Linux ATM API
Draft, version 0.4

Werner Almesberger
EPFL, LRC
werner.almesberger@lrc.di.epfl.ch

July 19, 1996

Contents

1 About this document **3**
1.1 Related APIs . . . 3
1.2 Design considerations . . . 3
1.3 Description of API elements . . . 4
1.4 Changes since version 0.3 . . . 4
1.5 Changes since version 0.2 . . . 5
1.6 Changes since version 0.1 . . . 5
1.7 Changes since first draft (version 0) . . . 6

2 Connection control **7**
2.1 Phases . . . 7
2.2 Address descriptors . . . 9
2.3 PVC addressing . . . 10
2.4 SVC addressing . . . 12
2.5 Attribution of incoming calls . . . 16
2.6 QOS descriptors . . . 16
2.7 Library functions . . . 18
2.8 Connection preparation . . . 20
2.9 Connection setup . . . 23
2.10 Connection status . . . 24
2.11 Connection teardown . . . 24
2.12 Connection control summary . . . 25

3 Data exchange **27**
3.1 Transfer scheduling . . . 27
3.2 Sending and receiving . . . 27
3.3 Alignment and size constraints . . . 28
3.4 Asynchronous I/O . . . 29
3.5 Example . . . 29

4 Administrative functions **31**
4.1 ioctl arguments . . . 31
4.2 Interface configuration . . . 31
4.3 Connection configuration . . . 33
4.4 Physical layer . . . 33
4.5 `/proc` . . . 33

5 Related services **34**
5.1 IP over ATM 34
5.2 Arequipa 34

A Acronyms **35**

1 About this document

This document defines an API for ATM-related system services under Linux. The basic design idea is to extend the 4.3 BSD release UNIX socket interface to support additional functionality needed for ATM wherever possible.

Currently, only CBR and UBR are specified. Support of ABR, ABT, VBR, etc. is left for further study. Similarly, SVC multicast signaling is not specified yet. Also, only an unreliable datagram transport is defined.

Detailed descriptions of system calls and library functions are only given for items that are additions specific for ATM or if their their use for ATM differs from their use for the INET protocol family. See [1] for a general introduction to the BSD socket API.

ATM functionality covered by this API is described in [2] and in [3], which are in turn based on several ITU-T documents.

Pointers to previous and future versions of this document can be found at `http://lrcwww.epfl.ch/linux-atm/`

1.1 Related APIs

This API is not directly aligned with the "Native ATM Services" API specification [4] by the SAA API ad-hoc work group of ATM Forum, but evolution of their work is being monitored and any development requiring significant changes to the API will be reflected in this proposal. The SAA API work group currently doesn't intend to specify an ATM API for the BSD-style socket interface.

The SAA/Directory work group of ATM Forum is defining an ATM name resolution service [5] and is also considering the issues of textual address representations. The Linux API document will be aligned with the results of that work.

The IETF currently faces similar compatibility issues in preparation of the transition from IPv4 to IPv6. The current Internet Draft describing IPv6 extensions for BSD sockets [6] therefore provides useful insights into problems which have to be addressed in the context of ATM too.

Finally, IEEE is drafting the standard POSIX socket API in 1003.1g. Alignment with this imminent important standard has priority over the other standards listed above.

Internal APIs of Linux ATM are covered by [7].

1.2 Design considerations

The main considerations driving the design of this API are the following:

- when porting applications written for INET domain sockets, only source code parts using addresses should need to be modified
- where existing semantics are changed, the original design philosophy and also the original terminology should be retained
- where semantics are modified or new ones are added, work from related standardization efforts is adapted if possible

Also, care is being taken to make sure to follow up design decisions with implementation experience as closely as possible.

1.3 Description of API elements

The API is described in terms of C data structures and function calls. Excerpts from the respective header files are included only to illustrate the use of the structures – it should not be assumed that any implementation uses identical declarations. In particular, no assumption should be made about the following properties:

- order of elements in structs
- size of structs, unless explicitly indicated
- alignment of struct elements
- absence of additional struct elements
- numerical values of manifest constants, except for (absence of) equality with other manifest constants from the same group of constants[1]

Include file names will change in the future.

Integers are assumed to have at least the following size:

Type	Bits
`char`	8
`short`	16
`int`	32
`long`	32

1.4 Changes since version 0.3

The following items have been added:

- described how `text2atm` changes the address structure
- documented `getsockname` and `getpeername`
- added `ATM_ANYCLASS` traffic class
- new `struct atm_qos` and socket option `SO_ATMQOS` for QOS specification

The following items have been changed:

- corrected the reference to `/usr/include/atm.h` (was mis-named as `/usr/include/atmlib.h`)
- terminology change: what was previously called "double `connect`" is now called "two-phase `connect`"
- relaxed parameter restrictions for the second call in a two-phase `connect`
- socket addresses no longer contain the QOS specification
- a "connection descriptor" is now either called "address descriptor" or "QOS descriptor", depending on which part of the former structure is used
- changed `SO_CIRANGE` into a pair of ioctls
- BHLI matching changed: SAPs without BHLI are now incompatible with calls that specify a BHLI

[1] E.g. it is safe to use manifest constants in `switch` constructs.

1.5 Changes since version 0.2

The following items have been added:

- `bind` and `connect` are no longer allowed to modify their arguments. Instead, `getsockname` has to be called to obtain the effective traffic parameters.
- wildcard syntax and rules for SVC addressing
- described new library function `atm_equal`
- described new constant `ATM_AAL0_SDU`
- the use of `max_sdu` is now strongly encouraged
- several minor changes and corrections
- renamed `sdu2cr` to `sdu2cell`

1.6 Changes since version 0.1

Known bugs:

- whitespace isn't properly arranged in some of the code excerpts
- there are no program examples for SVCs yet

The following items have been added:

- SVC addressing
- raw ATM cell ("AAL0") transport
- system call behaviour for SVCs
- described new library functions `text2atm` and `atm2text`
- added `max_pcr`
- added more details about `select`

The following items have been changed:

- fixed document version numbers
- data structures are now described by giving excerpts from include files
- major restructuring and added some pictures. The text should now be more "implementor-friendly" and also contains more introductory remarks.
- mis-use of two-phase `connect` (PVCs) now yields unspecified behaviour
- first connect (PVCs) now may allocate only some (or even none) of the resources
- changed many references from [2] to [3]
- CDV is now specified in microseconds, not in stupidities like "cell slots"
- many other minor changes and additions

1.7 Changes since first draft (version 0)

The following items have been added:

- a few introductory remarks
- indicated that PCR and CDV are ignored when UBR is chosen
- QOS guarantee only applies to network
- description of role and references to related APIs
- clarified that `connect` on incompletely specified address allocates resources
- all values are stored in host byte order
- `ATM_NONE` traffic class for unidirectional traffic

The following items have been changed:

- clarified use of address parameter in `sendto`, `recvfrom`, `sendmsg`, and `recvmsg`
- `max_pcr` has been changed to `min_pcr`
- maximum CDV explanation was ambiguous
- explained use of PVC/SVC and removed reference to ITU-T I.233
- changed "VCC" to "VC"
- several bugs in the program examples

2 Connection control

In ATM, the fundamental communication paradigm is the connection. This section describes the mechanisms to establish, use and release ATM connections.

