Darmstadt University of Technology

A Flexible, QoS-Aware IP/ATM Adaptation Module

Jens Schmitt

December 1999

Technical Report TR-KOM-1999-06

Industrial Process and System Communications (KOM)

Department of Electrical Engineering & Information Technology
Merckstraße 25 • D-64283 Darmstadt • Germany

Phone:	+49 6151 166150
Fax:	+49 6151 166152
Email:	info@KOM.tu-darmstadt.de
URL:	http://www.kom.e-technik.tu-darmstadt.de/

A Flexible, QoS-Aware IP/ATM Adaptation Module

Jens Schmitt

Industrial Process and System Communications
Department of Electrical Engineering and Information Technology
Darmstadt University of Technology
Merckstr. 25 • D-64283 Darmstadt • Germany
{Jens.Schmitt, Martin.Karsten, Lars.Wolf, Ralf.Steinmetz}@kom.tu-darmstadt.de

Abstract

Overlaying IP-based networks onto ATM subnetworks is a network configuration pattern found increasingly often. While IP networks traditionally only offer plain "best-effort" service they are now evolving to offer more sophisticated services. Nevertheless, the exact mechanisms for providing QoS are not yet settled and essentially non-existing in today's production-level networks, with the Internet being the most popular and important example. On the other hand, ATM networks have been designed from their inception to offer a wide range of QoS mechanisms. Thus, given the configuration of an IP overlay network over an ATM subnetwork, it is very attractive to leverage ATM's QoS mechanisms to alleviate IP's QoS problem, at least partially. The invocation of those mechanisms will be done on so-called IP/ATM edge devices which are exactly at the frontier between the IP and ATM network. In particular, these edge devices could map reservation requests within the context of the RSVP/IntServ architecture onto especially setup VCs.

In this report we describe the design and implementation of a flexible, QoS-aware IP/ATM adaptation module. This adaptation module allows an IP/ATM edge device to route IP datagrams depending on their contents onto especially setup VCs in a performant manner. To achieve performance it is necessary to implement this module in kernel space, at least partially. On the other hand, it should be easy to use, for e.g. an RSVP/IntServ over ATM, or a DiffServ over ATM mapping module. Therefore, the adaptation module was split into two parts, a kernel-level part that handles all the time-critical tasks of data forwarding and a user-level part which gives access to the functionality provided by the adaptation module.

The result is a very flexible and general IP/ATM adaptation module that can be integrated conveniently with earlier results of the project on approaches of how to map the RSVP/IntServ mechanism onto those of an ATM subnetwork.

This work is supported in part by a grant of Volkswagen-Stiftung, D-30519 Hannover, Germany.

Table of Contents

1 Introduction

1.1 Motivation

IP-based production networks essentially still offer only best-effort service, and so does the largest IP-based network - the Internet. However, the Internet is becoming or even already is a commercially used ubiquitous communication infrastructure. A fact which will eventually require the Internet (or also large IP-based intranets) to be able to accurately predict its performance for business-critical applications, i.e. deliver stringent Quality of Service (QoS) guarantees for those applications.

On the other hand, the Asynchronous Transfer Mode (ATM) technology offers a lot of QoS-enabling facilities. However, due to its homogeneity stipulation, it faces its degradation to a link layer which is being used by TCP/IP in the core of the network where its accurate QoS mechanisms are most needed. The result is:

> IP lacks QoS, but has a wide distribution - ATM has QoS, but is not available end-to-end. Hence it seems very reasonable that IP takes ATM's assist in order to provide QoS, so that its huge user base can profit from ATM's facilities without the need of introducing ATM end-to-end.

In previous reports ([SWS97a], [SKWS98]) of the IQATM project we used the term overlay model for that kind of operation for the special case when mapping the RSVP/IntServ architecture onto ATM sub-networks.

In general, the problem of providing QoS in packet-switched networks can be separated into the distinct but related problems on the control and the data path. Previous prototypical implementations within the project ([SWS97b], see also Appendix) mainly focused on solution approaches for the control path issues of the problem, while they used a very simple and inefficient solution for the problem of providing QoS on the data path. That was due to the non-availability of source code of the ATM network driver, which needs to be modified for that purpose. Now we have access to such source code (thanks to the Fore partner's programme) and thus it was decided to overcome the deficiencies of the previous implementations on the data path.

Therefore we developed in a first step an IP/ATM adaptation module that allows to instruct the forwarding path inside an IP/ATM edge device to "route" IP data flows according to some characteristics onto especially setup ATM VCs. While the adaptation module's provided functionality is sufficient for efficiently operating RSVP/IntServ on the data path, it is intentionally designed more flexible to allow for other QoS-conveying information in IP datagrams (as e.g. contained in the Differentiated Services (DiffServ) byte) to be a source of special handling in the ATM network as well.

1.2 Outline

In the next chapter, the overall architecture of the IP/ATM adaptation module is being presented. We provide the design goals which lead the development of this adaptation module and give their rationale. Then a macroscopic view on the components of the adaptation module is given and the general interface to the adaptation module is described. The adaptation module consists of two instances which work together:

- a kernel instance, and a
- user instance.

In chapter 3, we describe the design and implementation of the kernel instance by looking at both, its global architecture and the functionality provided locally by the modules which form that architecture.

In chapter 4, details on the user instance are presented. Again we start by looking at its architecture, before we go into the details of the implementation of the components which make up that architecture.

In addition, we illustrate the application of the IP/ATM adaptation module by a short example of how to use it via the library interface provided by the user instance of the adaptation module.

In the last chapter, we provide a short summary of the report and give the relevant literature references.

2 Architecture of the IP/ATM Adaptation Module

2.1 Overview

In this section we describe the architecture of the IP/ATM adaptation module. We start by giving a brief overview of the function of that module, which will further on also be called VCM (Virtual Circuit Management) module. We discuss its design goals and what it is aimed at, and at the same time we point out what it is not, respectively what have been secondary goals during the process of development and why some restrictions have been made. Following these discussions we present the different components of the VCM module and their relationships among themselves as well as their relationship with existing code in the operating system and the ATM network device driver. Furthermore, we shortly illustrate its interface that shall allow higher layer/level software to make use of ATM's advanced QoS capabilities in an easy and elegant manner. A more detailed view on that interface is given in chapter 4.

The code for the IP/ATM adaptation module is still a prototype and does not offer yet all the functionality one could wish. What is still missing and which restrictions apply is treated at the end of this section along with the system requirements of the implementation of the VCM module.

2.2 Design Goals

We can distinguish between problem-specific goals and general design goals. Problem-specific goals are related to what we actually want to achieve with respect to the functionality of our IP/ATM adaptation module. General goals are related to desirable characteristics any software system is thriving to achieve, however we highlight those that are of particular importance for the VCM module.

2.2.1 Problem-Specific Goals

The first and foremost design goal is certainly to offer a **rich functionality**, which is to have a means of using ATM's mechanisms and characteristics for any IP QoS related matters, examples of which could be:

- RSVP/IntServ,
- ST-II,
- DiffServ/IP precedence,
- policy-based configurational (static) QoS,
- secure communications (e.g. for VPNs),
- simple hybrid TCP/IP-ATM API.

From the pretty diverse sample potential uses of the VCM module it follows that **flexibility** should be one of the most important design goals for the adaptation module. Flexibility here is with regard to:

- mapping of flows onto VCs, i.e. many-to-many relationships between flows and ATM VCs should be possible,
- description of what constitutes a flow (arbitrary rules on IP and higher level headers), i.e. more or less arbitrary rules on IP and higher level headers should be possible to define a flow of data that shall be forwarded using one or more (in case of multicast) VCs.

Another more technically motivated design goal is to be **independent of IP convergence modules** used for best-effort IP traffic delivery, i.e. the VCM module should be capable of interworking interchangeably with any of the following (only the most prominent examples):

- ForeIP - the proprietary implementation from Fore of IP over ATM for uni- and multicast best-effort transmissions,

- Classical IP over ATM (CLIP) - the IETF standard solution for unicast best-effort IP traffic over ATM subnetworks,
- Multi-Protocol over ATM (MPOA) - the ATM Forum's standard for delivering IP (beyond others) over ATM networks (again only unicast best-effort IP traffic).

The idea behind the independence from the IP convergence module in use is to be able to make use of their different strengths, e.g.

- when using ForeIP, then IP multicast is available and we are able to test all the fancy RSVP over ATM multicast issues we derived conceptually in [SKWS98],
- when using CLIP, then we will certainly find a large installed base and should thus in principle be able to use the VCM module as a base for larger tests,
- when using MPOA, then we can make use of NHRP-initiated shortcuts for unicast IP transmissions and thus maximally switched paths, an interesting feature we have investigated conceptually in [SWK$^+$99] and would like to be able to test in practice.

2.2.2 General Goals

Of course, the list of general design goals is virtually endless, however what we want to do here is to emphasize those that are of special significance to the development of the IP/ATM adaptation module. These are:

- **Modularity** of the code, especially in order to ease portability and migration to new releases of the operating system and/or ATM network driver code.
- **Reusability** of the code, since some parts could also be interesting to filtering software for firewalls or similar environments that need to deal with customizable forwarding decisions within an edge router, therefore genericity in this part could be beneficial.
- **Minimization** of **kernel**-level part, while **maximizing** the **user**-level part without sacrificing **efficiency** on the data forwarding path, i.e. only the most necessary changes to the forwarding behavior should be realized inside the kernel, while all the control functionality should be handed over to the user-space part of the implementation. Rationale behind this goal is the ease of development and coding in user-space when compared to kernel space.
- **Extensibility** of the code, is certainly a must, as for example the rules constituting a QoS-worthy flow will certainly experience changes and extensions. Similarly, with the advent of IPv6 the header formats will change and that must be accommodated by future versions of the VCM module as well.
- **Minimal invasiveness** with respect to existing code, i.e. Fore's ATM driver code should not be modified unnecessarily if possible, the same applies for the Solaris operating system source code[*]. This is a pragmatic design goal which allows us in the first place to make the code available to our partners at Deutsche Telekom AG since modifications inside the Solaris source code and/or the Fore ATM network driver code would either necessitate the existence of a source code license or only the binaries could be delivered thus preventing further extensions and modifications of the IP/ATM adaptation module.
- **Simple**, but **flexible interface** to the services provided by the VCM module. We want an **object-oriented** interface since this represents the problem domain well.

*. Note however that it was certainly necessary to know that code, in particular in order to be able to fit the adaptation module so neatly into the existing software

2.2.3 Secondary Goals

After having stated the general and problem-specific design goals for the IP/ATM adaptation module, we now want to make clear what we did not aim at primarily and why. While the following points in general certainly are important goals to strive for we will explain why we did not focus on them as much as on the aforementioned goals. Those "neglected" design goals are:

- **100% optimized performance** is not aimed at, although architecturally it should be possible if some work in tuning the software is invested. As mentioned above the performance-critical parts of the code, which are represented by the decisions on the data path which packets to "route" on which VC shall be part of the kernel like the rest of the communication subsystem under the Solaris operating system. Thus there is no fundamental performance problem. However, we do not intend to tune all the data structures to their optimum performance. For example, if the filter set inside the kernel is represented by a simple linear list, there are only small performance penalties for a relatively small filter set, whereas of course for larger sets with complicated (with respect to matching) filters this is certainly a different story. In this area very recent research work is available on packet classification (e.g. [WVTP97], [SVSW98] which also contain many pointers to other work in this field) which could be easily incorporated in principle, yet in practice adding some implementation complexity (and for maximum performance it would have to be done in hardware anyway), which was not considered necessary for our prototypical solution, which will under all likelihood never experience such a large filter set.
- **Portability -** while being an honorable goal, this is in our case only possible to a very limited degree, i.e. the code should be readily portable to System V based Unix platforms (e.g. Solaris, HP-UX, SGI, Digital UNIX, ...) and ATM network drivers that offer an API to the UNI signalling facilities within the driver code and a DLPI (Data Link Provider Interface) interface to upper layer protocols. However, operating system platforms that do not implement their communication subsystem based on the STREAMS mechanism of System V, or ATM network drivers that do not conform to DLPI or do not offer an API to the UNI services will certainly represent a major problem when the VCM module code is to be ported on them. While portability to such platforms in principle could be achieved by isolating the platform-dependent code rigorously, this would lead to substantial implementation efforts which are not justified for a prototypical system as is projected for the IP/ATM adaptation module.
- **Completeness** - while the VCM module shall be flexible and extensible it is not aimed at being complete. For example not all possible filter rules one could imagine should be implemented and made readily accessible to the user, but rather a user should with a minimum of modifications to the VCM module be able to extend the code for new filter rules to be applicable.
- **Failure Handling** - since the IP/ATM adaptation module is not aimed at being production-level code, sometimes simple failure handling (defaulting to exiting in extreme failure situations instead of handing over to the user with detailed reports of what went wrong) for ease of implementation should be given preference over absolute protection against faults. Nevertheless, all faults should be detected, albeit their handling does not always have to be as sophisticated as one could desire.

To emphasize once more: while the above goals have not been the leading forces for the development of the IP/ATM adaptation module, they have nevertheless not been totally neglected but have been realized in a "best-effort" manner, whereas the aforementioned primary design goals have been viewed as strict requirements.

2.3 Components and their Relations

After having discussed the goals which led our development of the IP/ATM adaptation module let us now take an overall view on the actual design of the VCM module and how it fits into the existing code of the operating system and the ATM network driver. This is illustrated in Figure 1. Let us start from the bottom up. Here we have the hardware of the network adaptors, i.e. the Ethernet† respectively the ATM controller. This corresponds to the physical layer. In the case of Ethernet the link layer is almost trivial and is realized in a moderately sized character device driver called le0 (under Solaris) on top of which the higher layer protocols are stacked. In a System V UNIX (as Solaris is one) the link layer would provide a standard interface called DLPI in order to offer its services to the network layer (in the case of BSD-based systems the ifnet interface would be provided by the link layer module). Furthermore, as is depicted in Figure 1, the upper layer protocol stack in System V UNIXes, i.e. the network and transport layer is implemented in kernel space using the STREAMS framework. The IP STREAMS multiplexer driver serves as the central component receiving several streams from upstream and multiplexing them on the correct downstream directions to the corresponding network drivers (according to the routing/forwarding table). From the IP multiplexer there are always two streams leading downstream to the network device drivers where one is for the ARP control requests and the other is for the actual datagrams to be sent over the network. For the former stream the ARP module is pushed, while for the latter the IP and the ARP module are pushed, in that order. Upstream from the IP multiplexer there are the transport layer STREAMS device drivers, the most common of which certainly are the TCP and UDP drivers. There is one stream for each transport layer. On such a stream a module corresponding to the respective transport layer is being pushed as can be seen again in Figure 1. Applications that want to send data using the TCP/IP protocol suite are of course run in user space and transmit their data to the transport layer devices by opening up a stream to the transport layer STREAMS devices (it is also possible to build up a stream directly to the IP multiplexer using "raw" IP, as for example does the RSVP daemon to send its protocol messages) and crossing the user-kernel borderline by using the STREAMS head. An example of such an application depicted in Figure 1 is the conferencing tool vic, of which there is a special release that is RSVP-enabled, i.e. uses RSVP signalling in order to reserve resources for its video flows to experience uncongested transmission paths. This is achieved by contacting the RSVP daemon that is running in user space via an API called RAPI (which is under standardization by the X/OPEN group).

