPR implementation the hops must form a strict explicit route\(^1\).

- **PLUGIN**
  The plugin object describes one or multiple plugins that need to be installed on a node. It contains the plugin name, followed by initial configuration parameters.

On the arrival of a CREATE_PATH request, a node first verifies whether its address is indeed listed in the explicit path. If it is, the node then checks if

\(^{1}\) Loose routes could be supported without much implementation effort, however paths computed by the ANCS are strict anyway.
it is the last node along the path to be visited. On the last node, only plugins need to be installed, but no further routing decisions need to be made. In case we have not reached the end of the path, the node verifies if the next node listed in the path list is a valid neighbor. If true, the node then adds the flow to its internal flow table. If packet processing is required as well, the EPR daemon checks for an existing kernel plugin class, installs the plugin into the kernel if needed, creates a new instance of the plugin, and binds the plugin instance with the flow. Once all state has been established on the node, the EPR daemon forwards the request to the next neighbor listed in the CREATE_PATH message. This procedure is repeated until the last node along the path is reached.

Then, beginning at the end node, each node sends an ANSWER message upstream to the node the request originated from, and the status code indicates whether the service path could be established successfully.

**Release Path Message**

The RELEASE_PATH request tears down a previously established path, with a message format very similar to the create path request. The release path message contains a flow specification (which uniquely identifies the
flow) and an explicit route object to direct the message along its explicit path. However, no plugin objects are present in the request. Since each node stores the plugins associated with a flow, the active router is able to free all plugin instances bound to a flow when needed.

**Path Status Message**

The EPR protocol also supports the **STATUS** request for the retrieval of path information from remote nodes. This message type is used for debugging purposes. When received, the node returns all its internal flow and plugin state to the originator of the status request.

**Answer Message**

An **ANSWER** message is transmitted in response to the receipt of a **CREATE_PATH**, **RELEASE_PATH**, or **STATUS** message. It contains status information such as the successful establishment of the path segment or an error code in case the setup failed.

**8.1.3 Establishment of Node State**

State information on routers is stored as *soft-state* to deal with inherent network failures. The node state consists of the forwarding state such that flows are routed along explicit paths, the installed plugin code, and the binding of plugin instances with associated flows.

**Explicit Path Forwarding**

In the following, we describe how we can perform our own explicit path routing, that is, move packets along a predefined set of nodes. Routers are required to maintain state on a per-flow basis. When an active router receives a request to establish a new path, it records the flow description (flow filter), the outgoing interface to be used by the flow, and the plugins needed for processing (Figure 8.3). Packets matching the flow specification are then sent out of the given interface. This state is released if either an explicit tear down request is received or if the path request is not refreshed within the time-out
interval. To establish a complete path, filter entries are added to all nodes along the path.

<table>
<thead>
<tr>
<th>source address</th>
<th>destination address</th>
<th>source port</th>
<th>dest port</th>
<th>transport</th>
<th>incoming protocol interface</th>
<th>outgoing interface</th>
<th>plugins instances used for flow processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0/0</td>
<td>129.132.66/24</td>
<td>1050</td>
<td>80</td>
<td>TCP</td>
<td>eth0</td>
<td>eth1</td>
<td>plugin1</td>
</tr>
<tr>
<td>129.132/16</td>
<td>1050</td>
<td>2048</td>
<td>21</td>
<td>*</td>
<td>eth1</td>
<td>eth2</td>
<td>plugin2</td>
</tr>
<tr>
<td>0.0.0.0/0</td>
<td>0.0.0.0/0</td>
<td>*</td>
<td>25</td>
<td>UDP</td>
<td>eth2</td>
<td>eth0</td>
<td>plugin3</td>
</tr>
</tbody>
</table>

**Figure 8.3:** Filter table for flow-specific forwarding and processing

As discussed in Section 7.2.4, when considering processing sites a service path does not need to be a straight IP path, in which all of the nodes are distinct and no duplicated nodes exist. To support non-simple paths, the forwarding mechanism must consider the *incoming port* where the packet has been received from. For this reason, flow filters consist of a six tuple, including the incoming interface. The interface field can be used to build non-simple paths, where a packet can transit a node multiple times. The filter target also includes a plugin chain, that is, a list of plugin instances that need to be executed when the filter matches.

Considering the incoming interface into the forwarding decision allows setting up non-simple paths. However paths can enter a node via the same interface only once, otherwise a node could not distinguish whether a packet previously traversed the node. This limitation could be overturned if incoming packets were marked with a tag to be used for subsequent forwarding decisions\(^1\). Since this case seems to be rather rare, we currently support only non-simple paths where traffic enters the node from a different incoming interface.

**Plugin Installation and Flow Binding**

In addition of setting up flow-specific routes, the EPR protocol allows installing and configuring plugin modules on selected nodes. To support this feature, the path establishment message includes a list of nodes where plugins need to be installed. If the address for a plugin matches, the node checks

---

1. Another possibility of distinguishing whether a packet previously entered a node could be comparing the TTL.
whether the referred plugin class has already been loaded into the kernel. If it is not present, the plugin loader retrieves it from a remote code server and verifies the consistency by checking the module’s digital signature [31]. Then the module is loaded into the node’s execution environment. Subsequently, the EPR daemon creates a new plugin instance, invokes the configuration method, and binds the plugin instance with the filter describing the flow. Once the path has been established and the required plugins deployed, the application can begin transmitting data which will be forwarded along the path and processed by intermediate plugins.

8.2 Related Signaling Mechanisms

Several resource allocation protocols supporting explicit per-flow routing and allowing functions to be deployed in the network have been proposed in the literature. This section briefly describes a few of these mechanisms.

The IP source routing option [113] provides a means for the source of an IP datagram to supply routing information to be used by intermediate routers. The route data are composed of a series of Internet addresses present in the IP option header. Since there is an upper limit of the option header length, only eight hops can be explicitly routed. Also due to security concerns, IP source routing is often disabled on routers.

The ATM Private Network–Network Interface (PNNI) protocol [8] is a distributed resource allocation system that routes connections along paths that satisfy specific QoS constraints. PNNI includes a link-state protocol that distributes information about the network topology and resource availability (e.g., bandwidth, delay, jitter) of links, a routing algorithm that uses this information to select routes that satisfy QoS constraints (e.g., path with guaranteed minimum bandwidth, maximum delay), and a signaling protocol that reserves and allocates network resources along the predetermined path of the connection. PNNI uses explicit source routing, where the ingress switch computes the entire path to the destination, using information about the network topology and resource availability gathered from the link-state protocol. The route is then passed to other switches along the selected path, which in turn, allocate local resources and propagate the signaling method along the path to the destination. If during this phase an attempt to reserve local re-
sources fails (which is possible since the source chose the path based on its view of the network), the switch at the point where the allocation was unsuccessful initiates a crankback mechanism by computing a new path, allowing the path setup process to continue. Setting up explicit paths is seen as an attractive feature of ATM networks since each application can have its own specific QoS requirements which are fulfilled by the network. Nevertheless, ATM does not support the concept of processing resources as required by active networks.

The Multi-Protocol Label Switching (MPLS) [118] approach is based on a label-swapping paradigm implemented in the link layer. MPLS defines two label distribution protocols that support explicitly routed paths.

- **CR-LDP** [70], which is an extension of the Label Distribution Protocol [6], is a peer-to-peer protocol where messages are reliably delivered using TCP and state information associated with explicitly routed LSPs does not require periodic refresh. An explicit route can be strict, where all nodes are explicitly listed, or loose, allowing to define paths through portions of the network with partial knowledge of the topology.

- **RSVP-TE** [12] extends the original RSVP [18] protocol by setting up explicit label-switched paths which allocate network resources (e.g., bandwidth). The explicit route object encapsulated in a path message includes a concatenation of hops, describing a strict or loose route. RSVP-TE is based on soft state, where the state of each LSP must periodically be refreshed\(^1\).

CR-LDP and RSVP-TE are signaling protocols for MPLS that perform similar functions, but based on either hard or soft state distribution of labels.

**Beagle** [28] is a signaling protocol for the setup of structured multi-party, multi-flow applications described by an application mesh. The mesh formulates the resources to be allocated as a network graph. The Beagle protocol is based on RSVP and introduces a new route constraint object carrying explicit routing information. In contrast to protocols like MPLS and PNNI, Beagle allows applications allocating computation and storage resources required

\(^1\) Typically every 30 seconds.
for delegates. A delegate is a code segment that executes application-specific functionality on routers.

### Table 8.1: Comparison of explicit routing mechanisms

<table>
<thead>
<tr>
<th></th>
<th>IP Source routing</th>
<th>PNNI</th>
<th>MPLS CR-LDP</th>
<th>MPLS RSVP-TE</th>
<th>Beagle</th>
<th>EPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plugin deployment</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit routing</td>
<td>strict or loose</td>
<td>strict or loose</td>
<td>strict or loose</td>
<td>strict or loose</td>
<td>strict(^a)</td>
<td></td>
</tr>
<tr>
<td>Looping paths</td>
<td>yes(^b)</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Protocol messages</td>
<td>n/a(^c)</td>
<td>reliable</td>
<td>reliable</td>
<td>unreliable</td>
<td>unreliable</td>
<td>reliable</td>
</tr>
<tr>
<td>Router forwarding state</td>
<td>none</td>
<td>hard, VCI/VPI entry</td>
<td>hard, MPLS tag</td>
<td>soft, MPLS tag</td>
<td>soft, RSVP filter</td>
<td>soft, flow filter</td>
</tr>
</tbody>
</table>

- a. EPR does not implement loose routes, but this could be added quite easily.
- b. not explicitly specified but possible
- c. in-band signaling, thus no protocol messages required

From the related signaling mechanisms, only EPR and Beagle support the installation of code modules across the network, which is a strict requirement for our envisioned system. As we have seen when computing optimal paths with processing, paths can become non-simple where a given node is being visited multiple times. The EPR protocol has specifically been designed to support explicitly routed non-simple paths.

### 8.3 Summary

Active networks allow for programmability of the network layer by installing processing functions on selected nodes. Since the effort of manually establishing associated interior state is very high, we have designed the EPR protocol to automate this process. The EPR protocol supports explicit routing and deployment of processing functions on designated nodes. To guarantee that paths transit processing sites in a given order, active networks require routing that differs from the default IP path. Furthermore, when optimal allo-
8.3. Summary

cation of network resources is considered, service paths can be non-simple, meaning that some of the nodes are visited repeatedly. To take this into account, the incoming interface (where a packet was received from) needs to be considered by the forwarding decision, which is a distinct feature of the EPR protocol.
Part III:

Evaluation
Chapter 9

Implementation

To validate our proposed design concepts, we have implemented a fully functional prototype of our service framework. In the following sections, we give an overview of our implementation, identify the components involved, and describe their interactions.

9.1 Overall Node Architecture

Our implementation has been realized by various components that in combination offer applications an easy-to-use service framework. Figure 9.1 illustrates the overall node architecture consisting of the following core functions (components with a bold outline have specifically been added or modified):
Network device drivers implement hardware-specific send and receive functions. Packets that are received from an interface enter the IP stack for further processing and forwarding.

Netfilter classifies packets according to filter rules evaluated at various hooks (shown as dots) inside the IP stack. Packets matching a filter expression are passed to the PromethOS plugin framework for further processing.

The PromethOS plugin framework provides an environment for the dynamic loading of plugin classes, the creation of plugin instances as well as the execution of code for customized packet handling.

The plugin loader retrieves plugins from remote code servers as a reaction to a plugin installation request from the EPR signaling protocol.