2.1 Phases

A connection typically goes through the following four phases:

- connection preparation
- connection setup
- data exchange
- connection teardown

During *connection preparation*, parameters are set and general local resources (e.g. socket descriptors) are allocated. Neither local nor remote networking resources are allocated during preparation. During *connection setup*, local networking resources (bandwidth, connection identifiers, buffers, etc.) are allocated. Resource allocation in the network may be handled by network management (PVCs) or it may be done as part of the connection setup (SVCs). In the *data exchange*, data is sent over a previously established connection. Finally, during *connection teardown*, communication is stopped and resources are deallocated.

Connection preparation, connection setup and data exchange are typically performed in this order. Connection teardown is different in that it can be initiated at any time. It may even overlap with other phases (e.g. data may continue to flow in one direction after a `shutdown` until the final `close`).

2.1.1 Phases for PVC sockets

Opening a PVC socket means to attach an endpoint to a connection that normally already exists in the network or that is established by a third party. The state diagram in figure 1 illustrates the life cycle of PVC sockets.[2]

PVC sockets are created with the `socket` system call. Then the QOS requirements are indicated, an address descriptor is set up, and `bind` or `connect` is called (they have identical functionality when used with PVCs) to open the VC. If the VC cannot be opened, the operation may be retried, possibly after changing the address descriptor and/or the traffic parameters.

One particularity with PVCs is that they can be opened in two steps: first, only a part of the actual address is provided, so that only the resources needed for the connection can be allocated. In the second step, the complete address is provided. The address used in the first step is called an *incompletely specified address*.[3]

Once a VC is established, data can be exchanged until the socket is closed with the `close` system call. It is of course possible to close the socket in any state.

2.1.2 Phases for SVC sockets

SVCs are connections whose establishment in the network is controlled by the communicating parties. An endsystem can either accept an incoming connection (*passive open*) or it can try to establish an outgoing connection (*active open*). Figure 2 illustrates the life cycle of an outgoing connection.

[2] The states described in this document are only conceptual. They may not exist in an actual implementation.

[3] Note that the only way to undo the effect of binding to an incompletely specified address is to close and re-create the socket, i.e. it is not possible to bind to a different incompletely specified address or to an incompatible complete address.

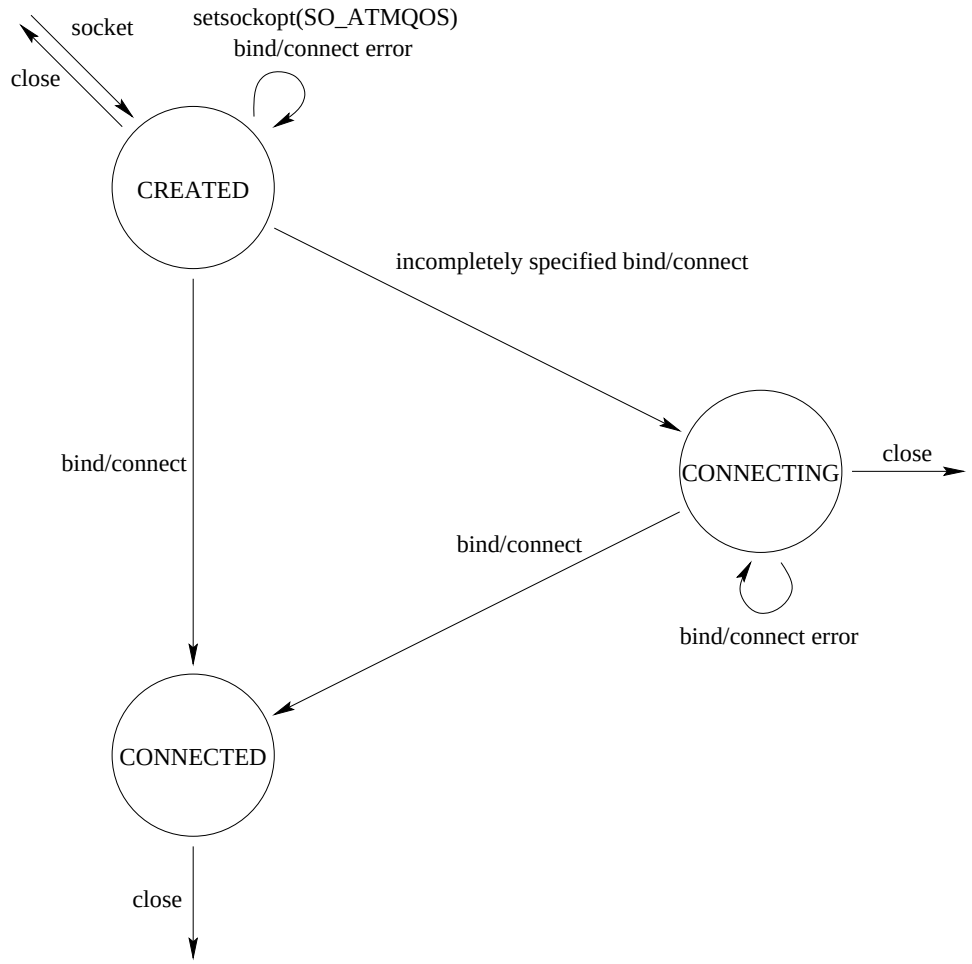

Figure 1: Opening a PVC socket.

The most common way to establish a connection is to create the SVC socket with `socket`, to specify the QOS requirements, to prepare the address descriptor, to request connection setup with a blocking `connect`, to block until the network accepts (or rejects) the connection, to exchange data, and eventually to close the socket with `close`.

The local endpoint can be bound to a specific address with the `bind` system call. If using a non-blocking connect, the process continues to run while the connection is being set up and it may also decide to close the socket prematurely. Note that there are two transitions for "connect error" from the CONNECTING state. The choice of which of them is taken depends on whether the socket is bound or not.

The passive open is illustrated in figure 3.

As usual, the socket is first created with the `socket` system call and the QOS parameters are set. Then, the address descriptor is set up to specify the type of connections that should be attributed to this socket, and the socket is bound with the `bind` system call. The *service access point* (SAP) is registered at the local signaling entity with the `listen` system call.