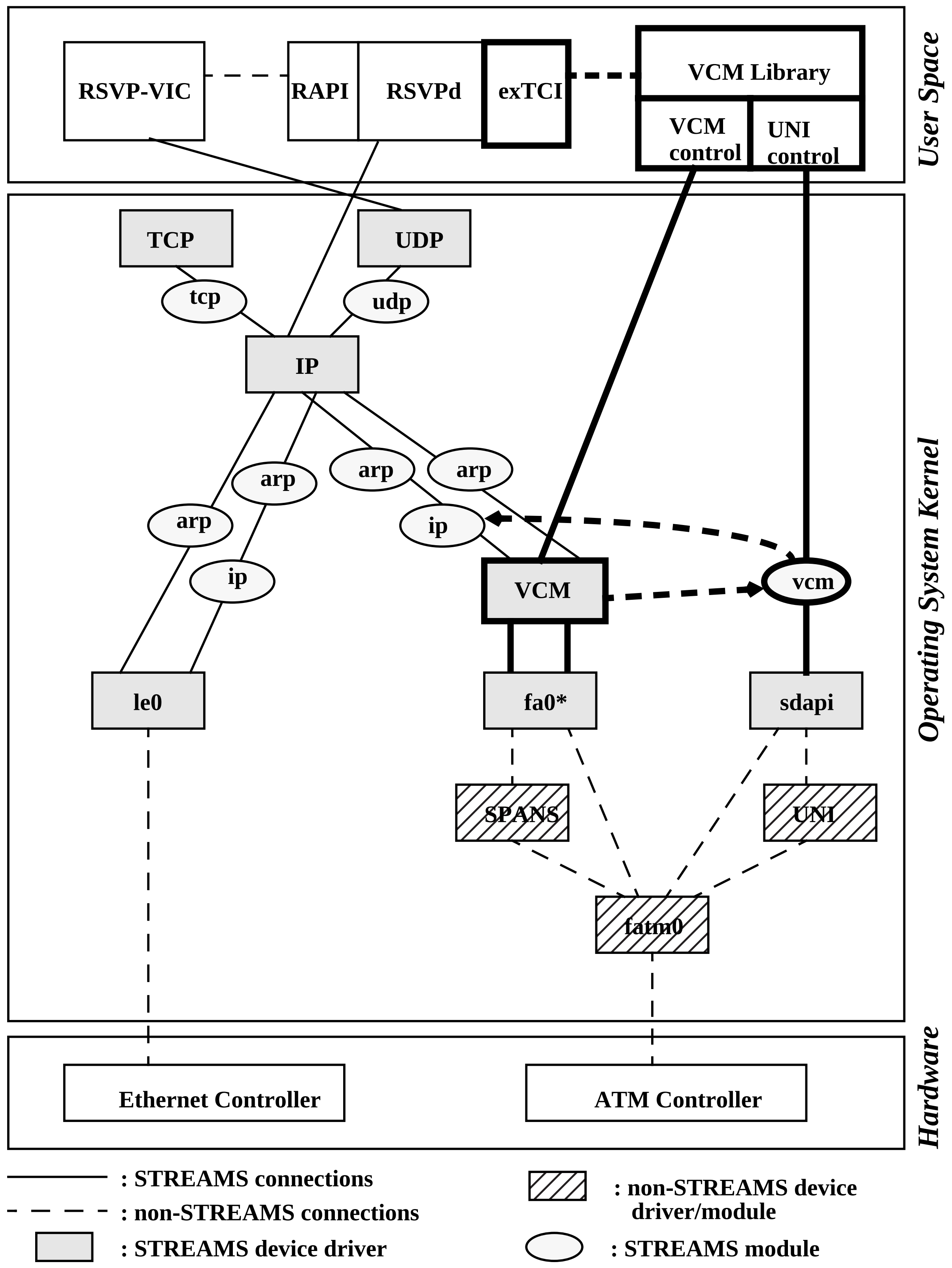
Let us now take a look on the ATM-side of our edge device. The ATM network driver is considerably more complex than the Ethernet network driver and is itself constructed of several modules that interwork with each other. Some of them, but not all, are depicted in Figure 1. The lowest, respectively closest to the hardware controller is a character device driver named fatm0 which interfaces directly to the hardware of the ATM network adaptor and makes available some very basic functionality like sending and receiving data, reserving local VC resources, etc. Eventually, all the other functionality as e.g. signalling is based on those functions in the fatm0 driver. The Fore code encompasses two different signalling modules (implemented as character device drivers):

- Fore’s proprietary SPANS signalling protocol, and
- the standard-conform UNI 3.0/3.1 signalling as specified by the ATM Forum.

On top of the signalling modules and the basic fatm0 module there are several modules for different kind of purposes, still implemented in kernel space. In Figure 1, two of them are depicted:

- the Fore IP device STREAMS driver (called fa0), implementing Fore’s proprietary convergence module for realizing best-effort IP transmissions over an ATM subnet by using facilities provided by the SPANS signalling,
- the SDAPI device STREAMS driver (called sdapi), implementing a signal-driven API that allows for direct access to the facilities provided by the UNI signalling.

†. We assumed the IP-side of the edge device to be connected to an Ethernet as it was actually the case in our test settings.



* : Here we could also have the classical IP or MPOA driver with our flexible design.

Figure 1: Overall view on IP/ATM adaptation module and its relationship to existing components.

All of the above were the existing components before the introduction of the components for the IP/ATM adaptation module. As already mentioned one of our most important design goals is to leave those components untouched in order to be able to distribute code of our IP/ATM adaptation module. The components realizing the functionality of our IP/ATM adaptation module are depicted in Figure 1 with bold frames. The most important is the VCM STREAMS device multiplexing driver which is located between the IP multiplexer and a convergence IP module (in the example Fore IP was taken). The task of the VCM device is to multiplex the IP data streams according to configurable parameters onto ATM VCs. The IP multiplexer essentially does not see the Fore IP driver any more but is now communicating directly to the VCM multiplexer which however provides the same interface (DLPI) as the Fore IP device driver so that the IP multiplexer does not realize it "talks" to someone else. The VCM multiplexer examines the IP datagram against a set of filters that are configured into it. The configuration of the filters is possible via an **`ioctl`** interface of the VCM multiplexer (more details in chapter 3). If any of the filter rules applies, the VCM multiplexer routes the datagram onto the respective VC which has been setup beforehand (see below on how), if none of the filters apply then the datagram is just passed on to the fa0 driver. For the "rerouting" of the data over especially setup VCs, the VCM multiplexer hands the successfully matched datagrams over to the VCM STREAMS module, which has been pushed on the SDAPI STREAMS device. In the VCM module the IP datagrams are prepared for being sent over their ATM VC by prepending an internal header required for the SDAPI driver. That is what has to be done for the ingress to an ATM network. For the egress from the ATM network, the inverse has to be done by the VCM module: stripping off the internal header and putting the IP datagram into the upward directed stream to the IP multiplexer. These actions are depicted in Figure 1 by the dotted arrows from the VCM device to the VCM module and from the VCM module to the data stream leading into the IP multiplexer.

The remaining question certainly is: who sets up the VCs and controls the filter configuration in the VCM device. This is done in user space by an instance of the VCM module that is being implemented as a library. This library uses the SDAPI provided by Fore to setup and manage VCs. These actions are recorded by the VCM module and thus it is able to construct the required internal headers for use of the especially set up VCs. The other task of controlling the VCM device by managing its filter set configurations is also done by the VCM user library. The VCM library is all a user as e.g. the RSVP daemon sees when implementing its extended Traffic Control Interface (exTCI) as described in [SKWS98]. Therefore the VCM library interface is a crucial part of the overall design of the IP/ATM adaptation module and we take closer look at it in the next section (for details see chapter 4).

2.4 Interface to the IP/ATM Adaptation Module

The interface to the IP/ATM adaptation module is implemented as a user level library that allows to set *filters* into the forwarding path from the IP-side of an edge device to the ATM-side. Here, filters consist of a number of rules which map data flows on a number ATM VCs that can each be setup with a certain, specified QoS. The user of that library only needs to supply the logic for which data streams/flows there should be special treatment by the ATM subnetwork, the VCM module takes all the necessary steps to setup corresponding VCs by using UNI signalling, rerouting the data path within the IP/ATM edge devices, and so on as described above. The logic is a simple restricted predicate logic, where the predicates are based on arbitrary conditions in the headers including and above the IP layer combined by logical ANDs, thus constituting a filter rule, while an ORed concatenation of such filter rules represents a filter which is mapped on a set of VCs, where the sets of the VC endpoints is disjunct. In a more formal way filters can be described as:

Let $A_{i,j}(p)$, $i=1,...,n$, $j=1,...,k$, be predicates defined on the contents of the IP packet p,

e.g. $A_{i,j}(p) = \begin{cases} 1 & \textit{if IP dest-addr} = a.b.c.d \\ 0 & \textit{otherwise} \end{cases}$

then $F_j = A_1(p) \wedge \ldots \wedge A_n(p)$ constitutes a filter rule for *j=1,..,k*,
and $F = (F_1 \vee \ldots \vee F_k; VC_1, \ldots, VC_v)$ constitutes a filter (with *endpoints(VC_i)* $\cap$ *endpoints(VC_j)* = {} for all *i,j*).

Since flexibility was the most important design goal for the interface towards the VCM, different kinds of matching actual packet header subfields against filters where introduced, i.e. predicate definitions are very general. For example it is possible to do mask matches which is particularly suited to address fields that are structured as e.g. IP's source and destination address fields, thus allowing for filter rules to be defined on whole IP subnets (e.g. "all traffic from subnet a.b.c shall take extra VC v when being forwarded to subnet d.e.f").

2.5 Functional Restrictions of the Current Implementation

In this subsection we want to outline restrictions of the current implementation, which should however not be critical for the use of the VCM module as a prototype and which will be overcome in the next releases of the IP/ATM adaptation module in the next phase of the IQATM project. These restrictions are:

- Ease of implementation was often given preference over efficiency, as e.g. in the case of the simple list of the filters inside the kernel module against which any incoming packets has to be matched. Here certainly is much room for improvement by using a more sophisticated, tuned data structures which allows for faster matching against the filter rules.
- The adaptation module is certainly very system-specific, however code at this level probably always is. Nevertheless, we made a serious try to isolate system-specific code from general code so that the porting task is simplified.
- At the moment only one user of the IP/ATM adaptation module per ATM network interface is possible at a time. However introducing multi-user capabilities into the library should be straightforward:
  - In the kernel module some additional identification for different users needs to be added in order to restrict users administrative operations on filters onto their set of filters.
  - In the user level instance/library it must be ensured that some actions, e.g. the set up of the plumbing are only executed once for all users and not once for each user.
- Furthermore, there maybe problems with filters that can apply at the same time, here a priority-based mechanism (set by policies: e.g. IntServ-related filters may be of higher priority than DiffServ filters) should be devised. Again this should be very easy to achieve by simply adding a priority field to the filter structure in the kernel and just propagating the configuration of that additional characteristic of a filter through the user library to the actual users of the adaptation module, which are the ones that have the knowledge to set such policies/priorities.

2.6 System Requirements

Having mentioned that the code is system-specific, we certainly have to specify which are the system requirements when actually running the code as it is provided:

- The edge devices must be running under the SUN Solaris 2.5.1/2.6 operating system (earlier version of Solaris should be no problem, but were not tested).
- The edge devices must use Fore network adapters with ForeThought's Software Release 5.0 (at the time of writing it is unfortunately not yet absolutely clear whether a source code license from Fore is required in order to access the SDAPI library interface).
- The previous point means we are using ATM's UNI 3.0/3.1 to signal VCs, thus a switch must be able to understand this signalling.

- During our development and testing we only used a Fore switch (Fore LE 155). We are not sure whether signalling would work if edge devices are connected to switches of other manufacturers, which in theory they certainly should. Therefore we recommend a test scenario where edge devices are connected to Fore switches while inside the ATM network other ATM switches could be used as well.

3 The VCM Kernel Instance

3.1 Overview

In this section the kernel part of the VCM module is presented in more detail. As already pointed out in the last section, one of the main goals for the design of the VCM module is to keep the VCM kernel instance as minimal and lean as possible and to extend its services by the user level module (layering principle). The reasons for that are:

- ease of development,
- comfort of programming,
- maximum use of existing facilities of the ATM network driver code (a user-level API allowing direct access to the UNI signalling is available).

In order to explain in more detail the inner working of the VCM kernel instance, we first give a short review of UNIX device drivers in general and the STREAMS mechanism found in System V Unix systems, as e.g. SUN's Solaris, in order to implement (among other things) the communication subsystem. Furthermore, we present a high level overview of the architecture of Fore's ATM network driver code in order to be able to show how the VCM kernel instance interfaces to that code, respectively how it makes use of the facilities provided by the driver.

Although the design goals for the IP/ATM adaptation module have already been stated and reasoned about in the discussion on the overall architecture of the VCM module, we again look at these in a more concrete and detailed fashion for the VCM kernel instance. Next, the architecture of the VCM kernel instance is being regarded with an emphasis on how the VCM kernel instance is related to existing modules which are either part of the operating system or part of the ATM network driver. In the rest of the section we then delve into the details of the implementation of the VCM kernel instance.

3.2 Review of UNIX Device Drivers and the STREAMS Mechanism

In this subsection we review the fundamental concepts of System V Unix device drivers in general and the STREAMS framework in order to ease understanding of the description of our implementation of the IP/ATM adaptation module. Of course, this is only a very shallow overview and without any pre-knowledge it might be difficult to follow later on, so for a more complete presentation of these general issues we refer to ([Sun96], [Sun95].

3.2.1 UNIX Device Drivers

A device driver consists of a set of routines that allows the kernel and user programs to communicate with peripheral devices. The purpose of a device driver is to hide the complex device-specific details from the user and the rest of the operating system. It allows the user to use regular file system calls to access a device by translating them into device specific commands. Device drivers are part of the kernel. There are two basic categories:

- block device drivers, and
- character device drivers.

Block device drivers are used for peripherals which must handle file systems, like hard disks. The read and write operations use fixed size data blocks. Character device drivers do not require fixed sized blocks for the read and write operation. Therefore, almost every device can be accessed via a character device driver. Even classical block devices can be handled by a special "raw" character device driver.

Character device drivers are typically used for asynchronous terminals (serial drivers), mice and network adapter cards. A special type of character device drivers are pseudo device drivers. They offer an entry point to the kernel, but do not really communicate with a device. The character device "/dev/kmem" for example is a pseudo driver that provides the possibility to read directly from the kernel memory. The kernel expects certain routines in a device driver. These driver entry points must follow the special naming conventions below. The following routines form the interface between the device driver and the rest of the kernel:

- Initialization (xx_init) [‡],
- Open and close (xx_open, xx_close),
- Read and write (xx_read, xx_write),
- Special input/output (xx_ioctl),
- Interrupt handler (xx_intr),
- Poll (xx_poll),
- Select (xx_select),
- Strategy (xx_strategy).

