ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PBRD
Path Based Routing Daemon

Amir Guindehi *<amir@guindehi.ch>*

Studentenarbeit SA-2002.17
Wintersemester 2002
Tutoren: Ralph Keller, Philipp Blum
Supervisor: Prof. Dr. Lothar Thiele

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Signature Not Verified

Amir Guindehi

Digitally signed by Amir Guindehi
DN: cn=Amir Guindehi, o=Digital Impact GmbH, c=US
Date: 2002.04.08 18:21:17 +01'00'

Abstract

Abstract (Deutsch)

Einführung

Aktive Netzwerke werden es in Zukunft ermöglichen, Netzwerke den eigenen Bedürfnissen nach zu konfigurieren und zu programmieren, um neue Dienste, Dienstgüte oder Diensterweiterungen rasch, unkompliziert und möglichst transparent für Dienst und Nutzer einführen zu können.

Die am TIK entwickelte Component Based Routing Architecture (COBRA) erlaubt es, sogenannte *Plugins*, welche ausführbaren Code enthalten, zur Laufzeit dynamisch in den Betriebssystemkernel eines Netzwerkknotens zu integrieren. Damit wird ermöglicht, den Netzwerkknoten auf flexibelste Art und Weise mit beliebiger Funktionalität zu erweitern.

Zum Beispiel kann ein Nutzer ein Ver- und Entschlüsselungs-Plugin im Netz installieren, um zu erreichen, dass sein Datenverkehr verschlüsselt übertragen wird. Ein Netzwerkbenutzer hat somit die Möglichkeit, Plugins in Routers zu installieren, um einzelne oder aggregierte Datenflüsse speziell nach seinen Bedürfnissen zu verarbeiten.

Diese dynamische, lokale Verarbeitung eines Datenflusses durch eines oder mehrere Plugins kann nun um Remote-Fähigkeiten dahingehend erweitert werden, dass es einem Nutzer frei steht, sowohl den Pfad des Datenflusses durch das Netzwerk, wie auch dessen Verarbeitung auf den einzelnen Knoten des Weges, dynamisch zu bestimmen. Um dies zu erreichen, ist ein Signalisierungsprotokoll von Nöten, das den erforderlichen Informationsaustausch zwischen den einzelnen cooperierenden Knoten des Netzes wahrnimmt.

Linux bietet mit dem Netfilter- und IPRoute2-Framework elegante Möglichkeiten, sowohl den Betriebssystemkernel zur Laufzeit mit beliebiger Netzwerk-Funktionalität zu erweitern, wie auch tiefgreifende Änderungen am Routing Verhalten des Systems vorzunehmen.

Ziele

Ziel dieser Arbeit ist es, ein Signalisierungsprotokoll für Aktive Netzwerke zu implementieren.

Das Signalisierungsprotokoll soll es ermöglichen, in einem heterogenen Netzwerk aus klassischen, wie auch aus COBRA basierten Netzwerkknoten, von einem der Knoten aus, dynamisch, beliebige Datenflusspfade durch das Netzwerk zu konfigurieren, wie auch beliebige Plugins entlang dieses Pfades zu laden, zu konfigurieren und in den Datenflusspfad einzubinden.

Dabei sollen die folgenden Kriterien erfüllt sein:
- **Flexibilität:** Es sollen die klassischen 6 Tupel Datenflussdefinitionen entlang von beliebigen Pfaden durch das heterogene Netzwerk konfigurierbar sein, soweit dies durch die Heterogenität des Netzes nicht verhindert wird.
- **Modularität:** Das Signalisierungsprotokoll soll so modular wie möglich implementiert werden, um zu erreichen, dass es auf unterschiedliche Systemarchitekturen portiert werden kann. Jener Protokollaspekt, der von jeder Client Applikation genutzt wird, soll in einer Client-Bibliothek gekapselt werden, um so eine schnelle Entwicklung allfälliger Applikationen und eine einfache Nutzung des Signalisierungsprotokolls zu fördern.

- **Effizienz:** Der Zusatzaufwand im eigentlichen Datenpfad der Implementation muss so effizient wie möglich gestaltet werden. Der Aufwand im Kontrollpfad ist natürlich ebenfalls zu minimieren, er ist aber nicht von eminenter Bedeutung für die Effizienz der Gesamtimplementation
- **Zuverlässigkeit:** Das Signalisierungsprotokoll muss berücksichtigen, dass Netzwerkknoten und Netzwerke inhärent unzuverlässig sind und jederzeit ausfallen können. Gleichzeitig ist es von eminenter Wichtigkeit, dass das Signalisierungsprotokoll trotz aller widriger Umstände stabil, effizient und zuverlässig funktioniert, allfällige Missstände erkennt und korrigiert, um weder Ressourcen zu blockieren, noch zu verschwenden.
- **Integration:** Die Implementation soll optimal unter Linux integriert werden und alle die vielfältigen Möglichkeiten ausschöpfen, die einer Implementation unter Linux 2.4 mit Netfilter und IPRoute2 offen stehen.

Resultate

In dieser Semsterarbeit wurde ein Signalisierungsprotokoll entworfen, das in Form eines unter Linux 2.4 lauffähigen Pfad basierten Routing Daemons, optimal, mit Hilfe des Netfilter-Frameworks und unter Berücksichtigung der Linux IPRoute2 Architektur, vollumfänglich implementiert wurde. Ein einfacher Kommandozeilenclient ermöglicht die Konfiguration und Wartung der Pfade, wie auch das Laden und Konfigurieren der Plugins entlang dieser Pfade.

Die modulare Implementation des Daemons erlaubt eine einfache Portierung auf die unterschiedlichsten Systemarchitekturen.

Das Signalisierungsprotokoll besitzt die folgenden Eigenschaften und Möglichkeiten:
- **Flexibilität:** Es sind beliebige Pfad-Definitionen durch das heterogene Netzwerk möglich, soweit dies durch die Heterogenität des Netzes nicht verhindert wird.
- **Modularität:** Das Signalisierungsprotokoll ist modular aufgebaut. Sowohl Client wie auch Server verwenden die selbe Kommunikationsbibliothek. Die Linux systemspezifischen Teile sind gekapselt und können einfach auf neue Systeme portiert werden. Die Implementation ist nicht an Linux gebunden, nutzt aber trotzdem die Vorzüge des Linux Kernels mit all seinen Möglichkeiten.
- **Effizienz:** Der eigentliche Aufwand des Signalisierungsprotokolls liegt im Pfad-Aufbau. Beim eigentlichen Datenverkehr wirkt lediglich der Overhead des Netfilter Frameworks als Bremse. Da die Arbeit des Netfilter Frameworks sich darauf beschränkt, Pakete dahingehend zu markieren, dass sie für eine der Routing Tabellen bestimmt sind, die für jeden Nachbarn generiert wurde, ist der Aufwand vertretbar. Er liegt in der Grössenordnung eines normalen, auf Linux Netfilter basierten Firewalls.
- **Zuverlässigkeit:** Das Signalisierungsprotokoll implementiert die Zustandsspeicherung der Pfad- und Plugin-Konfigurationen als Soft-States. Dies, vielfältige Fehlerbehandlung und klug gewählte Timeouts, ermöglicht dem Signalisierungsprotokoll trotz widriger Umstände stabil, effizient und zuverlässig zu funktionieren.
- **Integration:** Das Signalsierungsprotokoll wurde als reiner Userspace Daemon implementiert. Er verwendet die standardisierten Schnittstellen zum Netfilter- und COBRA Framework wie auch zur IPRoute2 Architektur.
- **Entwicklung:** Eine einfache und konfortable Userspace Kommunikationsbibliothek für die Client-Entwicklung wurde geschaffen.
- **Hilfe:** Aussagekräftige Fehlermeldungen und eine ausführliche Debug-Ausgabe erleichtern das Verständnis der Funktion- und Arbeitsweise und die Fehlersuche.

Weiterführende Arbeit

Als nächstes wäre es möglich, die Aspekte der Pfadaggregation innerhalb des heterogenen Netzwerkes zu betrachten. Sowohl Pfad- wie auch Plugin-Konfigurationsaggregation, wenn mehrere Datenflüsse das gleiche, gleichkonfigurierte Plugin verwenden, wäre denkbar, um dem klassischen Skalierungsproblem im Backbone vorzubeugen.

Weiterhin wäre die Implementation von dynamischen Nachbardefinitionen sehr wünschenswert. Durch Einführen eines weiteren Signalisierungspakettyps wäre es durchaus möglich, dynamisch, neue Nachbarn eines Netzwerkknotens zu definieren und so beispielsweise Netzwerktopologieänderungen im Pfad basierten Routing berücksichtigen zu können.

So könnten zum Beispiel die Topologieinformationen, die ein Linkstate Protokoll wie OSPF besitzt, verwendet werden um neue, eventuell kostenintensivere Pfade an das selbe Ziel zu berechnen und so Failover Routen für das Pfad basierte Routing zu finden, zu evaluieren und dann eventuell automatisch, mit Hilfe dieses Signalisierungsprotokolles, zu konfigurieren.

Abstract (English)

Introduction

In the future, Active Networks will allow users' requirements to be reprogrammed such that new network services, network service qualities or service upgrades can be implemented rapidly, efficiently and transparently for service and user.

The Component Based Routing Architecture (COBRA), designed and implemented at ETHZ by the Computer Engineering and Networks Laboratory (TIK), provides a framework which allows network code plug-in to be dynamically loaded and linked to the kernel code of a network node at runtime.

Network nodes can therefore be extended with a new functionality in a flexible way. For example, the user can install encryption and decryption plug-ins in a network for the encryption of data transfers. Using this new framework, the network user can install plug-ins in the routers such that individual or aggregate flows, can be processed according to pertinent needs.

This dynamic, local processing of the data flow through the use of one or more plug-ins can be extended by remote control possibilities, such that not the user can not only dynamically choose the the path of the data flow throughout the network but also has the choice of the processing capabilities on the individual nodes in the network. To accomplish this, a signaling protocol is needed which provides the necessary information exchange between the cooperating nodes of the network.

Linux with its Netfilter and IPRoute2 architecture provides elegant possibilities for extending the operating system kernel during runtime with any number of network functionality and for implementing far reaching changes in the routing behavior of that system.

Aims & Goals

The goal of this work is to design and implement a signaling protocol for active networks.

In a heterogeneous network of classic and COBRA based network nodes, the signaling protocol will enable the users to dynamically configure any number of paths for single or aggregated data flows and to load, bind and configure any number of plug-ins to those data flows, on any node of that path in a fast and flexible way.

To realise the aforementioned the signaling protocol must fulfill the following criteria:

- **Flexibility:** It shall be possible to configure the classic six-tupel data flow definitions along of freely chosen paths throughout the heterogeneous network, if the heterogeneity of the network nodes allow this.
- **Modularity:** Implementation of the signaling protocol should be highly modular to faciliate the porting of the protocol to different system architectures. A client communication library shall encapsulate those protocol aspects every client application using the new protocol will need. This will allow a fast development of applications using the signaling protocol and will assist in its usablility.

- **Performance:** The performance loss in the data path due to the implementation has to be minimized as much as possible. The expenditure in the control path has to be minimized also, however this is not of eminent significance for the overall efficiency of the implementation. The overhead of modularity should not seriously impact performance.
- **Reliability:** The signalling protocol has to take into account that network nodes and networks are inherently unreliable and that they can fail any time. At the same time it is of eminent importance that the signalling protocol is stable, efficient and reliable. Even in adverse circumstances the protocol should recognize any drawbacks and correct them, so that no resources are blocked nor wasted.
- **Integration:** The implementation shall be optimally integrated under Linux and it shall exploit the many possibilities an implementation under Linux 2.4 with Netfilter, COBRA and IPRoute2 has.

Results

In this semester thesis, a signaling protocol was designed and optimally implemented, in the form of a path based routing daemon running under Linux 2.4 with the help of the Netfilter and COBRA framework and the IPRoute2 architecture. A simple but fully functional command line client allows for the configuration, the maintenance as well the loading, configuration and combination of plug-in along those paths.

The modular implementation of the path based routing daemon allows for a fast port of the signaling protocol to other node architectures. The modularity of the existing Netfilter and COBRA framework allowed a permitted merger of both frameworks.

The targets set for the signaling protocol were archived as follow:
- **Flexibility:** Arbitrary path definitions troughout out the heterogeneous network are possible, as far as the heterogeneity of the network nodes permits.
- **Modularity:** The signaling protocol is modularly built. Client and server use the same communication library. The Linux specific parts are encapsulated and can be ported to new node architectures without effort. The implementation is not bound to Linux but uses all the advantages that Linux based implementation can provide.
- **Efficiency:** The main performance loss of the signaling protocol occurs in the setup phase of a path. The slow down of the real data transfer is mainly caused by the overhead of the Netfilter framework. Since the work of the Netfilter framework is restricted to mark packets as be destined for the specific routing table, that is generated for every neighbor, the expenses are justifiable. The expenses are in the order of magnitude of a normal Linux based firewall.
- **Reliability:** The signaling protocol implements the state storage of the path and plug-in configurations soft states. The manifold error handling and well chosen timeouts allow the signaling protocol to perform stably, efficiently and reliablly even in adverse circumstances.
- **Integration:** The signaling protocol was implemented as a pure user space daemon. The signaling protocol uses the standardized interfaces to the Netfilter framework as well as to the IPRoute2 architecture.
- **Development:** A simple and comfortable user space communication library for client development was created. It allows fast and easy programing of new clients.
- **Help:** Meaningful error messages and a very detailed debug output facilitate the understanding of function and mode of operation, as well the search for errors.

Further Work

As a next step, it would possible to investigate the aspects of path aggregation in the heterogeneous network. Path as well as plug-in configuration aggregation for multiple data flows using the same plug-ins are imaginable to prevent the classic scaling problems on the backbone.

Beyond that, implementation of dynamic neighbor definitions would be of further interest. Defining a new signaling packet type and implementing the program logic for dynamically creating and purging new neighbors of a network node would allow the path based routing to take into account network topology changes.

The topology information of a link state protocol as OSPF could be used to calculate new, eventually more expensive paths to the same destination and to find, evaluate and even, eventually configure fail over routes for the path based routing.

Inhaltsverzeichnis

1. Einführung

1. 1. Motivation

Heutige Netzwerk-Infrastrukturen sind sehr kostspielig im Unterhalt. Netzwerkknoten (Routers) werden in rascher Folge installiert. Kaum installiert sollte die Software (bzw. die Betriebsysteme) dieser Knoten auch fliessend an die neuesten Protokolle angepasst werden.

Am Beispiel von IPv6, das 'IP Extentions', das Erweiterungen der Protokoll Headers schon im Protokollentwurf selbst vorsieht, sieht man, wie schnell und wie oft solche Protokollerweiterungen vorkommen können. Diese Erweiterungen können nicht vorhergesehen werden, sind also auch nicht in den aktuellen Softwareversionen und Betriebsystemversionen der installierten Router implementiert und werden somit von den aktuell installierten Knoten nicht unterstützt. Dies führt zwangsläufig zu veralteten Netzwerkknoten, die die neuesten Varianten der Protokolle nicht unterstützten.

Es sollte möglich sein, Netzwerkknoten zu entwerfen, die sich automatisch an die neuesten Protokolle anpassen können. Dies würde den Gesamtsupport von modernen Protokollen im Netzwerk stark verbessern.

Heutige Netzwerkknoten sind sehr monolitisch aufgebaut und aus diesem Grund nicht einfach erweiterbar. Es ist sehr aufwendig neue Protokolle zu implementieren oder alte Protokolle zu erweitern. Meistens muss die ganze Software bzw. das ganze Betriebssystem ausgewechselt werden.

Dies führt dazu, dass Protokollwechsel mehrere Jahre dauern, wie man auch beim Wechsel von IPv4 zu IPv6 gesehen hat!

In heutigen Netzwerken sind solche Änderungen und Upgrades lokal am Netzwerkknoten vorzunehmen. Der Netzwerkknoten ist während des Upgrades nicht verfügbar und kann seine Aufgaben nicht wahrnehmen. Es muss also Vorsorge getroffen werden, um den Ausfall des Knotens zu kompensieren, falls der restliche Netzwerkverbund nicht durch das Upgrade tangiert werden soll. Es muss ein gesicherter, physikalischer Zugang zum Netzwerkknoten vorhanden sein, was einigen administrativen Aufwand erfordern kann, damit ein Techniker den Knoten warten, upgraden und wieder neu konfigurieren kann.

All dies führt dazu, dass Netzwerkknoten seltener den neuen Gegebenheiten angepasst werden. So kommt es, dass viele Knoten mit veralteten Betriebsystemen laufen, nicht den aktuellsten Sicherheitsanforderungen angepasst sind oder einfach hingestellt und vergessen werden.

1. 2. Problembeschreibung

In der Vergangenheit war die Hauptaufgabe eines Routers sehr einfach. Er musste auf Grund einer Zieladressentabelle Pakete auf die richtigen Interfaces weiterleiten.

In den heutigen modernen, schnellen Netzen müssen Netzwerkknoten, die ja oftmals an den neuralgischen Stellen des Netzes den Datenverkehr weiterleiten, neue, komplexere und vor allem aufwendigere Funktionalitäten zur Verfügung stellen.

Moderne Router haben unter anderem die folgenden Aufgaben zu erfüllen:
- Integrated / Differentiated Services
- Erweiterte Routing Funktionalität:
  Level 3 und Level 4 Routing und Switching, QoS Routing,
- Multicast
- Sicherheitsalgorithmen um VPN's - Virtuelle Private Netzwerke zu implementieren
- Erweiterungen zu existierenden Protokollen: RED - Random Early Detection
- Neue Kernprotokolle: IPv6

Aktive Netzwerke (Active Networks) werden es in Zukunft ermöglichen, Netzwerke den eigenen Bedürfnissen nach zu programmieren, um neue Netzwerkdienste rasch und unkompliziert einführen zu können.

Von der ursprünglich am TIK entwickelten Active Network Node (ANN) Architektur [1] [2] [3] wurde eine Portierung auf Linux vorgenommen. Die Component Based Routing Architecture [6], mit Kurznamen COBRA, basiert auf der Linux Netfilter Architektur und erlaubt, sogenannte Plugins, welche ausführbaren Code enthalten, in das Netzwerksubsystem zu installieren und an spezifische Datenflüsse zu binden, um so diese Datenflüsse zu manipulieren. So wird ermöglicht, den Netzwerkknoten auf flexible Art und Weise, mit beliebiger Funktionalität, dynamisch, zur Laufzeit zu erweitern.

Der gegenwärtige Stand von COBRA Plugins [6] bietet die Basis-Funktionalität, um Plugins in den Kernel zu laden, Instanzen von Plugins zu generieren und Plugin-Instanzen an einen Filter zu binden und zu konfigurieren. Dies geschieht alles lokal auf dem System.

Damit jedoch eine Applikation beliebige Funktionalität im Netzwerk installieren kann, ist ein Mechanismus notwendig, welcher die Installation und Konfiguration von Plugin-Instanzen vereinfacht und es ermöglicht, die Installation und Konfiguration von einem weit entfernten System aus zu kontrollieren und zu steuern.