Now the process can block in the `accept` system call until a connection request is received, or it can poll using non-blocking `accept`s or using `select`. If an incoming connection matches the specification of the SAP, the `accept` system call succeeds (or the socket becomes readable so that `select` returns and

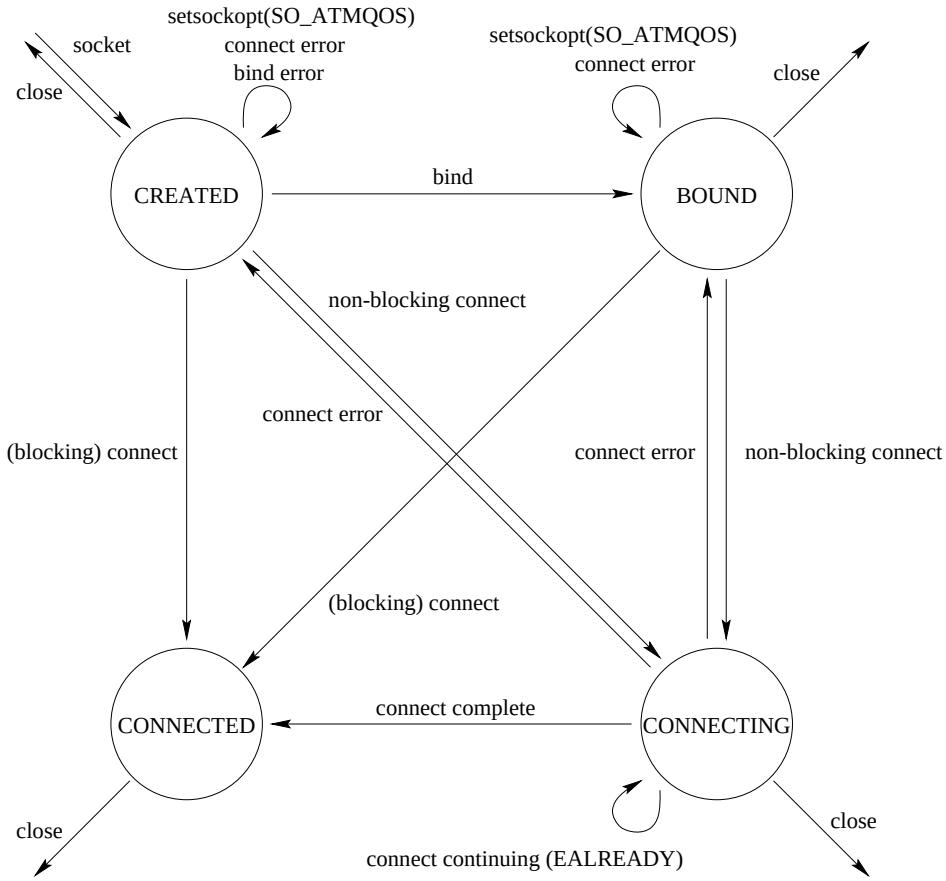

Figure 2: Active open of an SVC socket.

that `accept` can be invoked). If `accept` succeeds, a new socket is created for the connection and data exchange can begin.

The socket used to listen and the socket(s) used to exchange data can be individually closed with `close`. Note that closing the listen socket only removes the SAP registration and does not affect any connections which have already been set up.

2.2 Address descriptors

An address descriptor specifies the actual address of a destination entity. It is used wherever an address structure (`struct sockaddr`) is required. Its general structure is as follows:

```
struct sockaddr_atmtype {
        unsigned short prefix_family;
        struct {
                ...
        } prefix_addr;
};
```

type is the address type, i.e. `pvc` for PVCs and `svc` for SVCs. *prefix* is the corresponding field name prefix, which follows the convention used for other address families. The prefix is `sap` for PVCs and `sas`

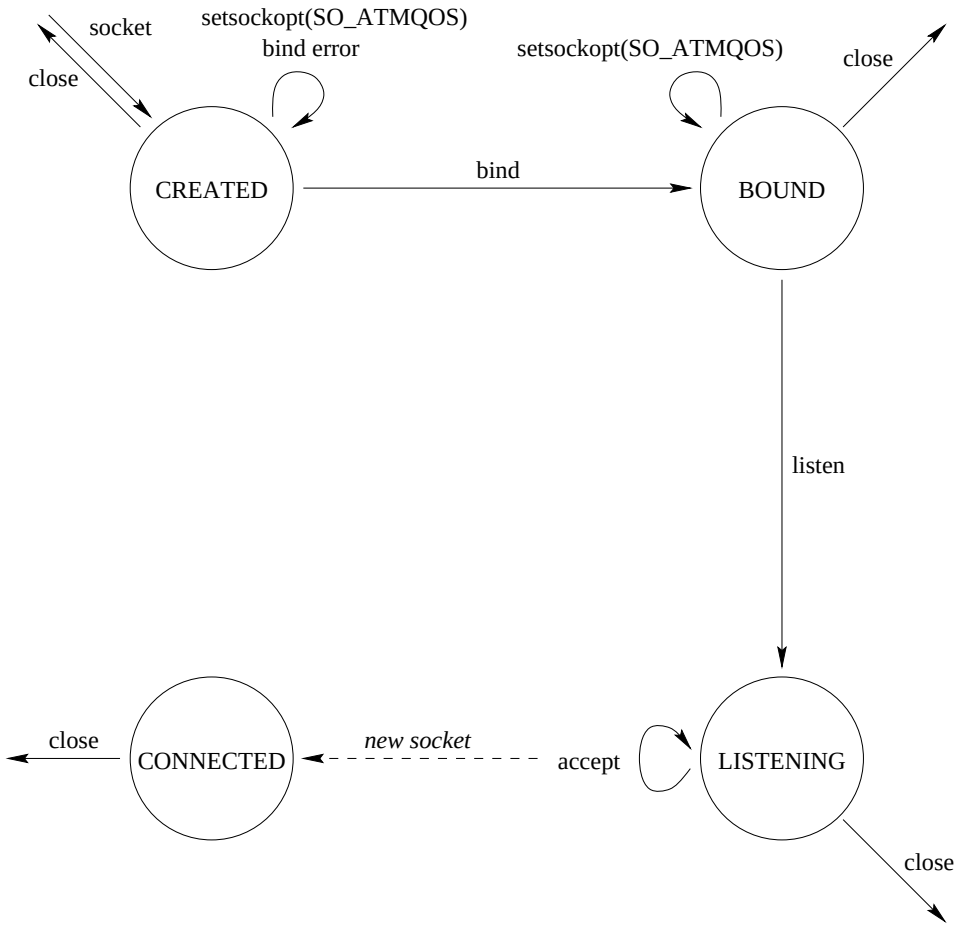

Figure 3: Passive open of an SVC socket.

for SVCs. *prefix*_`family` contains the address family. *prefix*_`addr` contains the actual address information. The address format differs significantly between PVCs and SVCs.

PVC addressing is described in section 2.3, SVC addressing is described in section 2.4.

2.3 PVC addressing

PVCs are addressed by specifying the local interface and the connection identifier (VPI and VCI) to use on that interface. The address structure is defined in `/usr/include/linux/atm.h`:

```
struct {
        short   itf;
        short   vpi;
        int     vci;
}               sap_addr;
```

The contents of all fields are stored in host byte order. `itf` is the number of the local interface (zero-based). `vpi` is the *virtual path identifier* in the range from 0 to 255. `vci` is the *virtual channel identifier*

in the range from 0 to 65535. Note that the VCIs from 0 to 31 are reserved by ITU-T or by ATM Forum and should not be used by applications. Therefore, only privileged processes are allowed to open VCs using VCIs below `ATM_NOT_RSV_VCI` (32).

The valid ranges for VCIs and VPIs may be further constrained by the interface and might be configurable (see section 4.2.3).