The initialization routine is called at boot time. This function checks if the device really exists in the system. It resets flags and counters and allocates the required resources. The open routine prepares the device for the input and output operations. The close function is called to deactivate the device. The read and write functions perform the data transfer to and from the device. They are invoked when a user process calls a read or write system call.

The input/output control routine offers some special functions for character devices. It is called when the user process issues an ioctl system call. This routine is often used to get information about the current status of the device.

The interrupt handler is called when the device sends an interrupt. The purpose of an interrupt is to indicate that the device requires the attention of the kernel. A device interrupts for example if it has completed an operation or if new data has arrived. If a device is not able to generate interrupts, the kernel can periodically call a poll routine in order to service the device. The poll routine is called at each clock tick. It is helpful for handling slow devices. The select routine allows the device driver to perform synchronous multiplexing. It checks if the device is ready for a read or write access. Block devices require a strategy routine to sort the read and write requests into a queue. Character device drivers do not require a strategy routine.

Not all functions must be present for each driver. For example it would not make sense to provide a write routine for a mouse or a read routine for a printer driver. Furthermore, the set of functions depends on the type of the device. The driver routines are divided into two groups according to the process context. The read, write and the ioctl routine form the top half of the driver. Strategy and interrupt handler routines can be called in interrupt context and belong to the bottom half of the driver.

3.2.2 STREAMS Framework and Mechanisms

The STREAMS mechanism was developed in 1983 by Dennis Ritchie. The first operating system that included STREAMS was UNIX System V Release 3.0. The STREAMS concept provides a convenient mechanism for the design of layered protocol stacks. It allows a clean separation between device independent and device specific code.

A stream is a full-duplex communication path between a user process and a device. It consists of a STREAMS head, a STREAMS driver and one or more connected modules, which can be pushed onto

‡. xx stands for the prefix of the device driver.

the stream dynamically by user programs (see Figure 2). The modular layered structure makes STREAMS a valuable tool especially for the development of network protocols.

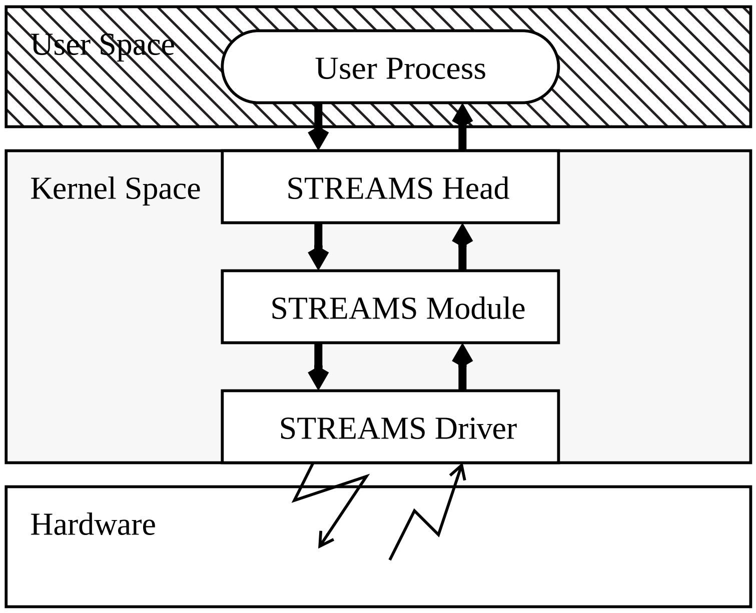


Figure 2: STREAMS Framework.

The communication between the modules and therefore between the user process and the driver is done by passing messages up or down the stream. Each message contains a message type to indicate its intended purpose. For example the message type M_DATA is usually used for messages that contain ordinary data. Messages of the type M_PROTO usually contain control information and associated data.

Special put routines are used to send STREAMS messages. A ‘write put’ (xx_wput) function is called to pass a message to the next module below (downstream) and a ‘read put’ (xx_rput) routine is used to send messages to the next module above (upstream).

STREAMS modules are used to manipulate the messages that are passed through the stream. They can be pushed onto or popped from the stream dynamically. Each STREAMS module consists of a pair of queues, one for the read side (rqueue) and one for the write side (wqueue). A queue consists of a message queue, which contains the messages that wait for service and a put routine. It may have a service routine to allow deferred message processing.

STREAMS head and STREAMS driver are special mandatory modules for each stream. The STREAMS head is the interface between a user process and the stream in kernel space. It is allocated and initialized when the stream is opened for the first time. The STREAMS head converts the user requests into STREAMS messages and it provides the STREAMS messages which were sent upstream in a user readable form to the process in user space.

The STREAMS driver provides the interface between the kernel and the device. Like regular character device drivers it hides the complexities of the underlying hardware from the kernel and the user processes. It converts the STREAMS messages into data structures that the device understands. Like the STREAMS head, the STREAMS driver is initialized via the open system call.

A special feature of the STREAMS mechanism is the possibility to multiplex streams. This is done by special modules called STREAMS multiplexers (Figure 3 shows a simplified implementation of the TCP/IP stack using the STREAMS framework). A multiplexer which multiplexes data from several upper streams to one single lower stream is called "N-to-1" or "upper" multiplexor. A multiplexer which has only one upper stream but several lower streams is called "1-to-M" or "lower" multiplexer. This property provides a useful service especially for the implementation of internetworking protocols which might route data over different network interfaces. For example one IP module can multiplex the incoming data to several network drivers. The appropriate driver module can be chosen in accordance to the forwarding decision.

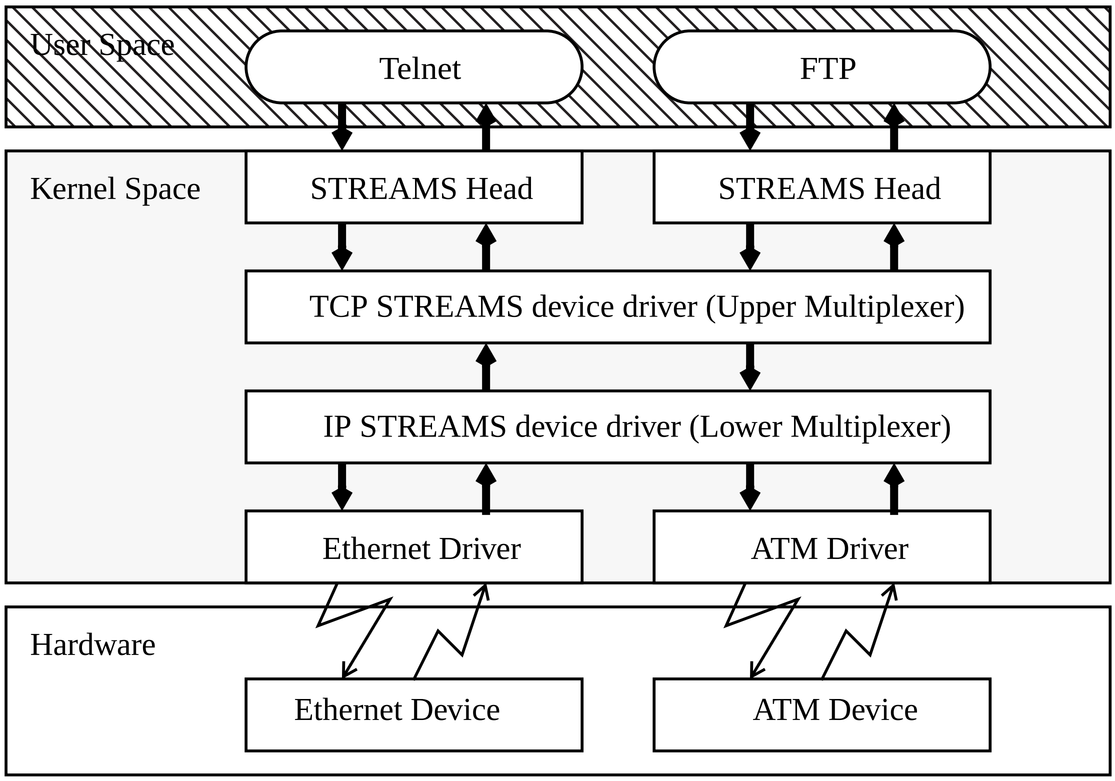


Figure 3: TCP/IP protocol stack STREAMS implementation

Of course STREAMS does not specify the "protocol" which neighbouring drivers/modules use to communicate with each other. This is opaque to the STREAMS mechanism and makes it such a generally utilizable facility. For our purposes, the most interesting communication is between the IP multiplexer and the device-specific drivers. Here, the entry point for IP traffic into the device driver depends again on the operating system. In a STREAMS environment it would be a Data Link Provider Interface (DLPI) [OSI91] and in a BSD based Unix system a BSD socket interface (the ifnet interface).

3.3 Architecture of the Fore ATM Network Driver

In order to understand how the VCM kernel instance fits into the existing ATM network driver code and how it makes use of that code, we give a brief overview on the architecture of the Fore ATM network driver. A view on the modular structure of that architecture based on functional groups is illustrated in Figure 4.

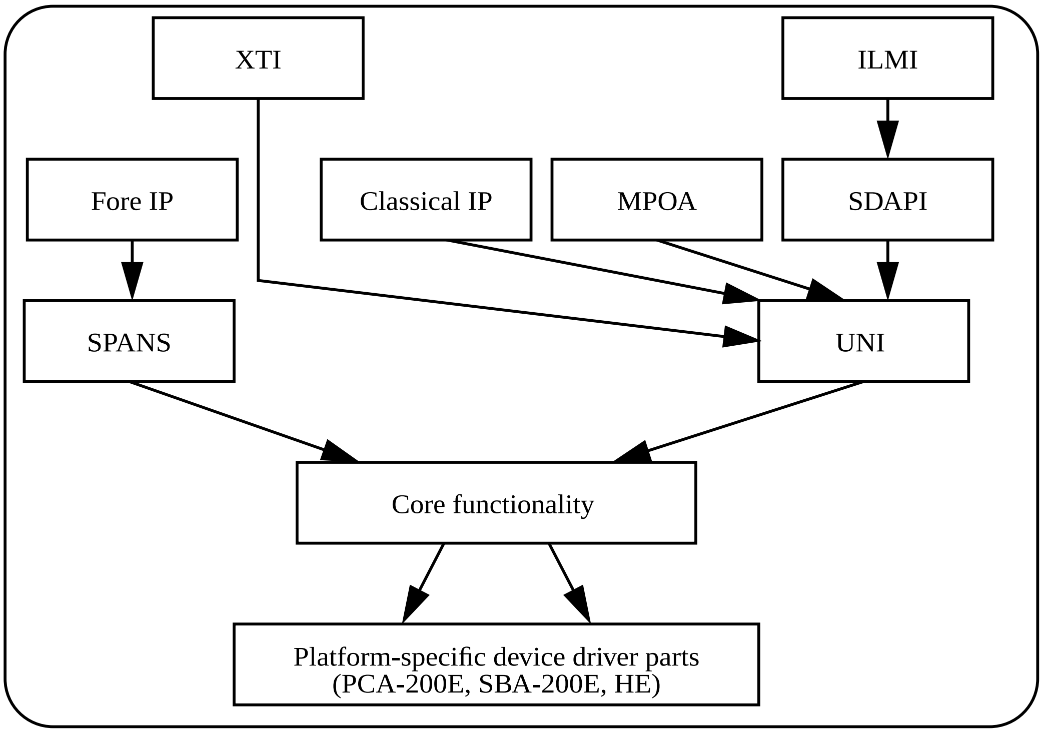


Figure 4: Modular Structure of Fore’s ATM network driver.

Here, we can see that device-specific, low-level code is well shielded from the code of convergence modules like MPOA, CLIP and ForeIP by using an abstract core module that provides a device-independent interface to functionality as e.g. reserving local resources when opening a VC or sending data onto a VC. While the arrows indicate which modules are using which other modules, not all the interdependencies have been drawn, but only those related to control path issues, while almost all modules use on their data path the services provided by the core functionality module.

The Fore ATM network driver supports two different signalling protocols:

- SPANS (Simple Protocol for ATM Network Signalling): Fore’s proprietary signalling protocol, and
- UNI 3.0/3.1 Signalling: the ATM Forum’s standard signalling protocol at the User-Network Interface.

Those two both use the functions provided by the core module in order to setup VCs, indicate willingness for reception of incoming VCs, etc. The only convergence module that is still using SPANS is the Fore IP module, a proprietary solution from Fore to realize best-effort IP over ATM for uni- as well as multicast. All the other convergence modules use the standard-conform UNI signalling module. Those are:

- the Classical IP over ATM module that implements unicast best-effort IP over ATM according to the IETF specifications RFC 1577 and RFC 1483.
- the ATM Forum’s MPOA solution which builds upon ATM Forum’s LAN Emulation standard and the IETF NHRP specification.
- the XTI (X/Open Transport Interface) which represents a standard API for native ATM programming in the tradition of System V’s TLI (Transport Layer Interface).

- the SDAPI which is the kernel-level component for a user-level library that allows to access directly the UNI 3.0/3.1 signalling services.

Another module is ILMI (Interim Local Managment Interface), which helps in autoconfiguring the ATM network by e.g. soliciting ATM addresses of a certain network interface, etc. The ILMI module is implemented on top of the SDAPI module, i.e. uses its interface to the UNI signalling to accomplish its task.

Internally, the ATM driver modules are not implemented as STREAMS modules/drivers. However all the modules that interface to upper layers of the protocol stack (CLIP, Fore IP, MPOA) or even directly to the user (XTI, SDAPI) provide a STREAMS interface, i.e. are ready to receive STREAMS messages. In the case of the IP convergence modules (CLIP, Fore IP, MPOA) the STREAMS message passing is based on the DLPI.

3.4 Design Goals and Decisions

Apart from the design goals for the overall architecture which also apply to the VCM kernel instance, there are also more specific design goals for the VCM kernel instance:

- The **functionality** provided by the VCM kernel instance should be kept **minimal**, but **complete** ("Keep it lean and clean"). The goal was to design **atomic functions** which can be composed to an enhanced higher level service provided by the VCM user instance. The rationale for this is the higher effort required for development and coding in the kernel space when compared to user space implementations.
- Despite minimality the VCM kernel instance should offer as much **flexibility** as possible, especially with regard to the **specification** of **rules** that specify which packets belong to a flow for which special VCs are available (virtually any information contained in IP and upper layer headers should be possible to qualify for such special treatment by the ATM network). In particular different kinds of granularity should be possible, e.g. traffic from certain subnets (identified by CIDR prefixes) should be a possible criterion as well as application subflows that are qualified by e.g. (transport protocol, source address, destination address, source port, destination port, and/or even RTP header fields).
- The STREAMS-related operations should be separated as far as possible from the operations which are needed to accomplish the required functionality of routing the IP packets according to configurable criteria on different especially setup VCs. Hence, the structure of the VCM user instance should be **modular**.