Dazu wird ein Signalisierungsprotokoll benötigt, welches es erlaubt, diese Schritte quasi ferngesteuert vornehmen zu können.

Dieses Signalisierungsprotokoll soll die folgenden Schritte ermöglichen:
- Von Plugins sollen Instanzen generiert werden, welche sich an einen bestimmten Paketfilter binden lassen.
- Das Routing durch das Netzwerk soll so modifiziert werden, dass Datenpakete eines speziellen Datenflusses, entlang einer vordefinierten Route weitergeleitet werden und auf entsprechenden Knoten, durch die an bestimmte Paketfilter gebundenen Plugins, verarbeitet werden. Es soll also ein Datenfluss und Pfad basiertes Routing möglich sein.

1. 3. Ziel

Ziel dieser Arbeit ist es, ein Signalisierungsprotokoll für Aktive Netzwerke innerhalb eines lauffähigen, flexiblen Frameworks zu implementieren, welches es erlaubt, Plugins mittels eines Signalisierungsprotokolls auf verteilten Knoten zu installieren, an Filter zu binden und zu konfigurieren. Diese Plugins können dann den Datenfluss verarbeiten und manipulieren.

Das Signalisierungsprotokoll soll es ermöglichen, in einem heterogenen Netzwerk aus klassischen, wie auch aus COBRA [6] basierten Netzwerkknoten, von einem der Knoten aus, dynamisch, beliebige Datenflusspfade durch das Netzwerk zu konfigurieren, wie auch beliebige Plugins entlang dieses Pfades zu laden, zu konfigurieren und in den Datenflusspfad einzubinden.

Dabei sollen die folgenden Kriterien erfüllt sein:

- **Flexibilität:** Es sollen die klassischen 6 Tupel Datenflussdefinitionen entlang von beliebigen Pfaden durch das heterogene Netzwerk konfigurierbar sein, soweit dies durch die Heterogenität des Netzes nicht verhindert wird.
- **Modularität:** Das Signalisierungsprotokoll soll so modular wie möglich implementiert werden, um zu erreichen, dass es auf unterschiedliche Systemarchitekturen portiert werden kann. Jener Protokollaspekt, der von jeder Client Applikation genutzt wird, soll in einer Client-Bibliothek gekapselt werden, um so eine schnelle Entwicklung allfälliger Applikationen und eine einfache Nutzung des Signalisierungsprotokolls zu fördern.
- **Effizienz:** Der Zusatzaufwand im eigentlichen Datenpfad der Implementation muss so effizient wie möglich gestaltet werden. Der Aufwand im Kontrollpfad ist natürlich ebenfalls zu minimieren, er ist aber nicht von eminenter Bedeutung für die Effizienz der Gesamtimplementation.
- **Zuverlässigkeit:** Das Signalisierungsprotokoll muss berücksichtigen, dass Netzwerkknoten und Netzwerke inhärent unzuverlässig sind und jederzeit ausfallen können. Gleichzeitig ist es von eminenter Wichtigkeit, dass das Signalisierungsprotokoll trotz aller widriger Umstände stabil, effizient und zuverlässig funktioniert, allfällige Missstände erkennt und korrigiert, um weder Ressourcen zu blockieren, noch zu verschwenden.
- **Integration:** Die Implementation soll optimal unter Linux integriert werden und alle die vielfältigen Möglichkeiten ausschöpfen, die einer Implementation unter Linux 2.4 mit Netfilter und IPRoute2 offen stehen.

Es soll des weiteren die Funktionsweise des Frameworks demonstriert werden, indem gezeigt wird, wie eine Applikation die Installation und Konfiguration eines Datenflusspfades, wie auch der Plugins entlang dieses Pfades, veranlassen kann.

1. 4. Gliederung dieses Berichtes

Der vorliegende Bericht ist in 7 weitere Kapitel gegliedert.

Kapitel 2 - “Erweiterbare Router” auf Seite 5 führt in einfacher Art und Weise in das Konzept der erweiterbaren Routerarchitektur [1] [2] [3] ein und stellt sowohl die Linux Netfilter Architektur [4], wie auch die Component Based Routing Architecture (COBRA) [6] vor.

In Kapitel 3 - “Signalisierungsprotokoll für erweiterbare Router” auf Seite 11 betrachten wir die Anforderungen für ein Signalisierungsprotokoll für Aktive Netzwerke [1] [2] [3].Weiter betrachten wir dann schon existierende Signalisierungsprotokolle und machen eine kleine Evaluation ihrer Möglichkeiten, die zeigen wird, weshalb wir schlussendlich eine eigene Implementation eines Signalisierungsprotokolles für Aktive Netzwerke bevorzugen.

Kapitel 4 - “Design des Signalisierungsprotokolles” auf Seite 13 beschreibt dann das genaue Design des gewählten Signalisierungsprotokolles. Es wird beschrieben, wie die Zuverlässigkeitsanforderungen mit Hilfe von Soft-States erfüllt werden, wie das Pfad basierte Routing in Zusammenhang mit dem Signalisierungsprotokoll funktioniert, und wie die Plugins entlang des Pfades mit Hilfe des Signalisierungsprotokolles geladen und konfiguriert werden. Dann werden die Design Ziele genauer spezifiziert und gegen Ende des Kapitels wird das Design sowohl des Daemons, des Clients wie auch der Kommunikationsbibliothek vorgestellt.

Im Kapitel 5 - “Implementation des Signalisierungsprotokolles” auf Seite 21 wird dann die eigentliche Implementation der drei Komponenten, Daemon, Client und Library, im Detail beschrieben.

Danach wird in Kapitel 6 - “Demonstration” auf Seite 37 das Framework in einem Netzwerk, bestehend aus mehreren Netzwerkknoten mit laufendem PBRD Daemon, demonstriert. Es wird gezeigt, wie ein Datenflusspfad definiert und aufgesetzt wird, wie ein Plugin an den Datenfluss gebunden wird, und wie man diese Konfiguration wieder löschen kann.

Schlussendlich diskutiert dann Kapitel 7 - “Schlussfolgerungen und Ausblick” auf Seite 43 die Schlussfolgerungen, die aus dieser Entwicklung zu ziehen sind und zeigt weitere Ausblicke und Möglichkeiten für zukünftige Weiterentwicklungen.

Im Kapitel 8 - “Anhang” auf Seite 45 sind dann weitere Ergänzungen und Referenzen zu finden.

2. Erweiterbare Router

2. 1. Was sind erweiterbare Router?

Bis zum heutigen Tage besitzen Router typischerweise sehr monolitisch aufgebaute Betriebssysteme, die aus diesem Grund natürlich nicht einfach erweiterbar sind.

Mit der immer schnelleren Protokollentwicklung und der zunehmend schnelleren Markteinführung neuer Entwicklungen, wird es immer wichtiger, dass Router Betriebssysteme dynamisch und zur Laufzeit erweitert und erneuert werden können.

Am TIK wurde in mehreren Arbeiten genau dieses Ziel verfolgt. Es wurde eine Active Network Node Architektur [1] [2] [3] entworfen und entwickelt. Das Hauptziel der vorgeschlagenen Architektur war es, ein modulares und erweiterbares System zu entwickeln, das das Konzept der Flows und die Fähigkeit bestimmte Komponenten abhängig von Flows auszuwählen, besass.

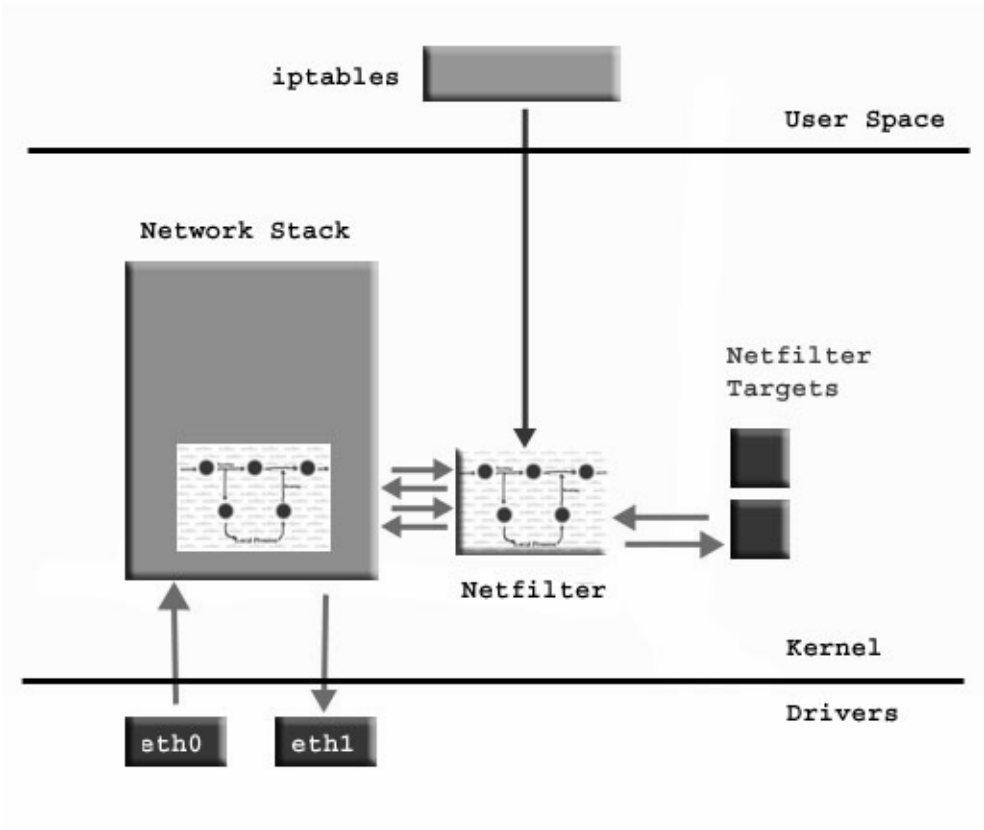
Die Active Network Node Architektur [1] [2] [3] besitzt die folgenden Eigenschaften:

- Dynamisches Laden und Freigeben von Plugins zum Betriebssystemkern zur Laufzeit des Systems. Plugins sind Code-Module, die eine spezifische Routerfuktionalität implementieren (z.B: Ver- und Entschlüsselung von Paketen). Nachdem ein Plugin geladen wurde, ist es nicht mehr von anderem Kernelcode zu unterscheiden.
- Instanzierung von beliebig vielen individuellen Instanzen eines Plugins. Eine Instanz ist eine spezifische Laufzeitkonfiguration eines individuellen Plugins. Es ist öfters sehr wünschenswert, mehrere Instanzen desselben Plugins im Kernel zu haben, wie beispielweise beim Paketscheduling. Dort kann ein und derselbe Paket Scheduler in verschiedenen Konfigurationen (und somit in verschiedenen Instanzen) mit verschiedenen Interfaces arbeiten. State-of-the-art Paket Schedulers sind normalerweise hierarchisch aufgebaut, mit möglicherweise verschiedenen Modulen, die in verschiedenen Hierarchielevels arbeiten. Unter den Knoten, auf dem selben Level, sind die Module unterschiedlich konfiguriert, können aber als Plugininstanzen ein und desselben Plugins koexistieren. Um ein einfaches und einheitliches Interface für die Allokation mehrerer Instanzen ein und desselben Plugins zur Verfügung stellen zu können, muss das Plugin auf ein Set von standardisierten Nachrichten reagieren können. Durch die Standardisierung dieser Nachrichten und durch die Implementation in allen Plugins, wird die Interoperabilität verschiedener Plugins garantiert.
- Effizientes Mapping der individuellen Daten Paket Flows und die Möglichkeit Flows an bestimmte Plugininstanzen zu binden. Sets aus speziellen Flows werden normalerweise durch Filter spezifiziert. Zum Beispiel kann ein Filter genau den TCP Datenverkehr vom Netzwerk 172.16.7.0/24 zum Host 172.16.0.65 spezifizieren. Filter können auch individuelle Ende zu Ende Applikationsflows spezifizieren. Filter werden immer durch 6-Tupels spezifiziert: <Source Adresse, Destination Adresse, Protokoll, Source Port, Destination Port, Interface>. Jedes Element des 6-Tupels kann auch als nicht relevant markiert sein. Für das vorherige Beispiel wäre das 6-Tupel also so anzugeben: <172.16.7.0/24, 172.16.0.65/32, TCP, *, *, *>. Natürlich hat ein Filter für einen Ende zu Ende Applikationsflow alle Felder (ausser eventuell das Interface Feld) ganz spezifiziert.
- Hoher Durchsatz im ganzen Datenpfad. Der hohe Durchsatz ist zum einen Teil durch die vollständige Kernelimplementation, die kostbare Kontext Switches verhindert, gegeben. Zum anderen Teil ist die schnelle Paketklassifikation nötig, um genügend schnell Pakete an Filter und Filter an Plugins zuordnen zu können.

2. 2. Was ist die Netfilter Architektur?

Das Netfilter Framework [4], entwickelt von Rusty Russel und seinem Netfilter Core Team, ist ein Framework zur Paketbehandlung unter Linux 2.4, gehört aber nicht zum normalen Berkeley Socket Interface.

Jedes vom Netfilter Framework [4] unterstützte Protokoll definiert 'Hooks', IPv4 definiert beispielsweise 5 Hooks, welches wohldefinierte Punkte auf dem Weg eines Pakets durch den Protokoll-Stack sind. An jedem dieser Punkte wird das Protokoll das Netfilter Framework [4] mit dem Paket und der Hook-Nummer aufgerufen. Teile des Kernels können sich an den verschiedenen Hooks für jedes Protokoll registrieren und werden so aufgerufen, wenn die den Filtern entsprechenden Pakete durch den Stack fliessen.



Figur A: Netfilter Framework Gliederung

Wenn ein Paket an das Netfilter Framework [4] weitergereicht wird, wird überprüft, ob irgend jemand für dieses Protokoll und/oder für diesen Hook registriert ist; wenn ja, bekommt jeder von ihnen der Reihe nach die Chance, das Paket zu untersuchen und es möglicherweise zu verändern, das Paket zu verwerfen, es durchzulassen oder Netfilter zu beauftragen, das Paket für den Userspace einzureihen. Die von Netfilter eingereihten Pakete werden vom ip_queue-Treiber gesammelt, um dann an den Userspace geschickt zu werden. Diese Pakete werden asynchron behandelt.

Das Netfilter Framework [4] ist sehr modular aufgebaut. Zusätzlich zu diesem einfachen Framework wurden verschiedene Module geschrieben, welche Funktionalitäten ähnlich zu früheren (pre-netfilter) Kernel bieten, im Besonderen ein erweiterbares NAT-System und ein erweiterbares Paketfilter-System (iptables).

Es ist möglich Netfilter Module für beliebigste Zwecke zu schreiben. Es sind ganz normale Linux Kernel Module mit einigen speziellen Funktionen, die vom Netfilter Framework [4] aufgerufen werden, wenn Pakete, die den entsprechenden Datenflussfiltern entsprechen, den Netzwerk-Stack durchfliessen.

2. 3. Was ist die COBRA Architektur?

Die Component Based Routing Architecture (COBRA) [6] basiert auf der Linux Netfilter Architektur und erlaubt, sogenannte Plugins, welche ausführbaren Code enthalten, in das Netzwerksubsystem zu installieren und an spezifische Datenflüsse zu binden, um so diese Datenflüsse zu manipulieren. So wird ermöglicht, den Netzwerkknoten auf flexible Art und Weise mit beliebiger Funktionalität dynamisch, zur Laufzeit, zu erweitern.

Um das Netfilter Framework [4] um Cobra Plugins [6] zu erweitern, mussten einige neue Module geschrieben werden.

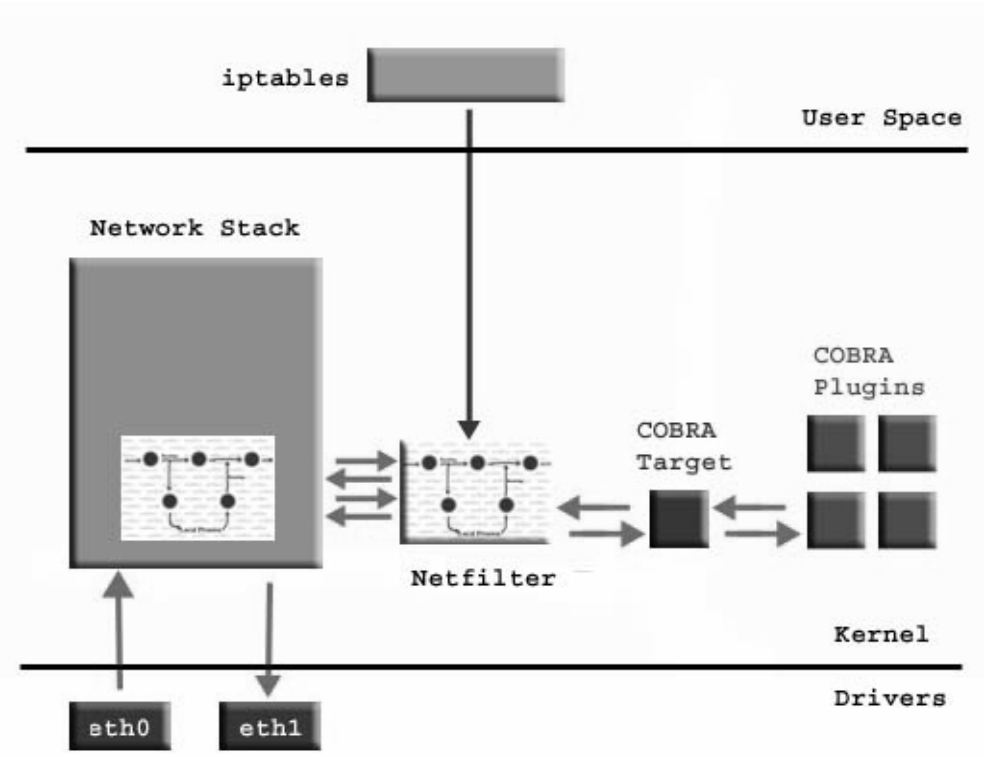


Bild B: Cobra Framework Gliederung

Im Speziellen musste sowohl ein Paket Dispatcher und Plugin Verwalter als *neues Netfilter Target*, wie auch eine *neue Netfilter Tabelle* mit den Filtern für die Flows als Modul implementiert werden. Ein Framework für das Design der Cobra Plugins [6] musste ebenfalls entworfen werden. Zusätzlich musste das Userspacetool ‘iptables’ noch um die Optionen für die neuen Funktionalitäten erweitert werden, was durch Hinzufügen einer Shared Library möglich war.

2. 3. 1. Cobra Netfilter-Tabelle

Um Cobra Flows und normale Filter sauber zu trennen und sie auch in allen 5 IPv4 Hooks registrieren zu können, wurde eine neue Netfilter-Tabelle mit dem Namen ‘cobra’ in Netfilter implementiert.

Die ‘cobra’ Tabelle meldet sich beim Laden des Moduls im Netfilter Framework an und beim Entfernen meldet sie sich auch wieder ab.