Two PVC sockets can share the same address if one of them uses only the receive direction (i.e. `qos.txtp.class == ATM_NONE`) and the other uses only the transmit direction. Note that such sharing may not be supported for all traffic classes. Also note that `qos.txtp.class` and `qos.rxtp.class` must not both be set to `ATM_NONE`.

2.3.1 Special PVC addresses

The following special values are recognized in PVC addresses:

`itf == ATM_ITF_ANY` selects the lowest-numbered interface on which the specified VPI/VCI pair is valid and not yet in use.

`vpi == ATM_VPI_ANY` selects the lowest-numbered free VPI on the specified interface on which the specified VCI is valid and not yet in use.

`vci == ATM_VCI_ANY` selects the lowest-numbered (non-reserved) free VCI on the specified interface for which the VPIs correspond.

`vpi == ATM_VPI_UNSPEC` does not allocate any VPI/VCI pair and uses the interface number for resource control only. `vpi == ATM_VPI_UNSPEC` also implies that the VCI is unspecified. Therefore, the VCI value is ignored.

`vci == ATM_VCI_UNSPEC` does not allocate any VPI/VCI pair and uses the interface and VPI numbers for resource control only.

Addresses containing `ATM_`*`part`*`_ANY` components are called *wildcard* addresses. Similarly, addresses containing `ATM_`*`part`*`_UNSPEC` components are called *incompletely specified* addresses.

An address may contain more than one wildcard component, e.g. `itf == ATM_ITF_ANY`, `vpi == ATM_VPI_ANY` and `vci == ATM_VCI_ANY` would select the lowest-numbered free PVC address. When using more than one wildcard component in an address, interface numbers are more significant than VPI numbers, and VPI numbers are more significant than VCI numbers when determining the lowest address. Furthermore, wildcards and incompletely specified components can be mixed. In this case, the wildcard selection is based on incomplete information.

2.3.2 Textual representation

The formats for textual representation of PVC addresses are

1. *vpi*`.`*vci*
2. *itf*`.`*vpi*`.`*vci*

itf, *vpi*, and *vci* are non-negative decimal numbers without leading zeroes, or the special characters `?` or `*`. indicating an unspecified or wildcarded address component, respectively. If the first format is used, the interface number is assumed to be zero.

2.4 SVC addressing

SVCs addressing occurs at up to three levels:[4]

- selection of a gateway relaying ATM connections between a public and a private network
- selection of an endsystem within a public or private network
- selection of a service on the selected endsystem

Public networks use E.164 (ISDN/telephony, [8]) numbers for addressing. Private networks use private ATM Forum NSAP addresses, as described in section 5.1.3.1 of [3]. As far as addressing is concerned, those public networks which also use NSAP addresses are treated like private networks.

2.4.1 Endsystem addressing

To select an endsystem, the following address information is required (examples in figure 4 are in parentheses):

- if the destination is on a private network: the private address (A→B)
- if the destination is on a public network (using E.164 addresses): the public address (A→D, D→E)
- if the destination is on a private network, which is reached via a public network using E.164 addresses: the private **and** the public address (A→F→G)

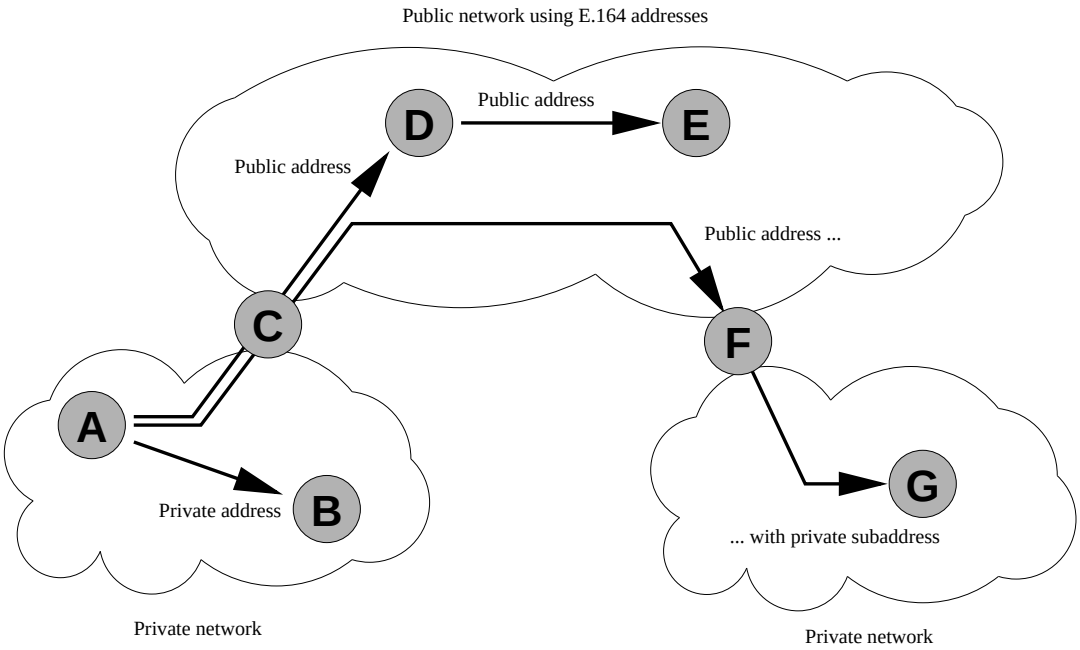

Figure 4: SVC addressing[5]

[4] A fourth level, the selection of a transit network, is specified in annex D of [3]. The Linux ATM API currently does not support the use of this mechanism.

The service on an endsystem is identified by specifying either the low-layer protocol (OSI layers 2 or 3), or the high-layer protocol, or both, or neither of them. The latter case is probably not very relevant in real networks.

The whole SVC address structure is defined in `/usr/include/linux/atm.h`:

```
struct {
    unsigned char   prv[ATM_ESA_LEN];
    char            pub[ATM_E164_LEN+1];
    struct atm_blli *blli;
    struct atm_bhli bhli;
}                   sas_addr;
```

The private address `prv` is an ATM Forum NSAP address of an ATM end system on a private network. Its length is always `ATM_ESA_LEN` (twenty) bytes. See section 5.1.3.1 of [3] for encoding rules. The private address is omitted by setting its first byte to zero.

The public address `pub` is an E.164 address. The public address is a NUL-terminated ASCII string of up to `ATM_E164_LEN` decimal digits. The public address is omitted by setting its first byte to zero (i.e. NUL).

`blli` and `bhli` identify the local service access point (SAP), e.g. the calling or called application[6] or a functional part of the latter. High-layer information is optional (its absence is indicated by setting `bhli.hl_type` to `ATM_HL_NONE`). Low-layer information is optional too (its absence is indicated by setting `blli` to `NULL`), but in addition to that, it can also be repeated. In the latter case, `blli` points to a linked list of (broadband) low-layer informations. See sections 2.4.3 and 2.4.4 for details.