The local resource manager monitors the node’s resources, and keeps track of what resources can be further allocated by signaling protocols.

Routing protocols compute route prefixes which are then merged into a central routing table managed by Zebra and propagated into the kernel forwarding table. Specifically, the OSPF protocol daemon has been extended such that it can provide information about the topology and capabilities of network resources.

Figure 9.1: Implementation overview of active node
• **Signaling protocols** provide resource reservation mechanisms for end-to-end QoS guarantees. In particular, the EPR protocol daemon establishes explicitly routed paths and installs plugins on selected nodes.

• The ANCS daemon accepts service establishment requests from user applications, obtains topology information from OSPF’s link-state database, determines suitable locations for placing processing functions, and interacts with the EPR protocol to establish state for explicitly routed service paths and to install the packet processing functions.

In the following, we describe the data and control planes of our architecture in more detail.

### 9.2 Data Plane: PromethOS

In the context of this thesis, we have designed and implemented PromethOS [77], a modular node architecture based on Linux 2.4. PromethOS provides a framework that can be dynamically extended by flow-specific packet processing functions called router plugins. In particular, PromethOS offers functions for loading new packet processing code into the system, creating instances of plugins, and binding them with flows. In addition, PromethOS implements mechanisms for flow-specific forwarding of data traffic along explicitly routed paths.

#### 9.2.1 Netfilter Packet Classification

PromethOS uses the netfilter [120] framework as its packet filtering mechanism. Netfilter is a set of hooks inside the kernel’s network stack allowing kernel modules to register callback functions invoked every time a network packet traverses one of those hooks. Packet classification is carried out at these hooks inside the IP stack. Packets that match these rules are handed to the corresponding target kernel module (such as the PromethOS plugin framework) for further processing. Using this mechanism, the usual packet flow through the network stack can branch to kernel modules implementing packet filtering, network address translation (NAT), or other advanced packet processing. For the IPv4 stack, netfilter defines five hooks called pre-routing, forwarding, post-routing, local-in and local-out as illus-
trated in Figure 9.1. The reason for multiple hooks is that different applications want to add processing steps at different locations. For example, encryption is performed before sending the packet out (post-routing), whereas NAT rewrites the destination address of incoming packets in pre-routing and the source address of outgoing packets in post-routing or local-out.

**Packet Processing in IP Stack**

The data path for processing packets looks as follows: As the packet enters the device driver and having passed some simple sanity checks (e.g., correct IP checksum), it is sent to the IP input processing that contains the netfilter framework's pre-routing hook. Whenever the packet traverses a hook, control is passed to the netfilter framework which looks whether a filter matches the packet. If a filter matches, the plugin framework is invoked which dispatches the packet further to the corresponding plugin. Once the plugin has processed the packet, control is returned to the location in the IP stack where the hook branched off. Next packets enter the routing code, which decides whether the packet is destined for another interface, or a local process. The routing code may drop packets that are unroutable. If the packet is destined to a local address, the netfilter framework is called again for the local-in hook, before being passed to a user space process. If it is destined to pass to another interface, the netfilter framework is called for the forwarding hook. The packet then passes the final post-routing hook, before it is sent off the interface. The local-out hook is called for packets that are generated locally.

**Limitations of Netfilter**

The netfilter framework provides the minimum mechanisms required to load modules into the kernel, to specify packet matching rules evaluated at hooks, and to invoke the matching target module. However, netfilter has a serious restriction since all loadable modules must be known at compile time to guarantee proper kernel symbol resolution for the linking process. Thus, only kernel modules that have been statically configured can be loaded into the networking subsystem. This is a significant limitation since it prevents the nodeOS from loading arbitrary new components at runtime.
9.2.2 PromethOS Plugin Framework

To overcome this limitation, we have extended the netfilter framework with a plugin framework that manages all dynamically loadable plugins and dispatches incoming packets to plugins according to matching filters. When a plugin initially gets loaded into the kernel, it registers the entry addresses of its virtual functions with the plugin framework. Once a packet arrives and needs to be processed by a plugin, the framework invokes a previously registered plugin-specific callback function. Since plugins register their entry-points, the entry functions do not need to be known at compile time, and for this reason, the plugin framework can load any plugin into the kernel while the node is running. The plugin then inspects or modifies this packet buffer as specified by the plugin code.

Plugin Classes and Instances

For the design of plugins, we follow an object-oriented approach. A plugin class is a dynamically loadable Linux kernel module that specifies the general behavior by defining how it gets initialized, configured, and how packets need to be processed. A plugin instance is a runtime configuration of a plugin class bound to a specific flow. An instance is identified by a node unique instance identifier. In general, it is desirable to have multiple configurations of a plugin, each having its own data segment for internal state. Multiple plugin instances can be bound to one flow, and multiple flows can be bound to a single instance. Through a virtual function table, each plugin class responds to a standardized set of methods to initialize, configure, reconfigure itself, and for handling packets. All code is encapsulated in the plugin itself, thus the plugin framework is not required to know anything about a plugin’s internal details. Once a packet is associated with a plugin, the plugin framework invokes the processing method of the corresponding plugin, passing it the current instance (data segment) and a pointer to the kernel structure representing the packet buffer (struct sk_buff).

Plugin Installation and Configuration

PromethOS plugins are managed at load-time by providing configuration parameters and at runtime through the control interfaces via the procfs. When the PromethOS plugin framework initially gets loaded, it creates the
entry /proc/promethos. Below this entry, the control and reporting ports of individual plugins are registered. PromethOS plugins can be manually loaded by iptables (which we extended with semantics required for the PromethOS plugin framework) or automatically by the EPR protocol. The communication to control plugins and report messages between user space and plugins follows a request-reply approach. A control message is addressed to the appropriate plugin by passing the plugin instance identifier as a parameter and the plugin then responds with a reply message.

Example Use of Encryption Plugin

To give the reader a feel for the simplicity and elegance with which plugins can be put into operation, we illustrate the shell commands necessary to load and configure an encryption plugin. Note that these commands can be executed at any time, even when network traffic is transiting through the system. As mentioned above, we use a PromethOS-enhanced iptables program\(^1\) that interacts with the iptables framework. This extension of iptables implements calls to the insmod program, which serves as the primary tool to install Linux kernel modules.

To encrypt data traffic, we use the Advanced Encryption Standard (AES) [105]. AES is a symmetric block cipher algorithm that uses a block length of 128 bits. In our implementation, only the user’s payload is encrypted, meaning that the IP header and UDP or TCP transport protocol headers remain intact. The reason for not encrypting the transport header is that the port numbers are also required by the filter matching scheme for explicit path routing.

In the following scenario, we assume a UDP traffic flow from MSR2 to MSR5 (see Section 10.1 for our test network setup). Since end system applications do not encrypt the stream, the traffic is visible in clear text by intermediate routers (in our example the packet contains “hello world”).

```
msr5: # tcpdump -x -X -vvv -i eth0 port 5555
msr2.ethz.ch.32777 > msr5.ethz.ch.5555:
 4500 0028 726f 4000 4011 4080 8184 4265 E..(ro@.@.@...Be
```

---

1. iptables can be extended by dynamic libraries to recognize module-specific configuration parameters.
To encrypt the data, the following commands are executed on MSR2:

- Loading and binding plugin with filter:
  
  ```
  msr2:# iptables -t promethos -A OUTPUT -s msr2 -d msr5 -j PROMETHOS --plugin CRYPT --autoinstance --config 'E/mykey'
  ```

  This command adds a filter specification to the PromethOS plugin framework for matching all traffic originating from MSR2 and destined for MSR5, requesting to install the CRYPT plugin in encryption (‘E’) mode at the OUTPUT hook, and creating an instance of this plugin to perform AES encryption based on the given secret key. If the PromethOS plugin framework is not yet loaded, the module dependency resolution of Linux installs it on demand.

- Upon successful completion of the plugin loading and instantiation, the plugin framework reports the plugin instance number:
  
  ```
  Router plugin instance is 1
  ```

The configuration of the current PromethOS table can be retrieved with iptables:

```
msr2:# iptables -t promethos -L
Chain OUTPUT (policy ACCEPT)
target prot opt source destination
PROMETHOS all -- msr2 msr5 PROMETHOS CRYPT#1
```

To install a corresponding decryption plugin on MSR5, the same commands are executed, with the difference that the CRYPT plugin is configured in decryption (‘D’) mode and the plugin instance is attached to the INPUT hook. Once both the encryption and decryption plugins are installed, the data traffic then looks as follows on MSR5:

```
msr5:# tcpdump -x -X -vvv -i eth0 port 5555
msr2.ethz.ch.32777 > msr5.ethz.ch.5555:
4500 0030 9212 4000 4011 20d5 8184 4265 E..0..@.@ Be
```

---

1. The plugin framework is implemented as a loadable kernel module itself.
2. tcpdump displays packets how they look on the wire, thus before they are decrypted at the INPUT hook within the network stack.
Since AES operates with blocks of 16 bytes, the payload size has been increased by a few bytes.

This example demonstrates the seamless integration of the PromethOS plugin framework in Linux, allowing to load arbitrary code at runtime. Plugins can be removed from the kernel by the standard mechanisms provided by iptables and the Linux kernel module framework.

9.3 Control Plane: Routing and Signaling Protocols

The control plane configures the data plane through an interface\(^1\) such that packets arriving at an interface can be handled efficiently.

Various routing protocols are supported on our node architecture. Our extended OSPF daemon [78] is required to provide topology information for the ANCS but protocols such as BGP [116] and RIP [92] can be used if desired. All routing protocols contribute route prefixes to a routing table managed by Zebra [115]. The Zebra daemon collects these route prefixes and computes a common forwarding table that is then installed in the data plane. Whenever routing protocols receive updates from neighboring routers that modify the routing table, Zebra recomputes the prefix set and propagates all changes to the kernel forwarding table. Updates to the kernel table are implemented using netlink\(^2\). Zebra also provides information for routing processes about interface properties (e.g., link type, cost metric) and events occurring on interfaces, such as when a link fails. Such events are then reported to all routing protocol daemons. As a consequence, routing protocols will compute new routes, and propagate these prefix changes back to Zebra.

To allow applications to explicitly reserve and allocate network resources, our implementation supports various signaling protocols. For example,

---

1. IETF has recently made an effort to standardize the interface between the control and data plane and has come up with ForCES [151]. This separation allows elements from different component vendors to be interoperable with each other.
2. The netlink protocol is used to transfer information between the kernel and user space, such as for modifications of the kernel tables.
RSVP [18] and MPLS [118] allow applications to make bandwidth reservations for guaranteed end-to-end flows. In particular, the EPR [77] protocol, which is required in our architecture, installs processing functions on designated nodes and establishes flow filters such that data traffic gets routed on explicit paths.

In the following subsections, we describe the modified OSPF and newly implemented EPR daemons.

### 9.3.1 Extended OSPF Protocol Daemon

Our prototype implementation executes a modified version on the OSPF daemon [78] that has been extended to provide topology and processing information for the ANCS. Using a socket-based API, external applications can retrieve the full or partial link-state database of the OSPF daemon as well as can originate application-specific LSAs. Opaque LSAs are then transparently distributed to other routers within an LSA’s flooding scope and received by other applications again through the OSPF API. Figure 9.2 illustrates the internal structure of the OSPF daemon.