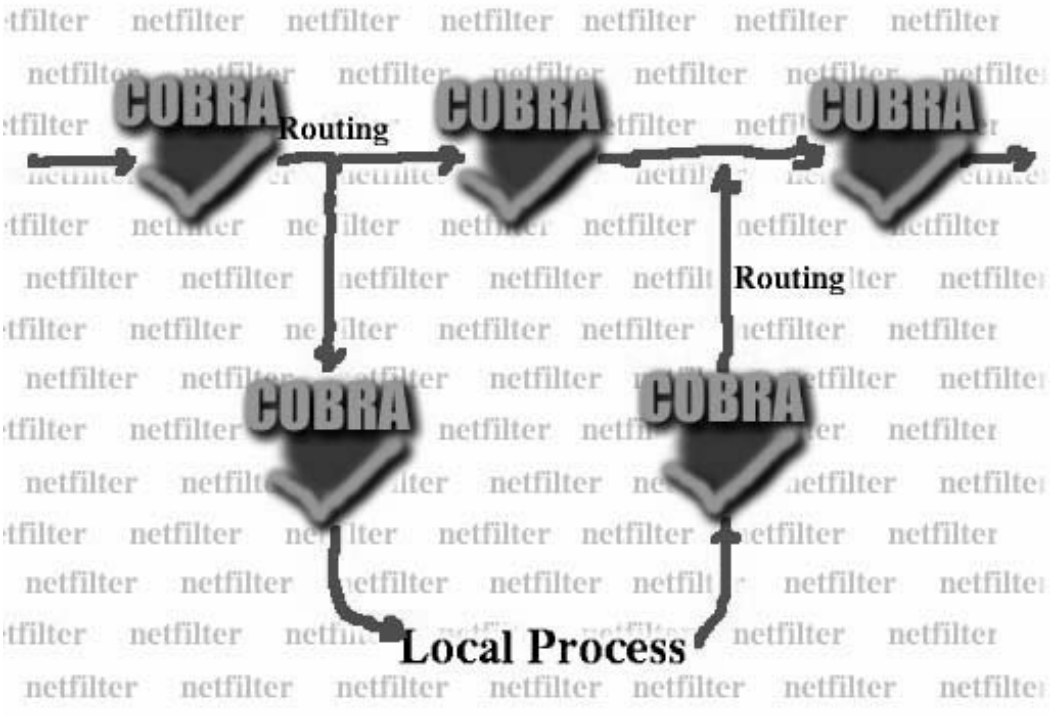


Bild C: Die Hooks der ‘cobra’ Tabelle

Da es Cobra Plugins möglich ist, an jeder Stelle im Netzwerk-Stack aktiv zu werden, registriert sich die ‘cobra’ Tabelle beim Initialisieren in allen 5 IPv4 Netfilter Hooks. Des weiteren löscht die Tabelle diese fünf Hooks auch wieder, wenn das Tabellen Modul wieder aus dem Kernel entfernt wird.

Sollte das Cobra Framework einmal performancemässig den Anforderungen nicht standhalten können, so ist es möglich, die in dieser Anwendung unbenutzten Hooks zu entfernen und so das Framework durch Ausschalten der unnötigen Aufrufe zu entlasten und zu beschleunigen.

Die ‘cobra’ Tabelle ist als separates Netfilter-Tabellen Kernel Modul implementiert, das auch die nötigen Datenstrukturen zur Verwaltung der Tabelle registriert.

2. 3. 2. Cobra Netfilter-Target

Flow Einträge in der ‘cobra’ Tabelle zeigen auf ein spezielles Netfilter-Target, das ‘COBRA’ heisst.

Dieses Target implementiert die Verwaltungstabellen, die nötig sind, um zu vermerken, welche Cobra Plugins geladen sind und wieviele Instanzen von ihnen initialisiert wurden.

Bild D: Das ‘COBRA’ Target

Dazu implementiert das Target zwei exportierte Funktionen. Die erste dieser Funktionen wird von jedem sich initialisierenden Cobra Plugin zur Registrierung desselben aufgerufen, die Zweite wird von jedem beendenden Cobra Plugin zur Deregistrierung aufgerufen. Auf diese Weise ist sichergestellt, dass das Cobra Target jederzeit über alle geladenen und instanzierten Cobra Plugins informiert ist.

Das Target erstellt eine Instanzdatei im /proc Dateisystem des Kernels. Dies ist nötig, da sichergestellt werden muss, dass neue Instanzen eines Plugins auch eine garantiert eindeutige Instanznummer erhalten. Um dies trotz Concurrent Programming garantieren zu können, wurde diese Aufgabe dem nur einmal im ganzen System existierenden Cobra Target übertragen. Ausser in Ausnahmefällen wird das Target die nächste freie Instanznummer ermitteln, und dies dem Userspacetool ‘iptables’ über die Instanzdatei im /proc Dateisystem mitteilen. Der Ausnahmefall ist der Fall, bei dem eine Plugin-Instanz Pakete von zwei verschiedenen Flows verarbeiten soll. Dann soll natürlich beim Erstellen des zweiten Flow Eintrags in der ‘cobra’ Flow Tabelle keine neue Instanz des Plugins erstellt werden, sondern es soll die schon existierende Instanz des Plugins auch für die Pakete dieses neuen Flows aufgerufen werden.

Bei der Registrierung eines Cobra Plugins teilt das Plugin dem Target alle Informationen mit, die nötig sind, damit der Dispatcher Teil des Targets allfällige ankommende Pakete für eine Instanz eines Plugins an das Plugin übergeben kann.

Ebenfalls mitgeteilt wird, welche Funktionen des Plugins zur Initialkonfiguration und zur Laufzeitkonfiguration vom Target aufgerufen werden sollen. Das Target initialisiert, nach der Registrierung eines neuen Moduls, dasselbe mit den, dem Userspacetool angegebenen Daten. Das Cobra Target erstellt auch eine Status-Datei im /proc Dateisystem des Kernels. Über diese Status-Datei ist es möglich, ein Plugin zur Laufzeit neu zu konfigurieren, beziehungsweise dem Plugin beliebigste Daten zukommen zu lassen. Durch Schreiben in die Status-Datei im /proc Dateisystem kann, durch Angabe der Instanznummer und der zu übergebenden Daten, ein Plugin jederzeit neu konfiguriert werden.

Des Weiteren ist es möglich, durch Lesen dieser Status Datei im /proc Dateisystem beim Cobra Target nachzufragen, welche Plugins in wie vielen Instanzen alloziert sind.

Das neue Cobra-Target ist ebenfalls als separates Netfilter-Target Kernel Modul implementiert.

2. 3. 3. Shared Library für das Userspace Tool ‘iptables’

Das Userspacetool ‘iptables’ ist erweitert worden. Die neuen Flags für das neue Cobra Target, wie auch die Hilfe-Texte wurden implementiert.

Des Weiteren ist es Aufgabe des Userspacetools das entsprechende Cobra Plugin zu Proben und gegebenfalls zu Laden.

Ebenfalls Aufgabe des Userspacetools ist es, bei entsprechenden Kommandozeilen-Parametern die Instanzdatei des Cobra Targets im /proc Dateisystem zu lesen und eine neue, im ganzen System einheitliche Instanznummer für die Instanzierung der neuen Plugininstanz zu verwenden.

Diese Erweiterungen des Userspacetools erfolgten durch Hinzufügen einer Shared Library zum Userspacetool ‘iptables’.

3. Signalisierungsprotokoll für erweiterbare Router

3. 1. Wozu ein Signalisierungsprotokoll?

Der gegenwärtige Stand von Component Based Router Architecture Plugins (COBRA) [6] bietet die Basis-Funktionalität, um Plugins in den Kernel zu laden, Instanzen von Plugins zu generieren und Plugin-Instanzen an einen Filter zu binden und zu konfigurieren. Dies geschieht alles lokal auf dem System.

Solange diese Konfigurationen statischer Natur sind, ist dies vertretbar. Sind diese Konfigurationen allerdings fluktuativer Natur, so wäre es sehr wünschenswert, wenn diese Konfigurationsarbeit von einem weiter entfernten System aus möglich wäre.

Damit jedoch eine Applikation beliebige Funktionalität im Netzwerk installieren kann, ist ein Mechanismus notwendig, welcher die Installation und Konfiguration von Plugin-Instanzen vereinfacht und es ermöglicht, die Installation und Konfiguration von einem weit entfernten System aus zu kontrollieren und zu steuern.

Des weiteren verlangt unsere Aufgabenstellung ein Signalisierungsprotokoll, das es ermöglicht, in einem heterogenen Netzwerk aus klassischen, wie auch aus COBRA basierten Netzwerkknoten [6], von einem der Knoten aus, dynamisch, beliebige Datenflusspfade durch das Netzwerk zu konfigurieren, wie auch beliebige Plugins entlang dieses Pfades zu laden, zu konfigurieren und in den Datenflusspfad einzubinden.

All diese Funktionen verlangen ein Signalisierungsprotokoll, dessen Aufgabe es ist, zwischen den einzelnen Netzwerkknoten zu vermitteln, den einzelnen Knoten die Informationen zukommen zu lassen, die sie benötigen, um die gewünschten Funktionen ausführen zu können und die Zuverlässigkeitsanforderungen der Aufgabenstellung einzubringen und zu realisieren.

3. 2. Schon existierende Signalisierungsprotokolle

Es gibt einige schon existierende Signalisierungsprotokolle, die für Aktive Netzwerke genutzt werden können. Im speziellen sind das RSVP - Ressource Reservation Protokoll, BEAGLE ein auf RSVP aufbauendes Signalisierungsprotokoll für Aktive Netzwerke, und MPLS - Multiprotokoll Label Switching.

Jedes dieser Signalisierungsprotokolle hat Vorzüge und Nachteile. Im folgenden Kapitel 3. 3. - “Evaluation der Signalisierungsprotokolle” auf Seite 12 werden diese besprochen.

3. 3. Evaluation der Signalisierungsprotokolle

3. 3. 1. Vergleich

Ich möchte nicht gross auf die einzelnen Signalisierungsprotokolle eingehen, wird doch aus der unten stehenden Tabelle schnell ersichtlich, dass für unsere Zwecke nur eine eigene Implementation eines Signalisierungsprotokolles in Frage kommt.

Des weiteren geben die Texte [7] [11] [12] [13] viel besser Auskunft über die Möglichkeiten der einzelnen Protokolle als es mir hier zusammengefasst möglich wäre.

Table 1: Vergleich der Signalisierungsprotokolle

	Dyn. Rerouting	Lücken im Pfad	AS Routing	Plugins	Soft State	Status	Leight Weight	Linux
RSVP	Ja	Ja	Ja	-	-	-	-	-
Beagle	Ja	Ja	Ja	Ja	Ja	-	-	Beta
MPLS	-	-	Ja	-	-	-	-	Alpha
PBR	-	-	-	Ja	Ja	Ja	Ja	Ja

Die rechten 5 Spalten der Tabelle sind die für uns am wichtigsten Punkte!

Da wir ein Signalisierungsprotokoll für aktive Netzwerke brauchen, kommt von den möglichen schon existierenden Protokollen lediglich Beagle in Frage. Beagle basiert auf RSVP und ist infolge dessen eines der komplexesten Protokolle zur Auswahl.

3. 3. 2. Schlussfolgerungen

Wie man dem obigen Vergleich entnehmen kann, fehlt den meisten schon existierenden Signalisierungsprotokollen ein für uns sehr entscheidendes Feature. Keines der schon existierenden Protokolle unterstützt Plugins ausser Beagle.

Da wir für aktive Netzwerke aber darauf angewiesen sind neuen Code, in Form von Plugins, auf den einzelnen Netzwerkknoten installieren zu können, sind Protokolle, die keine Plugins unterstützen, für unsere Zwecke nicht brauchbar. Leider wäre auch der Aufwand eines dieser Protokolle dahingehend zu erweitern sehr gross und für unsere Zwecke nicht vertretbar.

Dies schränkt die Auswahl auf Beagle ein. Beagle ist ein auf RSVP basierendes Signalisierungsprotokoll. Da wir uns ein Light-Weight Protokoll wünschen, das möglichst auch auf Linux Routern verfügbar ist, kommt leider auch Beagle nicht in Betracht.

In dieser Arbeit wird deshalb ein neues, einfach gehaltenes, für unsere Zwecke ausreichendes Signalisierungsprotokoll entworfen und implementiert.

4. Design des Signalisierungsprotokolles

4. 1. Anforderungen an ein neues Signalisierungsprotokoll

Das neue Signalisierungsprotokoll für Aktive Netzwerke soll eine Light-Weight Variante der schon existierenden Protokolle sein. Ziel dieser Arbeit ist es, ein Signalisierungsprotokoll innerhalb eines lauffähigen, flexiblen Frameworks zu implementieren, welches es erlaubt, Plugins mittels des Signalisierungsprotokolls auf verteilten Knoten zu installieren, an Filter zu binden und zu konfigurieren. Diese Plugins können dann den Datenfluss verarbeiten und manipulieren.

Das Signalisierungsprotokoll soll es ermöglichen, in einem heterogenen Netzwerk aus klassischen, wie auch aus COBRA basierten Netzwerkknoten, von einem der Knoten aus, dynamisch, beliebige Datenflusspfade durch das Netzwerk zu konfigurieren, wie auch beliebige Plugins entlang dieses Pfades zu laden, zu konfigurieren und in den Datenflusspfad einzubinden.

Obwohl es speziell für COBRA Plugins, Netfilter und Linux geschrieben wird, soll das neue Signalisierungsprotokoll so modular wie möglich entworfen werden, so dass alle systemspezifischen Teile in separaten Modulen liegen, die schnell und einfach auf andere Architekturen, Frameworks und Systeme portiert werden können. Sowohl Daemon wie auch Client-Library sollen möglichst viele dieser Module gemeinsam nutzen, um Coderedundanz zu vermeiden.

Die gewünschte Funktionalität lässt sich auf 3 Kernprobleme zurückführen, die in den folgenden drei Unterkapiteln einzeln, konzeptuell, besprochen werden:
- “Setup und Teardown” auf Seite 15
- “Zuverlässigkeit / Soft-States” auf Seite 17
- “Datenfluss-Routing entlang eines Pfades” auf Seite 18
- “Dynamisches Plugin/Datenfluss Binden entlang eines Pfades” auf Seite 19

4. 2. Design Ziele

4. 2. 1. Flexibilität

Das Signalisierungsprotokoll soll die klassischen 6 Tupel Datenflussdefinitionen, bestehend aus Source Adresse, Destination Adresse, Source Port, Destination Port, Protokoll und Incoming Interface, unterstützen. Das Netfilter Framework würde auch noch viele andere Möglichkeiten bieten, um Datenflüsse zu spezifizieren, aber da diese Möglichkeiten auf anderen Systemen nicht zur Verfügung stehen, auf die das Signalisierungsprotokoll noch portiert werden soll, werden wir uns auf den kleinsten gemeinsamen Nenner beschränken müssen und somit nur das klassische 6 Tupel unterstützen. Unterstützung für weitere Paketfiltertypen könnten in einer Fortsetzungsarbeit zu dieser Arbeit untersucht werden.

Es soll möglich sein, diese 6 Tupel Datenflussdefinitionen zur Spezifikation von beliebigen Datenflusspfaden durch das heterogene Netzwerk zu benutzen, soweit diese Pfade nicht durch die heterogenität des Netzes verhindert werden, weil beispielsweise einer der Knoten dazwischen das Signalisierungsprotokoll nicht unterstützt.

4. 2. 2. Modularität

Das Signalisierungsprotokoll soll so modular wie möglich implementiert werden, um zu erreichen, dass es auf unterschiedliche Systemarchitekturen portiert werden kann.

Sowohl Daemon wie auch Client-Library sollen möglichst viele dieser Module gemeinsam nutzen um Coderedundanz zu vermeiden.

Jener Protokollaspekt, der von jeder Client-Applikation genutzt wird, soll in einer Client-Bibliothek gekapselt werden, um so eine schnelle Entwicklung allfälliger Applikationen und eine einfache Nutzung des Signalisierungsprotokolls zu fördern.

Im Kapitel “Aufbau des Signalisierungsprotokolls” auf Seite 20 wird der genaue modulare Aufbau des Signalisierungsprotokolls und seiner Komponenten dargestellt und genauer erklärt.

4. 2. 3. Effizienz

Das ganze Signalisierungsprotokoll muss natürlich so effizient wie möglich gestaltet sein. Dabei gibt es zwei Orte die einen gewissen Aufwand erfordern, die Arbeit im eigentlichen Datenpfad der Implementation, wie auch Arbeit im Kontrollpfad des Signalisierungsprotokolls.

Der Zusatzaufwand im eigentlichen Datenpfad der Implementation muss so effizient wie möglich gestaltet werden, denn es ist dieser Aufwand, der wirklich zum Tragen kommt, schliesslich ist dieser Aufwand für jedes, den Pfad durchfliessendes Paket zu rechnen.

Der Aufwand im Kontrollpfad ist natürlich ebenfalls zu minimieren, er ist aber nicht von eminenter Bedeutung für die Effizienz der Gesamtimplementation, denn dieser Aufwand kommt nur einmal pro aufzusetzenden Pfad zum Tragen.

4. 2. 4. Zuverlässigkeit

Das Signalisierungsprotokoll muss berücksichtigen, dass Netzwerkknoten und Netzwerke inhärent unzuverlässig sind und jederzeit ausfallen können. Netzwerkknoten können abstürzen, ausser Betrieb genommen werden, Fehlverhalten an den Tag legen oder einfach aussteigen.

Gleichzeitig ist es von eminenter Wichtigkeit, dass ein Signalisierungsprotokoll für Aktive Netzwerke trotz aller widriger Umstände stabil, effizient und zuverlässig funktioniert, allfällige Missstände erkennt und korrigiert, um weder Ressourcen zu blockieren, noch zu verschwenden.

Dazu ist es notwendig das Signalisierungsprotokoll konzeptuell darauf vorzubereiten dass solche Ausfälle geschehen, wie wir es im Kapitel 4. 3. 2. besprechen.

4. 2. 5. Integration

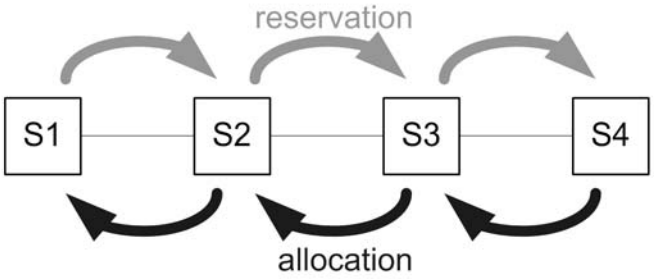
Die Implementation soll optimal unter Linux integriert werden und alle die vielfältigen Möglichkeiten ausschöpfen, die einer Implementation unter Linux 2.4 mit Netfilter und IPRoute2 offen stehen.

4. 3. Entwurf des Signalisierungsprotokolls

4. 3. 1. Setup und Teardown

Das neue Signalisierungsprotokoll ist ein klassisches Reservationsprotokoll entlang bestimmten Pfaden durch das Netzwerk. Die einzelnen Netzwerkknoten verwalten Ressourcen, wie Routing-Tabellen, Datenfluss-Tabellen etc., die mit Hilfe des Signalisierungsprotokolles reserviert und wieder freigegeben werden müssen, um eine faire Verteilung und Benutzung der vorhandenen Netzwerkressourcen zu ermöglichen.