2.4.2 Textual representation

The following formats can be used for textual representation of endsystem addresses:

1. *40_hex_digits*
2. *up_to_12_decimal_digits*
3. *up_to_15_decimal_digits*:*22_hex_digits*
4. *up_to_12_decimal_digits*+*40_hex_digits*
5. *up_to_12_decimal_digits*+*up_to_15_decimal_digits*:*22_hex_digits*

The first format is used for (private) NSAP addresses. The second format is used for (public) E.164 addresses. The third format is used for E.164 addresses embedded in the NSAP address format. The fourth and the fifth format are used for public addresses with private subaddresses.

Periods (`.`) can be used to separate hexadecimal or decimal digits. Periods must neither appear next to any non-digit (including other periods) nor at the beginning or at the end of the address string. Hexadecimal digits corresponding to letters can be written in upper or in lower case. A public E.164 address can optionally be prefixed with a colon to disambiguate it. A leading zero is only allowed in the NSAP address format and it is significant. Note that an NSAP address must not start with a zero byte, i.e. with two leading zeroes in hexadecimal representation.

In addition, in all contexts where wildcards are applicable, the textual representation of private NSAP addresses can be extended by appending a slash, followed by the number of valid bits in the binary form

[5] A fourth possibility would be to reach a private network from a public network by passing through a gateway using subaddressing, e.g. E→F→G.

[6] Which can be a protocol stack or a demultiplexer, e.g. IEEE 802.1 SNAP or IP.

of the address, e.g. `47000580ffe1000000f21510650020ea000ee000/104`. Entirely unused hex digits can be omitted, e.g. the example above could be abbreviated to `47000580ffe1000000f2151065/104`

When using embedded E.164 addresses, the number of bits must be greater than or equal to 72, i.e. binary wildcarding doesn't apply to the E.164 part.[7] Since all forms of E.164 addresses also imply the length (in digits), and the smallest unit is assumed to be one digit, no syntactical extension is necessary to mark wildcards.

There is no textual representation for high-layer and for low-layer information.

2.4.3 Low-layer information

Each element of a list of low-layer information elements has the following structure (defined in `/usr/include/linux/atmsap.h`):

```
struct atm_blli {
    unsigned char l2_proto;
    union {
        struct {
            unsigned char mode;
            unsigned char window;
        } itu;
        unsigned char user;
    } l2;
    unsigned char l3_proto;
    union {
        struct {
            unsigned char mode;
            unsigned char def_size;
            unsigned char window;
        } itu;
        unsigned char user;
        struct {
            unsigned char ipi;
            unsigned char snap[5];
        } tr9577;
    } l3;
    struct atm_blli *next;
};
```

The following values can be used for the layer 2 protocol (`l2_proto`):

```
#define ATM_L2_NONE     0
#define ATM_L2_ISO1745  0x01
#define ATM_L2_Q291     0x02
#define ATM_L2_X25_LL   0x06
#define ATM_L2_X25_ML   0x07
#define ATM_L2_LAPB     0x08
#define ATM_L2_HDLC_ARM 0x09
#define ATM_L2_HDLC_NRM 0x0a
#define ATM_L2_HDLC_ABM 0x0b
```

[7] The restriction is waived if the number of bits is less than eight and if a textual representation different from the one explicitly specifying an embedded E.164 address is used, i.e. if there is no way to tell that this is actually an E.164 address.

```
#define ATM_L2_ISO8802  0x0c
#define ATM_L2_X75      0x0d
#define ATM_L2_Q922     0x0e
#define ATM_L2_USER     0x10
#define ATM_L2_ISO7776  0x11
```

The value `ATM_L2_NONE` indicates that there is no layer 2 protocol information.

The following values can be used for the fields `l2.itu.mode` and `l3.itu.mode`:

```
#define ATM_IMD_NONE     0
#define ATM_IMD_NORMAL   1
#define ATM_IMD_EXTENDED 2
```

The value `ATM_IMD_NONE` indicates that the mode is not specified.

The following values can be used for `l3_proto`, the layer 3 protocol:

```
#define ATM_L3_NONE     0
#define ATM_L3_X25      0x06
#define ATM_L3_ISO8208  0x07
#define ATM_L3_X223     0x08
#define ATM_L3_ISO8473  0x09
#define ATM_L3_T70      0x0a
#define ATM_L3_TR9577   0x0b
#define ATM_L3_USER     0x10
```

The value `ATM_L3_NONE` indicates that there is no layer 3 protocol information.

The default window size `l3.itu.def_size` has to be in the range from 16 to 4096 and is encoded as $\log_2$ of the window size. The value 0 is used to indicate absence of default window size information.

The field `next` either contains a pointer to the next low-layer information element or it is `NULL`.

Note that setting both `l2_proto` and `l3_proto` to the respective null value in the same information element is invalid, i.e. if there is no information to convey, the entire element has to be omitted.

See section 5.4.5.9 of [3] for the exact coding of low layer information and for which fields are available depending on the protocol used.

2.4.4 High-layer information

`struct atm_bhli` is defined in `/usr/include/linux/atmsap.h`:

```
struct atm_bhli {
    unsigned char hl_type;
    unsigned char hl_length;
    unsigned char hl_info[ATM_MAX_HLI];
};
```

The following values can be used for `hl_type`:

```
#define ATM_HL_NONE     0
#define ATM_HL_ISO      0x01
#define ATM_HL_USER     0x02
#if defined(UNI30) || defined(ALLOW_UNI30)
```

```
#define ATM_HL_HLP      0x03
#endif
#define ATM_HL_VENDOR   0x04
```

The value `ATM_HL_NONE` is used to indicate absence of high-layer information. `ATM_HL_HLP` is only valid when using UNI 3.0 signaling.

See section 5.4.5.8 of [2] and of [3] for the exact coding of high layer information and for the content of `hl_info` depending on the information type used. The use of `hl_length` is only required if the content of `hl_info` has variable size. `hl_length` is always set in addresses returned by the operating system.

2.5 Attribution of incoming calls

An incoming call is attributed to the local SAP which fulfills the following criteria:

- It is either not registered for a specific local address (i.e. the public and the private address fields are null) or it is registered exactly for the called address.
- If high- or low-layer information is specified in the SAP or in the incoming call, both must match exactly, except for the negotiable fields `l2.itu.mode`, `l2.itu.window`, `l2.user`, `l3.itu.mode`, `l3.itu.def_size`, and `l3.itu.window`.[8] If a non-negotiable field is present in the local SAP but not in the incoming call or vice versa, they don't match.
- If the incoming call or the local SAP contains more than one low-layer information element, a pair-wise comparison is done. They match if at least one pair of low-layer information elements matches.

In case of a draw, an arbitrary SAP is selected.

2.6 QOS descriptors

QOS descriptors are used to express the QOS requirements or settings of a connection. They are encoded in a data structure of type `struct atm_qos`, which is defined in `/usr/include/linux/atm.h`. It contains the following fields:

```
struct atm_qos {
        struct atm_trafprm txtp;
        struct atm_trafprm rxtp;
};
```

`txtp` and `rxtp` are the traffic parameters in send and receive direction, respectively. Their structure is described in the following section.

2.6.1 Traffic parameters

Traffic parameters are added to the QOS descriptor by simply assigning values to the corresponding fields.