![Figure 9.2: Internal structure of OSPF daemon with API server](image)

- The OSPF core provides the basic protocol functions such as neighbor discovery, initial link-state database exchange, propagation of topology changes to neighbors, building internal link-state database, and computation of route prefixes from the link-state database.
• The opaque module provides functions for exchanging opaque LSAs between routers. Internally, opaque LSAs can be generated by either the MPLS-TE or the OSPF API server module. These modules then invoke the opaque handling code of the OSPF core to flood the information to neighbors within the flooding scope.

• The OSPF API server implements the server side of the OSPF API protocol (as described in Section 6.3.4). Due to multithreaded programming, the API server can handle multiple clients concurrently.

External applications (as the ANCS daemon) link against the OSPF API client library. The OSPF client library offers a convenient API to directly access the link-state database and to transparently distribute customized opaque LSAs. The client library establishes a connection to the OSPF daemon that then allows retrieving LSA updates and originating opaque data describing processing capabilities. This protocol has already been described in Section 6.3.4.

9.3.2 Explicit Path Routing Protocol Daemon

The EPR daemon runs on each active node and implements the EPR protocol as described in Section 8.1. It accepts path setup and tear down requests, and sets up an appropriate data plane configuration in the node’s kernel. Figure 9.3 illustrates the internal structure of the EPR daemon.

The daemon, server, thread modules, I/O library, and various data structures implement the EPR signaling protocol daemon. Modules within the multi-threaded box run concurrently as individual threads whenever a new EPR protocol request arrives. Internally, the EPR daemon keeps a list of established flows (k_flow) that require explicit routing, a list of plugins (k_plugin) that are installed in the kernel and bound to a flow, and a list of active neighbors (k_neighbor) that implement the EPR protocol. Interaction with underlying nodeOS-specific kernel functions are implemented by kernel_ip, kernel_ipt and kernel_ipp, respectively. In particular, the EPR daemon consists of the following components:

• The I/O library (libeprd) implements the functionality required to communicate between peering EPR daemons. This includes functions to assemble messages into protocol data units (PDUs), sending and re-
The daemon module includes EPR’s main function. When the daemon starts up, it parses its command line arguments, initializes the system logging facility, sets various handlers for proper signal handling, and then detaches itself from the controlling terminal and invokes the server module.

- The server module allocates a socket structure which it binds to a port, listens for incoming connections, and accepts new EPR protocol requests. For each request, it creates a new thread such that multiple requests can be processed in parallel.

- A thread gets invoked by the server module on the arrival of a new CREATE_PATH, RELEASE_PATH, or STATUS request. Each request is handled by its own thread.

- The kernel module controls all accesses from the EPR signaling daemon to the underlying kernel resources of the node. This includes func-
tions for adding or removing flow filters, loading or unloading plugins, and binding or unbinding plugin instances with flows. Since the EPR daemon can process multiple requests in parallel, the underlying kernel structures (k_flow, k_plugin, k_neighbor) need to be protected by a kernel semaphore.

9.3.3 Plugin Loader

Plugin code gets loaded into the networking subsystem as a reaction to an explicit request from the EPR signaling protocol. The active node then checks whether the referred plugin class has already been loaded into the kernel. If it is not present, the plugin loader looks for a locally cached copy (stored in the file system of the node). If this again fails, the plugin loader contacts a remote code server, retrieves the object file, and verifies the consistency by checking the module’s digital signature [31]. Then the module is loaded into the kernel and linked against the current kernel image. Subsequently, the node creates a new instance of the plugin, invokes the configuration method, and binds the plugin instance with the filter describing the flow.

A code server is a well-known node in the network that provides executable code fragments that can be loaded into a node’s execution environment. Code servers can be organized hierarchically, meaning that if a plugin cannot be found on a code server handling the local domain, it can be retrieved from a server higher up in the hierarchy.

9.3.4 Local Resource Manager

The local resource manager monitors the node’s local resources, keeps track of how much can be further allocated, and decides whether incoming reservation requests can be granted or need to be denied. This also includes the available processing resources. When processing code gets installed into the node and bound to a flow, the local resource manager deducts the required CPU cycles from the processing element’s pool of remaining cycles.
9.4 Active Network Control Software

As illustrated in Figure 9.4, the ANCS enables applications to create, refresh, and release path-based services which are explicitly routed within the network and can include several processing steps. Within a given network domain, at least one instance of an ANCS daemon needs to be running. Usually, a single ANCS instance is sufficient to perform the service mapping, since our algorithm is able to perform this mapping process quite efficiently (specific performance numbers about the mapping will be presented in Section 11.3). Nevertheless, multiple ANCS daemons can be used to serve a domain in case the overhead of mapping service requirements onto network resources should become excessive. When multiple ANCS instances are running, client applications can send service requests to one of the available ANCS daemons.

In the following, we describe the internal functions of the ANCS in more detail.

![Figure 9.4: Internal structure of ANCS daemon](image)

**9.4.1 Service Setup Manager**

Applications request services from the ANCS using the network programming interface (NPI). The NPI is implemented by the service setup
manager part of the ANCS daemon and a client stub (which is linked against each client application requesting network services). Data between these two parts are exchanged using the NPI protocol. The client stub translates requests from the application into protocol data units (PDUs) which are then transferred to the ANCS daemon. Internally, the service setup manager handles these requests by invoking appropriate routines of the routing and signaling subsystem.

### 9.4.2 Resource Discovery

The ANCS performs resource discovery by accessing the OSPF’s link-state database using the provided OSPF API. Based on the link-state database, the ANCS builds an annotated network graph. For each router-LSA and network-LSA present in the LSDB, we add a node to the graph and then for each link referred within these LSAs, we include also the edges. Finally, we attach attributes to the graph as encoded in opaque LSAs. As a result, the graph structure represents all information stored in the LSDB, but in a format that is better suited for graph traversal required by the SPF algorithm.

### 9.4.3 Routing

The routing subsystem of the ANCS computes the optimal location of the processing steps and a service path visiting these sites. As input, the routing algorithm requires a representation of the network topology as a graph. Based on the application’s processing requirements expressed as an active pipe, the routing system then transforms the network graph into a layered graph as explained in Section 7.2.3. Then, the routing subsystem runs Dijkstra’s shortest-path first (SPF) [46] algorithm on that layered graph, whose solution represents an optimal end-to-end path with required and conditional processing steps to be placed within the network. The final step consists of determining the exact path that is required by the signaling protocol. Such a path must be identified by the address of each node, along with the node’s incoming and outgoing interfaces.

---

1. To reduce the number of interconnecting links between routers, the internal graph structure includes network nodes as an abstraction for LANs that support native broadcast. The path to be signaled must be composed of router addresses only.
9.4.4 Signaling

Once a suitable path has been selected by the routing subsystem, the signaling component establishes the required network state based on the underlying EPR protocol. The EPR client library translates the signaling subsystem’s request into corresponding protocol data units (PDUs) which are then sent to the EPR daemon.

9.5 Summary

To validate the design concepts proposed in this thesis, we have implemented a fully functional node architecture consisting of various system components.

At the data plane, PromethOS is responsible for executing flow-specific packet handlers. Executable code comes in form of plugins, which are loadable kernel modules operating in the kernel address space. From a plugin multiple runtime configurations (instances) can be created, and bound to filters describing a flow. If the code is not present locally, it can be retrieved from a code server that provides a distributed library of plugin modules. PromethOS supports the concept of explicitly routed paths. Since the solution of an optimal service path can be non-simple, the flow forwarding scheme considers also the incoming interface.

At the control plane, mechanisms are needed to discover the capabilities of other active routers and protocols to reserve those resources. To locate active resources, we have extended the OSPF daemon with an API that allows the retrieval of the internal LSDB as well as the distribution of our own opaque LSAs describing processing capabilities. For the reservation of network resources, we have implemented the EPR signaling protocol that supports per-flow explicit path establishment routed through a predefined list of hops, and the installation of plugin modules along such paths. On behalf of applications, the ANCS daemon offers a conveniently to use programming interface and manages all the required resources from the active network.

In the following chapter, we will demonstrate how applications use the ANCS to install their own processing functions, without having to know
about the underlying topology and system-specific details of embedded processing sites.
Chapter 10

Applications with Network Processing

In this chapter, we present how two real-world applications benefit from placing processing functions into the network using the ANCS. We demonstrate how applications can conveniently express their processing requirements using active pipes, how our network control software maps these requirements onto available resources, and how processing modules and forwarding state interior to the network are finally configured.

We begin by introducing the setup of our test network used for all our applications. Then, we describe each of the application scenarios in more detail and illustrate how they can be established using the ANCS.
10.1 Description of Test Network

Figure 10.1 depicts our test network consisting of active nodes as described in the previous Chapter 9. The router hardware ranges from AMD K6/233 (MSR1, MSR3), Pentium II/450 (MSR4, MSR5), to Pentium III/800 (MSR2). All routers (except MSR1) are connected over 155 Mbit/s ATM links with a FORE 202WG switch. MSR1 is connected with MSR2 and MSR3 using crossed 100 Mbit/s Ethernet cables. Thus, for all traffic crucial for our evaluation, we have dedicated ATM or Ethernet links. In addition, we have connected all routers to a common Ethernet for traffic generated by NFS, NIS, and SSH terminal sessions. Using this setup, we make sure that traffic unrelated to our deployed services does not interfere with our evaluation.

![Figure 10.1: Test network used for evaluation](image)

In the figure, every interface has a unique address assigned, which also identifies the outgoing link. Each node is labeled with its router ID\(^1\), which corresponds to the lowest interface address of the node. Each link also has a cost metric\(^2\).

In the following, we describe two applications that we implemented using our active network.

---

1. The OSPF specification [101] does not require the router ID to be one of the node’s interfaces, but this is common practice.
2. In this example, OSPF assigns links a default metric of 10.
10.2 Video Distribution with Congestion Adaptation

In an active network, a video distribution application can benefit from placing application-specific congestion adaptation modules (as discussed in Section 5.3.1) directly into the network, rather than depend on end-to-end adaptation mechanisms (e.g., [93]) only. Customized modules interior to the network implement their own packet queuing and dropping behavior such that traffic streams get dynamically adapted as a reaction to network congestion. In the context of this dissertation, we have implemented a video distribution application that performs adaptation of video streams at locations with high network congestion. Figure 10.2 illustrates such a node performing video scaling.

![Figure 10.2: Media gateway performing network-interior video scaling](image)

Packets entering the node are first classified as either belonging to the video stream or ordinary traffic. Video traffic then gets processed by a scaling module and as a result, the video’s bandwidth is reduced to a certain target rate reported by the packet scheduler (e.g., DRR [125]). The exact details of how the bandwidth reduction algorithm works will be explained in the following sections. The adapted video stream is then forwarded to the packet scheduler. To each flow, the packet scheduler assigns a fair share of the link’s outgoing bandwidth. Internally, the packet scheduler keeps a separate queue for each traffic flow. The outgoing link’s bandwidth divided by the number of competing flows results in the guaranteed rate at which each queue will be served. This rate at which each flow’s queue is forwarded is periodically reported to the video scaling module. As a result, the video adaptation module always knows the rate that is guaranteed by the scheduler.

In the following, we describe the mechanism of how a video stream can be efficiently adapted to a specific target rate. First, we look at existing video
encoding schemes and then propose a new approach based on wavelet encoding.

10.2.1 Video Encoding Schemes

The method used to encode individual pixels of the video frames into network packets is central to video scaling. In this section, we elaborate on various encoding schemes and illustrate, how scaling can be performed efficiently.