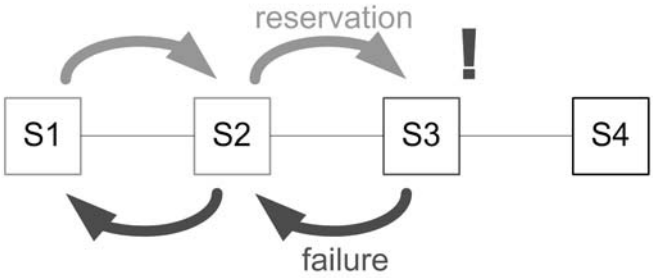
Beispielsweise kann ein Netzwerkknoten einen vollständig spezifizierten Datenfluss, dargestellt durch das klassische 6 Tupel, nur genau einmal in eine bestimmte Richtung routen. Das heisst also, dass von mehreren Benutzern, die kollidierende Anforderungen an das Netzwerk stellen, nur genau einem erlaubt werden kann, das Netzwerk entsprechend zu nutzen.



Figur E: Ablauf einer Reservation

Um nun dem neuen Signalisierungsprotokoll zu erlauben, diese Ressourcenanforderung zu prüfen, zu verifizieren und dann zuzulassen oder abzulehnen, muss die Reservation in zwei Phasen aufgeteilt werden, wie aus Figur E ersichtlich ist.

In einer ersten Phase prüft das Signalisierungsprotokoll ob die benötigten Netzwerkressourcen auf den einzelnen Netzwerkknoten des gewünschten Pfades vorhanden sind. Dies geschieht derart, dass das Signalisierungsprotokoll, vom Ersten bis zum Letzten, alle Knoten entlang des Pfades über die Ressourcenanforderungen informiert und jeden prüfen lässt, ob den Anforderungen Genüge getan werden kann. Ist dies der Fall, wird der nächste Knoten des Pfades bis hin zum Ende weitergeprüft.



Figur F: Abbruch einer Reservation

Sollte einer der Knoten melden, dass er die Ressourcenanforderungen nicht erfüllen kann, so wird abgebrochen, denn um den Benutzer zu befriedigen, müssen alle Netzwerkknoten entlang des Pfades die Ressourceanforderungen erfüllen. Dies wird in Figur F schematisch dargestellt.

Ist die erste Phase abgeschlossen, so ist garantiert, dass alle Netzwerkknoten entlang des Pfades die nötigen Ressourcen reserviert haben. In der zweiten Phase wird dann die eigentliche Allokation der entsprechenden Ressourcen vorgenommen. Dies geschieht rückwärts, also der letzte Netzwerkknoten alloziert zuerst seine Ressourcen, bevor der Zweitletzte dies tut, und so weiter. Dies garantiert, dass erst wenn der vorderste und somit letzte Knoten seine reservierten Ressourcen alloziert und konfiguriert hat, Daten, durch den so aufgesetzten Pfad, zu fliessen beginnen.

Natürlich müssen die Netzwerkknoten die Ressourcen, die sie in Phase 1 reservieren und somit versprechen, auch garantiert in Phase 2 allozieren können.

Um Fairness zu garantieren und um sicher zu sein, dass nicht wegen des Aufbaus eines langen Pfades für den einen Benutzer ein zweiter Benutzer ausgeschlossen wird, muss der, das Signalisierungsprotokoll implementierende Daemon multithreaded programmiert werden und allfällige gemeinsame Daten müssen bei Mehrfachzugriffen vor gleichzeitigen Zugriffen geschützt werden.

Natürlich muss das Signalisierungsprotokoll auch die Möglichkeit bieten, einmal aufgesetzte Pfade durch das Netzwerk wieder freigeben zu können. Dies wird genau gleich wie das Aufsetzten des Pfades realisiert. Ebenfalls aus zwei Phasen bestehend wird beim Tier-Down eines Pfades, in der ersten Phase geprüft, ob der Datenfluss auch wirklich auf allen Netzwerkknoten installiert ist und freigegeben werden kann. Dann werden in der zweiten Phase, ausgehend vom letzten Knoten - irgendwo muss man ja anfangen - auf jedem Knoten sowohl der Pfad, wie auch die ihm zugehörigen Ressourcen, wieder freigegeben.

4. 3. 2. Zuverlässigkeit / Soft-States

Das Signalisierungsprotokoll muss berücksichtigen, dass Netzwerkknoten und Netzwerke inhärent unzuverlässig sind und jederzeit ausfallen können. Gleichzeitig ist es von eminenter Wichtigkeit, dass das Signalisierungsprotokoll trotz aller widriger Umstände stabil, effizient und zuverlässig funktioniert, allfällige Missstände erkennt und korrigiert, um weder Ressourcen zu blockieren noch zu verschwenden.

Um diesen Ansprüchen gerecht zu werden, wird die Zustandsspeicherung im Signalisierungsprotokoll als Soft-States realisiert. Sowohl die Pfadreservationen der einzelnen Datenflüsse, wie auch die Plugin-Konfiguration und das Binding der Plugins an die Datenflüsse, müssen als Soft-States implementiert werden, um zu gewährleisten, dass ein einzelner, sich falsch verhaltender Netzwerkknoten, beispielsweise bei einem Crash desselben, nicht die Ressourcen der restlichen Knoten für Pfade verschwendet, die gar nicht mehr genutzt werden können, weil der Benutzer, der sie aufgesetzt hat und kontrolliert, bzw. dessen Host oder Netzwerkknoten, verschwunden ist.

Soft-States bedingen allerdings die Kooperation der beteiligten Clients. Ein Zustand, der mit Soft-States gespeichert wurde, muss von Zeit zu Zeit aufgefrischt werden, damit er nicht verfällt.

Ein Client, der einen Pfad über eine längere Zeit verwenden möchte, muss also hin und wieder den Netzwerkknoten entlang des Pfades mitteilen, dass er auch in Zukunft noch an diesem Pfad, beziehungsweise den dem Pfad zugeordneten Zuständen, interessiert ist und dass dieser Pfad somit erhalten bleiben soll.

Das Wiederauffrischen der Zustandsinformationen für einen Datenflusspfad ist so modelliert, dass es für das Signalisierungsprotokoll der identischen Aufgabe entspricht, einen Pfad zu erstellen, als auch ihn zu erneuern. Ein Client, der einen Pfad über eine längere Zeit nutzen will, kann also einfach den selben Aufruf, den er für die Erstellung des Pfades verwendet hat, später für das Wiederauffrischen desselben verwenden.

4. 3. 3. Datenfluss-Routing entlang eines Pfades

Das Routing eines Datenflusses entlang eines Pfades wird mit den Hilfsmitteln, die der Linux 2.4 Kernel zur Verfügung stellt, implementiert. Natürlich werden diese Funktionen getrennt implementiert, um durch den modularen Aufbau das einfache Portieren desselben zu gewährleisten.

Um das Pfad basierte Routing unter Linux 2.4 zu implementieren, werden wir die Möglichkeiten der Netfilter Architektur zum Markieren von Paketen, die gewissen Filtern entsprechen, ausnutzen.

Ebenfalls verwenden werden wir die Fähigkeit der IPRoute2 Architektur von Linux, die es ermöglicht, neue, spezielle Routing Tabellen dem Netzwerkstack zuzufügen, die dann den Marken der markierten Pakete zugeordnet sind.

Der Daemon, der auf dem Netzwerkknoten läuft, erstellt zur Startzeit eine Liste aller Nachbarn, die ebenfalls das Signalisierungsprotokoll unterstützen, und über die allfällige Pfade des pfadbasierten Routings führen können. Für jeden dieser Nachbarn erstellt der Daemon mit Hilfe der IPRoute2 Architektur eine Routing Tabelle, die genau einen einzigen Eintrag enthält: Die Default-Route zu genau diesem Nachbarn. Das heisst also, dass ein Paket, das die Routing-Tabelle eines Nachbarn X durchläuft, dank der Default-Route in dieser Routing-Tabelle, auch genau zu diesem Nachbar X weitergeschickt (forwarded) wird.

Später, zur Laufzeit des Daemons, wird eine allfällige Pfadreservation des Signalisierungsprotokolles so umgesetzt, dass mit Hilfe des Netfilter Frameworks genau die Pakete des gewünschten Datenflusses mit der Marke desjenigen Nachbarn markiert werden, der als nächster Knoten des Pfades vorgesehen ist.

Es gibt also zwei einfache Zuordnungen:
- Zu jedem Nachbar gehört eine Marke und eine Routing Tabelle
- Zu jedem Datenfluss entlang eines Pfades gibt es einen nächsten Nachbarn und eine Marke

Ersteres wird mit den Möglichkeiten der IPRoute2 Architektur, die normalerweise mit dem Userspace Programm ‘ip’ konfiguriert wird, das Zweite wird mit dem Netfilter Framework, das normalerweise mit dem Userspace Tool ‘iptables’ konfiguriert wird, und von den Filtern der Mangle-Tabelle realisiert. Die Mangle-Tabelle des Netfilter Frameworks ist diejenige Tabelle des Frameworks, in der Filter zum Markieren von Paketen eingetragen werden. Der das Signalisierungsprotokoll implementierende Daemon verwendet genau die selben Schnittstellen zur Wahrnehmung dieser Aufgaben, die auch die Userspace Programme ‘iptables’ und ‘ip’ verwenden.

Natürlich kann es passieren, dass die gewählten Filter in der Mangle-Tabelle für verschiedene Datenflüsse kollidieren. In dem Fall wird der zweite Benutzer beim reservieren des Pfades feststellen müssen, dass nicht alle nötigen Ressourcen vorhanden sind, da ja der erste Benutzer schon einen Filter verwendet hat, der mit dem Filter des zweiten Benutzers kollidiert. Das Signalisierungsprotokoll wird dies in der Reservationsphase bemerken und eine entsprechende Fehlermeldung zurückgeben.

4. 3. 4. Dynamisches Plugin/Datenfluss Binden entlang eines Pfades

Das dynamische binden von Datenflüssen an Plugins wird mit Hilfe der Component Based Routing Architecture (COBRA) [4] und dem Netfilter Framework [4] realisiert.

Das COBRA Framework [6] ermöglicht es einem lokal laufenden Programm mit der Unterstützung des Netfilter Frameworks beliebige Plugins zu laden und an Datenflüsse zu binden.

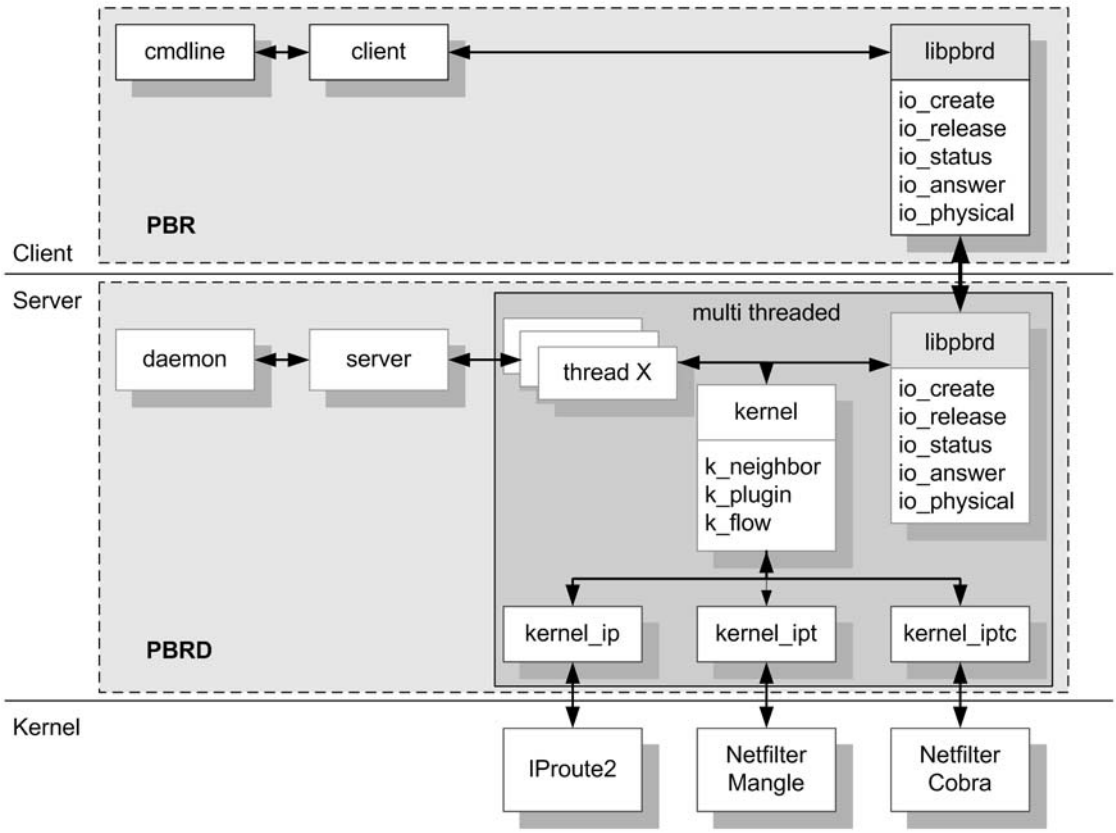
Der das Signalisierungsprotokoll implementierende Daemon nimmt diese Möglichkeiten wahr, um über die Netfilter Schnittstelle COBRA Plugins zu laden, zu konfigurieren, zu instanzieren und sie an Datenflüsse zu binden. Er tut dies in genau derselben Art und Weise wie auch das Userspace Tool ‘iptables’ dies tut.

Natürlich kann es auch beim Binden der Plugins an ihre Datenflüsse passieren, dass zwei Filter innerhalb der ‘cobra’-Tabelle kollidieren. Auch hier wird das Signalisierungsprotokoll dies in der Reservationsphase bemerken und es dem Benutzer durch eine Fehlermeldung zu verstehen geben.

Dieses Flow/Plugin Binding ist schon vollständig durch das COBRA Framework gelöst und kann vom, das Signalisierungsprotokoll implementierenden Daemon ganz normal, wie vom Netfilter Framework vorgesehen, verwendet werden.

4. 4. Aufbau des Signalisierungsprotokolls

Um einen Überblick über den modularen Aufbau des Signalisierungsprotokolles zu gewinnen, ist es wohl am besten, die folgende Module Schema-Zeichnung genauer zu betrachten.



Figur G: Modularer Aufbau des Signalisierungsprotokolles

Jedes der gezeigten Baublöcke entspricht einer logischen Abstraktion und ebenfalls genau einem Object File. Die Implementation hält sich an den genau gleichen, modularen Aufbau.

Wie man sieht, kann man die Module grob in 3 Sorten kategorisieren:

- Die Module ‘cmdline’, ‘client’ und libpbrd werden auf der Userspace Seite verwendet, um zusammen die Funktionalität des Kommandozeilen-Clients PBR zu implementieren.
- Die Module ‘daemon’, ‘server’, ‘thread’, ‘kernel’, ‘kernel_*’ und die Kommunikationsbibliothek ‘libpbrd’ implementieren den Userspace Daemon PBRD, der das eigentliche Signalisierungsprotokoll implementiert. Er ist multithreaded programmiert und die Module in der ‘multithreaded’ Box werden von den einzelnen Threads des Daemons aufgerufen.
- Die Module ‘kernel_*’ enthalten die systemabhängigen Teile, die auf die Netfilter, COBRA und IPRoute2 Frameworks aufbauen. Diese Module müssen neu geschrieben werden, wenn das Signalisierungsprotokoll auf eine andere Architektur portiert werden soll.

Am unteren Rand sind dann die Kernel-Teile schematisch dargestellt, die mit den systemabhängigen Modulen des Daemons kommunizieren: IPRoute2, Netfilter und COBRA. Sie gehören zum eigentlichen Linux Kernel bzw. zur COBRA Erweiterung desselben.

5. Implementation des Signalisierungsprotokolles

5. 1. Signalisierungspaket-Typen

Das Signalisierungsprotokoll verwendet 4 verschiedene Paket Typen, die zwischen den Daemons und den Clients verschickt werden. Diese Paket Typen haben einen Header Teil, der allen gemeinsam ist und der den Paket Typ enthält. Abhängig von diesem Feld kann dann der restliche Teil des Paketes anders interpretiert werden.

Die folgende Tabelle zeigt die 4 Paketarten und die ihnen zugehörigen Strukturen:

Table 2: Signalisierungspaket-Typen

Type	Struktur	Module
PBR_PBK_TYPE_CREATEPATH	**struct pbr_create_path_pkg**	**io_create**
PBR_PBK_TYPE_RELEASEPATH	**struct pbr_release_path_pkg**	**io_release**
PBR_PBK_TYPE_STATUS	**struct pbr_status_pkg**	**io_status**
PBR_PBK_TYPE_ANSWER	**struct pbr_answer_pkg**	**io_answer**

5. 2. Gliederung der PBRD IO Library Implementation

Im folgenden Bild ist die schematische Gliederung der Kommunikationsbibliothek dargestellt:

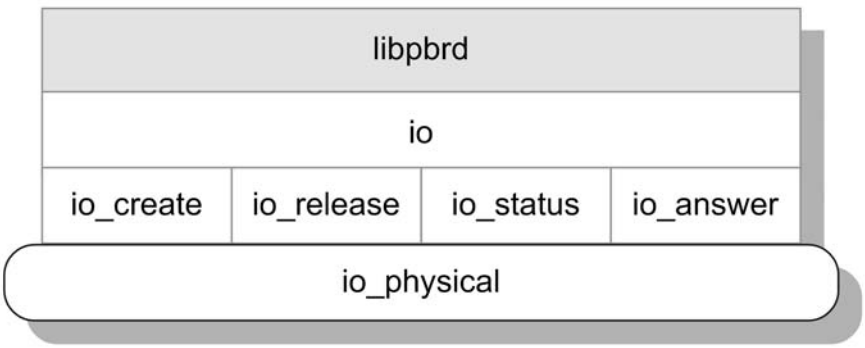


Bild H: Die Module der Kommunikationsbibliothek ‘libpbrd’

Die Bibliothek besteht aus den Modulen ‘io_create’, ‘io_release’, ‘io_status’ und ‘io_answer’, welche jeweils einen Paket Typ und die dazugehörenden Funktionen implementieren. Diese Module bauen auf dem Modul ‘io_physical’ auf, das dann die eigentliche Kommunikation stattfinden lässt.

Das Modul ‘io’ implementiert die Funktionen zum Verbindungsaufbau und Abbau, so wie allgemeine Paket Sende- und Empfangsfunktionen, die dann abhängig vom Paket Typ die entsprechenden Sende- und Empfangsfunktionen der Module ‘io_*’ aufrufen. Ebenfalls vom ‘io’ Modul implementiert ist eine Error-Routine die Error-Codes in Fehlermeldungen umwandelt.

Ich möchte noch auf einige Datenstrukturen eingehen, die im Header File io.h definiert werden und von den weiter unten beschriebenen Modulen, bei der Definition ihrer Datenstrukturen, mehrfach verwendet werden.