Traffic parameters are encoded in a data structure of type `struct atm_trafprm` (defined in `/usr/include/linux/atm.h`) which contains the following fields:

[8] See also annex C of [3].

```
struct atm_trafprm {
        unsigned char   class;
        int             max_pcr;
        int             min_pcr;
        int             max_cdv;
        int             max_sdu;
};
```

`class` is the traffic class, indicating general traffic properties. `min_pcr` and `max_pcr` indicate the desired peak cell rate (PCR), in cells per second. `max_cdv` is the maximum cell delay variation (CDV), in microseconds. `max_sdu` is the maximum service data unit (SDU) size, in bytes. The use of those fields is described in the following sections.

Note that compatibility of the requested QOS with available resources is not checked during connection preparation. This is only done during connection setup. Further, since only certain parameter values may be supported, connection setup may be performed with traffic parameters that differ from the requested ones. The effective parameters can be obtained by querying the socket option `SO_ATMQOS`. (See section 2.8.4.)

Further parameters (variable bit rate, maximum cell loss probability, end-to-end delay, etc.) may be added in the future. All parameter values are stored in host byte order.

Note that service guarantees (e.g. timely processing of CBR traffic) only apply to the network. It is the application's responsibility to ensure that sufficient host resources are available to properly generate or accept traffic streams.

2.6.2 Traffic classes

The following traffic classes are defined:

`ATM_NONE` no traffic in this direction

`ATM_ANYCLASS` accept any class

`ATM_CBR` constant bit rate (CBR)

`ATM_UBR` unassigned bit rate (UBR)

If no traffic class is specified (i.e. if the field is set to zero), `ATM_NONE` is used as the default. When the traffic class is `ATM_NONE`, all other traffic parameters are ignored.

`ATM_ANYCLASS` is used for specifying SVC SAPs that are compatible with any traffic class that is requested by the calling party.

Depending on the traffic class, only certain traffic parameter fields are used:

	ATM_NONE	ATM_CBR	ATM_UBR
max_pcr	–	Yes	–
min_pcr	–	Yes	–
max_cdv	–	Yes	–
max_sdu	–	Yes	Yes

2.6.3 Peak cell rate

The *minimum peak cell rate* `min_pcr` and the *maximum peak cell rate* `max_pcr` specify the bandwidth that will be consumed by this connection. They are counted in cells per second.

Hardware normally does not allow to set the peak cell rate (PCR) at a resolution of one cell per second. It will therefore choose a rate according to the following rules:

- if only `min_pcr` is specified (i.e. if it is non-zero), the next possible rate above `min_pcr` is used
- if only `max_pcr` is specified (i.e. if it is neither zero nor `ATM_MAX_PCR`), the next possible rate below the lower value of `max_pcr` and the remaining bandwidth on the link is used
- if both `min_pcr` and `max_pcr` are specified, the next possible rate above `min_pcr` is used

The operation fails if the selected rate would exceed either the remaining bandwidth or the maximum PCR (if specified).

For sending, the traffic shapers are adjusted to never exceed the selected cell rate. For receiving, the receiver prepares to accept cells arriving at least at this rate.

The minimum PCR normally is zero or a positive integer. The maximum PCR normally is a positive integer. The special value `ATM_MAX_PCR` indicates an unlimited cell rate (i.e. link speed). `ATM_MAX_PCR` is never used by the operating system for returning a cell rate. The maximum PCR is ignore if set to zero. `min_pcr` and `max_pcr` are ignored when using UBR.

2.6.4 Cell delay variation

The *maximum cell delay variation* (CDV) `max_cdv` specifies the upper bound for the maximum number of microseconds a cell can arrive ahead of its due time, relative to the preceding cell of the same VC,[9] i.e. the operating system may select a lower CDV than requested in `max_cdv`. When sending, cells will never be emitted at a pace exceeding this guarantee. For receiving, the receiver prepares to accept cells arriving with at least the specified CDV. The maximum CDV normally is a positive integer. Setting it to zero indicates an unspecified CDV. `max_cdv` is ignored when using UBR.

2.6.5 SDU size

The *maximum service data unit* (SDU) *size* specifies the maximum amount of data (counted in bytes) the API user will attempt to send at a time, i.e. with a single system call. For receiving and for sending, buffers are dimensioned accordingly. The SDU size is a positive integer. If set to zero, the maximum SDU size supported by the respective protocol is assumed. The operating system never returns a maximum SDU size of zero. Because this may lead to wasteful allocation of resources, `max_sdu` should always be set if the maximum message size is known or if it is implied by the application.

2.7 Library functions

A set of library functions is provided to encode, decode, and compare addresses and to convert SDU rates to cell rates.

In order to use the library, the header file `/usr/include/atm.h` has to be included. In addition to that, the library `libatm.a` has to be linked in.

Note that `text2atm` and `atm2text` might be changed in the future for better alignment with the ANS API. (See also [5].)

[9] This definition of CDV is based on the "peak cell rate monitor algorithms accounting for cell delay variation tolerance" described in annex A of [9].

2.7.1 text2atm

text2atm converts a textual presentation of a PVC or SVC address to the corresponding binary encoding.

```
int text2atm(const char *text,struct sockaddr *addr,int length,int flags);
```

text points to a NUL-terminated string containing the textual representation of the address. addr points to a data structure large enough to hold the resulting address structure, which can be either a struct sockaddr_atmpvc or a struct sockaddr_atmsvc. length indicates the length of the data structure. The conversion fails if not enough space is provided. flags is a set of processing options:

```
#define T2A_PVC          1
#define T2A_SVC          2
#define T2A_UNSPEC       4
#define T2A_WILDCARD     8
#define T2A_NNI         16
#define T2A_NAME        32
```

T2A_PVC and T2A_SVC enable the corresponding address formats. If neither of them is set, the presence of both is assumed. T2A_UNSPEC and T2A_WILDCARD allow the use of unspecified or wildcard parts, respectively, in PVC addresses. T2A_WILDCARD can also be used for SVC addresses, where it means to allow extensions of the form /*length*. T2A_NNI allows the use of 12 bit VPI values.[10] T2A_NAME allows further resolution using a directory service. Resolution based on numeric information takes precedence over directory lookups.

text2atm returns the length of the NSAP part on success (0 if there is no NSAP part, 160 if there is one, or something in between if using wildcards), a negative integer on failure. The definition of more detailed failure indications is for further study.

On success, text2atm initializes the following fields in PVCs: sap_family and everything in sap_addr. In SVCs, text2atm initializes sas_family, sas_addr.prv, and sas_addr.pub. Note that sas_addr.blli and sas_addr.bhli are not touched. In case of failure, text2atm may clobber the whole address structure. It is therefore recommended to initialize all other fields *after* calling text2atm.

2.7.2 atm2text

atm2text performs the reverse operation of text2atm: it converts a binary encoded ATM address to its textual representation.

```
int atm2text(char *buffer,int length,const struct sockaddr *addr,int flags);
```

buffer points to the buffer where the resulting string will be stored. length indicates the size of the buffer. It must be large enough to hold the entire string, plus the terminating NUL. MAX_ATM_ADDR_LEN is the maximum length of a string (excluding the terminating NUL) atm2text may generate. addr points to the address structure to convert. flags is a set of processing options:

```
#define A2T_PRETTY       1
#define A2T_NAME         2
```

A2T_PRETTY enables the addition of periods and the use of the colon notation for NSAP-embedded E.164 addresses. A2T_NAME enables directory lookups to translate the address to a name.

atm2text returns the length of the resulting string (without the terminating NUL) on success, a negative integer on failure. The definition of more detailed failure indications is for further study.