In general, scaling of video can be performed in spatial, color, and temporal space. Spatial scaling means decreasing the resolution, color scaling allows for reduction of color (which can go as far as turning a colored video into a grayscale or black and white video), while temporal scaling means modifying the frame rate. Only a few codecs allow efficient scaling. For example, coding in YC_bC_r color space makes adaptation in the color space very simple: a grey scale video can be generated by only decoding the luminance (Y) channel. However, adaptation of compressed video is problematic, since compression reduces redundancy and intensive computation is required to access the pixel information. This is especially true for schemes using motion vectors to compensate for motion in the temporal space. The problem is resolved by compression in different layers whereas each layer possesses well defined properties. Receivers then select specific layers to receive the desired video quality.

The most popular video standards for the Internet are proposed by the Motion Picture Expert Group called MPEG, and the technically similar ITU-T derivative for video telephony H.263 [68]. MPEG consists of a family of codecs for various applications. MPEG-1 has been designed for videodisk playback and does not offer bandwidth adaptation. MPEG-2 offers scalability with respect to Signal-to-Noise Ratio (SNR). SNR scaling refers to the fact that visual quality can be adapted other than by resolution reduction. Small changes in the picture are smoothed out, which is not immediately visible to the human eye, but greatly improves the compression rate. MPEG-4 [102] is a newer standard targeted at video conferencing and has to be seen as a meta standard defining a framework for various encoding schemes. As a baseline it includes H.263 like encoding, which is a standard for low bitrate communication using multiples of 64 Kbit/s channels. Technically, H.263
belongs to the same family of codecs as MPEG. Both use discrete cosine transformation (DCT) and similar motion prediction and compensation.

However, formats like MPEG and H.263 make scaling a computation-intensive task since specific quality features (such as the DCT and motion coefficients) cannot be easily extracted from a video stream. Partial decompression and buffering of frames is required to perform video scaling. The standard stream formats were neither designed for scaling nor loss tolerance (see [49] for a discussion of a loss-tolerant H.263 version), and thus lack important features. Since high-performance scaling is a crucial requirement, we favor a wavelet-based approach called WaveVideo featuring a simple difference-based temporal encoding.

10.2.2 Wavelet-Based Video Encoding

WaveVideo [39], [50] encodes a single video frame by first transforming the color channels from the spatial to the frequency domain and then quantizing and compressing the decorrelated output. A two-dimensional wavelet transform (WT) is applied to the image, which is implemented and approximated using iterated discrete-time filters. This 2D-process splits an image into a low-frequency (LL) and three high-frequency subbands (HL, LH, HH) and is repeated recursively on the LL-subband at each level of the transformation. For luminance (Y) and color difference (C_b, C_r) channels a tree consisting of low- and high-frequency subbands is generated (Figure 10.3).

Once an image has been transformed into frequency space, the correlation contained in typical natural pictures is dissolved and the actual compression step introducing loss by quantization of the high frequency coefficients is performed. Usually, transformed images have many zero-coefficients and only decorrelated features are represented by nonzero values. Statistical analysis of coefficient distributions manifests that not only zero values, but small values in general dominate the transformed image.

The quantized and entropy-coded (i.e., run-length encoded) leaves of the tree shown in Figure 10.3 are finally assembled into datagrams and marked with a tag describing the video segment of the packet.
The applied compression and channel coding schemes make WaveVideo suitable for environments with high packet losses such as heavily congested Internet links and wireless communication [95]. Independently of the use of video gateways in the network, the WaveVideo decoder running on the receiver tries to conceal packet losses in two ways:

- The loss of high-frequency wavelet-coded coefficients results in a smooth degradation of the whole image area. No artifacts like missing blocks or wrongly colored blocks (which can be noticed with MPEG under heavy loss) are visible.

- If packets containing low-frequency parts are lost, a caching algorithm (LL-cache) supplies a previous version of the missing low-frequency coefficient.

More and more products are using wavelet-based codecs implementing proprietary coding formats, especially for scalable streaming over the Inter-
10.2 Video Distribution with Congestion Adaptation

Further, several open standards now include this scheme as well (e.g., JPEG 2000 [71], MPEG-4 [102]).

10.2.3 WaveVideo Bandwidth Scaling

An active network node can apply two different schemes to scale the video stream to a lower transmission rate: It can either reduce the frame rate or the image quality of individual frames. Dropping complete image frames is simple to implement since the node has to examine only the sequence number of a frame. However, frame rate filtering has to take into account that filters can be cascaded and that packets have been eliminated by upstream routers. Thus, to create an equidistant spacing of video frames, the node has to buffer outgoing frames and forward them at a constant rate.

The second approach reduces the image quality by eliminating high-frequency parts from the video stream. This scheme drops packets from high-frequency subbands first, since these coefficients result in only a slight degradation of the image, perceivable by a minor loss of details. Contrary, low-frequency parts are forwarded with the highest priority to ensure the overall consistency of the image.

Eliminating high-frequency parts from the video allows scaling the stream by approximately a factor of 50 (depending on the frame size of the video): for example, a small-sized Quarter Common Intermediate Format (QCIF) video frame has four wavelet transformation levels for the luminance channel (Y) and three transformation levels\(^1\) for the chrominance channels (\(C_bC_r\)). Each transformation level contains three high-frequency parts (HL, LH, HH). Having four transformation levels, there are potentially 12 high-frequency parts to eliminate from the Y-channel and 9 high-frequency parts from the \(C_b\)- and \(C_r\)-channels. All in all, this allows adapting the video stream from full-quality frames consisting of 33 segments to just one low-frequency segment per frame. Figure 10.4 depicts a representation of video frames successively scaled from 33 segments (requiring 33011 bytes) to a single small segment (227 bytes). Using this scheme, an average sample vid-

---

\(1\) The color channels are initially subsampled both horizontally and vertically by a factor of two, resulting that fewer wavelet transformation levels are needed.
eo stream can be scaled from 2.6 Mbit/s down to a minimum of about 50 Kbit/s.

![Successive scaling of video frames](image)

Besides providing a high level of scalability, WaveVideo stream adaptation can be implemented efficiently by labeling each individual network packet with a tag. The tag describes the frame's content such as the corresponding frequency subband, color channel, and depth in the wavelet tree (Figure 10.5).

The tag allows for reduction of the stream's bandwidth by selectively dropping packets belonging to a particular quality set. In our case, we define a quality set such that the transmitted output stream does not exceed an upper bandwidth limit. In other words, we want to prevent the packet scheduler from randomly dropping packets to ensure that datagrams containing low-frequency coefficients (which define the overall image) reach the video sink.
even under congestion. Our scheme continuously monitors the forwarded video stream and periodically adapts the quality profile to fit the bandwidth assigned by the packet scheduler. Thus, if the forwarded bandwidth is higher than the bandwidth that the scheduler can guarantee, we modify the quality profile such that more high-frequency parts are eliminated from the video stream. Similarly, we include more high-frequency coefficients into the profile if the filtered video stream falls below the limit set by the scheduler. With a profile update period of 50 ms, the WaveVideo plugin can quickly respond to both fluctuations in the available link bandwidth and variations in the video stream encoding.

This scheme can be implemented in a very simple and efficient way using a quality profile encoded as a boolean lookup table with 128 (=\(2^7\)) entries to hold all combinations of color channel, recursion depth, and subband (see Figure 10.5, whereas scaling does not use the sequence number). Thus, if a packet is transiting through the node, we use 7 bits from the tag as an index into the quality profile table and lookup the entry: if the value is set, the node forwards the packet, otherwise it drops it. Modifying the quality profile is simple: to add or remove high-frequency coefficients from the forwarded video stream, we simply set or reset the corresponding table entries, beginning with the highest frequency coefficients. In summary, the overhead imposed by scaling the video stream is just a table lookup and can be executed very quickly.

10.2.4 Deployment of Video Adaptation Scenario

As we have presented a scheme for adapting a video stream under congestion, we now describe how a video distribution application can install
such congestion control modules at the most favorable places within the network.

As depicted in Figure 10.6, the scenario assumes that a sender attached to MSR1 transmits a video stream of 2.6 Mbit/s towards the receiver’s access router MSR5. Along the default IP path, the link between MSR3 and MSR5 is restricted to 3 Mbit/s (using ATM’s hardware-based cell pacing). Thus, when no other traffic is active, the receiver at MSR5 will display a disturbance-free video. As cross traffic, MSR2 sends traffic to MSR5 routed via MSR3. Hence, when cross traffic is active, the 3 Mbit/s link will be congested since it needs to be shared in a fair manner by the video and cross traffic. Thus, one cross traffic stream results in a residual video bandwidth of 1.5 Mbit/s, making the link moderately congested. Two concurrent cross traffic streams originating from MSR2 lead to heavy congestion, with only 1 Mbit/s remaining for the video.

We assume that congested links are labeled with an attribute, marking them as congested. This is usually done automatically by the local resource manager monitoring the link load, but here we define the labels using the ANCS configuration manager that enables manual changes to ANCS’s internal network information base (NIB). The value of this attribute corresponds
to the ratio of the traffic entering the node divided by the traffic that can be forwarded on the link:

\[
\text{congestion} = \frac{\text{offered link load}}{\text{link bandwidth}} = \frac{4.6\text{Mbit/s}}{3\text{Mbit/s}} = 153\%
\]

To attach an attribute to the link describing the congestion level (in percent), we invoke the ANCS configuration manager:

```
# acm setattr link 10.0.3.1 congestion 153
```

Now, we install video scaling modules along the path using:

```
# acm create --dport 5555 --from 10.0.5.1 --to 10.0.2.2
   --plugin conditional link wv init 5
   --constraint congestion ge 100
```

This establishes a path from MSR1 to MSR5 for all traffic that matches the flow specification with destination port 5555 and installs video scaling modules on all nodes preceding congested links. Video scaling modules need to be installed whenever the congestion threshold on links equals or exceeds 100%. For now, we assume that the cost for installing the `wv` plugin is 5. The path computed by the ANCS then looks as follows:

```
# acm status
============ Active Services ==============
active pipe from: 10.0.5.1 to: 10.0.2.2
flowspec src: 0.0.0.0/0 dest: 0.0.0.0/0 sport: 0 dport: 5555 proto: 0 if: 0
  conditional processing:
    link plugin: wv init: init cost: 5
    congestion ge 100

Established path:
10.0.5.1 (msr1) in: 0.0.0.0 out: 10.0.5.1 distance: 0
10.0.0.2 (msr3) in: 10.0.7.1 out: 10.0.3.1 distance: 15
  plugin: wv init: init cost: 5
10.0.2.2 (msr5) in: 10.0.3.2 out: 0.0.0.0 distance: 25
```

1. Each node can have a `cpucost` attribute attached, specifying the `unit cost` for processing. The cost for processing a plugin on a node is computed by multiplying the cost of the plugin times the node’s processing unit. If no `cpucost` attribute is present, the cost unit 1.
In the process of computing an optimal service path, the ANCS considers all the resources needed to install the service and balances between processing and link costs. In this scenario where processing requires 5 units, the ANCS establishes the requested active pipe from MSR1 to MSR5, and installs the video adaptation module on MSR3. The total cost of this setup is 25, represented by the distance (sum of link and processing costs) computed from the active pipe’s source node MSR1.

It is important to note that the selected path is not required to go through the congested link but in our scenario this seems to be optimal even when considering the additional cost for processing on the congested link. In fact, if we assumed a processing cost for video scaling of more than 10, then the route MSR1, MSR2, MSR4, MSR5 around the congested link would be better:

```
# acm create --dport 5555 --from 10.0.5.1 --to 10.0.2.2
   --plugin conditional link wv init 11
   --constraint congestion ge 100
```

As is visible, a path routed around the congested area (without any processing) is then advantageous:

Established path:

10.0.5.1 (msr1) in: 0.0.0.0 out: 10.0.6.1 distance: 0
10.0.0.1 (msr2) in: 10.0.8.1 out: 10.0.1.1 distance: 10
10.0.1.2 (msr4) in: 10.0.1.2 out: 10.0.2.1 distance: 20
10.0.2.2 (msr5) in: 10.0.2.2 out: 0.0.0.0 distance: 30

Note that trading between link and processing costs is a distinct feature of the ANCS.