Die Struktur **`struct pbr_plugin`**, die eine Liste von Plugins modelliert, sieht folgedermassen aus:

```
struct pbr_plugin
{
  char name[PBR_MAX_PLUGIN_NAME_LEN];
  char init[PBR_MAX_PLUGIN_INIT_LEN];

  struct pbr_plugin *next;
};
```

Die Struktur **`struct pbr_hop`**, die einen Pfad modelliert, sieht folgendermassen aus:

```
struct pbr_hop
{
  struct in_addr ip;
  struct pbr_plugin *plugins;

  struct pbr_hop *next;
};
```

Die Struktur **`struct pbr_flow`**, die einen Datenfluss modelliert, sieht folgendermassen aus:

```
struct pbr_flow
{
  struct in_addr src_ip;
  u_int16_t src_mask;
  struct in_addr dst_ip;
  u_int16_t dst_mask;
  struct in_addr interface;
  u_int16_t src_port;
  u_int16_t dst_port;
  u_int8_t proto;
};
```

Nun gibt es noch eine letzte Datenstruktur, und sie modelliert ein allgemeines Paket unbekannten Typs:

```
struct pbr_pkg
{
  u_int8_t type;
};
```

5. 2. 1. IO Physical

Das ‘io_physical’ Modul besteht aus dem Source File io_physical.c und dem zugehörigen Header File mit Namen io_physical.h.

Die Aufgabe dieses Modul ist es, Kommunikationsfunktionen zur Verfügung zu stellen, die es den anderen Modulen ermöglicht, Daten strukturiert an einen Socket zu senden.

Die folgende Liste zeigt die implementierten Funktionen und ihre Aufgaben:

- **`pbr_send()`**
  Diese Funktion schreibt den übergebenen Buffer bestimmter Länge auf den Socket
- **`pbr_sendip()`**
  Diese Funktion schreibt die übergebene IP Adresse auf den Socket
- **`pbr_sendshort()`**
  Diese Funktion schreibt einen Short (2 Byte) auf den Socket
- **`pbr_sendbyte()`**
  Diese Funktion schreibt ein Byte auf den Socket
- **`pbr_sendstring()`**
  Diese Funktion schreibt einen String auf den Socket
- **`pbr_receive()`**
  Diese Funktion liest einen Buffer bestimmter Länge vom Socket
- **`pbr_receiveip()`**
  Diese Funktion liest eine IP Adresse vom Socket
- **`pbr_receiveshort()`**
  Diese Funktion liest einen Short vom Socket
- **`pbr_receivebyte()`**
  Diese Funktion liest ein Byte vom Socket
- **`pbr_receivestring()`**
  Diese Funktion liest einen String vom Socket

5. 2. 2. IO Create

Das ‘io_create’ Modul besteht aus dem Source File io_create.c und dem zugehörigen Header File mit Namen io_create.h.

Die Aufgabe dieses Moduls ist es, alle Funktionen zu kapseln, die mit Paketen des Typs **`PBR_PKG_TYPE_CREATE`** zu tun haben.

Das Header File ‘io.h’ definiert die Struktur **`struct pbr_create_path_pkg`**, die ein solches Paket modelliert.

Sie sieht folgendermassen aus:

```
struct pbr_create_path_pkg
{
  u_int8_t type;
  struct pbr_flow *flow;
  struct pbr_hop *hops;
};
```

Die folgende Liste zeigt die implementierten Funktionen und ihre Aufgaben:

- **`pbr_newCreatePathPkg()`**
  Generiert ein neues Paket und setzt seine Felder auf die übergebenen Werte.
- **`pbr_freeCreatePathPkg()`**
  Gibt ein Paket und seine Felder wieder frei.
- **`pbr_sendCreatePathPkg()`**
  Sendet ein Paket mit Hilfe der Funktionen des ‘io_physical’ Moduls, das in Kapitel 5. 2. 1. - “IO Physical” auf Seite 23 besprochen wird.
- **`pbr_receiveCreatePathPkg()`**
  Empfängt ein Paket mit Hilfe der Funktionen des ‘io_physical’ Moduls, das in Kapitel 5. 2. 1. - “IO Physical” auf Seite 23 besprochen wird.

5. 2. 3. IO Release

Das ‘io_release’ Modul besteht aus dem Source File io_release.c und dem zugehörigen Header File mit Namen io_release.h.

Die Aufgabe dieses Moduls ist es, alle Funktionen zu kapseln, die mit Paketen des Typs **`PBR_PKG_TYPE_RELEASE`** zu tun haben.

Das Header File ‘io.h’ definiert die Struktur **`struct pbr_release_path_pkg`**, die ein solches Paket modelliert.

Sie sieht folgendermassen aus:

```
struct pbr_release_path_pkg
{
  u_int8_t type;
  struct pbr_flow *flow;
  struct pbr_hop *hops;
};
```

Die folgende Liste zeigt die implementierten Funktionen und ihre Aufgaben:

- **`pbr_newReleasePathPkg()`**
  Generiert ein neues Paket und setzt seine Felder auf die übergebenen Werte.
- **`pbr_freeReleasePathPkg()`**
  Gibt ein Paket und seine Felder wieder frei.
- **`pbr_sendReleasePathPkg()`**
  Sendet ein Paket mit Hilfe der Funktionen des ‘io_physical’ Moduls, das in Kapitel 5. 2. 1. - “IO Physical” auf Seite 23 besprochen wird.
- **`pbr_receiveReleasePathPkg()`**
  Empfängt ein Paket mit Hilfe der Funktionen des ‘io_physical’ Moduls, das in Kapitel 5. 2. 1. - “IO Physical” auf Seite 23 besprochen wird.

5. 2. 4. IO Status

Das ‘io_status’ Modul besteht aus dem Source File io_status.c und dem zugehörigen Header File mit Namen io_status.h.

Die Aufgabe dieses Moduls ist es, alle Funktionen zu kapseln, die mit Paketen des Typs **`PBR_PKG_TYPE_STATUS`** zu tun haben.

Das Header File ‘io.h’ definiert die Struktur **`struct pbr_status_pkg`**, die ein solches Paket modelliert.

Sie sieht folgendermassen aus:

```
struct pbr_status_pkg
{
  u_int8_t type;
  char status[PBR_MAX_STATUS_LEN];
};
```

Die folgende Liste zeigt die implementierten Funktionen und ihre Aufgaben:

- **`pbr_newStatusPkg()`**
  Generiert ein neues Paket und setzt seine Felder auf die übergebenen Werte.
- **`pbr_freeStatusPkg()`**
  Gibt ein Paket und seine Felder wieder frei.
- **`pbr_sendStatusPkg()`**
  Sendet ein Paket mit Hilfe der Funktionen des ‘io_physical’ Moduls, das in Kapitel 5. 2. 1. - “IO Physical” auf Seite 23 besprochen wird.
- **`pbr_receiveStatusPkg()`**
  Empfängt ein Paket mit Hilfe der Funktionen des ‘io_physical’ Moduls, das in Kapitel 5. 2. 1. - “IO Physical” auf Seite 23 besprochen wird.

5. 2. 5. IO Answer

Das ‘io_answer’ Modul besteht aus dem Source File io_answer.c und dem zugehörigen Header File mit Namen io_answer.h.

Die Aufgabe dieses Moduls ist es, alle Funktionen zu kapseln, die mit Paketen des Typs **`PBR_PKG_TYPE_ANSWER`** zu tun haben.

Das Header File ‘io.h’ definiert die Struktur **`struct pbr_answer_pkg`**, die ein solches Paket modelliert.

Sie sieht folgendermassen aus:

```
struct pbr_answer_pkg
{
  u_int8_t type;
  u_int8_t error_no;
  char error_text[PBR_MAX_ANSWER_ERR_LEN];
};
```

Die folgende Liste zeigt die implementierten Funktionen und ihre Aufgaben:

- **`pbr_newAnswerPkg()`**
  Generiert ein neues Paket und setzt seine Felder auf die übergebenen Werte.
- **`pbr_freeAnswerPkg()`**
  Gibt ein Paket und seine Felder wieder frei.

- **`pbr_sendAnswerPkg()`**
  Sendet ein Paket mit Hilfe der Funktionen des ‘io_physical’ Moduls, das in Kapitel 5. 2. 1. - “IO Physical” auf Seite 23 besprochen wird.
- **`pbr_receiveAnswerPkg()`**
  Empfängt ein Paket mit Hilfe der Funktionen des ‘io_physical’ Moduls, das in Kapitel 5. 2. 1. - “IO Physical” auf Seite 23 besprochen wird.

5. 3. Gliederung der PBRD Daemon Implementation

Im folgenden Schema der Daemon-Implementation ist der modulare Aufbau des Daemons einfach ersichtlich.

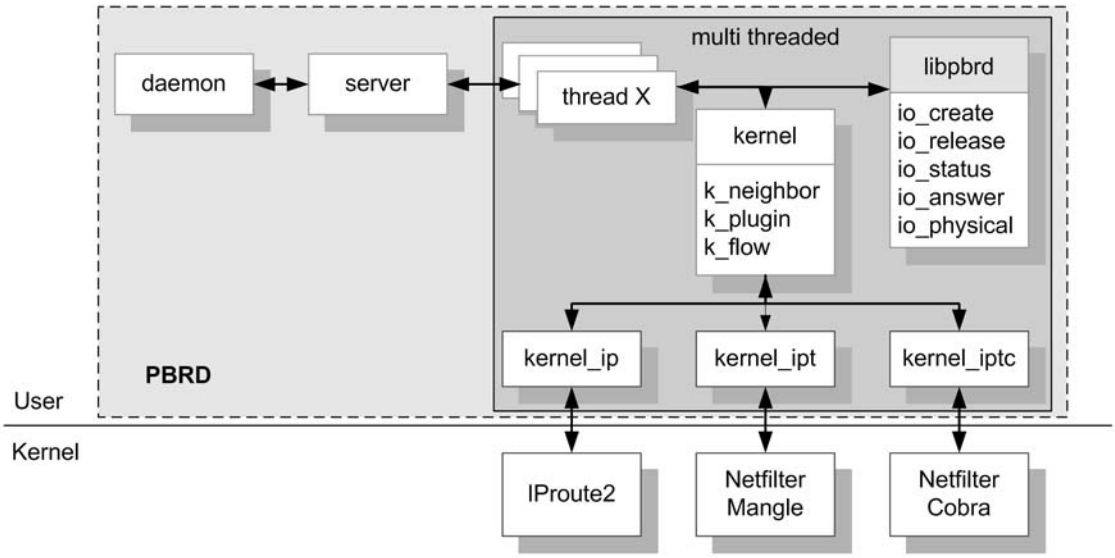


Bild I: Die Module des Signalisierungsprotokoll-Daemon ‘PBRD’

Der PBRD Daemon besteht aus den Modulen ‘daemon’ und ‘server’, die den eigentlichen Daemon implementieren.

Dieser startet dann für jede eingehende Verbindung einen Thread der diese Verbindung bedient. Der gestartete Thread, der das eigentliche Signalisierungsprotokoll implementiert, besteht wiederum aus mehreren Modulen.

Die eigentliche Kontroll-Logik ist im Modul ‘Thread’ implementiert. Dieses Modul implementiert das eigentliche Signalisierungsprotokoll. Er verwendet sowohl die Bibliothek ‘libpbrd’ wie auch die Schnittstellen zum Kernel in den ‘kernel_*’ Modulen und ein weiteres Modul ‘kernel’ zur Verwaltung der nötigen Datenstrukturen, die für die Zustandsspeicherung der Datenflüsse, Nachbarn, Marken und Plugins in den Datenstrukturen ‘k_neighbor’, ‘k_plugin’, ‘k_flow’ nötig sind.

Natürlich muss der Zugriff der verschiedenen Threads auf die Datenstrukturen ‘k_neighbor’, ‘k_plugin’, ‘k_flow’ durch Semaphoren geschützt werden, damit beim Mehrfachzugriff auf die Variablen keine Probleme entstehen. Die Funktionen des ‘kernel’ Modules sorgen dafür, dass dies geschieht.

Das Module 'Thread' verwendet dann die Kernel-Schnittstelle der 'kernel_*' Module um den Zugriff auf die systemspezifischen Abstraktionsmodule 'kernel_*' zu regeln.

Die systemspezifischen Teile des Signalisierungsprotokoll-Daemons sind in den folgenden drei Modulen gekapselt:

- 'kernel_ip' regelt den Zugriff auf die Funktionen der IPRoute2 Architektur, ist also zuständig für jene Funktionen, die mit dem Kommandozeilen Kommando 'ip' konfigurierbar sind. Über dieses Modul werden die neuen Marken abhängigen Routing Tabellen generiert und mit einer Default Route zum jeweiligen Nachbarn gefüllt. Kapitel 5. 3. 10. - "Kernel IP Route2 - Pfad basiertes Routing" auf Seite 34 geht weiter darauf ein.
- 'kernel_ipt' regelt den Zugriff auf die Funktionen des Netfilter Frameworks [4]. Dieses Modul ist also dafür zuständig, dass Filter für das Marking in der Mangle-Tabelle von Netfilter erstellt werden. In Kapitel 5. 3. 8. - "Kernel Netfilter Mangle Table - Markieren der Pakete" auf Seite 33 wird dies genauer beschrieben.
- 'kernel_iptc' regelt dann schlussendlich den Zugriff auf die Funktionen des COBRA Frameworks [6]. Über die Funktionen dieses Modules werden COBRA Plugins geladen, konfiguriert und an Datenflüsse gebunden. Kapitel 5. 3. 9. - "Kernel Netfilter Cobra Table - Plugin Zuordnung" auf Seite 34 beschreibt dies im Detail.

Die systemspezifischen Module sind einfach aufgebaut und sollten einfach zu portieren sein.

5. 3. 1. Daemon

Das Modul 'daemon' besteht aus dem Source File daemon.c und dem zugehörigen Header File daemon.h.

Das 'daemon' Modul definiert die **`main()`** Funktion des Daemons sowie einen Signal Handler der die Signale **`SIGTERM`**, **`SIGQUIT`** und **`SIGINT`** abfängt und ein sauberes Beenden durch Freigeben aller Speicherbereiche ermöglicht. Dieser Signal Handler ruft die Funktion **`server_stop()`** des Moduls 'server' auf, das im Kapitel 5. 3. 2. - "Server" auf Seite 28 besprochen wird.

Die Aufgaben der **`main()`** Funktion sind wenige. Sie interpretiert die ihr übergebenen Optionen der Kommandozeile und generiert allfällige Datenstrukturen aus diesen. So wird beispielsweise eine verkettete Liste aller auf der Kommandozeile angegebenen Nachbarn erstellt.

Falls beim Interpretieren der Optionen ein Fehler auftritt, wird der im Header File definierte Hilfstext ausgegeben.

Danach wird das Log Modul über die Funktion **`logopen()`** initialisiert und erste Log Meldungen ausgegeben.

Im weiteren führt der Prozess ein **`fork()`** aus und schliesst den ursprünglichen Prozess, um sich vom kontrollierenden TTY zu trennen, der Signal Handler wird installiert und daraufhin der eigentliche Server gestartet.

Dies geschieht durch den Aufruf der Funktionen **`server_init()`**, der die verkettete Liste der Nachbarn übergeben werden, **`server_run()`**, die wiederum der entsprechende Server Port mitgegeben wird und erst wieder zurückkehrt, wenn der Server beendet, und schlussendlich **`server_exit()`**, der wieder alle Nachbarn, in Form der verkettetten Liste, mitgeben wird.

Danach wird die verkette Liste der Nachbarn wieder freigegeben, das Log geschlossen und der Prozess beendet.

5. 3. 2. Server

Das ‘server’ Modul, besteht aus dem Source File server.c und dem zugehörigen Header File server.c. Es implementiert lediglich 4 Server Funktionen.

Die beiden Funktionen **`server_init()`** und **`server_exit()`** dienen der Initialisierung, beziehungsweise der Freigabe allfälliger Datenstrukturen. Beide Funktionen rufen ihre Pendants im ‘kernel’ Modul auf, das im Kapitel 5. 3. 4. - “Kernel” auf Seite 30 genauer beschrieben wird. Beide Funktionen geben die verkettete Liste den zu konfigurierenden bzw. wieder freizugebenden Nachbarn mit.

Die Funktion **`server_stop()`** wird vom Interrupt Handler, der im ‘daemon’ Modul definiert und im Kapitel 5. 3. 1. - “Daemon” auf Seite 27 beschrieben wurde, aufgerufen um die Server Funktion server_run() zu veranlassen sich zu beenden.

Die Funktion **`server_run()`** implementiert die eigentliche Serveraufgabe. Sie alloziert einen **`AF_INET`** Socket, füllt eine **`struct sockaddr_in`** mit dem ihr übergebenen Port, setzt die Socket Option **`SO_REUSEADDR`** auf dem Socket und bindet dann den Socket mit **`bind()`** an den Port. Anschliessend setzt der Prozess den Socket mit der Funktion **`listen()`** in den Listen State und startet eine nur durch **`server_stop()`** unterbrechbare Endlosschleife. Innerhalb dieser Endlosschleife, ruft der Prozess die Funktion **`accept()`** des Sockets auf, die erst retourniert, wenn eine neue Verbindung am Socket anfällt und startet dann mit Hilfe der Bibliothek ‘libpthread’ und der Funktion **`pthread_create()`** einen Thread, dessen Funktion **`thread_run()`** aufgerufen wird, und der als Argument den akzeptierten Socket mit der neuen Verbindung erhält.

5. 3. 3. Thread

Das ‘thread’ Modul besteht natürlich wieder aus dem Source File thread.c und dem zugehörigen Header File mit Namen thread.h und exportiert an andere Module lediglich die Funktion **`thread_run()`**. Alle weiteren Funktionen sind privat und werden nur vom Modul ‘thread’ selber verwendet.

Die Funktion **`thread_run()`** implementiert die eigentliche Signalisierungsprotokoll-Logik.

`thread_run()` startet gleich als Erstes eine Endlosschleife, die lediglich dann verlassen wird, wenn der Socket geschlossen und somit die Kommunikationsbeziehung beendet wird. Jeder Start eines Threads und der dazugehörende Aufruf von **`thread_run()`** bedingt, dass ein neuer Request anliegt und somit ein anderer Daemon oder ein Client einen Request an diesen Daemon senden möchte.

Der Ablauf ist einfach. Zuerst wird ein Paket von beliebigem Typ mit der Funktion **`pbr_receivePkg()`** aus der Kommunikationsbibliothek, die im Kapitel 5. 2. - “Gliederung der PBRD IO Library Implementation” auf Seite 21 genauer beschrieben ist, empfangen um dann an die Funktion **`thread_process_pkg()`** weitergegeben zu werden, die das Paket verarbeitet und ein Antwortpaket zurücksendet.

Die Funktion thread_process_pkg() liest dann den Paket Typ aus dem Header des Paketes und übergibt das Paket dann der entsprechenden Paket Typ abhängigen Funktion, währendem der Paket Point auch noch gerade in den richtigen Pointer Typ umgewandelt wird.