A design decision we made was to implement the VCM kernel instance in C. This was motivated by the fact that the STREAMS framework and the kernel entry points are to be specified in C anyway and that it would make only limited sense to have a hybrid design by introducing another language, as e.g. C++. Furthermore, performance is an argument for using C.

3.5 Architecture of the VCM Kernel Instance

As illustrated in Figure 1, the VCM kernel instance's functionality is distributed over two distinct kernel

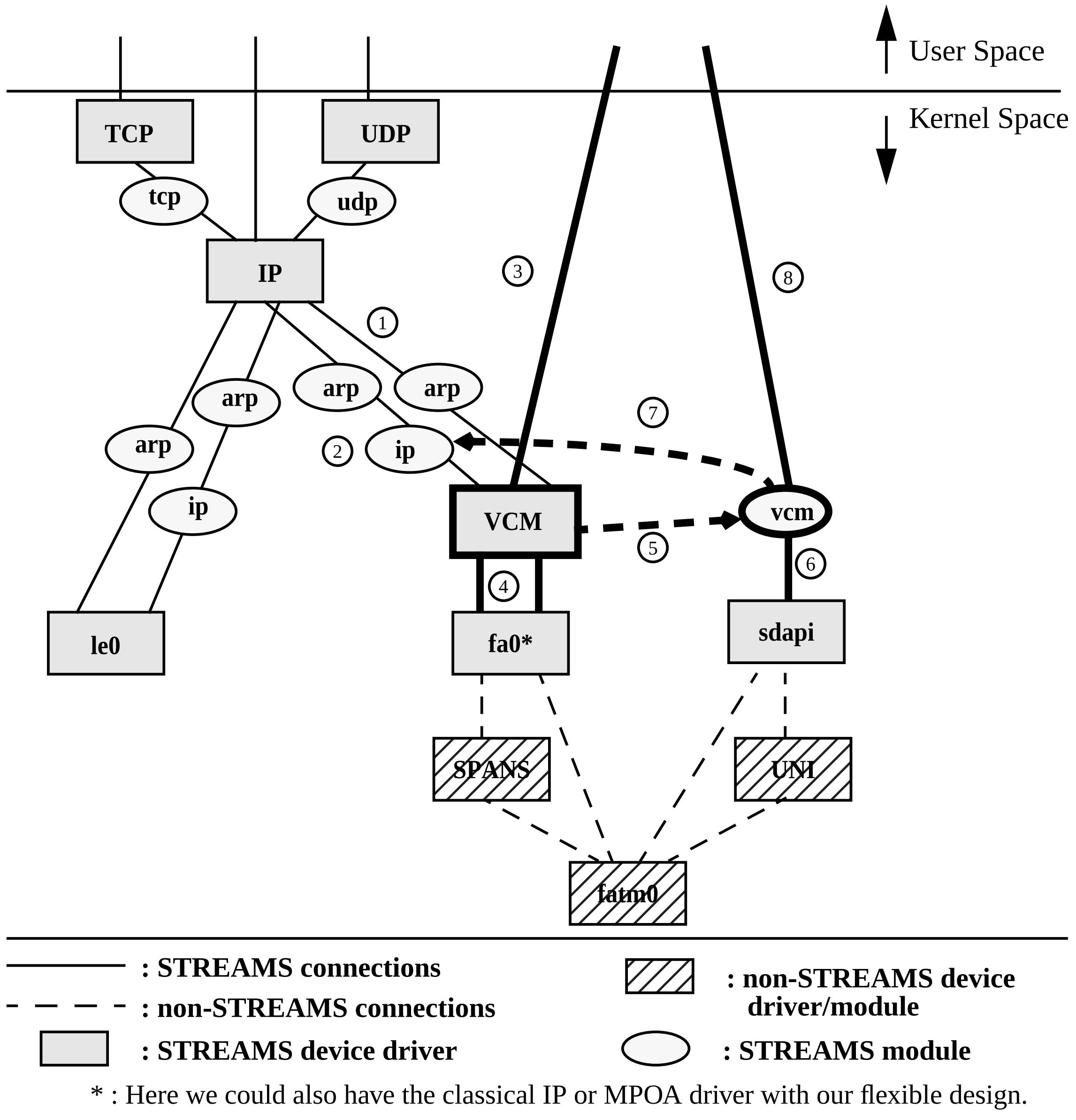


Figure 5: View on the VCM kernel instance and its relationship to existing components.

modules (bold frames respectively lines represent new components respectively relations):

- **VCM STREAMS multiplexing device driver**: this component is located just beneath the IP STREAMS multiplexer, linked to it by two streams, one for the ARP requests (1) and one for the actual flow of IP datagrams to the ATM network interface (2). Since it is to be configured from user space (3), it needs to be realized as a driver, since otherwise ioctl commands could not pass the IP multiplexer unless we do not want to change the Solaris source code of the IP STREAMS multiplexer. Furthermore, it needs to be implemented as a STREAMS multiplexer since it multiplexes messages from the IP multiplexer onto the utilized IP convergence module (in Figure 1 this is the Fore IP module, here called fa0) corresponding on whether they arrived through the IP datagram stream or the ARP message stream (4). For best-effort IP datagrams which do not correspond to any of the rules being configured, the VCM device will just pass on the IP datagram to the utilized IP convergence module. Similarly, for all the configuration when the plumbing between the IP STREAMS device and the ATM network interface is set up, the VCM device utilizes the services provided by the IP convergence module and again just passes the related STREAMS messages exchanged between the IP device driver and the IP convergence module driver back and forth. However, if an IP datagram passes by that satisfies any of the rules configured into the VCM device, that IP datagram will be taken out of the default best-effort data path and be handed over to the other component of the VCM kernel instance: the VCM STREAMS module. Note that by "squeezing" the VCM device between the IP multiplexer and the IP convergence module there are no modifications necessary, neither for the Solaris operating source code nor for the ATM network driver code.

- **VCM STREAMS module**: this component "sits" on top of the SDAPI device driver on the stream from the VCM user instance, which manages the VCs by using the SDAPI library, to the SDAPI device. The main task of the VCM module is to receive IP datagrams from the VCM device (5) and passing them on to the SDAPI device (6), thus representing an entry point into the queues of the SDAPI device for the different VCs. In the reverse direction, for data arriving on those especially setup VCs the VCM module again acts as an entry point of the received IP datagrams back into the IP device driver's queues (7). Furthermore, the VCM module listens to all SETUP requests from the VCM user instance (8) and copies the negotiated headers for sending on those VCs into internal buffers that can later on be associated to filter rules thus allowing IP datagrams satisfying the filter rule to be actually sent on that VC.

It is crucial to understand how these two component work together, so let it be emphasized once more: the VCM STREAMS multiplexing driver examines the flow of IP datagrams passed from the IP multiplexer to the ATM network interface represented by any of the possible IP convergence modules. If it detects an IP datagram that matches any of the filter rules in its filter set, this datagram is taken out of the default path to the respective IP convergence module and handed over to the VCM STREAMS module which passes it on to the SDAPI STREAMS device driver. To enable the SDAPI device to send the datagram on the associated VC of the matching filter rule, an internal header indicating the VC descriptor and other information needs to be prepended. This header is recorded by the VCM module when the VC is set up by the VCM user instance, which is possible since the VCM module is pushed onto the stream between the VCM user instance and the SDAPI device driver.

The configuration of the filter set is being done by the VCM user instance via a controlling stream to the VCM device driver (3) using ioctl system calls with various VCM-specific commands corresponding to actions as e.g. introduction of a new filter, deletion of an existing filter, etc. Whenever a new filter is set up by issuing the corresponding ioctl, the SDAPI-specific headers being recorded by the VCM module for the last SETUP requests are associated with that filter (note that those can potentially be multiple).

3.6 Modules of the VCM Kernel Instance

After having introduced the fundamental global architecture of the VCM kernel instance let us now take a more local and detailed view on the internal modules that implement the VCM kernel instance.

3.6.1 Solaris Device Driver Specifics Module: `vcm_ddi.c`

This module implements all the necessary functions for a Solaris device driver, respectively in general a Solaris loadable kernel module. This code is partially shared between the VCM STREAMS driver and the VCM STREAMS module and partially specific for each of those two. It is very system-specific code that is why it was isolated in a separate module.

The shared functionality for the VCM STREAMS device and module is with regard to the kernel entry points for loadable kernel modules, which is the same for both. This code is extremely Solaris-specific and certainly needs a major porting effort when desiring to port the VCM kernel instance.

The specific code for the device respectively the module is concerned with their different kernel entry points as e.g. for attaching, detaching, probing, etc. a device. All of these functions are written in conformance to the DDI (Device Driver Interface) and DKI (Device Kernel Interface) specifications for System V Unix systems, so that at least for those systems the porting of these parts should be straight-forward.

Since the Solaris operating system kernel is multi-threaded it is important to realize that the code of the VCM kernel instance needs to be MT-safe. The STREAMS framework offers some mechanisms to restrict the parallelism among the different entry points for the STREAMS mechanisms: MT perimeters (for more details see [Sun95]). We have tried to allow as much parallelism as possible, yet not allowing for race conditions inside our VCM kernel instance. Therefore we specified an inner perimeter always spanning a pair of queues and an outer perimeter with exclusive access to the open and close procedure of the VCM STREAMS driver and module.

3.6.2 VCM STREAMS Multiplexing Device Driver: `vcm_dev.c`

This module contains all the VCM STREAMS device specific functionality. This is composed of the various STREAMS entry points typical for a STREAMS multiplexing device driver and some additional other functions mainly dealing with configuration of the VCM STREAMS device and handing over the datagrams to the VCM STREAMS module.

3.6.2.1 Kernel Entry Points

The first entry point that is called when the VCM STREAMS device is opened (by using the normal system call open()) is

```
int vcm_dev_open(queue_t *q, dev_t *devp, int flag, int sflag,
                 cred_t *credp)
```

which does all the necessary initialization tasks. What this routine does depends upon how often it has been called already. When it is called the first time it assumes that the control queue from the VCM user instance is to be opened and makes the necessary initialization for that queue. For further open()'s it detects that the control queue is already opened and opens as next queues the IP and ARP queue, in that order (which is the order in which the TCP/IP stack is built up (plumbed) in the Solaris operating system, which however could certainly be different for other operating systems or even other releases of Solaris). Those two queues are initialized as well and further open()'s are rejected by the VCM device indicating that it is already busy, i.e. can service only one user instance at the same time.

To ensure that the order of the open()'s is correct is not part of the VCM STREAMS device but is the task of the VCM user instance using the VCM device. When vcm_dev_open() has been called for all three queues then the upward STREAMS connections in Figure 1 have been set and the VCM device is now able to monitor the IP data stream from the IP multiplexer to the ATM network interface, receive ioctl commands from the VCM user instance and pass selected datagrams to the VCM STREAMS module. Again it is under the responsibility of the VCM user instance to ensure that the plumbing in the downward direction from the VCM device to the utilized IP convergence is set correct.

The logically inverse entry point is

```
int vcm_dev_close(queue_t*, int, cred_t*)
```

which just closes the queue that it was called for and releases some internal data structures that were associated with that queue. It is verified that the queue is one of control, ARP or IP queue, otherwise the close is rejected.

Let us now come to the entry points that are really working on the data streams being exchanged between the IP multiplexer, the VCM device and the IP convergence module respectively the VCM STREAMS module. The first one is

`int vcm_dev_uwput(queue_t *q, mblk_t *mp)`

This function is always called when there are messages to be delivered downwards from either the IP multiplexer or the VCM user instance. Let us start with the case of the control queue (from the VCM user instance to the VCM device). Here the main task is to receive different ioctl commands that can be issued by the VCM user instance. Those ioctls can be:

- **`I_LINK`**: This will, when received the first time, link the VCM STREAMS multiplexer to the lower IP queue leading to the IP convergence module. When being called with the lower IP queue already being setup it is assumed that now the lower ARP queue again leading to the IP convergence module is to be linked to the VCM device. Again, ensurance of the correct order is under the responsibility of the VCM user instance.
- **`I_UNLINK`**: This will unlink the lower queues from the VCM device to the IP convergence module again. The order of unlinking is the reverse order of the linking. Further unlinks will be rejected.
- **`VCM_NEWFILTER`**
  `VCM_ADDVC2FILTER`
  `VCM_CHANGEFILTER`
  `VCM_ADDFILTER`
  `VCM_CHANGEVC4FILTER`
  `VCM_DELFILTER`
  `VCM_DELETEVCFROMFILTER`,
  `VCM_EXISTFILTER`
  `VCM_LISTFILTER`
  `VCM_FLUSH`

  These are ioctl commands that trigger functions that deal with the configuration and management of the filter set maintained by the VCM STREAMS device. However that is delegated to the function **`vcm_dev_user_ioctl()`** described below.

Success or failure of those ioctl commands is then passed back upstream using STREAMS mechanisms. It has to be noted that for the control queue, messages are never passed on to the lower queues instead they are always terminated in the VCM device.

For the IP queue and the ARP queue, if a message from upstream is of a message type that possibly contains data (M_DATA, M_PROTO or M_PCPROTO), it is examined more closely. If it actually contains an IP datagram, then the actual data forwarding is delegated to the function **`vcm_dev_dataforwarding()`** which is described below.

Of course, the VCM device does the necessary flush handling for multiplexing drivers if an M_FLUSH message is received. Any other messages for the IP and ARP queues are just being passed on to the respective lower queues.

Unless they are not high priority messages all actions described above on passing messages on to the lower queues are first queued in the queue serviced by the VCM device. The entry point that services this queue is

`int vcm_dev_uwsrv(queue_t *q)`

This function just checks the upper queue on which an enqueued message arrived and then sends on the message to the respective lower queue subject to the flow control mechanism of the STREAMS framework.

The entry point for the service routine on the lower queue of the VCM STREAMS multiplexer is

`int vcm_dev_lwsrv(queue_t *q)`

This function is needed in order to ensure that, after the STREAMS flow control mechanism has back-enabled a previously congested lower queue to the ATM network driver, the upper queue is back-enabled again. A lower put routine however is not necessary.

Those were all the entry points on the write side, let us now turn to the read side of the VCM device. The first routine that is invoked for messages coming from downstream (i.e. from the IP convergence module) is

`int vcm_dev_lrput(queue_t *q, mblk_t *mp)`

The main task of that function is to pass on the messages received from the IP convergence module to the corresponding upper queues, i.e. either to the IP or ARP queue. Another task of the lower read put routine is to record the replies of the IP convergence module to DLPI requests from the IP multiplexer concerning the length of the link layer header, which serve for optimizing the data path in the kernel implementation. That knowledge is required for our purposes in order to be able to recognize all IP datagrams and to be able to locate them in STREAMS messages. Furthermore, processing of M_FLUSH messages is provided.

Again as in the downstream case, messages are first queued in the VCM device unless they are not high priority messages. The kernel entry point that services these enqueued messages is

`int vcm_dev_lrsrv(queue_t *q)`

This function checks for the lower queue on which the message was received and directs it to the corresponding upper queue subject to the flow control mechanism enforced by the STREAMS framework.