In the following sections, we compare the video quality for plain- and active dropping, both under moderate and heavy congestion, and also demonstrate how video scaling can quickly adapt to bursty cross traffic flows.
10.2.5 Video Quality Measurements

While the previous section has illustrated how a video congestion control service can be deployed in the network quite easily, we provide some video quality measurements that demonstrate how the video distribution application benefits from installing those video congestion control modules. The measurements provided here have been taken on a predecessor [74] of our extensible router platform implemented on NetBSD. Since the video scaling algorithm is identical, the same results can be expected for the PromethOS architecture.

Figure 10.7 shows two representative frames taken from the test sequences Akiyo (top row) and Foreman (bottom row) under three different conditions. Both frames are shown in the losslessly decoded version, the received video without scaling support (i.e., plain-queued packets), and the received video with scaling support (from left to right) in the case of moderate congestion. Akiyo is a low-motion video sequence used here to demonstrate the static image degradation incurred by packet losses on the network. As we can see, the plain-dropped video of Akiyo shows a general fuzziness with smoothed and blurred edges. Looking at the actively scaled version of the video, transmitted at the same bandwidth, a minor degradation in high-frequency details is observable but the general definition is close to the losslessly decoded video. Looking at the Foreman test sequence, the same explanation applies. However, due to the higher degree of motion, the video without scaling support suffers not only from loss of definition, it also shows motion-blur artifacts. The same video, transmitted under the same conditions with scaling support, shows still a little motion artifact (right side of foreman’s ear) but is otherwise almost perfectly decoded.

To compare the quality of the received test sequences, we use the Peak Signal-to-Noise Ratio (PSNR) which is the most commonly used metric of image quality in the video compression literature. It measures how closely an image resembles an original uncorrupted image. In general, the higher the PSNR, the better the image quality. If two images are identical, the PSNR would be infinite.

\footnote{However, note that the subjective image quality can be improved by adding noise and reducing the PSNR in certain situations such as dithering. Thus, PSNR is not the final word in comparing the quality of encoding schemes.}
The Mean Squared Error (MSE) is the mean of the sum of the squares of the differences between the values of pixels in two images:

\[
\text{MSE} = \frac{1}{n} \sum_{i,j} |P_{i,j} - Q_{i,j}|^2
\]

where \(P\) and \(Q\) are two images, \(i\) and \(j\) are the horizontal and vertical locations of a pixel, and \(n\) is the total number of pixels in the image. The Root Mean Squared Error (RMSE) is the square root of the mean squared error:

\[
\text{RMSE} = \sqrt{\text{MSE}}
\]

The RMSE for images is essentially the average change in a pixel caused by the compression and decompression process. For example, a RMSE of 1 means that on average a pixel is changed by one level, such as an average pixel 128 is shifted to 127 or 129.
The PSNR is usually quoted in decibels (dB) on a logarithmic scale:

$$\text{PSNR} = 20 \cdot \log_{10}\left(\frac{b}{\text{RMSE}}\right) = 10 \cdot \log_{10}\left(\frac{b^2}{\text{MSE}}\right)$$

where $b$ is the peak value for a pixel, typically 255 for 8 bit pixels.

Based on this metric, we compared the received test sequences with the original videos used for encoding and calculated the PSNR for the luminance channel (the color channels usually follow the same behavior). The results are shown in Figure 10.8 for the Foreman sequence.

Knowledgeable packet dropping in congested situations clearly demonstrates its benefits: in the moderately congested case, the video quality almost always follows the PSNR of the lossless video\(^1\). Also, visual results are very good. On the other hand, plain-queuing exhibits quality losses of more than 25 dB which are perceivable by either disturbing artifacts or a general fuzziness.

In the heavily congested case with two cross traffic streams, the situation is even worse. Congestion is so high that the Foreman video sequence with a high degree of motion is no longer usable when applying plain queuing on the router. During several seconds, the video is almost unrecognizable. In contrast, using active queuing, the receiver shows almost undisturbed video playback. Subjective visual results show some minor artifacts (little spikes due to motion blur, or small jumps on camera moves), but otherwise an intact video.

### 10.2.6 Fast Reaction to Congestion

One major advantage of network-interior video scaling is the fact that nodes have local knowledge about the current load situation. Because the WaveVideo plugin can directly interact with the packet scheduler, it can react to congestion much more quickly without requiring an end-to-end control

\(^1\) Lossless here means that all frequency subbands have been received and are used for decoding. Note that the process of encoding the original video followed by the decoding introduces some small variations due to the limited precision of representing wavelet coefficients (16 bits). For this reason, the lossless video does not have a PSNR of infinity.
To demonstrate the node’s ability to quickly respond to an overload situation, we inject cross traffic bursts and measure the video quality of the Akiyo sequence perceived at the receiver attached to msr5. When cross traffic is active, the packet scheduler assigns the cross traffic an equal share of the link bandwidth, thus limiting the available bandwidth for the video flow to 1.5 Mbit/s. Figure 10.9 depicts the quality of the received video
streams both for moderately and heavily congested situations, and the burst activity.

![Reaction to Bursts](image)

**Figure 10.9: Reaction to bursty cross traffic**

The quality of the plain-queued video stream suffers PSNR drops of 5–10 dB whenever the cross traffic is active, disturbing the video seriously. Further, the video quality does not recover until the burst is finished, making the video defective for the complete duration of the burst. On the other hand, active video scaling follows closely the video quality of the original video with only minor falls right after the cross traffic is turned on. As soon as the WaveVideo plugin discovers a decline in bandwidth (which is in the worst-case the 50 ms profile update period), it scales the video to the new available bandwidth. Doing so, the video stream quality recovers rapidly to a level very close to the original video stream showing no disturbing artifacts during the bursts.

To further demonstrate that quality is indeed gained by active dropping, we analyze the received frequency subbands at the receiver. Figure 10.10 ([74]) depicts a histogram of the subband distribution of plain-dropping (top) and active dropping (bottom) when routers are exposed to cross traffic bursts. Our test sequence consists of 33 subbands. The lowest-frequency subband (containing the most crucial image information) is shown at the bottom
of each graph and the highest frequency subband is displayed on top. The gray level of each cell indicates how many times a specific subband was received during a period of 8 frames: if a cell is white, the subband was never received at all, and if a cell is completely black, the subband was present in all the last 8 frames. Active dropping now clearly shows its benefits: during burst activity, plain-dropping shows a random distribution of the frequency subbands, forwarding all subbands with a more or less equal probability and thus not taking into account the frequency subband. On the other hand, active dropping ensures that low-frequency subbands (which are crucial for the general image definition) are always forwarded to the receiver.

10.3 Security Gateway Application

In this scenario, which we described in Section 5.3.2, we would like to interconnect two dislocated computer networks over an insecure channel. To ensure privacy, encryption and decryption steps are needed within the trusted domains.
As illustrated in Figure 10.11, nodes MSR1 and MSR5 are assumed to be security gateways that interconnect domain A and B and provide secure communication for various attached end system terminals. The channel must be established in a way that no unencrypted data ever transits any untrusted area of the network. To make the scenario somewhat more interesting, we assume that the links MSR1 to MSR2 and MSR2 to MSR4 have failed and cannot be used.

We require encryption to be placed on either MSR2 or MSR3 (here MSR1 does not offer processing), and decryption on MSR4. We assume that node MSR2 has faster CPU capabilities and therefore set the processing costs on MSR2 to 3 units while on MSR3 to 5 units, meaning that processing on MSR2 is favored if possible.

10.3.1 Encryption of Data Traffic

For the encryption of data traffic, we make use of the Advanced Encryption Standard (AES) [105]. AES is a symmetric block cipher algorithm that
uses a block length of 128 bits and a key length that can be 128, 192, or 256 bits\textsuperscript{1}. In our implementation, only the user’s payload data are encrypted, meaning that the IP header and UDP or TCP transport protocol headers remain intact. The reason for not encrypting the transport header is that the port numbers are also required for explicit path routing\textsuperscript{2}.

Since AES processes one block of data at the time, it is necessary to break the payload data (plaintext) into 128-bit blocks, and padding the last block with zeros if necessary. To reconstruct the original payload length, the plaintext is prepended with a length field. Figure 10.12 illustrates the packet format used in our implementation.

![Figure 10.12: Format of AES encrypted packet](image)

In our implementation, the plaintext is handled in blocks of 128 bits at a time and encrypted using the same key, also known as the Electronic Code-Book (ECB) mode. With ECB, if the same 128-bit block of plaintext appears more than once in the message, it always produces the same ciphertext. If messages are highly structured, it may be therefore possible for a cryptanalyst to exploit these regularities. To overcome the security deficiencies of ECB, the Cipher Block Chaining (CBC) mode could be used. In CBC, the input to the encryption algorithm is the exclusive OR of the current plaintext block and the preceding ciphertext block, however the same key is used for each block. The input of the encryption function for each plaintext block bears no relationship to the plaintext block. Therefore, repeating patterns of 128 bits are not exposed. For decryption, each cipher block is passed through the decryption algorithm. The result is XOR-ed with the preceding ciphertext block to produce the plaintext block.

---

1. Our implementation uses 128 bits.
2. Note that IPSec [10] encrypts the complete IP payload (including the transport header). However, to support explicit per-flow forwarding based on 5-tuples, port numbers are needed. If port numbers are encrypted as well, flow matching is restricted to source and destination addresses.
Since AES is a symmetric encryption algorithm, the encryption and decryption plugins must have the same key, and that key must be protected from access by others. Currently we assume that the two plugins have obtained a common secret key. To accomplish this key exchange, several mechanisms have been proposed such as Diffie-Hellman [45], RSA [117], and Kerberos [107].

When the plugins share a common secret key, the signaling protocol can transmit mykey (which is used as an initialization vector) in clear text. Each plugin then generates a common session key from the combination of the secret key and mykey.