Abhängig vom Paket Typ wird eine der folgenden Funktionen aufgerufen:
- **`thread_process_create_path_pkg()`** für den Typ **`PBR_PKG_TYPE_CREATEPATH`**
- **`thread_process_release_path_pkg()`** für den Typ **`PKG_PKG_TYPE_RELEASEPATH`**
- **`thread_process_status_path_pkg()`** für den Typ **`PBR_PKG_TYPE_STATUS`**

Die einfachste dieser Funktionen ist **`thread_process_status_pkg()`**, denn sie ruft lediglich die Funktion **`kernel_status()`** des ‘kernel’ Moduls auf, das im Kapitel 5. 3. 4. - “Kernel” auf Seite 30 genauer beschrieben wird, alloziert ei neues Status Paket, das mit dem Rückgabewert der Kernel Status Funktion gefüllt wird, und sendet dieses neue Antwortpaket mit der Funktion **`pbr_sendPkg()`** an den anfragenden Kommunikationspartner. Nach dem Senden gibt sie das neu allozierte Paket wieder frei.

Die beiden anderen Funktionen arbeiten ähnlich, implementieren aber viel mehr Funktionalität. Auch sie erhalten ein Paket und auch sie senden am Schluss ein Paket zurück, allerdings ist das Antwort Paket vom Typ **`PBR_PKG_TYPE_ANSWER`**. Beide Funktionen verhalten sich gleich, ist die Aufgabe der einen doch genau die umgekehrte Aufgabe der zweiten Funktion. Wir werden in diesem Text nur auf die Funktion **`thread_process_create_path_pkg()`** eingehen, die Funktion **`thread_process_release_path_pkg()`** funktioniert analog.

Als erstes wird geprüft, ob wir wirklich der nächste Knoten sind, der im Paket vermerkt ist. Ist dies nicht der Fall, wird ein entsprechendes Antwort Paket generiert, mit einer Fehlermeldung gefüllt und zurückgesendet. Dann liest die Funktion die Pfad-Liste des Paketes und überprüft, ob dieser Knoten der letzte Knoten des aufzusetzenden Pfades ist oder nicht.

Auf dem letzten Knoten des Pfades müssen nur die Plugins an den Datenfluss gebunden werden. Es müssen keine Markierungen und Routing Entscheide mehr getroffen werden. Ist dieser Knoten wirklich der Letzte des Pfades, so wird, falls auf diesem Knoten Plugins an den Datenfluss zu binden sind, die Funktion **`kernel_install_plugins()`** aufgerufen, falls das Plugins noch nicht installiert ist, ansonsten wird die Funktion **`kernel_renew_plugins()`** aufgerufen, die die TTL des Plugin erfrischt und an den Datenfluss bindet. Dann wird ein Antwort Paket generiert, und mit dem Rückgabewert der einen oder anderen Funktion gefüllt. Dieses Anwort Paket wird dann am Ende der Funktion **`pbr_sendPkg()`** übergeben, die die Anwort an den anfragenden Host zurückschickt.

Ist dieser Knoten nicht der Letzte des Pfades, so ist mehr zu tun. Dann wird geprüft, ob der im Paket angegebene Nachbar, also der nächste Knoten des Pfades, auch wirklich als Nachbar dieses Knotens konfiguriert ist. Ist dies nicht der Fall, wird wieder ein Antwort Paket generiert, mit einer Fehlermeldung gefüllt und zurückgesendet.

Nun ist sicher, dass der Pfad noch weiter geht und somit das Paket weitergesendet werden muss. Um das Paket weiterzusenden, nachdem dieser Knoten aus der Pfad-Liste entfernt wurde, wird das Paket der Funktion **`thread_send_to_nexthop()`** aufgerufen und ihr das Paket übergeben. Die Funktion liefert einen Rückgabewert, der dem Antwort Paket des nächsten Knotens entspricht. Ist in dieser Antwort keine Fehlermeldung enthalten, so wird weitergefahren, ansonsten wird die in der Antwort enthaltene Fehlermeldung wieder als Anwort an den aufrufenden Host zurückgesendet.

Nachdem dies geschehen ist, wird unterschieden ob der entsprechende Pfad schon aufgebaut ist oder nicht. Falls er schon aufgebaut ist, werden die TTLs von Pfad, wie auch Plugins mit Hilfe der Funktionen **`kernel_renew_flow()`** und **`kernel_renew_plugins()`** erfrischt, so dass die zugehörigen Soft-States nicht verfallen. Ist der Pfad und die Plugins neu, so wird der entsprechende Datenfluss und die zugehörigen Plugins mit den Funktionen **`kernel_install_flow()`** und **`kernel_install_plugins()`** installiert. Allfällige Fehler werden in Antwort Pakete gekapselt und zurückgeschickt.

Schlussendlich bleibt noch die Funktion **`thread_send_to_nexthop()`** zu besprechen. Sie macht nichts anderes, als sich mit der Funktion pbr_connect() zum nächsten Knoten zu verbinden, ihm das Paket mit pbr_sendPkg() zu senden und dann mit pbr_receivePkg() eine Antwort vom nächsten Knoten zu erhalten. Dieses Antwort Paket wird retourniert.

5. 3. 4. Kernel

Das ‘kernel’ Modul besteht natürlich wieder aus dem Source File kernel.c und dem zugehörigen Header File mit Namen kernel.h.

Dieses Modul implementiert vier Gruppen von Funktionen:
- Funktionen zum Installieren, Deinstallieren und Erneuern von Pfaddefinitionen
- Funktionen zum Installieren, Deinstallieren und Erneuern von Plugindefinitionen
- Funktionen zum Prüfen und Finden von Nachbarn und ihrer Marken
- Einige Hilfsfunktionen

Zur ersten Gruppe gehören die Funktionen **`kernel_install_flow()`**, **`kernel_uninstall_flow()`** und **`kernel_renew_flow()`**. Zur zweiten Gruppe gehören die Funktionen **`kernel_install_plugins()`**, **`kernel_uninstall_plugins()`**, **`kernel_renew_plugins()`**. Zur dritten Gruppe gehören die Funktionen **`kernel_checkneighbor()`** und **`kernel_neighbormark()`**. All diese Funktionen sind durch Semaphoren geschützt, so dass Concurrent-Threads, die die Funktionen aufrufen und damit auf gemeinsame Speicherbereiche zugreifen, gegeneinander geschützt sind. So wird garantiert, dass sich immer nur genau ein einziger Thread im kritischen Bereich befindet.

Beiden Gruppen gemeinsam ist, dass sie sich um das Schützen der entsprechenden Datenstrukturen vor Concurrent Prozessen kümmern, funktional in beiden Gruppen jeweils ein Pendant zu finden ist und sie die jeweiligen Funktionen in den systemspezifischen Modulen ‘kernel_ip’, ‘kernel_ipt’ und ‘kernel_iptc’ aktivieren, die nötig sind, um die jeweiligen Datenflüsse, Markierfilter und Plugins aufzusetzten. Die dazu verwendeten Funktionen werden im Kapitel 5. 3. 8. - “Kernel Netfilter Mangle Table - Markieren der Pakete” auf Seite 33, Kapitel 5. 3. 9. - “Kernel Netfilter Cobra Table - Plugin Zuordnung” auf Seite 34 und Kapitel 5. 3. 10. - “Kernel IP Route2 - Pfad basiertes Routing” auf Seite 34 besprochen.

Die Funktion **`kernel_checkneighbor()`** prüft ob ein gegebener Knoten ein Nachbar ist. Die Funktion **`kernel_neighbormark()`** retourniert die dem übergebenen Nachbarn zugeordenete Marke und **`kernel_status()`** retourniert den aktuellen Zustandsspeicher des Knoten in einer für Menschen lesbaren Form indem sie die Funktionen **`kernel_pluginstatus()`**, **`kernel_flowstatus()`**, und **`kernel_neighborstatus()`** nacheinander aufruft und ihre Zurückgegebenen Informationen zusammengefügt retourniert.

Des Weiteren sind einige Hilfsfunktionen in diesem Modul implementiert, namentlich **`kernel_init()`** und **`kernel_exit()`**, die den Timer des ‘kernel’ Moduls aktivieren und deaktivieren, der dafür sorgt, dass die Zustände altern. Dazu existieren die nicht exportierten Funktionen **`kernel_timer_handler()`**, die den Timer Signal Handler implementiert und der, um die Zustände altern zu lassen, die Funktionen **`kernel_flowdec()`** und **`kernel_plugindec()`** aufruft. Dieser Handler wird von den Funktionen **`kernel_init_timer()`** und **`kernel_exit_time()`** installiert und deinstalliert.

5. 3. 5. Kernel Flow - Datenfluss Zustandsspeicher

Das ‘kernel_flow’ Modul besteht natürlich wieder aus dem Source File kernel_flow.c und dem zugehörigen Header File mit Namen kernel_flow.h.

Das Header File definiert die Struktur **`struct kernel_flow`**, die eine doppelt verkettete Liste der Datenflussdefinitionen modelliert.

Sie sieht folgendermassen aus:

```
struct kernel_flow
{
  struct pbr_flow flow;
  long timer;
  struct in_addr nexthop;
  long mark;
  struct kernel_flow *next;
  struct kernel_flow *prev;
};
```

Des weiteren definiert das Header File noch die initiale TTL der Datenflüsse im Precompiler-Makro **`KFLOW_TTL`** und setzt es auf den Wert 3600, was einer Stunde entspricht.

Die folgende Liste zeigt die implementierten Funktionen und ihre Aufgaben zur Manipulation der Datenstrukturen:

- **`kernel_flowadd()`**
  Ermöglicht das Hinzufügen eines neuen Datenflusses.
- **`kernel_flowrenew()`**
  Erfrischt den Soft-State Zustandsspeicher durch Setzen des **`time`** Feldes auf **`KFLOW_TTL`**.
- **`kernel_flowremove()`**
  Ermöglicht das Löschen eines eingetragenen Datenflusses.
- **`kernel_flowremoveall()`**
  Entfernt alle Datenflüsse.
- **`kernel_flowdec()`**
  Dekrementiert die TTL des entsprechenden Datenflusses.
- **`kernel_flowcheck()`**
  Prüft alle Datenflüsse und löscht diejenigen, deren TTL abgelaufen ist.
- **`kernel_flowfind()`**
  Sucht einen Datenfluss aus der Liste und retourniert ihn.
- **`kernel_flowfind_no()`**
  Sucht einen Datenfluss aus der Liste heraus und retourniert seinen Index.
- **`kernel_flowstatus()`**
  Retourniert den aktuellen Zustand der Datenflussdefinitionen in textueller Form.

5. 3. 6. Kernel Plugin - Plugin Zustandsspeicher

Das ‘kernel_plugin’ Modul besteht natürlich wieder aus dem Source File kernel_plugin.c und dem zugehörigen Header File mit Namen kernel_plugin.h.

Das Header File definiert die Struktur **`struct kernel_plugin`**, die eine doppelt verkettete Liste der Datenflussdefinition für das entsprechende, in der Datenstruktur vermerkte, Plugin modelliert.

Sie sieht folgdendermassen aus:

```
struct kernel_plugin
{
  struct pbr_flow flow;
  long timer;
  struct pbr_plugin plugin;
  struct kernel_plugin *next;
  struct kernel_plugin *prev;
};
```

Des Weiteren definiert das Header File noch die initiale TTL der Datenflüsse im Precompiler-Makro **`KFLOW_TTL`** und setzt es auf den Wert 3600, was einer Stunde entspricht.

Die folgende Liste zeigt die implementierten Funktionen und ihre Aufgaben zur Manipulation der Datenstrukturen:

- **`kernel_pluginadd()`**
  Ermöglicht das Hinzufügen eines neuen Plugins.
- **`kernel_pluginrenew()`**
  Erfrischt den Soft-State Zustandsspeicher durch Setzen des **`time`** Feldes auf **`KPLUGIN_TTL`**.
- **`kernel_pluginremove()`**
  Ermöglicht das Löschen eines eingetragenen Plugins.
- **`kernel_pluginremoveall()`**
  Entfernt alle Plugins.
- **`kernel_plugindec()`**
  Dekrementiert die TTL des entsprechenden Plugins.
- **`kernel_plugincheck()`**
  Prüft alle Plugins und löscht diejenigen, deren TTL abgelaufen ist.
- **`kernel_pluginfind()`**
  Sucht ein Plugin aus der Liste und retourniert es.
- **`kernel_pluginfind_no()`**
  Sucht ein Plugin aus der Liste heraus und retourniert seinen Index.
- **`kernel_pluginstatus()`**
  Retourniert den aktuellen Zustand der Plugindefinitionen in textueller Form.

5. 3. 7. Kernel Neighbor - Nachbar Zustandsspeicher

Das ‘kernel_neighbor’ Modul besteht natürlich wieder aus dem Source File kernel_neighbor.c und dem zugehörigen Header File mit Namen kernel_neighbor.h.

Die Aufgabe dieses Moduls ist es, die verkettete Liste der konfigurierten Nachbarn zu verwalten und die in der Datenstruktur enthaltenen Informationen zu verwerten, oder den aufrufenden Funktionen zurückzugeben.

Die folgende Liste zeigt die implementierten Funktionen und ihre Aufgaben:

- **`kernel_neighborinit()`**
  Speichert die ihr übergebene verkettete Liste der Nachbarn.
- **`kernel_neighborcheck()`**
  Prüft ob der übergebene Knoten ein Nachbar ist und retourniert dies.
- **`kernel_neighbormark()`**
  Sucht die dem Nachbarn zugeordnete Marke und retourniert sie.
- **`kernel_neighborstatus()`**
  Retourniert den aktuellen Zustand der Nachbardefinitionen in textueller Form.

5. 3. 8. Kernel Netfilter Mangle Table - Markieren der Pakete

Das ‘kernel_ipt’ Modul besteht natürlich wieder aus dem Source File kernel_ipt.c und dem zugehörigen Header File mit Namen kernel_ipt.h.

Die Aufgabe dieses Modules ist es, den Zugriff auf das Netfilter Framework zu abstrahieren, zu kapseln und dem ‘kernel’ Modul zugänglich zu machen. Die dabei manipulierte Netfilter Tabelle hat den Namen ‘mangle’ und kann in der Komandozeile mit dem Kommando ‘**`iptables -t mangle -L`**’ eingesehen werden.

Die folgende Liste zeigt die vom Modul implementierten Funktionen und ihre Aufgaben:

- **`kipt_init()`**
  Initialisiert das Netfilter Framework, löscht allfällig noch vorhandene Chains und Regeln, generiert eine ‘pbrd’ Chain innerhalb der Mangle Tabelle und fügt die neue ‘pbrd’ Chain in die **`PREROUTING`** und **`OUTPUT`** Chain ein.
- **`kipt_exit()`**
  Löscht die ‘pbrd’ Chain und die auf sie verweisenden Zeiger in der **`PREROUTING`** und **`OUTPUT`** Chain der Mangle Tabelle.
- **`kipt_install_flow()`**
  Installiert eine Markier-Regel in der ‘pbrd’ Chain und setzt als Filter genau die Spezifikation des Datenflusses. Markiert wird mit der übergebenen Marke. Dazu werden die entsprechenden **`Netfilter-Matches TCP und UDP`**, wie auch das **`Netfilter-Target MARK`** verwendet.
- **`kipt_uninstall_flow()`**
  Löscht die Markier-Regel des entsprechenden Datenflusses in der ‘pbrd’ Chain.

5. 3. 9. Kernel Netfilter Cobra Table - Plugin Zuordnung

Das ‘kernel_iptc’ Modul besteht natürlich wieder aus dem Source File kernel_iptc.c und dem zugehörigen Header File mit Namen kernel_iptc.h.

Die Aufgabe dieses Modules ist es, den Zugriff auf das COBRA Framework zu abstrahieren, zu kapseln und dem ‘kernel’ Modul zugänglich zu machen. Der Zugriff auf das COBRA Framework erfolgt in genau der selben Art und Weise wie beim Netfilter Framework, ist doch COBRA eine Erweiterung desselben. Die dabei manipulierte Netfilter Tabelle hat den Namen ‘cobra’ und kann in der Komandozeile mit dem Kommando ‘**`iptables -t cobra -L`**’ eingesehen werden.

Die folgende Liste zeigt die vom Modul implementierten Funktionen und ihre Aufgaben:

- **`kiptc_init()`**
  Initialisiert das COBRA Framework, löscht allfällig noch vorhandene Chains und Regeln, generiert eine ‘pbrd’ Chain innerhalb der Cobra Tabelle und fügt die neue ‘pbrd’ Chain in die **`PREROUTING`** und **`OUTPUT`** Chain ein.
- **`kiptc_exit()`**
  Löscht die ‘pbrd’ Chain und die auf sie verweisenden Zeiger in der **`PREROUTING`** und **`OUTPUT`** Chain der Cobra Tabelle.
- **`kiptc_install_plugin()`**
  Installiert ein Plugin und eine Plugin-Regel in der ‘pbrd’ Chain und setzt als Filter genau die Spezifikation des Datenflusses. Der Name des Ziel-Plugins wird ebenfalls spezifiziert. Dazu werden die entprechenden **`Netfilter-Matches TCP und UDP`**, wie auch das **`Cobra-Target COBRA`** mit den entsprechenden Argumenten zum Plugin-Namen und Konfiguration verwendet.
- **`kiptc_uninstall_plugin()`**
  Löscht die Plugin-Regel des entsprechenden Plugins in der ‘pbrd’ Chain.

5. 3. 10. Kernel IP Route2 - Pfad basiertes Routing

Das ‘kernel_ip’ Modul besteht natürlich wieder aus dem Source File kernel_ip.c und dem zugehörigen Header File mit Namen kernel_ip.h.

Die Aufgabe dieses Modules ist es, den Zugriff auf die IPRoute2 Architektur zu abstrahieren, zu kapseln und dem ‘kernel’ Modul zugänglich zu machen. Die aktuelle Konfiguration kann in der Komandozeile mit dem Komando ‘**`ip rule ls`**’ und ‘**`ip route ls table X`**’ eingesehen werden.

Die folgende Liste zeigt die vom Modul implementierten Funktionen und ihre Aufgaben:

- **`kip_init()`**
  Erstellt für jeden konfigurierten Nachbarn eine Routing Tabelle. Dies entspricht einem ‘ip route add default via N’, wobei N des Nachbars IP Adresse ist. Dann konfiguriert die Funktion den Netzwerkstack dahingehend, jeweils die entsprechende Routing Tabelle für die entsprechend markierten Pakete zu wählen. Prinzipiell entspricht dies einem ‘ip rule add fwmark M table T’ mit der Marke M und der Routing Tabellennummer T.
- **`kip_exit()`**
  Löscht alle Routing Tabellen und ihren Aufruf für entsprechende Marken.