[10] Note that such VPIs values are only allowed at the NNI, not at the UNI.

2.7.3 atm_equal

atm_equal is used to test ATM addresses for equality:

```
int atm_equal(const struct sockaddr_atmsvc *a,const struct sockaddr_atmsvc *b,int len,int flags);
```

a and b point to the two addresses to compare. len is the length (in bits) of the prefix of the NSAP address to consider when using wildcards. flags is a set of processing options:

```
#define AXE_WILDCARD     1
#define AXE_PRVOPT       2
```

If AXE_WILDCARD is set, only len bits of an NSAP address are compared. Also, for E.164 addresses, only as many digits as are present in the shorter address are compared. This also applies to NSAP-embedded E.164 addresses. Because of the coding of the latter, len must be greater or equal than 68.

If AXE_PRVOPT is set, two addresses are still considered to be equal if their public parts match, even if only one of them has a private part. Note that addresses with the same private part are always equal, regardless of the public part.

atm_equal returns a non-zero integer if the addresses match, or zero, if they don't, or if no comparison is possible.

2.7.4 sdu2cell

sdu2cell calculates the number of data cells that would be generated by sending a set of SDUs on a given socket.

```
int sdu2cell(int s,int sizes,const int *sdu_size,int *num_sdu);
```

s is the socket descriptor that has previously been returned by the socket system call. sizes is the number of SDU sizes described in the sdu_size and num_sdu arrays. Each element of sdu_size indicates the length (in bytes) of an SDU. Each element of num_sdu indicates the number of times an SDU with the corresponding size is sent.

sdu2cell computes the total number of ATM data cells that would be sent on the connection and returns either that number or -1 if there is an error (e.g. an overflow). Note that sdu2cell does not check whether the specified SDU sizes are valid for the connection.

2.8 Connection preparation

The connection preparation phase consists of the following parts:

- socket creation
- QOS descriptor initialization
- address descriptor initialization
- traffic parameter setting

Socket creation and *QOS descriptor initialization* must always be performed before *traffic parameter setting*.

2.8.1 Socket creation

Sockets are created with the `socket` system call.

`socket(int domain,int type,int protocol);`

The `domain` indicates whether signaling shall be used for this connection (SVC), or not (PVC). The following domains are defined:

`PF_ATMPVC` ATM PVC connection

`PF_ATMSVC` ATM SVC connection

Merging of both domains into a single `PF_ATM` domain is left for further study.

The `type` selects general transport layer protocol characteristics. Only the `SOCK_DGRAM` transport protocol type is currently available, specifying an unreliable datagram transport. Note that no explicit sequence guarantees are given, although the use of ATM generally implies that sequence will be preserved.

With `protocol`, a specific protocol is selected. Currently, only the protocols `ATM_AAL0` and `ATM_AAL5` are defined, for details see section 3.2. Support of additional protocols is for further study.

2.8.2 Address descriptor allocation and initialization

The address descriptor is a data structure containing the actual address information. See section 2.2 for a detailed description.

Depending on the address family, one of the following data types is used for the connection descriptor:

- `struct sockaddr_atmpvc` for a PVC
- `struct sockaddr_atmsvc` for an SVC

The data structures are defined in `/usr/include/linux/atm.h`:

```
struct sockaddr_atmpvc {
        unsigned short  sap_family;
        struct {
                short   itf;
                short   vpi;
                int     vci;
        }               sap_addr;
        struct atm_trafprm sap_txtp;
        struct atm_trafprm sap_rxtp;
};
```

`sap_family` contains the address family and must be set to `AF_ATMPVC`. `sap_addr` is the PVC address as described in section 2.3. `sap_txtp` and `sap_rxtp` are obsolete and support of them will be discontinued in the future.

```
struct sockaddr_atmsvc {
    unsigned short      sas_family;
    struct {
        unsigned char   prv[ATM_ESA_LEN];
        char            pub[ATM_E164_LEN+1];
        struct atm_blli *blli;
```

```
        struct atm_bhli bhli;
    }                   sas_addr;
    struct atm_trafprm  sas_txtp;
    struct atm_trafprm  sas_rxtp;
};
```

The fields are the same as for PVCs, with the obvious exceptions that the family has to be set to `AF_ATMSVC`, and that the address is an SVC address as described in section 2.4. `sas_txtp` and `sas_rxtp` are obsolete and support of them will be discontinued in the future.

Note that the address structure as described here conforms to the definition of socket address structures of the 4.3 BSD release, which has also been adopted by Linux.[11]

The address descriptor is initialized by setting all bytes to zero using the `memset` C library function. (See [10].) Note that setting all fields to their respective null values (e.g. `0` or `NULL`) is not a valid way to initialize the address descriptor.[12] After that, its fields can be set to the desired values.

2.8.3 QOS descriptor allocation and initialization

The QOS descriptor is a data structure describing traffic properties (e.g. offered data rate, SDU size) and connection requirements (e.g. maximum cell delay variation). See section 2.6 for a detailed description.

The QOS descriptor is initialized by setting all bytes to zero using the `memset` C library function. Note that setting all fields to their respective null values (e.g. `0` or `NULL`) is not a valid way to initialize the QOS descriptor. After that, its fields can be set to the desired values.

2.8.4 QOS parameter setting

The QOS parameters of a connection are set using the `setsockopt` system call. Likewise, they can be queried with `getsockopt`.

For setting and for querying QOS parameters, the level `SOL_ATM` and the option name `SO_ATMQOS` are used. The option value is a `struct atm_qos` (see section 2.6), the option length is its size in bytes.

Merely setting the QOS parameters does not allocate any resources or check their availability.

Note that QOS parameters must be set **before** actually trying to establish a connection or to listen for incoming calls. It is an error to attempt such an operation without successfully setting the QOS parameters before.

The QOS parameters may be altered any number of times before issuing a call to `connect` or `bind`, but they must not be changed when connection establishment is in progress and after the connection has been established. Modification of QOS parameters for active calls may be supported in the future.

2.8.5 Example

```
struct sockaddr_atmpvc addr;
struct atm_qos qos;
int s;

if ((s = socket(PF_ATMPVC,SOCK_DGRAM,ATM_AAL5)) < 0) {
    perror("socket");
    exit(1);
}
```

[11] 4.4 BSD introduced an additional length field, which is not supported by Linux.

[12] Because new fields, which may be added in the future, would not be initialized.

```
memset(&qos,0,sizeof(qos));
qos.txtp.class = ATM_UBR;
qos.txtp.min_pcr = ATM_MAX_PCR;
qos.txtp.max_sdu = 8192;
qos.rxtp = qos.txtp;
if (setsockopt(s,SOL_ATM,SO_ATMQOS,&qos,sizeof(qos)) < 0) {
    perror("setsockopt(SO_ATMQOS)");
    exit(1);
}
memset(&addr,0,sizeof(addr));
addr.sap_family = AF_ATMPVC;
```

2.9 Connection setup

For PVCs, connection setup consists simply of adding address information to the connection descriptor and invoking `connect` or `bind`. For SVCs, the concept known from INET stream sockets involving `bind`, `connect`, `listen`, and `accept` is used.