The kernel entry point

`int vcm_dev_ursrv(queue_t *q)`

is again just needed in order to ensure that after a previously congested IP multiplexer is back-enabled again that this action is propagated to the lower queues as well.

3.6.2.2 Other Functions

Let us now turn to the other functions that are not kernel entry points, but which are however triggered by the kernel entry point functions described above. The first of those functions is

`int vcm_dev_user_ioctl(mblk_t *mp, struct iocblk *iocp)`

Its task is mainly to direct the different ioctl commands to the functions actually handling those requests (located in the module vcm_filter.c). Furthermore, the correct format and number of parameters of those requests is checked and in case they are incorrect a corresponding error is triggered to be delivered upstream to the VCM user instance. Another task accomplished in this function is that the buffer of headers recorded by the VCM STREAMS module is marked as already used if the respective commands requires to do so (that is the case for the VCM_NEWFILTER, VCM_ADDVC2FILTER, VCM_CHANGEVC4FILTER commands).

The function

`int vcm_dev_dataforwarding(queue_t *q, mblk_t* mp)`

is concerned with messages possibly containing IP datagrams. Those messages are of type M_DATA, M_PROTO or M_PCPROTO. For M_DATA messages it immediately follows that they contain data, while for the other two it must first be checked whether they really contain an IP datagram by calling the function **`check_for_data()`**. If an IP datagram is actually contained in the message being received than that datagram is being compared against the filter set in order to find out whether there is an especially setup VC for delivery of that datagram. This filtering is done by calling the function **`fil-`**

`ter()`, which is part of the module vcm_filter.c. If that function signals success than the message is routed to the SDAPI device via the VCM STREAMS module using the function

```
int route2SDAPI(vcm_filter_t* f, mblk_t* mp)
```

This function prepends the stored headers for the identified filter to the message that matches the filter's rule. In case there are multiple headers, i.e. multiple VCs on which the IP datagram has to be sent, then the replication of the IP datagram takes place here. The messages constructed from this are then passed to the downstream queue of the VCM STREAMS module finally leading to the SDAPI device.

As already mentioned the function

```
int check_for_data(mblk_t* mp)
```

checks whether a given message of type M_PROTO or M_PCPROTO actually contains an IP datagram.

3.6.3 VCM STREAMS Module: `vcm_mod.c`

This module contains the functionality specific for the implementation of the VCM STREAMS module. Again we distinguish between the kernel entry points for the STREAMS mechanism and other functions, although the VCM module consists almost exclusively of the STREAMS-related kernel entry points.

3.6.3.1 Kernel Entry Points

The kernel entry point

```
int vcm_mod_open(queue_t *, dev_t *, int, int, cred_t *)
```

is called when the open() system call is issued for the SDAPI device (in case the VCM STREAMS module is configured to be pushed on the SDAPI device). Besides the STREAMS-specific initializations this routine also sets the pointer (called **`sdapi_q`**) to its write queue such that the VCM device is able to pass filtered IP datagrams over to the VCM module. Furthermore the buffers for recording headers of recently setup VCs, the so-called VC template buffers, are initialized.

The inverse kernel entry point is

```
int vcm_mod_close(queue_t *)
```

being called when the SDAPI device is being close()'d. The **`sdapi_q`** is invalidated and the VC template buffers are emptied.

The function

```
int vcm_mod_wput(queue_t *, mblk_t *)
```

monitors the stream from the VCM user instance (which uses the SDAPI user library) to the SDAPI device for messages of type M_PROTO which then necessarily have to be messages containing data for a certain VC being setup before via the SDAPI user library. Such messages contain exactly the SDAPI-internal format needed to send data onto VCs setup by the SDAPI device. We therefore record this header in a VC template buffer and "swallow" the respective M_PROTO message. Again, to ensure a correct operation of that mechanism we need the VCM user instance to assert that per VC being built up only one time data is being sent from the VCM user instance onto that VC. Messages from other type than M_PROTO are just passed on (to not interfere with the communication between the SDAPI user library and the SDAPI device) and are enqueued for service by the kernel entry point

```
int vcm_mod_wsrv(queue_t *)
```

which does not do much besides STREAMS-specific flush handling and enforcing the flow control mechanism provided by the STREAMS framework.

The entry point for the upstream directed put routine

```
int vcm_mod_rput(queue_t *, mblk_t *)
```

is very simple and just enqueues STREAMS messages coming from the SDAPI device directed to the VCM user instance in the VCM module's upstream queue unless they are not of the high priority type.

The such enqueued messages are then serviced by

```
int vcm_mod_rsrv(queue_t *)
```

which checks whether the messages are of type M_PROTO and have in their SDAPI-internal header a type field indicating that they are data messages. If that is the case then the respective message is stripped off its SDAPI-internal header and passed on to the IP device directly, which is possible due to the fact that the lower IP upstream read queue was recorded when the VCM device was linked under the IP multiplexer (during the **`vcm_dev_open()`** function).

3.6.3.2 Other Functions

The auxiliary function

```
void flush_VC_template_buffer()
```

purges the buffers containing the recently recorded headers for data that is to be sent to the SDAPI device.

3.6.4 Filter Configuration and Management Module: `vcm_filter.c`

This module contains all the functionality related to filter-specific operations, which mainly deals with configuration and management of the filters, i.e. their rules and associated VCs. The VCM kernel instance maintains a set of all active filters. This set is currently implemented as a simple linked list of the following data structure

```
typedef struct vf {
  predicate_list_t rule; /* Conjunction of packet predicates */
  mblk_t** VC_template;
  int no_VC_templates;
  struct vf* next;
} vcm_filter_t;
```

Obviously, this is not efficient if the filter set becomes large. However, by separating the filter-specific operations from the rest of the code, it will be easy to optimize that data structure for fast matching against IP datagrams passing by using the latest research results on packet classification under multiple criteria [SVSW98].

3.6.4.1 Filter Configuration and Management functions

The functions described here correspond directly to the ioctl commands that are recognized by the upper write put routine of the VCM device. They can be further divided into actual operations on the filter set and into diagnostic functions. Let us start with the former, supposedly more important ones.

The function

```
int vcm_newfilter(vcm_filter_t*)
```

inserts a new filter into the existing (possibly empty) filter set (at the beginning of the linked list of filters). It first checks whether this is really a new filter or whether it is already contained in the filter set thus ensuring that no duplicates are stored in the filter set. Furthermore, it associates the set of SDAPI-internal headers contained in the VC template buffer with the filter rule passed down from the VCM user instance. The VCM user instance further ensures that the VC template buffer contains the desired contents for which VCs where set up by itself.

For the case that one or even several VCs shall be added to an existing filter, the function

`int vcm_addvc2filter(vcm_filter_t*)`

has been implemented. It copies the contents of the VC template buffer which has been populated by corresponding operations in the VCM user instance, to the already existing VC templates of the given filter. Hence from now on, IP datagrams satisfying the filter's rule will also be sent on the newly added VC(s).

The function

`int vcm_changefilter(vcm_filterpair_t*)`

allows to completely exchange the filter's rule with the given filter's rule without losing the association to the existing VCs on which data satisfying the filter's rule are sent. The function is given both, the existing filter and the new filter rule. The existing filter rule is needed in order to locate the filter for which the rule has to be exchanged.

The function

`int vcm_addfilter(vcm_filterpair_t*)`

adds a given filter to the existing filter set which however shares the same VC set as an already existing filter, that is also given when it is being called. This allows for a disjunction of filter rules, i.e. if any of those filter rules is satisfied than a matched IP datagram is being sent on the shared VCs of those filters.

If the VC set shall be exchanged while the filter rule is retained, then the function

`int vcm_changevc4filter(vcm_filter_t*)`

is needed. As usual it ensures again that the given filter exists and if it does it removes its existing VC set and copies instead the contents of the VC template buffer in place. Note again that it is the VCM user instance responsibility to populate the VC template buffer adequately.

The function

`int vcm_delfilter(vcm_filter_t*)`

deletes a given filter from the filter set. Of course it is first asserted that this filter is actually existing within the filter set.

If a VC is to be removed from a given filter, then the function

`int vcm_deletevcfromfilter(vcm_deletevc_t*)`

is appropriate. It is given the filter for which a VC is to be removed and the index of the VC which template shall be removed from the associated VC set of the filter. That index represents the order in which the VCs have been set up and must be tracked by the VCM user instance.

As already mentioned there also diagnostic functions that enable the VCM user instance to elicit the state of the VCM kernel instance with regard to the filter set. A simple, but useful function to find out whether a certain filter is actually existing in the VCM kernel instance's filter set is

`int vcm_existfilter(vcm_filter_t*)`

If the filter exists it returns **`FILTEREXISTS`** otherwise it return **`FILTERDOESNOTEXIST`**.

The function

`int vcm_flush()`

allows to totally purge the filter set of the VCM kernel instance and should always be called when the VCM user instance finishes for whatever reason.

A further useful function is

`int vcm_listfilter()`

It dumps the filter set onto the console (and thus also in the log files of the Solaris operating system) of the machine that is running the VCM module. An improved version of this could be to return that list of filters to the VCM user instance.

3.6.4.2 Other Functions

The above functions all corresponded directly to ioctl commands, now we describe some other functions of the vcm_filter.c module.

The first function is

`vcm_filter_t* filter(mblk_t *mp, int offset)`

which traverses the filter set and tries to match the given IP datagram (in **`mp`** at the offset **`offset`**) against one of the filters by calling the function **`filter_match()`** which is located in the module vcm_rule.c (see below). If a matching filter is found the search can be stopped since no duplicates are ever inserted into the filter set.

The auxiliary function

`vcm_filter_t* find_filter(vcm_filter_t*)`

is used in all locations where it is necessary to locate a given filter either to ensure whether it already exists or because some configurational action is to be applied to it. It uses the function **`rule_match()`** from the module vcm_rule.c (see below).

Another auxiliary function is

`void kill_filter(vcm_filter_t*)`

which frees all the heap-allocated memory for a given filter.

3.6.5 Filter Rule Matching Module: `vcm_rule.c`

This module contains all the functionality related to matching filter rules and the like. Whenever the VCM shall be extended for new predicates defined on the contents of an IP datagram, then this is the only module that needs to be touched. That is the reason why it was separated from the filter-specific module vcm_filter.c.

As already described in Section 2.4, a rule is composed of conjuncted predicates. Currently three different kinds of predicates are supported:

- address-mask match predicates, which allow to specify a value and a mask and a matching value has to be equal to the specified value at the positions defined by the mask,
- exact match predicates which require a matching value to be exactly the same as the specified value, and
- range match predicates which allow to define a matching range in which the matching value has to be located in order for the predicate to be true.

We assumed that with those three types of predicates most of the required semantics needed by users of the VCM module should be covered, however if not, it is straightforward to introduce new kinds of predicates. Everything that is needed, from the perspective of the VCM kernel instance, is the definition of an according data type and a matching function, as we have specified them for the three predicate types given above:

For the address-mask matching predicate we have the function

```
int address_match(address_predicate_t, ipaddr_t)
```

For the exact match predicate we have the function

```
int exact_match(exact_predicate_t, u_char)
```

For the range match predicate we have the function

```
int range_match(range_predicate_t, u_short)
```

Furthermore, there are functions needed that serve as interface to the filter-specific functionality contained in vcm_filter.c. One of them is the function

```
int rule_match(predicate_list_t, predicate_list_t)
```

that evaluates whether two given rules are the same or not and returns the result.

The function

```
int filter_match(vcm_filter_t *f, mblk_t *mp, int offset)
```

evaluates whether a given IP datagram (contained in **`mp`** at offset **`offset`**) can be matched against the given filter by using the functionality provided by the predicate matching functions described above.

Note that the latter two functions also need some minor modifications if a new predicate type shall be introduced respectively if filter rules shall be extended by new fields in an IP datagram even if the existing predicates are used.

4 The VCM User Instance

4.1 Overview

In this section the VCM user instance is being presented and discussed. The basic idea of the VCM user instance is to add one more abstraction level above the services provided by the VCM kernel instance. The rationale of this is to do as much as possible of the complex parts of the IP/ATM adaptation module in user space, while only time-critical parts are done in kernel space by the VCM kernel instance. Therefore, the following tasks are done by the VCM user instance:

- signalling message handling,
- enforcement of the rules necessary for the correct operation of the VCM kernel module,
- extension/refinement of the capabilities/services of the VCM kernel instance - easier setting of the filter rules,
- address resolution,
- optionally IP-ATM QoS mapping (not yet implemented).

Again we start by discussing the design goals specific to the VCM user instance in a more detailed fashion than possible when presenting the overall architecture of the IP/ATM adaptation module. Hereafter, we present the architecture of the VCM user instance with an emphasis on how a user interfaces to the VCM library. In the following, we discuss the implementation of the VCM user instance in more detail in order to allow for possible modifications and extensions. At the end of the section we conclude with an illustrative example on how to actually use the VCM library in order to set up a special VC for a certain flow of IP datagrams.

4.2 Design Goals and Decisions

Besides the overall design goals for the VCM module as a whole (as described in Section 2.2), we also have had some more specific design goals for the VCM user instance. Those are:

- The **interface** to the VCM user instance should be **flexible** and **easy** to use. It should be **extensible** for user code since not all potential uses of the VCM module can be anticipated now. Therefore we decided to design it in an **object-oriented** fashion.
- The VCM user instance should **hide** all the "knitty-gritty" **details** of the VCM kernel instance. In principle, the VCM kernel instance already provides an interface to the user space. However, it is not very convenient to use and many rules between using the VCM and the setup of VCs (which is not part of the VCM kernel instance) must be obeyed, as e.g. the fact that all actions to kernel filters are applied to the group of VCs that have been set up since the last action (ioctl) demanded from the VCM kernel instance. Therefore one of the main tasks of the VCM user instance is to **enforce** the **rules** implied by the overall design of distributing the functionality into user and kernel space and by the lean design of the VCM kernel instance. Furthermore, the VCM user instance **enriches** the services provided by the VCM kernel instance by e.g. providing the disjunctive association between filter rules applying to the same set of VCs (whereas the VCM kernel instance has no logical linkage between such filters).
- Another important goal when developing the VCM user instance must be the **decent handling of failure** conditions as e.g. the case where a switch breaks down and all the VCs are torn down. The VCM user instance must be able to signal these asynchronous events to a potential user of its services and must be able to indicate which VCs are actually affected.

With respect to the programming language we decided to use C++, since we wanted an object-oriented interface design to be accompanied with object-oriented coding. However, since the system-level interfaces to the VCM kernel instance and to the UNI services as provided by the SDAPI library are in C, the lower part of the VCM user instance is rather procedural. Therefore C++ which supports both paradigms of programming, procedural and object-oriented, was ideal for our case.

We furtherly decided to implement the VCM user instance as a library instead of, e.g. a daemon, because it is in our view a more fundamental way of offering its services, since a daemon could be implemented by using the VCM library functions.