5. 4. Gliederung der PBR Client Implementation

Der Komandozeilen-Client besteht aus den Modulen ‘cmdline’ und ‘client’. Das ‘client’ Modul kommuniziert direkt mit der Kommunikationsbibliothek ‘libpbrd’, wie im folgenden Schema ersichtlich ist:

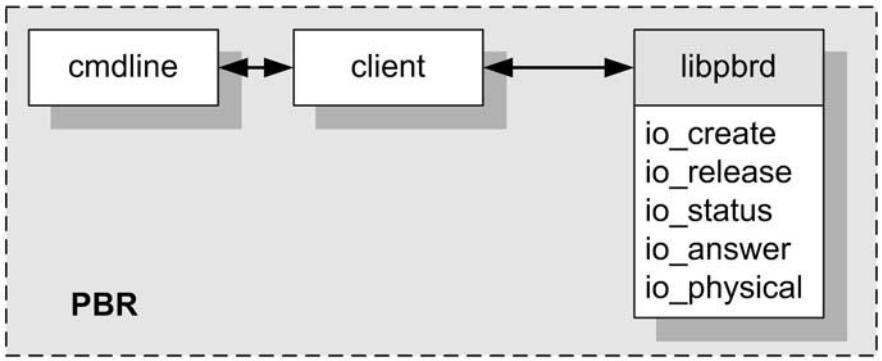


Bild J: PBR - Die Module des Kommandozeilen Client

5. 4. 1. Cmdline

Das ‘cmdline’ Modul besteht natürlich wieder aus dem Source File cmdline.c und dem zugehörigen Header File mit Namen cmdline.h.

Die Aufgabe dieses Moduls ist es, einen Signal Handler zu installieren, die Kommandozeilen Parameter zu interpretieren und dann Kontakt mit einem PBRD Daemon aufzunehmen und die vom Benutzer gewünschten Funktionen wahrzunehmen.

Dieses Modul definiert die main() Funktion des Clients. Sie interpretiert zuerst alle übergebenen Kommandozeilenparameter und validiert diese. Falls beim interpretieren der Optionen ein Fehler auftritt, wird der im Header File definierte Hilfstext ausgegeben.

Dann wird das Log Modul über die Funktion **`logopen()`** initialisiert und erste Log Meldungen ausgegeben.

Danach wird der Signal Handler installiert, der in der Funktion **`signal_handler()`** implementiert ist.

Abhängig von der vom Benutzer gewünschten Funktion wird dann eine, der folgenden drei Funktionen, mit den auf der Kommandozeile angegebenen Parametern aufgerufen:

- **`client_create_path()`**
- **`client_release_path()`**
- **`client_status()`**

Danach wird das Log mit **`logclose()`** geschlossen und der Client beendet.

5. 4. 2. Client

Das ‘client’ Modul besteht natürlich wieder aus dem Source File client.c und dem zugehörigen Header File mit Namen client.h.

Die Aufgabe dieses Modules ist es die vom Benutzer gewünschte Funktion dem entsprechenden Daemon mitzuteilen und dessen Antwort wieder dem Benutzer zu präsentieren.

Die folgende Liste zeigt die vom Modul implementierten Funktionen und ihre Aufgaben:

- **`client_create_path()`**
  Zuerst wird ein neues Paket vom Typ **`PBR_PKG_TYPE_CREATE`** erzeugt, das mit den übergebenen Informationen wie Pfad und Plugins gefüllt wird. Anschliessend wird eine Verbindung zum ersten Knoten des Pfades erstellt und das Paket zu diesem Knoten gesendet. Die erhaltene Antwort wird ausgegeben.
- **`client_release_path()`**
  Zuerst wird ein neues Paket vom Typ **`PBR_PKG_TYPE_RELEASE`** erzeugt, das mit den übergebenen Informationen wie Pfad und Plugins gefüllt wird. Anschliessend wird eine Verbindung zum ersten Knoten des Pfades erstellt und das Paket zu diesem Knoten gesendet. Die erhaltene Antwort wird ausgegeben.
- **`client_status()`**
  Zuerst wird ein neues Paket vom Typ **`PBR_PKG_TYPE_STATUS`** erzeugt. Anschliessend wird eine Verbindung zum ersten Knoten des Pfades erstellt und das Paket zu diesem Knoten gesendet. Die erhaltene Antwort wird ausgegeben.

6. Demonstration

In diesem Kapitel soll der Daemon PBRD und der Kommandozeilen Client PBR demonstriert werden. Um diese Demonstration möglichst alle Elemente zeigen zu lassen, haben wir am TIK ein kleines Testnetzwerk aufgebaut, dessen Knoten alle das neue Signalisierungsprotokoll unterstützen.

Zuerst möchte ich auf den Aufbau dieses Testnetzes eingehen. Wir wollten ein möglichst flexibles Testbett haben. Dazu wurden die 4 Rechner msr[12345] an das selbe Ethernet gehängt. Jeder der Rechner msr[2345] besitzt eine ATM Karte und hängt auch noch an einem ATM Switch. Die dichte Vermaschung der Maschinen untereinander haben wir mit ATM durch Virtual Circuits (VC) modelliert. Jede Maschine besitzt 2 - 3 ATM Interfaces mit dem atm[012].

Im folgenden Bild ist die vollständige Topologie des Netzes dargestellt:

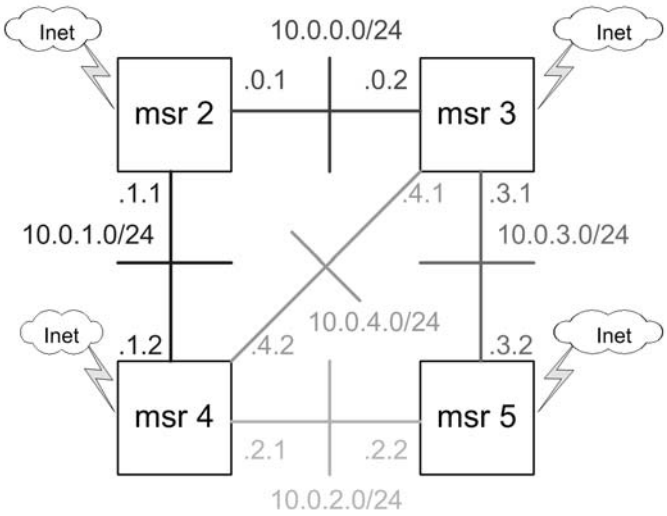


Bild K: Topologie des Test-Netzwerkes

Im obigen Topologieschema sind sowohl die IP Adressen der jeweiligen ATM Interfaces, wie auch die jeweiligen Netze verzeichnet. Zum Verständnis der Demonstration ist es von Nöten wenigstens einen kleinen Überblick über die verwendeten IP Adressen zu haben, um den Debug Output im Log zu verstehen.

Auf jeder der Maschinen msr[2345] läuft der PBRD Daemon. Der Yellow Page Server msr1 ist Log-Host für alle Maschinen. Durch Beobachten der Datei /var/log/messages auf dem Server msr1, kann man alle Log-Meldungen der PBRD Daemons einsehen, die ja alle per Syslog auf den Log-Host loggen.

Jeder der PBRD Daemons ist mit seinen Nachbarn konfiguriert und kennt also alle ATM Interfaces seiner Nachbarn.

6. 1. Unbeeinflusstes Routing

Um den Wandel des Routing-Verhaltens der einzelnen Maschinen besser zu sehen, möchte ich zuerst einmal zeigen, was ohne neu aufgesetzte Pfade geschieht.

Wenn wir auf der Maschine msr2 einloggen und uns dort mit einem Telnet auf den Port 12345 der Maschine msr1 verbinden, so erhalten wir ein ‘Connection Refused’ was ja auch korrekt ist, da auf dem Port nichts läuft:

```
msr2:/# telnet msr1 12345
Trying 129.132.66.100...
telnet: Unable to connect to remote host: Connection refused
```

Wir halten fest: das Paket wurde dank der Default Route der Maschine msr2 auf das Ethernet geroutet, erreicht dort direkt msr1 und wird von ihr mit einem ICMP Paket beantwortet.

6. 2. Aufsetzen eines Pfades für einen speziellen Datenfluss

Nun wollen wir einen neuen Pfad aufsetzen, der dafür sorgt, dass ein Datenfluss der Form **<*,*,*,12345,*,*>** entlang eines speziellen Pfades gerouted wird.

Dieser Datenfluss soll entlang unseres neu aufzusetzenden Pfades geroutet werden. Des Weiteren wünschen wir uns, dass am Ende des Pfades ein Plugin geladen wird, das den Datenfluss verarbeitet. Im COBRA Framework enthalten ist ein Log-Plugin, dass es ermöglicht, jedes Paket eines Datenflusses zu loggen. Falls das Paket nicht an den letzten Knoten des Pfades adressiert ist, so wird das Paket nach dem letzten Knoten wieder dem Standard-Routing der entsprechenden nächsten Knoten unterworfen sein und so weiter seinen Weg zur Zieladresse suchen.

Der Pfad soll so verlaufen, wie es im folgenden Bild gezeigt ist:

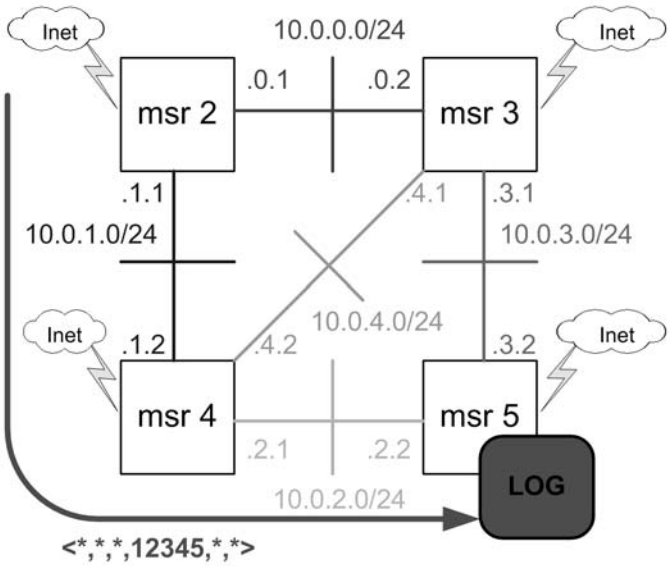


Bild L: Pfad und Log-Plugin im Test-Netzwerk

Um dies alles zu erreichen, müssen wir als erstes die Datenfluss-Spezifikation, wie auch den Pfad und die Plugin-Definition in Kommandozeilen Parameter umwandeln.

Um den gewünschten Pfad zu erstellen sind die folgenden Paramter nötig:

```
create 10.0.1.1:10.0.1.2:10.0.2.2
```

Die obige Spezifikation des Datenflusses entspricht den folgenden Kommandozeilenparametern des PBR Clients:

```
-p TCP --dport 12345
```

Des weiteren wollen wir auf dem letzten Knoten des Pfades ein Plugin mit Namen ‘LOG’ laden und es mit der Zeichenfolge ‘init’ initialisieren:

```
--plugin 10.0.2.2:LOG:init
```

Zusammengesetzt ergibt sich die folgende Kommandozeile, die wir auch gerade auf der Maschine msr2 ausführen:

```
msr2:/# pbr create 10.0.1.1:10.0.1.2:10.0.2.2 -p TCP --dport 12345 --plugin
10.0.2.2:LOG:init
client: CREATE PATH SUCCESS!
```

Das Erstellen des Pfades war also erfolgreich! Der Pfad kann nun verwendet werden.
Die folgende Ausgabe erscheint im Log-File auf msr1:

```
Apr  6 12:38:08 msr2 pbrd[11825]: thread#8: connecting to next hop at 10.0.1.2
Apr  6 12:38:08 msr4 pbrd[2270]: thread#10: connecting to next hop at 10.0.2.2
Apr  6 12:38:08 msr5 pbrd[2271]: thread#8: install/renew: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=10.0.2.2
Apr  6 12:38:08 msr5 pbrd[2271]: kernel_iptc: installing plugin 'LOG' with init 'init'
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: init() called...
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: init() - create_proc_entry('/proc/net/
cobra_instance') succeeded...
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: init() - create_proc_entry('/proc/net/
cobra_status') succeeded...
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: init() succeded...
Apr  6 12:38:08 msr5 kernel: COBRA: cobra_init(LOG) called...
Apr  6 12:38:08 msr5 kernel: COBRA_LOG: load() called...
Apr  6 12:38:08 msr5 kernel: COBRA_LOG: load() succeded...
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: cobra_register(LOG) called...
Apr  6 12:38:08 msr5 kernel: COBRA: cobra_init(LOG) succeded...
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: pf_instance_read - returning instance 1
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: configuring plugin LOG instance 1.
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: config() dispatching to plugin LOG#1
Apr  6 12:38:08 msr5 kernel: COBRA_LOG: config() called for plugin LOG#1 with config
'init'
Apr  6 12:38:08 msr5 kernel: COBRA_TARGET: checkentry(LOG): adding plugin instance 1
Apr  6 12:38:08 msr4 pbrd[2270]: thread#10: install/renew: src=0.0.0.0/
0:0,dst=0.0.0.0/0:12345,pro=6,if=10.0.1.2
Apr  6 12:38:08 msr4 pbrd[2270]: kernel_ipt: installing flow...
Apr  6 12:38:08 msr2 pbrd[11825]: thread#8: install/renew: src=0.0.0.0/
0:0,dst=0.0.0.0/0:12345,pro=6,if=0.0.0.0
Apr  6 12:38:08 msr2 pbrd[11825]: kernel_ipt: installing flow...
```

Im Folgenden werden im Log-File immer mal wieder solche und ähnliche Texte erscheinen:

```
Apr  6 12:39:22 msr2 pbrd[11772]: kernel-flow-ttl: ok: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=0.0.0.0
Apr  6 12:39:52 msr4 pbrd[2252]: kernel-flow-ttl: ok: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=10.0.1.2
Apr  6 12:40:15 msr5 pbrd[2255]: kernel-plugin-ttl: ok: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=10.0.2.2
```

Mit ihnen verkünden die einzelnen PBRD Daemons auf den Maschinen, dass sie die TTL des Pfades beziehungsweise des Plugins prüfen. Sollte die TTL ablaufen, so werden der Pfad und das Plugin wieder gelöscht.

Natürlich können wir den Pfad erneuern, seine TTL erfrischen. Dies erfolgt durch genau die gleiche Kommandozeile, die den Pfad auch erstellt hat. Die folgenden Log-Meldungen erscheinen dann:

```
Apr  6 12:42:51 msr2 pbrd[11828]: thread#8: connecting to next hop at 10.0.1.2
Apr  6 12:42:51 msr4 pbrd[2272]: thread#10: connecting to next hop at 10.0.2.2
Apr  6 12:42:51 msr5 pbrd[2273]: thread#8: install/renew: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=10.0.2.2
Apr  6 12:42:51 msr5 pbrd[2273]: kernel-plugin-ttl: renew: src=0.0.0.0/
0:0,dst=0.0.0.0/0:12345,pro=6,if=10.0.2.2
Apr  6 12:42:51 msr4 pbrd[2272]: thread#10: install/renew: src=0.0.0.0/
0:0,dst=0.0.0.0/0:12345,pro=6,if=10.0.1.2
Apr  6 12:42:51 msr4 pbrd[2272]: kernel-flow-ttl: renew: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=10.0.1.2
Apr  6 12:42:51 msr2 pbrd[11828]: thread#8: install/renew: src=0.0.0.0/
0:0,dst=0.0.0.0/0:12345,pro=6,if=0.0.0.0
Apr  6 12:42:51 msr2 pbrd[11828]: kernel-flow-ttl: renew: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=0.0.0.0
```

Wir können nun Daten, die der Datenflussdefinition entsprechen, generieren. Diese Daten sollten dem Pfad entlang bis zum Ende fliessen und dort geloggt werden.

Versuchen wir es! Ein erneutes Telnet von der Maschine msr2 an die Maschine msr1 mit dem Port 12345 liefert Folgendes:

```
msr2:/# telnet msr1 12345
Trying 129.132.66.100...
telnet: Unable to connect to remote host: Connection refused
```

Genau das selbe Verhalten! Schauen wir ins Log-File sehen wir das folgende:

```
Apr  6 12:44:59 msr5 kernel: COBRA_TARGET: target() called for plugin LOG#1
Apr  6 12:44:59 msr5 kernel: COBRA_TARGET: target() dispatching to plugin LOG#1
Apr  6 12:44:59 msr5 kernel: COBRA_LOG: target() called for plugin LOG#1 for hock 0
Apr  6 12:44:59 msr5 kernel: IN=eth0 OUT=eth1 SRC=129.132.66.101 DST=129.132.66.100
LEN=60 TOS=0x10 PREC=0x00 TTL=50 ID=51450 DF PROTO=TCP SPT=1025 DPT=12345 WINDOW=5840
RES=0x00 CWR ECE SYN URGP=0
```

Wie man sieht, hat alles geklappt! Das Paket ist nicht mehr direkt über das Ethernet Interface zur Maschine msr1, sondern das Paket hat den vorgeschriebenen Pfad über die ATM Interfaces zurückgelegt, ist auf der Maschine msr5 gesehen worden, ist dort vom Log-Plugin verarbeitet worden, was zu obigen Log-Zeilen geführt hat, und ist dann dort am Ende des Pfades wieder mit der Default Route der Maschine msr5 über das Ethernet Interface am Ziel angekommen.

Die Maschine msr1 hat natürlich wieder ein ICMP Paket generiert, das dann auf direktem Wege, über das Ethernet, als Antwort bei der Maschine msr2 ankam und die ‘Connection Refused’ Meldung erzeugte. Wir hatten ja auch nur einen unidirektionalen Pfad aufgesetzt.

6. 3. Abfragen des Pfad Status eines PBRD Daemons

Das Abfragen des Status eines Netzwerkknotens, der das neue Signalisierungsprotokoll unterstützt, ist einfach:

```
msr2:/# pbr status msr2

PBRD status of msr2

Flow status:

Flow #1: src=0.0.0.0/0:0, dst=0.0.0.0/0:12345, p=6, if=0.0.0.0
         ttl=2400, mark=2, nexthop=10.0.1.2

Plugin status:

No plugins installed

Neighbor status:

Neighbor 1: 10.0.0.2
Neighbor 2: 10.0.1.2
```

Unser aufgesetzter Pfad ist klar zu sehen. Der Status von msr5 sieht so aus:

```
msr2:/prodlocal/pbrd# pbr status msr5

PBRD status of msr5

Flow status:

No flows installed

Plugin status:

Plugin #1: src=0.0.0.0/0:0, dst=0.0.0.0/0:12345, p=6, if=10.0.2.2
         ttl=2400, plugin=LOG, init=init

Neighbor status:

Neighbor 1: 10.0.2.1
Neighbor 2: 10.0.3.1
```

Wie man sieht, hat msr5 keine Pfad Eintrageungen mehr, was ja auch logisch ist, denn msr5 muss keine Routing Entscheide mehr treffen, da er sich am Ende des Pfades befindet, aber dafür einen Eintrag für die Bindung des Log-Plugin an den Datenfluss.

6. 4. Löschen einer Pfad Definition

Nun wollen wir den Pfad, den wir aufgesetzt haben, auch wieder löschen. Natürlich könnten wir auch warten bis die TTL des Pfades abgelaufen ist. Wir würden dann aber in der Zwischenzeit, in der wir den Pfad ja gar nicht mehr brauchen, Netzwerk-Ressourcen verschwenden.