### 10.3.2 Deployment of Security Gateway Scenario

The use of the AES encryption plugin has already been explained in Section 9.2.2, however the plugins had to be configured manually. Here, we use the ANCS to configure the appropriate network state. The following statement establishes the desired secure communication channel:

```plaintext
# acm create --src 10.1.0.0/16 --dest 10.2.0.0/16
   --from 10.0.5.1 --to 10.0.2.2
   --plugin required node CRYPT E/mykey 10
   --constraint addrrange subset 10.0.0/24
   --constraint cpucost lt 10
   --plugin required node CRYPT D/mykey 10
   --constraint addrrange subset 10.0.2/24
   --constraint cpucost lt 10
```

Both processing steps are mandatory and need to be deployed on nodes, but they have distinct installation conditions. Encryption is required within the address range 10.0.0/24, decryption within 10.0.2/24. In addition, the unit for CPU cycle costs must be less than 10, thus excluding msr5. The path that the ANCS establishes for this scenario looks as follows:

```plaintext
# acm status
============= Active Services ==============
active pipe from: 10.0.5.1 to: 10.0.2.2
flowspec src: 10.1.0.0/16 dest: 10.2.0.0/16 sport: 0 dport: 0 proto: 0 if: 0
required processing:
```
node plugin: CRYPT init: E/mykey cost: 10
constraint <addrrange, subset, 10.0.0.0/24>
required processing:
node plugin: CRYPT init: D/mykey cost: 10
constraint <addrrange, subset, 10.0.2.0/24>

Established path:
10.0.5.1 (msr1) in: 0.0.0.0 out: 10.0.5.1 distance: 0
10.0.0.2 (msr3) in: 10.0.7.1 out: 10.0.0.2 distance: 10
10.0.0.1 (msr2) in: 10.0.0.1 out: 10.0.0.1 distance: 20
   plugin: CRYPT init: E/mykey cost: 10
10.0.0.2 (msr3) in: 10.0.0.2 out: 10.0.4.1 distance: 60
10.0.1.2 (msr4) in: 10.0.4.2 out: 10.0.2.1 distance: 70
   plugin: CRYPT init: D/mykey cost: 10
10.0.2.2 (msr5) in: 10.0.2.2 out: 0.0.0.0 distance: 150

============= Active Attributes ===============
node 10.0.0.2 addrrange 10.0.0.0/24
node 10.0.0.1 addrrange 10.0.0.0/24
node 10.0.1.2 addrrange 10.0.2.0/24
node 10.0.2.2 addrrange 10.0.2.0/24
node 10.0.0.2 cpucost 6
node 10.0.0.1 cpucost 3
node 10.0.1.2 cpucost 8
node 10.0.2.2 cpucost 12

Clearly, the selected path is different from the IP default path. In fact, the
path is even non-simple, since MSR3 is being visited twice. The reason for
this route selection is caused by the fact that processing on MSR2 is less ex-
pensive than on MSR3, even when considering the additional link costs of the
detour (twice the link cost between MSR2 and MSR3). The attraction of a
cheaper processing node causes a non-simple path.

10.4 Summary

In this chapter, we have demonstrated two applications that benefit from
placing processing functions at appropriate locations directly in the network.
With the ANCS, the establishment of the processing functions and forward-
ing state is made very easy, since users can express processing requirements
using a high-level active pipe description, without the need to know about the underlying topology and location of processing sites.

The ANCS implements a deployment scheme that differs from the systems found in the literature and is in many ways much more sophisticated. First, it makes the distinction between mandatory and conditional processing steps, which can either be deployed on a node or when traversing a link. Second, it installs plugins only on selected nodes that offer sufficient resources and satisfy all constraints rather than all nodes along a path. Third, the constraint-based selection process leads to paths that are different from the default IP path and can even be non-simple. Fourth, the ANCS is the only system that optimizes route selection with respect to both link and processing costs, thus considering the trade-off that it might be beneficial to take a somewhat longer path in order to reach cheaper processing sites. This has also an impact on the selection of conditional modules that provide functionality that is not absolutely necessary for the operation, but can improve the quality of the connection. For example, if the processing cost of a congestion control module is high, the route selection algorithm then tries to route around congested links since such links would require plugins with additional processing costs.
Chapter 11

Performance Evaluation

In this chapter, we evaluate the performance of our system with the objective of drawing conclusions about the feasibility of our approach. We consider adequate performance as a strong indicator that our approach can be implemented efficiently.

Our goal is to evaluate whether the user-perceived latency for establishing a service is reasonable. For that reason, we have carried out a number of performance measurements based on application scenarios that represent typical use cases. For each of these application scenarios, we measure the time required to establish the services in the network. Since establishing a service includes multiple phases, we divide the service setup into request handling, routing, and signaling tasks. Then, for each of those subtasks, we analyze what internal functions contribute mainly to the execution time.
11.1 Benchmarks

To evaluate our system, we use the application scenarios listed in Table 11.1 as benchmarks. For each of the scenarios, the table describes the number of required and conditional processing steps, since the number of processing steps directly influences the routing time. The table also shows the computed path with the location of the installed plugins (marked with “p”), since the length\(^1\) of the service path has a direct impact on the signaling time.

Table 11.1: Application scenario benchmarks

<table>
<thead>
<tr>
<th>Application Scenario</th>
<th>Scenario Description</th>
<th>Active Pipe</th>
<th>Computed Path with Installed Plugins</th>
</tr>
</thead>
<tbody>
<tr>
<td>No processing</td>
<td>routing along default IP path, no plugins</td>
<td>-</td>
<td>MSR1, MSR3, MSR5</td>
</tr>
<tr>
<td>Congestion control</td>
<td>congestion control before congested links (MSR3 to MSR5)</td>
<td>- 1</td>
<td>MSR1, MSR3(p), MSR5</td>
</tr>
<tr>
<td>All links</td>
<td>plugins on all nodes preceding outgoing links on default path</td>
<td>- 1</td>
<td>MSR1(p), MSR3(p), MSR5</td>
</tr>
<tr>
<td>Transcoding</td>
<td>processing on single designated node MSR2</td>
<td>1</td>
<td>MSR1, MSR2(p), MSR3, MSR5</td>
</tr>
<tr>
<td>Security gateway</td>
<td>encryption and decryption processing</td>
<td>2</td>
<td>MSR1, MSR3(p), MSR5(p)</td>
</tr>
<tr>
<td>DiffServ</td>
<td>processing on ingress and egress routers, and scheduler plugins</td>
<td>2 1</td>
<td>MSR1, MSR2(p,p), MSR3(p), MSR5(p)</td>
</tr>
<tr>
<td>Complex</td>
<td>Four required processing steps with individual location constraints</td>
<td>4</td>
<td>MSR1, MSR2(p), MSR4(p), MSR5(p), MSR3(p), MSR5</td>
</tr>
</tbody>
</table>

The “no processing” scenario includes no processing functions at all, which is useful to establish baseline overhead introduced by our system. The

---

1. The cost for an individual link is retrieved from the OSPF link-state database and is usually defined as a function of the bandwidth and latency.
congestion control and security gateway scenarios have already been described in Chapter 10. The “all links” scenario installs plugins on all nodes with outgoing links along the service path. “Transcoding” installs a single plugin with the intention of making video formats compatible. “DiffServ” installs processing on the ingress and egress routers plus scheduler plugins within the DiffServ domain. The “complex” scenario includes multiple required processing steps, intended of producing an internal layered graph with a large number of nodes and a rather long signaling path.

11.2 Measurement Methodology

Our objective is to evaluate the time required to establish network services to draw conclusions about the feasibility of our approach. For our measurements, we use the network illustrated in Figure 10.1 on page 144. This test network consists of (rather slow) PC-based PromethOS routers ranging from 233 MHz to 800 MHz CPU frequency.

As explained in Section 9.4, at least one instance of the ANCS daemon needs to be running within the test network domain. In our setup, we run a single ANCS instance on msr2 since this is the fastest machine in our network and therefore best suited for computing the service paths.

To generate our measurement samples, we ran each benchmark 25 times on otherwise idle machines. For each of the functions taking part in the service establishment process, we produce a measurement sample representing its elapsed execution time. To do this, the code of the ANCS daemon has been instrumented at various locations to invoke gettimeofday which offers a time resolution of 1 μs. Since the distributions of the elapsed execution times are known to heavy-tailed [61], [88] due to effects such as demand paging, the benchmarks cannot be simply evaluated based on the average of the execution times. We thus base our evaluation on the median which is a

1. For the ANCS, the network domain refers to OSPF’s flooding scope of link-state advertisements.
2. The function gettimeofday calculates the time down to the nearest μs based on the accurate TSC CPU-tick counter, even so the timer interrupt in Linux occurs only every 10 ms.
3. The median (also called 50th percentile) of ordered samples $x_1, x_2, ..., x_n$ with $(x_1 \leq x_2 \leq ... \leq x_n)$ is defined as $x_m$ with $m = \frac{n}{2}$ if $n$ is even, or $m = \left\lfloor \frac{n + 1}{2} \right\rfloor$ if $n$ is odd.
better estimation for the execution time of a specific function and less sensitive to outliers. That is, all our reported time measurements are calculated as the median values from the generated measurement samples, representing the most typical execution times.

11.3 Service Establishment Measurements

To gain a more thorough insight of where the overhead comes from, we break the service establishment process down into:

- **Request handling** includes the time required to accept service path requests from client applications, exchanging service parameters, checking for duplicate requests, and returning the status on whether the request succeeded or failed.

- **Path routing** refers to the time of generating the topology graph from the link-state database, preparing the layered graph (including the evaluation of all installation conditions), running the shortest-path-first (SPF) algorithm on the layered graph, and computing a path that is suitable for signaling.

- **Path signaling** involves establishing forwarding state of flows by installing filters, creating plugin instances on selected nodes, and binding those instances with filters.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No processing</td>
<td>402</td>
<td>112</td>
<td>13 267</td>
<td>13 781</td>
</tr>
<tr>
<td>Congestion control</td>
<td>933</td>
<td>109</td>
<td>145 988</td>
<td>147 030</td>
</tr>
<tr>
<td>All links</td>
<td>640</td>
<td>157</td>
<td>279 567</td>
<td>280 364</td>
</tr>
<tr>
<td>Transcoding</td>
<td>842</td>
<td>106</td>
<td>41 515</td>
<td>42 463</td>
</tr>
<tr>
<td>Security gateway</td>
<td>1 548</td>
<td>141</td>
<td>225 425</td>
<td>227 114</td>
</tr>
<tr>
<td>DiffServ</td>
<td>1 512</td>
<td>184</td>
<td>208 984</td>
<td>210 680</td>
</tr>
<tr>
<td>Complex</td>
<td>2 311</td>
<td>170</td>
<td>327 934</td>
<td>330 415</td>
</tr>
</tbody>
</table>
Table 11.2 shows the measurements for the establishment of our network service scenarios. Overall, the total establishment time is small in absolute terms, even though we designed our implementation for simplicity and modularity rather than ultimate speed. All services are set up in less than 330 ms. Clearly, the signaling phase is the most expensive task since it involves exchanging EPR protocol data between nodes and the configuration of plugin code. The length of the path and the number of plugins to be installed has a direct impact on the signaling time. For example, the “no processing” scenario with a path length of 3 hops but without code installation can be set up in just 13 ms. In contrast, the “complex scenario” has a path of length 6 hops with 4 plugins to be installed and requires 327 ms. On the other hand, the handling of client requests requires typically less than 2 ms. Finally, constraint-based path computation is fairly minimal, for all the scenarios is requires less than 0.2 ms and seems to be negligible compared to the signaling time.

In the following, we examine each of the service establishment phases in more detail.

### 11.3.1 Request Handling Measurements

The first phase of the service establishment is the handling of requests from user applications. This involves accepting connections from clients, exchanging and parsing packets containing application parameters, invoking internal functions in the daemon, and returning status information as a packet to the application. Measurements indicate that the request handling phase is typically shorter than 2 ms. The details depend on the number of parameter bytes transmitted between the client application and the ANCS daemon.

### 11.3.2 Path Routing Measurements

The second phase that contributes to the service establishment time is the computation of the path and location of plugins. This routing process can be separated in the generation of a layered graph based on the network topology and the running of the SPF algorithm on that layered graph. Table 11.3 illustrates particular measurements of the routing phase (again, all time measurements represent medians).
Table 11.3: Measurements for routing

<table>
<thead>
<tr>
<th>Application Scenario</th>
<th>Building Layered Graph</th>
<th>Routing Layered Graph</th>
<th>Total a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time for initial topology graph [μs]</td>
<td>#modifications required steps</td>
<td>time for running SPF [μs]</td>
</tr>
<tr>
<td>No processing</td>
<td>73</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>Congestion control</td>
<td>77</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>All links</td>
<td>87</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Transcoding</td>
<td>65</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Security gateway</td>
<td>80</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>DiffServ</td>
<td>85</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>Complex</td>
<td>77</td>
<td>4</td>
<td>50</td>
</tr>
</tbody>
</table>

a. Total path routing total from Table 11.2. Note that the sum of all medians may not be identical with the median of all sums.
b. Includes “network nodes” representing networks.
c. Includes inter-layer links representing processing steps.