4.3 Architecture of the VCM User Instance

In this section we present the architecture of the VCM User Instance as a whole, before going into the detailed description of the implemented classes. We start by taking a global view on the VCM user instance's functionality and its relationships to its potential users and the modules on which it is based. After that, the static model of the VCM user instance is presented, thus simplifying the understanding of the implementation description in the following sections.

4.3.1 Global View

In Figure 6, the relevant part of the overall architecture for the VCM user instance is shown. The VCM

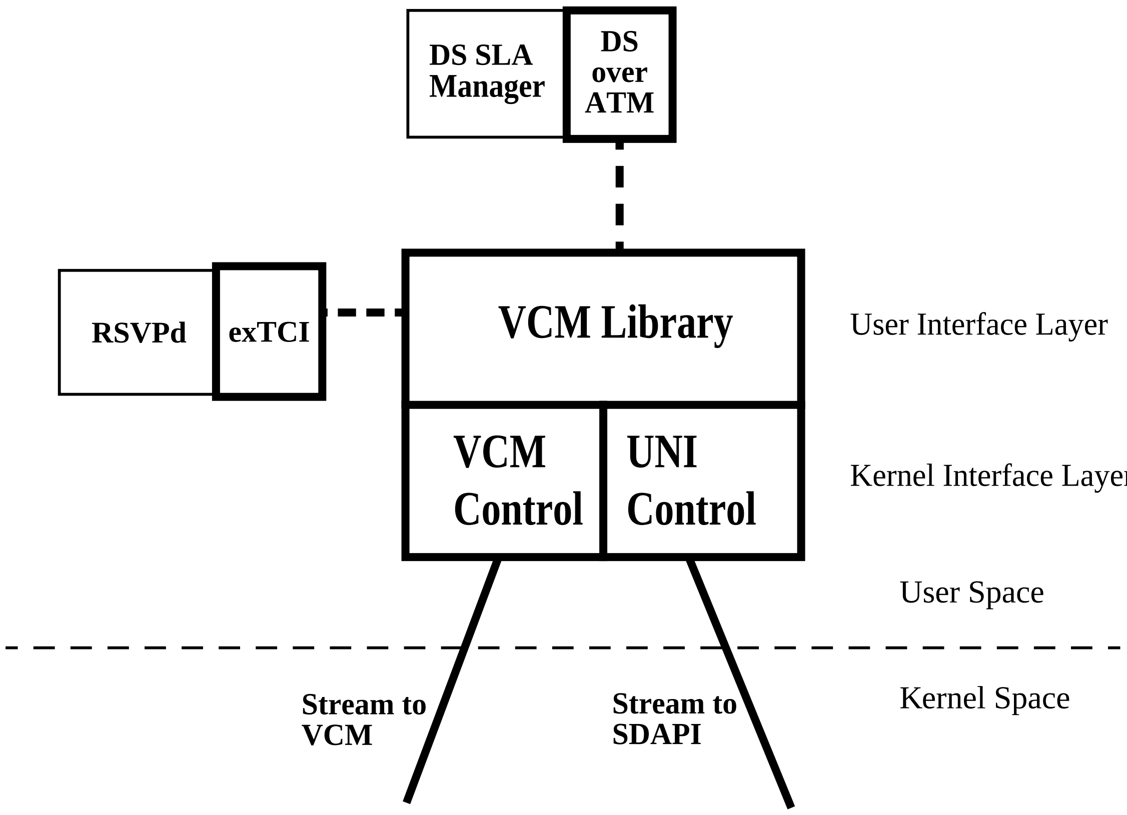


Figure 6: Global View on Architecture of VCM User Instance.

user instance can on a macroscopic level be divided into two layers:

- The *user interface layer* which is directed towards the user of the VCM services as, e.g. the RSVP daemon that "talks" via an extended Traffic Control Interface [SKWS98] to the VCM module in order to set up special VCs for RSVP-signalled IP data flows according to the IntServ specifications carried by the RSVP messages. Another example (as depicted in Figure 6) could be a DiffServ SLA (Service Level Agreements) Manager that maps the given SLAs into specific ATM VCs.

- The *kernel interface layer* on the other hand is directed towards the kernel-level modules, respectively their user-level interface for managing the data forwarding inside the kernel and the setup and tear down, etc. of the especially customized VCs. Along those two different tasks it can be divided even further into subcomponents:
  - UNI Control, which handles everything that is concerned with the UNI-signalled VCs and which therefore uses the services provided by the user-level front-end to the SDAPI device (which itself communicates to the SDAPI device via the STREAMS concept), and
  - VCM Control, which handles all the necessary actions in order to invoke the VCM kernel instance functionality of rerouting IP datagrams on specifically setup VCs depending on the contents of these datagrams. Again the communication across the user-kernel space barrier is based on the STREAMS mechanism.

Since the SDAPI library provides an upcall-based interface that requires a user to "listen" on specific file descriptors periodically, it was decided to put this "polling" into an extra thread in order to not lock a user of the VCM while doing signalling for VCs in the ATM network. That means the UNI Control subcomponent of the kernel interface layer runs as a separate thread whereas all the other functionality runs as the main thread (separating the VCM Control subcomponent is possible but not necessary since the interface to the VCM kernel instance works in a synchronous, immediate request-response fashion so that locking for substantial periods is not an issue).

4.3.2 Static Model

A quite detailed static model, according to the Coad/Yourdon notation (see [CY91]), of the VCM user instance is given in Figure 7. Since it is almost in a one-to-one correspondence to the actual implementation of the VCM user instance and shows all relevant classes it is more than just an analysis result, but rather encompasses already some design decisions.

The division between those classes that implement the kernel interface layer versus those classes that represent the user interface layer is illustrated by the bold dashed frame which encompasses all the classes that compose the kernel interface layer while (almost) all the surrounding classes belong to the user interface layer.

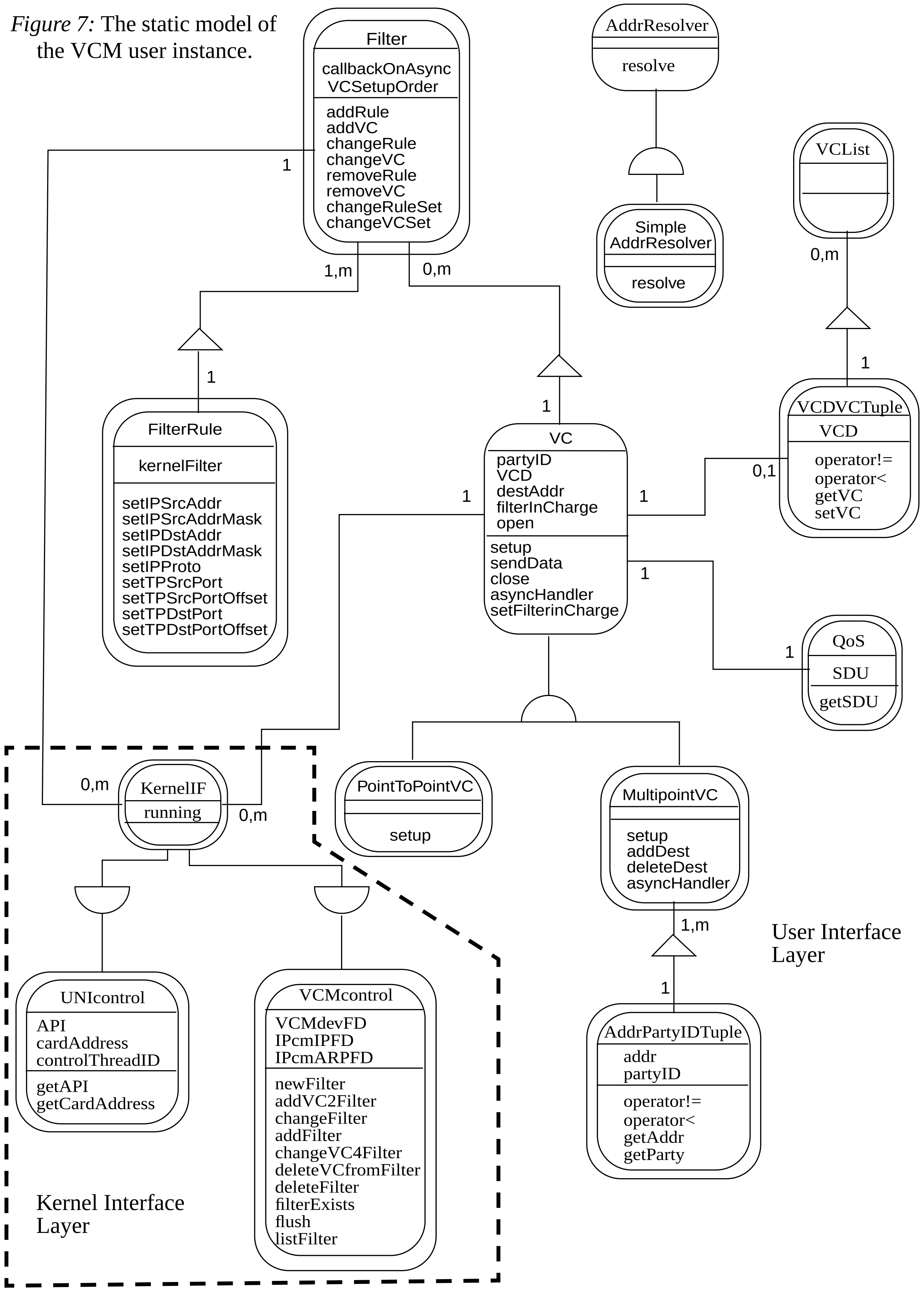
While the description of the internals of the classes is being done in the following sections, let us look here at the relations between the classes, where by the term relation we mean here: inheritance, aggregation and object relations. The class **`KernelIF`** represents the kernel interface layer's interface. It inherits its functionality from the **`VCMcontrol`** and the **`UNIcontrol`** class which have a direct correspondence to the subcomponents forming the kernel interface layer's functionality as described above. Theses classes are implemented quite monolithic in a style that is not object-oriented in nature. That is due to their proximity to system-level interface, and actually the main task of that classes lies in building a convergence "layer" towards the purely object-oriented user-interface layer from the C-style system-level commands applying to the use of the SDAPI library and the VCM kernel instance.

The user interface layer is composed of much more classes. The most important one being the **`Filter`** class. **`Filter`**'s consist of an arbitrary number (but at least one) of **`FilterRule`**'s and an arbitrary number of **`VC`**'s[**], thus storing the association between the rules on which IP datagrams are allowed to use which set of VCs. **`Filter`**'s have a relationship to the **`KernelIF`** in order to access its functionality for setting up kernel filters.

The **`VC`** class is an abstract base class for either **`PointToPointVC`** or **`MultipointVC`**. It has a one-to-one relation to the class **`QoS`** and a relation to the **`KernelIF`** in order to access its services for setting up VCs via UNI signalling. Furthermore, it has a relationship to the class **`VCDVCTuple`**, whose

**. If the number of VCs is equal to 0, then the semantic of that filter is to discard a matching IP datagram, thus accomplishing the usual functionality of a packet filter for firewalling purposes.

Figure 7: The static model of the VCM user instance.



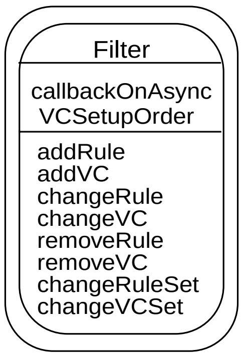
instances are kept in a container called **`VCList`** in order to track all the VCs irrespective of whether they were built up by the local VCM module or by another remote VCM module, i.e. for the first case they are outbound while for the second case they are inbound. Only the outbound VCs are instances of the class **`VC`** which explains that a **`VCDVCTuple`** object does not necessarily have a relation to a **`VC`** object. The class **`MultipointVC`** which represents a specialization of **`VC`** contains an arbitrary number (but at least one) of **`AddrPartyIDTuple`** objects in order to track the association between ATM addresses and party IDs within a point-to-multipoint VC.

Isolated from the above classes of the user interface layer are the abstract base class **`AddrResolver`** and its specialization **`SimpleAddrResolver`**, whose task it is to resolve an IP address into the corresponding ATM address. The base class **`AddrResolver`** just specifies the abstract interface how such a request is to be made, whereas **`SimpleAddrResolver`** implements a very simple (and inefficient) way of eliciting the ATM address for a given IP address. Here a better scheme is required for actual use. However by separating the interface from the implementation such an extension should be easily possible.

4.4 User Interface Layer Classes

Let us now look more closely into the classes which form the user interface layer. We will restrict ourselves here to the classes actually being accessed by user code, while auxiliary classes will be dealt with in a separate section (see section 4.6).

4.4.1 The Filter Class



The **`Filter`** class is the most important class keeping track of the association between filter rules and VCs. This is achieved by holding a sorted list on both the **`FilterRule`**'s and **`VC`**'s. Furthermore, it is being tracked in which order the VCs are set up in a separate list called **`VCSetupOrder`**. Since the **`Filter`** class is the only place where the link between filters and VCs is stored (as being described in chapter 3 the VCM kernel instance does not know this association), it is also the place where asynchronous failures with respect to VCs as e.g. link failures can be brought to the attention of all the filter rules that are affected by such events. However what exactly has to be done in case of such a failure cannot be decided within the VCM user instance but must be specified by the user of the VCM library. This is being done by specifying a handler for such situations which is being stored in the **`callbackOnAsync`** function variable. Note that whenever asynchronous failure conditions are signalled by the ATM network this handler will be called with information on what happened and which VC respectively which parties were affected, in the run-time context of the signalling management (UNI Control) thread.

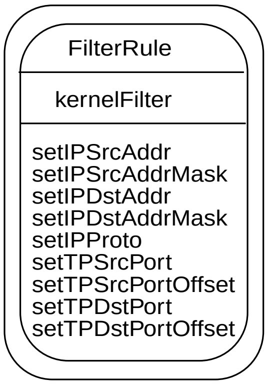
The interface of the **`Filter`** class provides all potentially useful operations on both the filter rule and the VC set (after a **`Filter`** has been constructed initially using its constructor):

- **`addRule`**: add a filter rule to the existing disjunction of filter rules which maps certain IP datagrams on the set of VCs for the existing filter.
- **`addVC`**: add a VC to the existing set of VCs (note that the VCs should have mutually exclusive sets of endpoints).
- **`changeRule`**: exchange one of the filter rules against a new filter rule but leave the existing VC set in place.
- **`changeVC`**: exchange a specified VC against a new one but leave the set of filter rules in place.
- **`removeRule`**: delete one of the filter rules constituting the filter rule set. Note that if the last filter rule is removed then the filter does not make much sense any more.

- **`removeVC`**: delete one of the VCs in the VC set of the filter. Note that if the VC set is empty after that operation than IP datagrams that match one of the filter rules of the filter are being discarded which may or may not be desired.
- **`changeRuleSet`**: exchange the existing filter rules against a new disjunction of filter rules but leave the VC set in place.
- **`changeVCset`**: exchange the existing VC set against a new VC set but leave all filter rules in place.

It needs to be mentioned that the above operations can fail in which case the methods "throw" adequate exceptions.