Mit der folgenden Kommandozeile wird der Pfad wieder abgebaut. Die Kommandozeile ist genau die selbe wie beim Aufsetzten des Pfades, lediglich das Wort ‘create’ wird durch ‘release’ ersetzt:

```
msr2:/prodlocal/pbrd# pbr release 10.0.1.1:10.0.1.2:10.0.2.2 -p TCP --dport 12345 --
plugin 10.0.2.2:LOG:init
client: RELEASE PATH SUCCESS!
```

Im Log-File ist dabei das Folgende zu sehen:

```
Apr  6 13:06:09 msr2 pbrd[11840]: thread#8: connecting to next hop at 10.0.1.2
Apr  6 13:06:09 msr4 pbrd[2277]: thread#10: connecting to next hop at 10.0.2.2
Apr  6 13:06:09 msr5 pbrd[2282]: thread#8: uninstalling: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=10.0.2.2
Apr  6 13:06:09 msr5 pbrd[2282]: kernel_iptc: uninstalling plugin 'LOG'
Apr  6 13:06:09 msr4 pbrd[2277]: thread#10: uninstalling: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=10.0.1.2
Apr  6 13:06:09 msr4 pbrd[2277]: kernel_ipt: uninstalling flow...
Apr  6 13:06:09 msr2 pbrd[11840]: thread#8: uninstalling: src=0.0.0.0/0:0,dst=0.0.0.0/
0:12345,pro=6,if=0.0.0.0
Apr  6 13:06:09 msr2 pbrd[11840]: kernel_ipt: uninstalling flow...
```

6. 5. Optionen des Kommandozeilen Clients PBR

Der Kommandozeilen Client gibt, falls er ohne Argumente gestartet wird, eine kleine Hilfe-Seite aus, die alle Optionen des Client erklärt:

```
pbr [--debug] [-d] [--help] [-h] [--server <server>] [--port <port>] <cmd>

Where <cmd> is one of: create, release, status

Usage:  create  hop1:hop2[:...:hopX] <flow-spec> [<plugin-spec>]
        release hop1:hop2[:...:hopX] <flow-spec> [<plugin-spec>]
        status  hop

Where <flow-spec> is one of: -s, -d, --sport, --dport, -p, -i:

Usage:  -s <src-ip>[/<src-mask>] --sport <src-port>
        -d <dst-ip>[/<dst-mask>] --dport <dst-port>
        -p <protocol>
        -i <in-interface>

Where <plugin-spec> is one of: --plugin

Usage:  --plugin <hop>:<name>:<init>[:<name>:<init>]
```

7. Schlussfolgerungen und Ausblick

Das Resultat dieser Arbeit ist eine vollständige Implementierung des von uns entworfenen Signalisierungsprotokolles. Das Signalisierungsprotokoll wurde einigen Stress-Tests ausgesetzt und könnte die in es gesetzten Erwartungen zu unserer Zufriedenheit erfüllen.

Alle Ziele, die wir uns gesetzt hatten, konnten erreicht werden:

- Es sind beliebige Pfad-Definitionen durch das heterogene Netzwerk möglich, soweit dies durch die Heterogenität des Netzes nicht verhindert wird.
- Beliebige Datenflüsse können an beliebige Plugins gebunden werden.
- Die Plugins können individuell konfiguriert werden und es können beliebig viele Plugins an beliebig viele Datenflüsse gebunden werden.
- Das Signalisierungsprotokoll ist modular aufgebaut. Sowohl Client, wie auch Server verwenden die selbe Kommunikationsbibiliothek. Die Linux systemspezifischen Teile sind gekapselt und können einfach auf neue Systeme portiert werden. Die Implementation ist nicht an Linux gebunden, nützt aber trotzdem die Vorzüge des Linux Kernels mit all seinen Möglichkeiten.
- Der eigentliche Aufwand des Signalisierungsprotokolles liegt im Pfadaufbau. Beim eigentlichen Datenverkehr wirkt lediglich der Overhead des Netfilter Frameworks als Bremse. Da die Arbeit des Netfilter Frameworks sich darauf beschränkt, Pakete dahingehend zu markieren, dass sie für eine der Routing Tabellen bestimmt sind, die für jeden Nachbarn generiert wurde, ist der Aufwand vertretbar. Er liegt in der Grössenordnung eines normalen, auf Linux Netfilter basierten Firewalls.
- Das Signalisierungsprotokoll implementiert die Zustandsspeicherung der Pfad- und Plugin-Konfigurationen als Soft-States. Dies, vielfältige Fehlerbehandlung und klug gewählte Timeouts, ermöglicht dem Signalisierungsprotokoll trotz widriger Umstände stabil, effizient und zuverlässig zu funktionieren.
- Das Signalisierungsprotokoll wurde als reiner Userspace Daemon implementiert. Er verwendet die standardisierten Schnittstellen zum Netfilter Framework, wie auch zur IPRoute2 Architektur.
- Eine einfache und konfortable Userspace Kommunikationsbibliothek für die Client-Entwicklung wurde geschaffen.
- Aussagekräftige Fehlermeldungen und eine äusserst ausführliche Debug-Ausgabe erleichtern das Verständnis der Funktion- und Arbeitsweise und erleichtert die Fehlersuche.

Sowohl das Netfilter-Framework [4], wie auch die COBRA Architektur [6] haben durch ihr offenes Design und ihre vielfältige Dokumentation die Entwicklung des Signalisierungsprotokolles sehr erleichtert.

Dadurch, dass sowohl Daemon wie auch Client als reine Userspace Prozesse entwickelt werden konnten und keinerlei Kernel-Manipulationen, ausser dem Compilieren eines Linux Kernels mit COBRA und Netfilter Support, nötig waren, hatten wir bei der Entwicklung keinerlei Probleme mit abstürzenden Systemen und ähnlichem. Sowohl der Daemon wie auch der Client sind äusserst stabil und selbst während der Entwicklung musste das Entwicklersystem nie rebooted werden.

Durch die sehr modulare Implementation und der sauberen Trennung der einzelnen Protokoll-Schichten konnte eine strenge Kapselung der einzelnen Module erfolgen, was die Entwicklung stark begünstigte!

Die Kommunikationsbibliothek ist sehr einfach zu benutzen. Mit ihrer Hilfe ist es leicht Client-Unterstützung für das neue Signalisierungsprotokoll zu implementieren, wie der Kommandozeilen Client PBR beweist.

Der hier gezeigte Entwurf des Signalisierungsprotokolles lässt Aspekte der Sicherheit aussen vor! Natürlich ist es von eminenter Bedeutung, dass alle Signalisierungsnachrichten authentisiert und autorisiert sind. Des Weiteren sind sicher auch Überlegungen betreffend der Privatsphäre des Benutzers anzustellen, was wahrscheindlich zu einer zusätzlichen Verschlüsselung der Signalisierungsnachrichten führen wird. Diese Aspekte könnten in einer weiteren Arbeit zusätzlich untersucht werden.

Des Weiteren wäre es möglich, die Aspekte der Pfadaggregation innerhalb des heterogenen Netzwerkes zu betrachten. Sowohl Pfad- wie auch Plugin-Konfigurationsaggregation, wenn mehrere Datenflüsse das gleiche, gleichkonfigurierte Plugin verwenden, wäre denkbar, um dem klassischen Skalierungsproblem im Backbone vorzubeugen.

Weiterhin wäre die Implementation von dynamischen Nachbardefinitionen sehr wünschenswert. Durch Einführen eines weiteren Signalisierungspaket-Typs wäre es durchaus möglich, dynamisch neue Nachbarn eines Netzwerkknotens definieren zu können und so beispielsweise Netzwerktopologieänderungen im Pfad basierten Routing berücksichtigen zu können.

So könnten zum Beispiel die Topologieinformationen, die ein Linkstate Protokoll wie OSPF besitzt, verwendet werden, um neue, eventuell kostenintensivere, Pfade ans dasselbe Ziel zu berechnen und so Failover Routen für das Pfad basierte Routing zu finden, zu evaluieren und dann eventuell automatisch, mit Hilfe dieses Signalisierungsprotokolles, zu konfigurieren.

8. Anhang

8. A. Offizielle Problemstellung

Aufgabenstellung:	Prof. Dr. Lothar Thiele, Ralph Keller
Titel:	**Signalisierungsprotokoll für Aktive Netzwerke mit Router Plugins**
Beginn der Arbeit:	23. Oktober 2001
Abgabetermin:	8. Februar 2002
Betreuung:	Ralph Keller, Philipp Blum, Lukas Ruf
Arbeitsplatz:	ETZ G69
Umgebung:	Linux, C

8. A. 1. Einleitung

Aktive Netzwerke (Active Networks) werden es in Zukunft ermöglichen, Netzwerke den eigenen Bedürfnissen nach zu programmieren, um neue Netzwerkdienste rasch und unkompliziert einführen zu können. Von der ursprünglich am TIK entwickelten Active Network Node (ANN) Architektur [1] [3] wurde eine Portierung auf Linux vorgenommen [5]. Das Linux Router Plugin Framework basiert auf der Netfilter Architektur und erlaubt, sogenannte Plugins, welche ausführbaren Code enthalten, in das Netzwerksubsystem zu installieren. Damit wird ermöglicht, den Netzwerkknoten auf flexible Weise mit beliebiger Funktionalität zu erweitern.
Der gegenwärtige Stand von Linux Router Plugins bietet die Basis-Funktionalität, um Plugins in den Kernel zu laden, Instanzen von Plugins zu generieren und Plugin-Instanzen an einen Filter zu binden. Damit jedoch eine Applikation beliebige Funktionalität im Netzwerk installieren kann, ist ein Mechanismus notwendig, welcher die Installation und Konfiguration von Plugin-Instanzen vereinfacht. Dazu wird ein Signalisierungsprotokoll benötigt, welches es erlaubt, diese Schritte quasi ferngesteuert vornehmen zu können. Dieses Signalisierungsprotokoll soll die folgenden Schritte ermöglichen:

- Falls sich der Code für das Plugin noch nicht auf dem Knoten befindet, soll das Plugin von einem Code-Server heruntergeladen werden.
- Von Plugins sollen Instanzen generiert werden, welche sich an einen bestimmten Paketfilter binden lassen.
- Das Routing durch das Netzwerk soll so modifiziert werden, dass Datenpakete entlang einer vordefinierten Route weitergeleitet und auf entsprechenden Knoten verarbeitet werden.

8. A. 2. Aufgabenstellung

Entwickeln Sie ein Framework, welches es erlaubt, Plugins mittels eines Signalisierungsprotokolls auf bestimmte Knoten zu laden, instanzieren und konfigurieren. Bestimmen Sie, welche Modifikationen und Erweiterungen am bestehenden Linux Router Plugin Framework und Plugin Management notwendig sind. Implementieren Sie schliesslich ihr Framework aufbauend auf Linux Router Plugins. Eine mögliche Architektur mit Signalisierungskomponenten ist in Abbildung 1 dargestellt. Dabei sind die folgenden Funktionalitäten notwendig:

- Ein API, welches von Applikationen benutzt wird, um aktive Pfade aufzusetzen. Dieses API sollte einfach gestaltet sein.
- Ein Signalisierungsprotokoll, welches erlaubt, Pfade auf entfernten Knoten aufzusetzen. Definieren Sie hierzu die entsprechenden Meldungstypen.
- Ein User-Kernel-Mechanismus zur Installation und Konfiguration von Kernelplugins.
- Ein Mechanismus zum Aufsetzen und Löschen von pfadspezifischem Routing. Basieren Sie diesen Mechanismus wenn möglich auf bereits bestehenden Linux Routing Komponenten

- Auf ein modulares Design wird in dieser Arbeit besonderes Gewicht gelegt. Trennen Sie dabei das API und Signalisierungsprotokoll von den entsprechenden Linux-Mechanismen, damit Ihr System einfach auf andere Architekturen portiert werden kann.

8. A. 2. a. Ziel

Ziel dieser Arbeit ist ein lauffähiges, flexibles Framework, welches es erlaubt, Plugins mittels eines Signalisierungsprotokolls auf verteilten Knoten zu installieren und an Filter zu binden. Demonstrieren Sie die Funktionsweise ihres Frameworks, indem Sie zeigen, wie eine Applikation die Installation und Konfiguration eines Plugins veranlassen kann.

8. A. 2. b. Vorgehen

- Lesen Sie sich in die Literatur zum Thema ein. Machen Sie sich zuerst mit der bestehenden Router Plugins Plattform vertraut ([1], [5]), dem Plugin Management [7] und bereits existierenden Signalisierungsprotokollen für Aktive Netze ([2], [6]).
- Erstellen Sie sich einen Zeitplan.
- Betrachten Sie die bestehende Router Plugins-Architektur [5] und erstellen Sie ein Design für ihr zu entwickelndes Framework. Bestimmen Sie die neuen Komponenten im User und Kernel Space, welche für ihr Framework notwendig sind.
- Definieren Sie ein Protokoll wie die verschiedenen Komponenten miteinander kommunizieren sollten.
- Definieren sie ebenfalls ein API, welches eine beliebige Applikation benutzen kann, um mittels ihres Frameworks Plugins auf bestimmten Knoten zu installieren.
- Implementieren Sie die notwendigen Komponenten unter Linux. Achten Sie auf ein modulares Design.
- Demonstrieren Sie die Lauffähigkeit ihres System, indem Sie zeigen, wie Sie mittels ihres APIs ein Plugin auf einen Knoten laden können, eine Instanz eines Plugins erzeugen und die Instanz an einen Filter binden können und das Routing durch das Netzwerk entlang des vorgeschrieben Pfades geschieht.
- Auf eine klare und ausführliche Dokumentation wird besonders Wert gelegt. Es wird empfohlen, diese laufend nachzuführen und insbesondere die entwickelten Konzepte ausführlich schriftlich festzuhalten.

8. A. 3. Bemerkungen

- Mit dem Betreuer sind wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen soll mündlich über den Fortgang der Arbeit berichtet und anstehende Probleme diskutiert werden.
- Es ist ein Zeitplan für den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Am Ende des zweiten Monats der Arbeit soll ein kurzer schriftlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt.
- Die Dokumentation ist vorzugsweise mit dem Textverarbeitungsprogramm FrameMaker oder Latex zu erstellen.

8. A. 4. Ergebnisse der Arbeit

- Neben einem mündlichen Vortrag von 20 Minuten Dauer im Rahmen des Fachseminars Kommunikationssysteme sind die folgenden schriftlichen Unterlagen abzugeben:

- Ein Bericht (in 3 Exemplaren). Dieser enthält eine Darstellung der Problematik, eine Beschreibung der untersuchten Entwurfsalternativen, eine Begründung für die getroffenen Entwurfsentscheidungen, sowie eine Liste der gelösten und ungelösten Probleme. Eine kritische Würdigung der gestellten Aufgabe und des vereinbarten Zeitplanes runden den Bericht ab.
- Ein Handbuch zum fertigen System bestehend aus Systemübersicht und Implementationsbeschreibung,
- Eine Sammlung aller zum System gehörender Software.
- Eine englischsprachige Zusammenfassung von 1 bis 2 Seiten, die einem Aussenstehenden einen schnellen Überblick über die Arbeit gestattet. Die Zusammenfassung ist wie folgt zu gliedern: (1) Introduction, (2) Aims & Goals, (3) Results, (4) Further Work.

8. A. 5. Literatur

[1] ANN - A Scalable, High Performance Active Network Node. http://king.ccrc.wustl.edu/~ann/

[2] Active Reservation Protocol. http://www.isi.edu/active-signal/ARP/index.html

[3] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B., “Router Plugins - A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers”, Washington University Tech Report WUCS-98-08, February 1998

[4] Decasper, D., Parulkar, G., Choi, S., DeHart, J., Wolf, T., Plattner, B., “A Scalable, High Performance Active Network Node”, In IEEE Network, January/February 1999

[5] Guindehi A., “COBRA - Component Based Routing Architecture”. Semester Thesis SA-2001.30, Swiss Federal Institute of Technology Zurich http://cobra.digital-impact.ch

[6] Chandra P., Fisher A., Steenkiste P., “Beagle: A Resource Allocation Protocol for an Advanced Services Internet”, technical report CMU-CS-98-150, August 1998. http://www.cs.cmu.edu/afs/cs/project/cmcl/archive/Darwin-papers/98beagletr.ps

[7] Choi, S., “Plugin Management”, Technical Report WUCS-00-04. http://www.arl.wustl.edu/arl/projects/ann/Prep/pm/pm_doc.pdf

[8] Choi, S., Plugin Management Software. http://www.arl.wustl.edu/arl/projects/ann/Prep/pm/pm.html

8. B. PBRD Sourcen

Die Sourcen des Cobra Frameworks, wie auch dessen Vortrags-Presentation und Dokumentation, können auf der folgenden Website bezogen werden:

http://cobra.digital-impact.ch/

Die Sourcen des PBRD, wie auch dessen Vortrags-Presentation und Dokumentation, können auf der folgenden Website bezogen werden:

http://pbrd.digital-impact.ch/

Für allfällige Fragen und Anregungen bezüglich des Cobra Frameworks oder bezüglich des Pfad basierten Routing Daemons PBRD ist der Autor unter den folgenden Email Adressen erreichbar:

amir@digital-impact.ch

und

amir@guindehi.ch

8. C. Liste aller Figuren

8. D. Liste aller Tabellen

8. E. Referenzen

[1] "**ANN - A Scalable, High Performance Active Network Node"**.
http://www.arl.wustl.edu/arl/projects/ann/ann.html

[2] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B.,
"Router Plugins - A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers",
Washington University Tech Report WUCS-98-08, February 1998.

[3] Decasper, D., Parulkar, G., Choi, S., DeHart, J.,Wolf, T., Plattner, B.,
"A Scalable, High Performance Active Network Node".
In IEEE Network, January/February 1999

[4] Rusty Roussell, **"The Netfilter Project"**.
http://netfilter.samba.org/

[5] Guindehi Amir,
"Ein durch Quality of Services (QoS) erweitertes Internet".
http://amir.ch/papers/QoS-Internet-V2.pdf

[6] Guindehi Amir,
"COBRA - Component Based Routing Architecture",
Semester Thesis SA-2001.30, Swiss Federal Institute of Technology Zurich.
http://cobra.digital-impact.ch

[7] Guindeh Amir,
"PBRD - Path Based Routing Daemon",
Semester Thesis SA-2002.17, Swiss Federal Institute of Technology Zurich.
http://pbrd.digital-impact.ch

[8] Prof. Dr. Burkhard Stiller,
"Protokolle für Multimediakommunikation",
WS 2000/2001, Eidgenössische Technische Hochschule Zürich, Departement für Elektrotechnik und Departement für Informatik.

[9] **"Router Plugins Toolkit"**.
http://www.tik.ee.ethz.ch/~crossbow/rp/

[10] Shreedhar M., Varghese G.,
"Efficient Fair Queueing using Deficit Round Robin",
Proc. ACM SIGCOMM, August/September, 1995.

[11] Chandra P., Fisher A., Steenkiste P.,
"Beagle: A Resource Allocation Protocol for an Advanced Services Internet",
technical report CMU-CS-98-150, August 1998.
http://www.cs.cmu.edu/afs/cs/project/cmcl/archive/Darwin-papers/98beagletr.ps$

[12] Choi, S.,
"Plugin Management",
Technical Report WUCS-00-04.
http://www.arl.wustl.edu/arl/projects/ann/Prep/pm/pm_doc.pdf

[13] Choi, S.,
"Plugin Management Software".
http://www.arl.wustl.edu/arl/projects/ann/Prep/pm/pm.html