The first step of the routing phase involves creating a topology graph from the link-state advertisements stored in the network information base (NIB). Since the topology graph is identical for all scenarios (with the exception of a few scenario-specific link or node attributes), the overhead is constant in the range of 80 μs. In principle, this topology graph could be built and updated incrementally whenever the content of the NIB changes, that is, when link-state advertisements arrive or get removed. Currently, our implementation builds this graph on demand, but a system ultimately tailored for speed would build it in advance, before service setup requests arrive.

In the second phase, each processing step modifies the graph as described in Section 7.2.3. For each required step, a new layer is added by making a copy of the initial network topology graph, adding those nodes and links to the layered graph, evaluating suitable processing sites, and adding edges between layers. For each conditional step, node and link constraints are evalu-
ated and if they satisfy, suitable links are marked with the appropriate plugin processing and link weights are adjusted to include the cost for the additional processing. Graph transformations for required processing steps depend on the number of layers that are added to the graph. Adding a layer takes roughly $6 - 8 \mu s$. Currently, we internally duplicate a layer by dynamically allocating new node and link objects. As an optimization, a reference counting scheme could be applied, reducing the time for the graph copy operation significantly. Transformations for conditional steps involve evaluating attributes and if they match, marking links with plugin processing and increasing the link weight with the plugin cost. In the congestion control scenario, the evaluation and marking is performed in 7 $\mu$s. The "all links" scenario marks all links in the network graph for processing, which requires 26 $\mu$s.

Once the layered graph has been created, the SPF algorithm can be executed. Since the size of the layered graph has a direct impact on the SPF running time, we list the number of nodes and links present in the internal layered graph. In our scenarios, the running time ranges from 6 $\mu$s (layered graph with 10 nodes and 24 links) to 25 $\mu$s (50 nodes and 124 links). As expected, these measurements indicate that the SPF's running time is polynomial with the size of the graph. In detail, Dijkstra's SPF algorithm partitions vertices in two distinct sets, the set of unsettled vertices and the set of settled vertices. Initially all vertices are unsettled, and the algorithm ends once all vertices are in the settled set. A vertex is considered settled and moved from the unsettled set to the settled set, once its shortest distance from the source has been found. The algorithm requires all unsettled vertices to be sorted according to their distance from the source kept in a priority queue. With a priority queue ordered at all times (as we implemented it), the complexity averages $O(|V| \cdot \log |E|)$.

Once the SPF algorithm has been completed, the path can be determined by projecting all layers onto a single layer. Required processing steps are located on nodes where the SPF path crosses different layers. The path suitable for the signaling phase must be composed of physical nodes only (no "network nodes") and be described using the router's real interfaces. This post-processing phase after the SPF algorithm has been executed requires 18 – 45 $\mu$s.
All in all, the path routing time is in the range of less than 0.5 ms, even when multiple processing steps with various application constraints need to be considered. Although our measurements were performed on a network with a few nodes, we do not expect the running time to increase dramatically for larger networks. We base this assumption on two observations: First, the runtime complexity remains polynomial, and second, we have seen that the number of layers remains small. Even for rather complex scenarios we assume that the number of layers is typically less than ten.

These measurements and the theoretical analysis of the routing algorithm give strong evidence that the routing process imposes only a small execution penalty. Although current performance numbers are promising and the computation is low, there is still room for optimizations. In particular, the internal graph data structure could be improved, organizing node and link objects in a tree (rather than a list) and using hierarchical prefix matching as a more efficient scheme to look up nodes in the graph.

11.3.3 Path Signaling Measurements

The third phase contributing to the service establishment overhead is the path signaling. It differs from the request handling and routing phases because the signaling process requires configuration of distributed network state. The installation of a plugin requires loading of the object code into the address space of the PromethOS kernel, and linking it against the kernel image (symbol table resolution). For our experiments, we assume that the corresponding plugin class has already been loaded into the kernel (since this needs to happen only once). Table 11.4 shows the time required to establish and release flow and plugin state on different nodes.

Looking at the measurements, adding explicit forwarding state to nodes is clearly faster than installing and configuring plugin modules. This is a promising observation since network services need forwarding state to be configured on all nodes along the path, while plugin state only on a specific selection of nodes. Adding a flow filter requires between 0.7 – 2.3 ms, 1To minimize the number of links between routers attached to a network, the OSPF protocol uses network-LSAs to describe the set of routers that have a direct interface on that network. This reduces the number of links from \( n^2 \) to \( 2n \) but adds additional “network nodes” to the topology graph.
11.4 Summary

In this chapter, we have presented experimental performance measurements of our system, tested under different scenarios that deploy network-interior processing functions. Our measurements indicate that the services we have considered can all be configured in less than 330 ms. Considering the tediousness of setting up network services by hand, we think that this is sufficiently fast for the use cases we have envisioned. We have identified that most of the overhead can be accounted to the signaling phase for setting up distributed network state. The overhead introduced by the constraint-based

| Table 11.4: Measurements for signaling on individual nodes |
|----------------|----------------|----------------|----------------|----------------|
|                | MSR1 (K6/233) [μs] | MSR2 (PIII/800) [μs] | MSR3 (K6/233) [μs] | MSR4 (PII/450) [μs] | MSR5 (PII/450) [μs] |
| Install forwarding state | 2 074          | 719           | 2 290          | 1 224          | 1 294          |
| Uninstall forwarding state | 707           | 251           | 859           | 452           | 76781          |
| Create plugin instance     | 133 723        | 26 670        | 127 664        | 77 331        | 446           |
| Free plugin instance       | 669            | 235           | 736            | 446           | 471           |

whereas creating an instance of a plugin requires 26.6 – 133.7 ms. As can be observed, the measurements depend heavily on the CPU speed of the node, with measurements on MSR2 (800 MHz) being roughly 4 times faster than on MSR3 (233 MHz). Removing a flow filter and freeing a plugin instance are efficient operations, requiring between 0.2 – 0.9 ms.

Our measurements indicated that the signaling penalty in our scenarios can be as large as 330 ms. Clearly, our test network represents a local area network where signal propagation is low. We are convinced that the signaling overhead will be reasonable even for wide area networks with much higher latencies. The time required to signal a service path corresponds to roughly twice the round trip time plus the time needed to install plugins on selected nodes (which can happen concurrently). Since in a wide area networks the latency is generally in the range of less that a few 100 ms, we think that this leads to signaling times that are still reasonable for most applications requiring network-based processing.
route calculation takes less than 0.5 ms in all of our scenarios, thus minimally impacts the total service establishment time.
Chapter 12

Conclusions

In recent years, many applications have emerged that require at least some packet-level processing interior to the network. As of today, these services are mostly provided by dedicated routers (implementing the required functionality as part of their system software) and which need to be configured manually by network administrators. In an effort to provide a more flexible infrastructure, nodes in an active network include computational capabilities that can be programmed to provide application-specific packet handling. Applications can significantly benefit from an active network because they are able to install parts of their program logic on routers rather than having to depend on end-to-end mechanisms only. To enable effective programmability of the active network, there is a need for control mechanisms that allow applications deploying customized processing functions at the most appropriate locations within the network.
12.1 Review of Contributions

This dissertation contributes to the design of the control mechanisms needed to manage and provide advanced services within programmable networks. In particular, the contributions of this thesis are as follows:

**Active Pipe Paradigm**

We proposed active pipes as an intuitive high-level abstraction for specifying transmission and processing requirements of user applications. An active pipe is a sequence of processing functions executing on the data stream and which is routed through the network. Contrary to composition schemes found in the literature, processing steps in an active pipe can be either required or conditional and be executed when either transiting a node or traversing a link. Since applications can have stringent requirements on the location of processing functions within the network, each processing step can have any number of installation conditions that need to be considered. The processing steps in an active pipe can be combined arbitrarily to build rather complex network services. We have modeled various application scenarios using active pipes and have demonstrated that active pipes are simple enough to be used effectively, while powerful enough to express typical application scenarios.

**Resource Discovery Mechanism**

To automate the establishment of network services, there needs to be a mechanism for the discovery of the underlying network topology along with localizing active routers and obtaining information about their processing capabilities. We proposed to extend link-state routing protocols such that they are able to distribute capabilities about network-embedded processing resources. The processing capabilities are included as opaque link-state advertisements and disseminated using the routing protocol’s flooding scheme to other active routers. Based on this mechanism, each active router learns about the location and capabilities of other active routers, and can then use this information for selecting processing resources when establishing network services. In particular, we illustrated how we extended the OSPF protocol daemon such that external processes can access the link-state database and how processes can disseminate their own opaque LSAs for announcing
processing capabilities. Based on these mechanisms, our network control software is then able to build a topology graph of the underlying network and distribute information describing the processing sites. Obtaining this network's topology graph is a fundamental requirement for automating the service deployment process.

**Resource Mapping Algorithm**

We presented an algorithm that maps an active pipe describing the processing requirements of an application onto the network topology. The algorithm guarantees that all application constraints, which can be formulated for each processing step individually, are considered in the mapping process. In an active pipe, a processing step can be either required or conditional, and deployed on either a node or along a link. Our algorithm is based on the layered graph technique in whose design we were involved. For both required and conditional processing steps, we have defined a graph transformation that expresses the mapping problem to an ordinary shortest path problem. By combining these graph transformations, our scheme allows a complete active pipe specification with multiple different processing steps to be transformed into a layered graph, in which each step has its individual location constraints. The solution of this graph then represents the optimal location for placing processing code and a path through the network transiting all processing sites. The proposed mapping algorithm is computationally feasible since the problem is expressed as an ordinary shortest-path problem, which can be solved in polynomial time. As we have seen, computed paths can be non-simple, where a given node is visited multiple times. Such solutions can be even desired, since they represent an optimal usage of network resources with respect to both link and processing costs. For example, our system might favor a somewhat longer path to reach more attractive processing sites. This optimization between link and processing costs is a distinct feature of our framework. To our knowledge, no other system automatically trades between link and processing costs. We have also demonstrated how the algorithm can be applied to a hierarchical network.

**Resource Deployment Protocol**

We have designed and implemented the *explicit path routing* (EPR) protocol that deploys processing code on selected nodes and establishes for-
warding state across the network as determined by the resource mapping algorithm. Such a signaling mechanism is needed since the current IP model does not support forwarding of traffic on paths other than on the default forwarding path. Within active networks, traffic needs to be directed on paths that transit the processing sites in a given order, which are (typically) not located on the IP default path. This path establishment process is based on two-phases: In the first phase, the protocol verifies whether enough resources are available on the forward path. If all resources are indeed available, the requested resources are then allocated on the reverse path during the second phase. For sending control messages between protocol daemons, EPR uses TCP to guarantee reliable delivery of messages. However, both path and plugin state is stored as soft-state to take into account that nodes and links can inherently fail. EPR also supports non-simple paths meaning that some of the nodes can be visited repeatedly. To support non-simple paths, the forwarding decision also takes into account the incoming interface from which a packet enters a node. To our knowledge, no other path signaling protocol has specifically been designed to support non-simple service paths.