4.4.2 The FilterRule Class



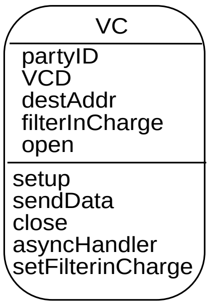
This class is the user-level front-end for specifying the criteria against which IP datagrams should be matched in order to decide whether they take the default VC or an especially setup VC over an ATM subnetwork. A **`FilterRule`** object is in direct correspondence to a kernel-level filter **`kernelFilter`**, more or less just building an object-oriented wrapper around this C structure. The added functionality is to intelligently initialize the **`kernelFilter`** with adequate wild cards for the different kind of predicates which are supported by the VCM kernel instance. The actual interface provided by a **`FilterRule`** object depends upon which and how many predicates are supported. At the moment the VCM kernel instance and thus also the VCM user instance "only" supports rules that are built from the predicates: IP source and destination address, transport protocol, and transport protocol source and destination port. We have discussed how the VCM kernel instance has to be modified for inclusion of other predicates and have illustrated how simple that is. For the VCM user instance, the necessary modifications are even simpler. The only class that needs to be changed is **`FilterRule`**. And here only a minor modification which provides the initialization of that new predicate and a set function for the predicate is necessary.

Correspondingly, the interface of the **`FilterRule`** class at the moment consists of the following methods (excluding again the constructor and destructor of **`FilterRule`**):

- **`setIPSrcAddr`**: this method sets the IP source address predicate to the specified value (where the mask is set to 32 bit, i.e. the match is exact).
- **`setIPSrcAddrMask`**: this method sets the mask value for the IP source address predicate.
- **`setIPDstAddr`**: this method sets the IP destination address predicate to the specified value (where the mask is set to 32 bit, i.e. the match is exact).
- **`setIPDstAddrMask`**: this method sets the mask value for the IP destination address predicate.
- **`setIPProto`**: this method sets the 8-bit value against which the protocol field of the IP header should be matched exactly in order to qualify for a certain transport protocol.
- **`setTPSrcPort`**: this method sets the transport protocol source port predicate to the specified 16-bit value (where the offset is initially set to 0, thus degenerating the range match into an exact match).
- **`setTPSrcPortOffset`**: this method sets the offset value of the transport protocol source port predicate, thus effectively specifying the magnitude of the range of that predicate.
- **`setTPDstPort`**: this method sets the transport protocol destination port predicate to the specified 16-bit value (where the offset is initially set to 0, thus degenerating the range match into an exact match).
- **`setTPDstPortOffset`**: this method sets the offset value of the transport protocol destination port predicate, thus effectively specifying the magnitude of the range of that predicate.

Note that by just constructing a **`FilterRule`** and setting the desired predicates it is not yet activated in the kernel, which is of course obvious since it is not associated with any VCs yet.

4.4.3 The VC Class



This class is an abstract base class for the **`PointToPointVC`** and **`MultipointVC`** classes for shared functionality between them and in order to be able to hold those two types of VCs in the same container if that is desired by a user of the VCM. Every **`VC`** is contained in exactly one **`Filter`** object and has a relation with a **`KernelIF`** object in order to access the UNI signalling functionality required for setting up and tearing down VCs. Furthermore, it has a relation to a **`VCDVCTuple`** object in order to track the outbound VCs in the global **`VCList`**, and it has a one-to-one relation to a **`QoS`** object thereby expressing the fact that every VC has to have a certain QoS. In addition, a VC keeps track of its party ID **`partyID`** and its VC descriptor **`VCD`**. A further attribute of a VC is the destination ATM address **`destAddr`** it is connected with (in case of a point-to-multipoint VC it is the address of that party for which the initial SETUP message is being sent). In the attribute **`open`** it is tracked whether the VC is already set up or not. The attribute **`filterInCharge`** stores the Filter object to which the VC belongs. This is needed in order to be able to signal asynchronous events that happen on a VC to the **`Filter`** object that "knows" which **`FilterRule`** objects are affected by that.

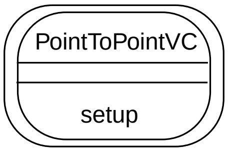
The public interface offered by the **`VC`** class consists of the following methods (excluding the **`VC`** constructor and destructor functions):

- **`setup`**: this is a pure virtual function that must be specified by the concrete setup functions of derived classes. However, since the setup procedure for point-to-point and point-to-multipoint VCs only differs with respect to a single flag (the bearer capability connection configuration flag) that must be set for point-to-multipoint VCs while it must not be set for point-to-point VCs, we have summarized that shared functionality in a protected **`setup`** method that can be called by the derived classes with the flag as parameter. This setup procedure consists of filling in the required Information Elements (IE) for the SETUP message as e.g. the calling party address, the called party address, the desired QoS, etc., and then actually sending the constructed SETUP message via the routines offered by the **`KernelIF`** object. While the SETUP message is in transit, the **`setup`** procedure "waits" on a condition variable until either a CONNECT message or a RELEASE/RELEASE COMPLETE message is received, those signalling success respectively failure of the VC setup. If the setup was successful, then an empty data buffer is sent on that VC (using the protected function **`sendData`**), which however is "swallowed" by the VCM STREAMS module and stored as a VC template for the next filter being set up in the VCM kernel instance.
- **`close`**: this method allows to close a VC explicitly, in contrast to the VC destructor which among other things also closes the VC (by calling the **`close`** method when it was still **`open`**).
- **`asyncHandler`**: this function is called when an asynchronous event happens on a VC. It then calls depending on the event the corresponding function of its **`filterInCharge`**.
- **`setFilterInCharge`**: this method sets the filterInCharge attribute to the specified value. It is being called from the **`Filter`** object which "knows" which VCs belong to it.

Besides the public interface there is one protected function that is important to be mentioned:

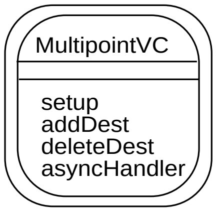
- **`sendData`**: this protected method allows to send data onto the VC. It is protected, because users of the VCM library should not send data onto VCs, because those VCs are only for special IP data streams flowing through the IP/ATM edge device.

4.4.4 The PointToPointVC Class



This class is a specialization of the **`VC`** class described above. Actually, it does not offer very much added functionality. It just concretizes the **`setup`** method from its base classes by calling its protected member function **`setup`** with the correct flag for point-to-point VCs. An alternative to the design with an abstract base class VC and specialized **`PointToPointVC`** and **`MultipointVC`** classes could have been to take the **`PointToPointVC`** class as a base and derive the **`MultipointVC`** class from it. This however seemed counterintuitive to us, since a point-to-multipoint VC does not have a ISA-relationship to a point-to-point VC.

4.4.5 The MultipointVC Class

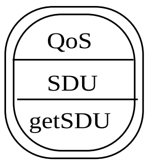


This class is another specialization of the **`VC`** base class. It also concretizes the **`setup`** procedure specified by the base class by calling its protected **`setup`** procedure, but now with a flag indicating that a point-to-multipoint VC shall be set up. In contrast to the PointToPointVC class there is actually some more extra functionality, mainly for adding and dropping parties during the lifetime of a point-to-multipoint VC:

- **`addDest`**: this function allows to add a party to an existing point-to-multipoint VC. To achieve that, the necessary IEs are filled in, as e.g. the party's ATM address, and then an ADDPARTY message is sent using the functionality provided by the **`KernelIF`** object with which the VC is associated. The **`addDest`** function then "waits" on a condition variable for the result of that ADDPARTY message, which can either be an ADDPARTYACK or an ADDPARTYREJECT message, signalling success respectively failure of the operation. In case the party could be added to the existing point-to-multipoint VC, its party ID is being kept track of in conjunction with its ATM address in an **`AddrPartyIDTuple`** object that belongs to the **`MultipointVC`** object.
- **`deleteDest`**: this method allows to drop a specified party from the point-to-multipoint VC. In addition it releases the association between the party ID and ATM address for that party.
- **`asyncHandler`**: in contrast to the **`PointToPointVC`** a specialized version of the **`asyncHandler`** method is required for the **`MultipointVC`** class due to the fact that there can be additional asynchronous events for this kind of VCs. So, besides a RELEASE message, also a DROPPARTY message can be a potential source of an asynchronous event, which needs to be handled differently, since now the VC could potentially remain in place if the user wishes so. The user's preference with respect to that are specified as already described by passing the asynchronous failure handler code to the **`Filter`** object which is in charge for the **`VC`** object.

In all the methods described above the **`MultipointVC`** object makes sure that it always keeps its sorted list of **`AddrPartyIDTuple`** objects up to date.

4.4.6 The QoS Class



The **`QoS`** Class is a lean class providing an easy interface to the QoS model of the UNI 3.0/3.1 specifications of the ATM Forum. While these require a lot of cumbersome filling in of IEs that apply to the QoS of a VC that is built up via a SETUP message, the **`QoS`** class frees a user of the VCM library of that burden without hiding any of the QoS capabilities of those specifications. A user can create a **`QoS`** object by just specifying

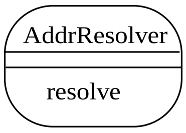
- the traffic type it desires (which is one of {**`CBR`**,**`VBR`**,**`UBR`**}[††]) to send,
- the traffic class (which is one of **`qos_class{0-4}`**) it wants to use, and

- the corresponding traffic parameters (which are depending on the traffic type a subset of {**pcr**,**scr**,**mbs**}) that apply to the connection.

It has to be noted here that we always perform uni-directional reservations, although for point-to-point VCs bi-directional reservation would also be possible. This would however complicate our design and we therefore decided to support only unidirectional VCs.

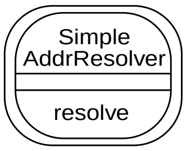
The only method offered by the QoS class besides a constructor and destructor is **getSDU** which returns a handle on the system-level representation of the IEs of a SETUP message as it is needed for the routines offered by the SDAPI library.

4.4.7 The AddrResolver Class



The **AddrResolver** class is just a simple interface class that describes how an object that resolves IP addresses into ATM addresses should be invoked, respectively the other way around how such an object can offer its services to the user. Hence, the only pure virtual method being provided is **resolve**.

4.4.8 The SimpleAddressResolver Class



This class provides a very simple way of eliciting the ATM address of an ATM interface for which only the IP address is known. It is derived from the abstract base class **AddrResolver** and thus inherits its interface description.
The **resolve** method is based upon the precondition that CLIP is running on the local ATM interface as well as on the remote interface. It works the following way:

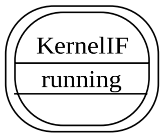
- it invokes a shells script which in turn performs a **traceroute** for the IP address that is to be resolved,
- this causes the CLIP cache to contain the required IP-ATM address association,
- from the cache this association is written to a local file,
- which is in turn read by the **resolve** routine.

It is obvious that this is not the most efficient and elegant way of doing address resolution, however it was simple. Yet, we will work on more sophisticated mechanism to do address resolution but for our first prototypical solution it was sufficient to show that all the rest of the implementation is working. The address resolution issue can be solved independently.

4.5 Kernel Interface Layer Classes

Let us now turn to the classes which form the kernel interface layer. As already mentioned, these classes tend to be hybrids between C and C++ programming style due to their proximity to system-level interfaces.

4.5.1 The KernelIF Class



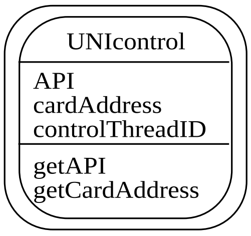
The **KernelIF** class is something like an "entrance door" to the parts of the VCM user instance that communicate with the kernel-level instance of the VCM. It encompasses the functionality of the VCM control subcomponent and the UNI control subcomponent by inheriting the interface from its base classes **VCMcontrol** and **UNIcontrol**. Note that there is always only one **KernelIF** object per ATM interface, i.e. if the edge device

††. With the network adaptors we used for testing (Fore’s PCA200E and SBA200E) it was not possible to setup VBR VCs, whereas our ATM switch (Fore’s LE155) seemed to have no problem with it.

has only one ATM card, then there can be only one **KernelIF** object. This restriction is controlled by the **running** flag, which is a class variable indicating whether a **KernelIF** object is already existing for a certain interface or not. When a **KernelIF** object is created it must be passed the IP convergence module on top of which the VCM device shall be "sitting", the IP address and netmask that shall be assigned to that interface and the unit number of the interface on which the VCM module shall be installed.

A possible design alternative would have been not to unify the interfaces of **VCMcontrol** and **UNI-control** via the **KernelIF** class. However, we decided it to be more convenient if anything that is directed towards kernel-level functionality should be sent to a common interface.

4.5.2 The UNIcontrol Class

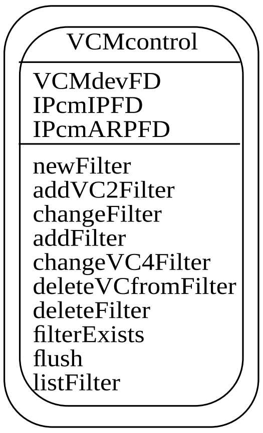


This class essentially builds an object-oriented wrapper around the functionally provided by the C-style SDAPI library which in turn provides an API to the UNI signalling services provided by the ATM network driver. When a **UNIcontrol** object is created (something that can only be done by a **KernelIF** object since the **UNIcontrol** constructor is protected an thus only derived classes can access it), it is passed the unit number of the interface on which UNI services are desired. From that unit number the first action is to elicit its own ATM address using the SDAPI library and storing this address in **cardAddress**. It then creates an SDAPI instance by specifying what is the callback function that shall be invoked on UNI signalling events. The result is a handle on that instance which is stored in **API** for later reference in calls to the SDAPI library. Since the creation of an SDAPI instance corresponds to the setting up of a stream from the SDAPI library user part to the SDAPI device, it is made sure that the VCM STREAMS module is pushed on that stream just above the SDAPI device by configuring the Solaris **autopush** facility accordingly. After that operation **autopush** must be reconfigured immediately in order to avoid pushing the VCM STREAMS module onto other SDAPI streams which have nothing to do with the operation of the VCM. Henceforth, the **API** instance is set up to listen to all UNI events that pertain to VCM related signalling activities (this is achieved by using a certain, common preconfigured selector byte for all instances of VCMs) by issuing the corresponding SDAPI library calls. If that was successful, the UNI control thread which handles the UNI signalling events is created and the necessary synchronization primitives for communicating between the control thread and the main thread are initialized.

This control thread, implemented by the class function **controlThread** goes into a dispatch loop where it "sleeps" on a **select** system call until something happens on the **API** file descriptor. In that case it calls the class function **SDAPIcallback** which then handles that signalling event. This handling is actually quite complicated and depends on the state in which the VCM user is in. The main question here is whether the event is the answer for a certain request as e.g the setup of a VC, i.e. a synchronous event that must then be signalled upwards to the main thread which is already waiting for it on a condition variable, or whether it is an asynchronous event, i.e. the main thread is not waiting for it. Furthermore the signalling events can either pertain to inbound or outbound VCs which must be distinguished as well. In fact, all these complexities where the original motivation for locating that functionality into the user-level part of the VCM since the development is very error-prone anyway and should thus not furtherly be aggravated by the difficulties of kernel-level coding and debugging.