**Network Architecture**

We have identified the components needed to configure network services automatically and proposed an appropriate node architecture based on Linux 2.4. We have implemented the all required control operations and protocols on top of our modular and extensible PromethOS router architecture (which supports the concept of flow-based routing along non-simple path and execution of application-specific packet handling code). Our implemented service framework provides a set of well defined programming interfaces to request processing resources and associated communication channels from the active network. In an evaluation, we demonstrate that network services can be established quite efficiently.

**Active Applications**

We have demonstrated the viability of our architecture in a realistic environment by implementing two applications that benefit from processing on intermediate nodes. For a video distribution application, we have designed a scaling algorithm that efficiently adapts a wavelet-encoded video stream to the available outgoing link bandwidth on routers. Based on an extensive
evaluation, we demonstrate that the perceived video quality at receivers can be improved by up to 25 db in PSNR compared to traditional router forwarding (without scaling), and that the video scaling scheme adapts very quickly (within 50 ms) to changes in available link bandwidth. In a second application we have implemented a security gateway that performs data encryption in a way completely transparent for end system applications.

12.2 Critical Assessment

In the following, we critically assess our approach with respect to applicability, performance, and scalability.

12.2.1 Applicability

We have demonstrated that using active pipes we can formulate a wide spectrum of application scenarios, including application-specific congestion control for video streaming, a security gateway, traffic engineering, and differentiated services. These application scenarios need multiple required and conditional processing steps placed in the network, with each processing step having individual constraints to be considered. Using active pipes, these placement constraints can be formulated in a way abstracting from the underlying network topology. For example in the case of the security gateway, the constraints for placing the encryption and decryption steps can be formulated such as “on routers within our trusted Intranets but it does not matter where exactly”. In our view, it is crucial that system-specific details of the network are hidden such that the deployment of application-specific code remains a simple task for users. Also, the optimal resource selection should be delegated to the network control, freeing applications from this complicated task.

Currently, network services that are established using our service framework are limited to point-to-point paths. However, we do not consider this as a serious restriction. First, as we have seen, a wide range of scenarios can be expressed using paths, and second, more complex scenarios can be built using multiple active pipes if some of the nodes are pre-determined. Our focus is on providing a service framework with a minimal set of mechanisms that serves a large fraction of applications, while avoiding unnecessary complexity.
Another advantage of our approach is that services can be established in a way completely transparent for end system applications. As in the example of the security gateway, end system applications do not need to be aware that their data traffic gets encrypted when leaving the trusted area of the network. This makes our system especially interesting for network administrators that want to impose certain traffic engineering policies for specific flows.

12.2.2 Performance

We evaluated the performance of our system by measuring the time to establish network services. Our evaluation revealed that all our application scenarios could be configured in less than 330 ms. Overall, we think that this is sufficiently fast for most use needs, since we assume that services are set up less frequent than transport layer flows, and once established, will be used for a long period of time (e.g., while video distribution takes places). Compared to the effort needed to setup services by hand, we consider the overhead as acceptable as long as it is less than a few seconds.

Considering the different tasks involved in establishing a service, we have identified that most of the overhead can be accounted to the signaling phase for setting up distributed network state. The constraint-based route calculation takes less than 0.2 ms in all of our scenarios, and thus minimally impacts the total service establishment time.

12.2.3 Scalability

In the following, we look at each of the individual tasks that are involved when establishing network services with respect to scalability.

Scalability of Resource Discovery Protocol

We have illustrated how the OSPF link-state routing protocol can be extended such that it is able to disseminate information about network-interior processing resources. This approach is feasible within an autonomous system (AS). However, to make our approach scalable for very large networks with thousands of nodes, we would require a hierarchical network representation, with aggregation of network state for portions of the network that are further away. For our approach, this would mean the introduction of addi-
tional hierarchies on the ANCS level (but keeping the current OSPF-based dissemination protocol), with an inter-ANCS protocol for exchanging aggregated topology and processing site information. In the literature, several hierarchy building methods have been proposed, ranging from simple two-level hierarchies (OSPF areas [101]) to an arbitrary number of hierarchies (PNNI [8], HIGCS [60]). Efficient methods for the aggregation of node and link attributes have also been described in PAR [9] and HIGCS [60]. Although our current system is based on a flat topology graph, we are convinced that our system could be extended to a hierarchical network for supporting very large networks.

**Scalability of Routing Algorithm**

The measurements performed and the theoretical analysis of the routing algorithm give strong evidence that the routing can be done quite efficiently, even for large networks. This is based on two observations: First, the runtime complexity of all graph transformations, even when combined arbitrarily, remains polynomial. Second, the number of layers remains small, since we do not expect that even rather complex scenarios have many required processing steps (which produce a new graph layer). For these reasons, we consider our routing algorithm to be feasible for even large networks.

**Scalability of Signaling Protocol**

To support explicitly routed paths, which are needed in the context of active networks to direct traffic flows through processing sites, our signaling protocol installs flow-specific filter entries into the forwarding table on each node. Since the number of flow entries could become very large, this impacts the scalability of the system. However, there are several observations that ameliorate this issue: First, we do not assume that all traffic requires active processing. Second, many application scenarios operate on aggregated traffic flows (e.g., encryption of traffic between two domains), meaning that only a single flow filter is required. Third, we expect that processing is performed mostly within access networks rather than on backbone routers. Fourth, with memory prices decreasing, routers will be able to handle even hundred of thousands of flow table entries. While all these arguments do not make the flow table scalability problem disappear, we believe that there is strong evi-
dence that with routers supporting a large number of flow entries the problem will not be critical.

12.3 Future Work

One future research direction is extending the current path-based active pipe model to a more expressive composition scheme for supporting multicast trees or even arbitrary service graphs. While this would give users more flexibility in expressing application scenarios, it also makes the resource mapping problem more complex. For instance, optimal solutions for mapping multicasting trees are proven to be NP-complete, and thus cannot be applied to larger networks. Therefore, computationally efficient heuristics need to be found that produce feasible (but not necessarily optimal) graph mappings. As distributed applications are becoming more elaborate, it can be expected that the requirements posed to the network will be more specific (e.g., QoS requirements for links and processing nodes), increasing the complexity of service descriptions. To keep the establishment of services simple (as is currently the case with active pipes), there should be mechanisms to store typical service patterns within a database. Such service patterns could also include policies and resource limits that need to be enforced by the network when establishing services, thus giving administrators more control over the usage of the network and regulating how resources can be used by applications.

Another interesting direction would be to port the ANCS system to a very realistic router platform such as the Dynamically Extensible Router [82]. The DER provides one ideal platform for implementing a high performance router with a scalable system architecture supporting dynamic packet processing and flow classification. Built on top of the processing capabilities, the ANCS then coordinates all the router resources such that the underlying network is capable of providing the expected network services on behalf of applications.
12.4 Final Remarks

In our opinion, programmable routers are a key component of the future communication infrastructure. The integration of communication and processing capabilities will allow the construction of a more flexible Internet platform in the years ahead. It is envisioned that applications will become more distributed, with parts of the program logic operating at numerous locations within the network. For the Internet to continue its success in supporting emerging applications, a paradigm shift is required to create service infrastructures capable of coordinating network-embedded services and utilizing network resources efficiently. Our service framework can be seen as one promising strategy for the development of such advanced network services. We are convinced that our proposed network control mechanisms represent a significant step towards the solution of these challenges.
I am deeply indebted to a number of people who supported me in the accomplishment of this dissertation. First and foremost, I was fortunate to find a great advisor in Prof. Dr. Bernhard Plattner, who motivated and guided me during the course of this thesis. I am especially grateful for initiating and supporting the collaboration with Washington University in St. Louis, which proved to be a tremendous profit for myself.

Heartfelt appreciation and thanks go to Prof. Dr. Jonathan Turner for his guidance while staying in St. Louis. I continue to be amazed with his knowledge of routers and their implementation, his passion for research, and brilliance with coming up with simple solutions for challenging problems.

I would also like to thank Dr. James Sterbenz for very thoroughly reviewing this thesis. His in-depth knowledge about all the activities within the active networking community, and his precious comments greatly improved this dissertation.

Also, thanks to Prof. Dr. Urs Hölzle who gave me the unique opportunity to come to University of California, Santa Barbara for two years as a post graduate researcher. He taught me how to do research and – equally important – how to write about it. He succeeded in creating an inspiring atmosphere and he sparked my decision to undertake a Ph.D.

Many thanks go to all my fellow colleagues at ETHZ and WashU. Thanks to Dan Decasper for the great time we spent in St. Louis and the windsurfing lessons in the San Francisco Bay. Thanks to Tilman Wolf for having me stay
at his place and helping with the endless issues of moving to St. Louis forth and back. Thanks to Dave Taylor for organizing the “fools” river-float trip and stimulating a great work environment by supplying the after lunch coffee at WashU. Thanks to Sumi Choi and Jai Ramamirtham for the great discussions about configuring programmable networks and being able to stay at Sumi’s apartment during the summer. Many thanks also to Marcel Waldvogel and his wife Nicola for having me stay at their house. Thanks to Fred Kuhrs for the many motivating discussions about the MSR’s router plugin environment, and to John DeHart for all the great work on the ANN project. Thanks to the rest of the groups both at ETHZ and WashU for their support. Many thanks to Ulrich Fiedler, Placi Flury, Lukas Ruf, and Matthias Bosshardt for their comments on earlier drafts, and Vincent Lenders, my office mate, for the inspiring discussions on all kinds of computer-related topics.

In particular, I would like to thank Masahiko Endo from Suri Giken Co., Ltd. for helping with the design and implementation of the extended OSPF daemon. His wonderful ASCII graphs explaining the internal data structures proved to be crucial. Once, he even took the plane from Tokyo so that we could have face-to-face discussions, which was very inspiring. Thanks also to Daniel Bauer and Robert Haas from IBM’s Zürich Research Laboratory for discussing specific aspects of network services and being willing to test the extended OSPF code in one of their projects. And not to forget Amir Guindehi, who greatly contributed with his exceptional programming skills to the implementation of PromethOS and EPR.

From the bottom of my heart thanks to my parents, Silvia and Emil, and the rest of my family, Martin and Bettina. Without their unconditional help and understanding this thesis would never haven been possible. Also thanks to Susan for all the proof-reading.

And finally, I would like to thank Caroline for her continuous motivation and invaluable moral support throughout the writing of this dissertation. Jag tycker du är så go min älskling!
Biography

Ralph Keller was born in Zürich, Switzerland on March 14, 1971 where he attended primary and high schools, and graduated in October 1990 with a Matura degree in Economics (Type E). After the compulsory military service, he started his studies in Computer Science at ETH Zürich in November 1991. He spent one year in Lausanne, where he worked during an internship for Siemens in the field of telecom management and continued his studies as an exchange student at EPFL. In October 1996, he graduated with a master’s degree (Dipl. Informatik-Ing. ETH).

In November 1996, he joined the Reliable Software Laboratory at University of California, Santa Barbara (UCSB), focusing on software engineering issues to improve programmer productivity for object-oriented systems. He developed Binary Component Adaptation [73] for Java, an approach that enables components to be customized even in binary form, thus improving software reusability while retaining a strict separation between component provider and reuser.

In November 1998, he became a research assistant at the Computer Engineering and Networks Laboratory (TIK) at ETH Zürich. He worked on the Active Network Node (ANN) and Multiservice Router (MSR) projects in close collaboration with Washington University in St. Louis, where he spent all in all more than a year. In 2001, he started his Ph.D. work on programmable services for active networks.

After completion of his Ph.D. he joined Google Inc.
References


192 References


References


References