The interface methods **getAPI** and **getCardAddress** do just what their names suggest, they return the **API** handle respectively the ATM address of the interface on which a **UNIcontrol** object is located.

4.5.3 The VCMcontrol Class



The **`VCMcontrol`** class is an object-oriented wrapper class for the functionality provided by the C-style interface of the STREAMS head of the stream leading to the VCM device. Like the **`UNIcontrol`** class it is a base class for the **`Kerne-lIF`** class, which is the only one that is allowed to construct objects of it.
When a VCMcontrol object is constructed it is passed the IP convergence module the VCM device shall "sit" upon, the IP address and the netmask of the interface on which the VCM shall be installed. It then makes sure that no VCM device exists already on that interface. Next, it constructs the STREAMS plumbing below the VCM device by opening the VCM device and two STREAMS to the IP convergence module, one for the ARP and the other for the IP stream. All the file descriptors from these operations are stored in the **`VCMcontrol`** object for later reference (in **`VCMdevFD`**, **`IPcmIPFD`** and **`IPcmARPFD`**, respectively). Then the IP and ARP stream leading to the IP convergence module are linked below the VCM device, which is then plumbed into the TCP/IP protocol stack under the specified IP address and netmask (using **`ifconfig`**).

The interface which the VCMcontrol class provides for communicating with the VCM device corresponds one-to-one with the available ioctl commands available at the STREAMS head of the stream leading to the VCM device (see also Section 3.6.4):

- **`newFilter`**: insert a new kernel filter into the VCM device’s filter set.
- **`addVC2Filter`**: add newly setup VC(s) for a specified kernel filter.
- **`changeFilter`**: change the filter rule for a given kernel filter without changing its VC set.
- **`addFilter`**: add a kernel filter that "routes" to the same set of VCs as a given filter.
- **`changeVC4Filter`**: change the VC set for a given kernel filter.
- **`deleteVCfromFilter`**: delete a specified VC from a given kernel filter.
- **`deleteFilter`**: delete a kernel filter from the filter set.
- **`filterExists`**: test whether a given filter exists in the VCM device’s filter set.
- **`flush`**: flush all the entries in the VCM device’s filter set.
- **`listFilter`**: dump the current filter set onto the console.

The "added value" of that interface is that the parameters of those methods can however be specified more conveniently when compared to the STREAMS head interface.

4.6 Auxiliary Classes

In this section we present the implementation of auxiliary classes that were needed for the implementation of the functionality of the classes for the user and kernel interface layers described above. These are mainly container classes (mostly used for the implementation of relations with variable cardinality), and classes for storing logical associations between logically linked entities.

4.6.1 The List Class

An object of type List provides a list of type ‘Value’.

```
template <class Value>
class List {
```

Standard constructor:

```
   List();
```

Copy constructor:

```
List( const List& l );
```

Destructor:

```
~List();
```

Assignment operator:

```
List& operator=( const List& l );
```

Compare operations:

```
bool operator==( const List& l ) const;
bool operator!=( const List& l ) const;
```

Class Iterator and ConstIterator provide a similar interface and can be used to iterate through a container having access to its elements. ConstIterator have to be used for iterating through a constant container object. Consequently the dereference-operator returns a reference to a constant object of type ‘Value’. When removing an element from a container, all iterators pointing to that element are invalidated and must not be used anymore.

```
class Container::Iterator {
   Iterator();
   Iterator& operator++();
   Iterator& operator--();
   Value& operator*() const;
   operator== ( const Iterator& i ) const;
   operator!= ( const Iterator& i ) const;
};
```

Empty the list:

```
void clear();
```

Insert element at beginning of list:

```
void push_front( const Value& elem );
```

Remove first element:

```
void pop_front();
```

Append element at rear end:

```
void push_back( const Value& elem );
```

Remove last element:

```
void pop_back();
```

Access first element:

```
Value& front();
const Value& front() const;
```

Access last element:

```
Value& back();
const Value& back() const;
```

Get iterator pointing to first element:

```
Iterator begin();
ConstIterator begin() const;
```

Get iterator pointing to last element:

```
Iterator end();
```

```
   ConstIterator end() const;
```

Insert element before ‘pos’. Returns iterator pointing to a new element:

```
   Iterator insert( ConstIterator pos, const Value& elem );
```

Erase element at ‘pos’. Returns iterator pointing to next element after ‘pos’:

```
   Iterator erase( ConstIterator pos );
```

Insert range of elements from another list, before ‘pos’:

```
   void insert( ConstIterator pos,
      ConstIterator first, ConstIterator last );
```

Erase range of elements. Returns iterator pointing to next element after last deleted:

```
   Iterator erase( ConstIterator first, ConstIterator last )
```

Number of elements:

```
   unsigned int size() const;
```

Boolean value to indicate whether container is empty:

```
   bool empty() const;
};
```

4.6.2 The SortedList Class

An object of type SortedList provides a sortable list of its value type. Type ‘Key’ should be set to the same type as ‘Value’ to indicate that the ordering is inherent to the value type. If a different key is used to access elements of a list, type ‘Value’ must either be inherited from ‘Key’ or provide a conversion operator to type ‘Key’. The type ‘Compare’ must be a class adhering to the following interface:

```
struct CompareType {
   operator bool( const Key&, const Key& );
};
```

The boolean function operator must model the ‘less than’ relation between two objects of type ‘Key’. A standard compare class exists, named ‘Less’, which uses the default ‘less than’ operator.

Given the requirements above for ’Key’ and ‘Value’, access methods (find, erase, etc.) can take both, an element or a key as an argument.

```
template <class Value, class Key, class Compare>
class SortedList : public List<Value> {
```

SortedList provides with some restrictions the interface of class List and additionally, the following methods are exported:

Find an element ‘elem’. Returns an iterator pointing to that element. If ‘elem’ does not exist in the list, an iterator to the next-larger element is returned.

```
   ConstIterator lower_bound( const Key& elem ) const;
```

Find an element ‘elem’. Returns an iterator pointing to that element. Returns point to ‘end()’, if element is not found.

```
   ConstIterator find( const Key& elem ) const;
```

Find or insert an element ‘elem’. Returns an iterator pointing to that element.

```
   ConstIterator find_or_insert_sorted( const Value& elem );
```

Boolean value to indicate whether a sorted list contains a specific element:

```
   bool contains( const Key& key ) const;
```

Insert an element at the correct position. Returns pointer to new element:

```
   ConstIterator insert_sorted( const Value& elem );
```

Insert an element at the correct position and ensure uniqueness. Returns pointer to new element. Returns iterator to ‘end()’, if element already exists:

```
   ConstIterator insert_unique( const Value& elem );
```

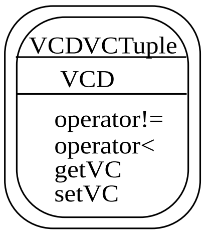
Erase element at position ‘pos’. Returns iterator pointing to next element after ‘pos’:

```
   ConstIterator erase( const Key& elem );
};
```

The restrictions to the interface of SortedList when compared to List is that the following methods are *not accessible*, because they could corrupt the internal ordering:

```
   void push_front( const Value& elem );
   void push_back( const Value& elem );
   Value& front();
   Value& back();
   Iterator insert( Iterator pos, const Value& elem );
```

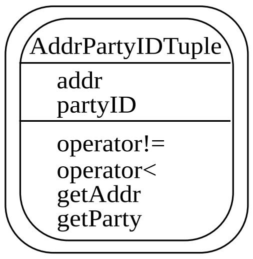
4.6.3 The VCDVCTuple Class



This class holds the logical association between a VC and its VC descriptor as it is used with the SDAPI library. As already mentioned we can distinguish inbound and outbound VCs, where outbound VCs are initiated by the local VCM module and are kept track of in a **VC** object which is then related to the corresponding **VCDVCTuple** object. Inbound VCs are initiated by remote VCM modules and thus have no relation to a corresponding **VC** object. Nevertheless they must also be tracked since UNI signalling is generated for them as well and those signals must be distinguishable from signals for outbound VCs. For that reason all the **VCDVCTuple** objects are kept in a global container called **VCList** and when a signalling event occurs its VCD is used in order to decide whether it is for an inbound or an outbound VC and if it is for an outbound VC then for which one.

The operator functions are needed in order to be able to hold the **VCDVCTuple** objects in a **SortedList** container for faster access. The **getVC** and **setVC** methods are used for setting up and querying the relation to a **VC** object.

4.6.4 The AddrPartyIDTuple Class



This class holds the logical association between the party ID for a certain party within a point-to-multipoint VC connection and its ATM address. A **MultiPointVC** object holds a **SortedList** of **AddrPartyIDTuple**’s in order to keep track of that association. This is done, e.g. in order to be able to inform a user in case of a DROPPARTY message which party has actually been dropped. The user can then decide how to handle that situation, i.e. whether the VC is still valuable to him or whether the whole VC shall be torn down because all parties are essential for the communication to take place.

The operator functions are needed in order to be able to hold the **AddrPartyIDTuple** objects in a SortedList container for faster access. The **getAddr** and **getParty** methods are used for querying the relation between ATM addresses and party IDs.

4.7 Example of Use

In order to illustrate the use of the VCM user instance, i.e. its library interface, we present a simple example of its usage in a hypothetical user code:

First of all a kernel interface must be created

```
KernelInterface *ki = new KernelInterface("qaa", "192.168.230.20",
"255.255.255.192", 0);
```

Here we take as an example CLIP as IP convergence module (the interface of which in our configuration is called **qaa**) with the specified IP address and netmask, furthermore we use the ATM card with **unit** number 0.

Next, an object for address resolution is created

```
SimpleAddressResolver sar;
```

Since we only have the **SimpleAddressResolver** available we use that one. We now use that object in order to resolve two IP address into ATM addresses for peer edge devices for which we want to setup special filters:

```
in_addr_t fiddleIP = inet_addr("192.168.230.20");
atm_addr_t* fiddleATM = sar.resolve(fiddleIP);
in_addr_t violaIP = inet_addr("192.168.230.30");
atm_addr_t* violaATM = sar.resolve(violaIP);
```

Now, a filter rule is created

```
FilterRule* f = new FilterRule;
```

and its IP destination address and transport protocol port predicates are set

```
f->setIPDestAddress(inet_addr("224.6.6.6"));
f->setTPDestPort(10000);
```

Next a QoS object is created and a point-to-multipoint VC object is created to the first peer edge device with that QoS

```
QoS* qos = new QoS(CBR, qos_class_0, 11000);
MultipointVC* mpvc = new MultipointVC(fiddleATM, qos, ki);
```

Then a filter object is created which associates the specified filter rule with that VC

```
Filter* f = new Filter(Filter::RuleList(f), Filter::VCList(mpvc), ki,
                       callbackOnAsync);
```

where the last parameter is a function variable specifying how asynchronous events on the given VC for that filter should be handled. From now on, IP datagrams that satisfy the filter rule (i.e. are addressed to the IP multicast group 224.6.6.6 with port 10000) are forwarded using the especially setup point-to-multipoint VC.

If now the second peer edge device shall be added to the point-to-multipoint VC then

```
mpvc->addDest(violaATM);
```

does the job. Or if the first destination shall be deleted, then

```
mpvc->deleteDest(fiddleATM);
```

is appropriate.

This was only an illustrative example of some operations that are possible when using the described VCM library.

5 Summary

In this report we provided a detailed description of the design and implementation of an IP/ATM adaptation module which allows to leverage the QoS facilities provided by ATM for an overlaid IP-based network. This description was structured according to the internal structure of the VCM module. The VCM module is divided into a user instance and a kernel instance. In chapter 2 we provided the global view on the interworking of those two, before we gave detailed descriptions of each and their internal components in chapter 3 and chapter 4. While the kernel instance is designed according to the STREAMS paradigm and implemented using the C programming language, the user instance's design is object-oriented offering a library interface for the C++ programming language. We think that this hybrid design suits the respective needs of those two distinct instances best, since in kernel space efficiency and ease of use of existing interfaces has preference over elegant design and reusability, whereas for the user instance those two metrics are crucial ones. One goal of the design was also to source out all complex operations which are non-critical with respect to performance on the data forwarding path from kernel into user space. That ensured maximum ease of development and coding and certainly will ensure this for future modifications as well, thus making the IP/ATM adaptation module a valuable tool for further experimentation with IP/ATM edge device functionality.

The adaptation module was intentionally designed as flexible and general as possible so that it can be used for virtually any "QoS signal" given by the IP network in order to trigger special handling by the ATM network. Our motivation to do so was that while overlaying the RSVP/IntServ architecture was the motivation to develop such an adaptation module in the first place it is conceivable that RSVP/IntServ will only be one of several "tools" within IP-based networks for users to convey their desires with regard to QoS provisioning by the network. Thus flexibility was one of the most important design goals for the IP/ATM adaptation module and has been like all the other design goals that were set in Section 2.2 achieved from our point of view.

We think that together with the sophisticated solution approaches devised in [SKWS98] for the overlaying of RSVP/IntServ onto ATM networks which were also partially implemented for our former prototype implementation (see Appendix), the IP/ATM adaptation module now also provides the potential to support those approaches on the data path in a performant manner. Thus a complete solution is now achievable by the integration of those two parts. Of course, this still remains on a prototypical level, since many of the operations of an actual production-level IP/ATM edge device will be assisted by special hardware. However, the overall architecture should remain the same and the same algorithms will be applicable.

References

[CY91] P. Coad and E. Yourdon. Object-Oriented Analysis, 1991. Prentice Hall, Englewood Cliffs.

[OSI91] OSI Work Group. Data Link Provider Interface Specification Rev. 2.0.0, August 1991. UNIX International.

[SKWS98] Jens Schmitt, Martin Karsten, Lars Wolf, and Ralf Steinmetz. Internet Integrated Services Multicast on ATM Networks and RSVP Extensions for Charging, August 1998. 1st Milestone Report of IQATM Project (Phase 2).

[Sun95] SunSoft. STREAMS Programmer's Guide , November 1995.

[Sun96] SunSoft. Writing Device Drivers , August 1996.

[SVSW98] S. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *Proceedings of SIGCOMM'98*. ACM, September 1998.

[SWK+99] Jens Schmitt, Lars Wolf, Martin Karsten, Ralf Steinmetz, Yann-Olivier Lorcy, and Christian Siebel. Shortcutting IP Flows over Large ATM Networks. In *Proceedings of the 2nd IEEE International Conference on on ATM (ICATM'99), Colmar, France*. IEEE, June 21–23 1999.

[SWS97a] Jens Schmitt, Lars Wolf, and Ralf Steinmetz. Interaction Approaches for Internet and ATM Quality of Service Architectures , June 1997. 2nd Milestone Report of IQATM Project (Phase 1).

[SWS97b] Jens Schmitt, Lars Wolf, and Ralf Steinmetz. Design and Implementation of an RSVP over ATM Prototype, October 1997. 3rd Milestone Report of IQATM Project (Phase 1).

[WVTP97] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proceedings of SIGCOMM'97*. ACM, September 1997.