2.9.1 System calls for PVC setup

The `connect` and `bind` system calls are used to set up connections. The behaviour of both is identical for PVCs. The following peculiarity has to be noted:

A connection to an incompletely specified address is not yet ready to transport data after calling `connect` for the first time; it must be connected or bound a second time without any unspecified components. On this second call, only previously unspecified address components are used (i.e. the values of all others are ignored). It is an error (yielding unspecified behaviour) if there are still unspecified components left. The first call to `connect` may reserve some or all of the resources that have been explicitly specified. If no second `connect` is to be attempted, the resources have to be released by calling `close`. Performing a `connect` in two steps as described above is called a "two-phase `connect`".

The following values of `errno` have a special meaning when using `connect` or `bind` on ATM PVC sockets:

`EADDRNOTAVAIL` the requested address cannot be assigned on the existing interface(s) in the present configuration.

`EOPNOTSUPP` on the second call, either address parts already specified in the first call have been changed or there are still unspecified address parts.

`ENETDOWN` the specified interface exists but is currently not operational.

`ENETUNREACH` the requested QOS criteria could not be met.

2.9.2 System calls for active SVC setup

The address of the caller (i.e. the local address) can optionally be set with the `bind` system call. Traffic parameters and high- and low-layer information specified in the call to `bind` are ignored. The address must correspond to one of the addresses the system recognizes as local. If a public address with a subaddress is specified, only the subaddress is checked. If the socket is not explicitly bound to a local address, an arbitrary local address will be passed in the signaling messages.

The connection is set up with the `connect` system call. A non-blocking `connect` (i.e. if the socket has previously been set to non-blocking) returns immediately with `errno` either set to some error condition or to `EINPROGRESS`, indicating successful initiation of the connection setup. The status of a non-blocking

connect can be queried using select (see section 3.1) or by invoking connect again. In the latter case, a negative value is returned and errno is set to EALREADY if the connection setup is still in progress, or – if the setup has completed or failed – the result is indicated like by a blocking connect.

2.9.3 System calls for passive SVC setup

A passive open is performed in three steps. First, the local SAP is defined using bind. Traffic parameters and SAP information will later be used to select incoming calls. If a local address is specified, only calls with exactly this destination address will be attributed to the socket. (Equality of addresses is determined using atm_equal, see also 2.7.3.) If using a subaddress, the public address is ignored in the comparison.

After the bind, the SAP has to be registered using listen. Note that listen may return an error if the SAP conflicts with another registered SAP.[13]

After an incoming connection has been attributed to a listening socket (which can be determined by using select (section 3.1) or by polling with accept), that connection can be accepted with the accept system call.

2.9.4 Example

```
addr.sap_addr.itf = 0;
addr.sap_addr.vpi = ATM_VPI_UNSPEC;
if (connect(s,(struct sockaddr *) &addr,sizeof(addr)) < 0) {
    perror("connect(1)");
    exit(1);
}
/* some other activities */
addr.sap_addr.vpi = 0;
addr.sap_addr.vci = 42;
if (connect(s,(struct sockaddr *) &addr,sizeof(addr)) < 0) {
    perror("connect(2)");
    exit(1);
}
```

2.10 Connection status

The addresses of the parties communicating over a socket can be queried using the system calls getsockname and getpeername.

The local address can be obtained with getsockname after a successful bind, or (for SVCs), after a successful accept or connect. The remote address can be obtained with getpeername after bind (or connect) for PVCs, and after accept or connect for SVCs, respectively.

The traffic parameters of a connection can be obtained with getsockopt for the option SO_ATMQOS (see section 2.8.4).

Note that getsockname, getpeername and getsockopt may return partial or invalid information if used prematurely.

2.11 Connection teardown

Two steps are distinguished when connections are torn down:

[13] Unlike INET domain sockets, ATM SVC sockets only occupy SAP "address" space when listening. Therefore, conflicts cannot be detected in the bind system call and bind only verifies general validity of the address.

- connection shutdown
- closing

Connection shutdown stops data transmission in either or both directions. Resources associated with a connection are deallocated when *closing* the connection. A connection shutdown is performed implicitly when closing a connection.

2.11.1 Connection shutdown

When a connection is shut down for sending, no further data is accepted for sending. Data can still be received unless the connection has also been shut down for receiving.

When a connection is shut down for receiving, no further data is accepted from the network. Data can still be sent unless the connection has also been shut down for sending.

The `shutdown` system call is used to shut down connections.

Whether and how connection shutdown also implies releasing of resources allocated to the connection is defined by the implementation.

2.11.2 Closing

When a connection has been closed, no further access to it is possible and all resources associated with it are freed.

Connections are closed by closing all sockets referencing them with the `close` system call or by terminating the processes owning the sockets.

A socket can be closed in any state, except if it has not yet been created or if it has already been closed before.

2.11.3 Example

```
(void) close(s);
```

2.12 Connection control summary

All steps in connection handling are summarized below along with the principal system calls or library functions used in each step.

- connection preparation
  - socket creation (`socket`)
  - connection descriptor initialization (`memset`)
  - QOS descriptor initialization (`memset`)
  - traffic parameter specification (assignments)
  - traffic parameter setting (`setsockopt`)
  - address descriptor initialization (`memset`)
- connection setup
  - PVC
    - addressing (assignments)

    - optional: partial setup (`bind`, `connect`)
    - address completion (assignments)
    - full setup (`bind`, `connect`)
    - optional: check traffic parameters (`getsockopt`)
  - SVC, active open
    - optional: binding of local side (addressing and `bind`)
    - addressing of remote side (assignments)
    - connection setup (`connect`)
    - optional: polling of non-blocking `connect` with `select`
    - optional: check traffic parameters (`getsockopt`)
  - SVC, passive open
    - addressing of local side (assignments)
    - binding of local side (`bind`)
    - SAP registration (`listen`)
    - optional: polling with `select`
    - connection acceptance (`accept`)
- data exchange (see section 3)
  - transfer scheduling (`select`)
  - sending and receiving (`read`, `write`, ...)
  - alignment and size constraints (`getsockopt`)
- connection teardown
  - connection shutdown (`shutdown`)
  - closing (`close`)

3 Data exchange

The data exchange paradigm for ATM sockets is modeled as closely as possible after the one for BSD sockets. The only significant exceptions are addresses, and additional buffer alignment and size constraints which apply when optimizing for throughput.

3.1 Transfer scheduling

The `select` system call can be used to schedule receive and send operations in order to minimize blocking delays.

If `select` indicates that an ATM socket is readable, this means that at least one subsequent read operation will succeed on it without blocking. If the remote side closes a socket, that socket becomes readable but will return zero, indicating EOF. `select` also indicates readability if a listening socket has incoming connections to accept.

If `select` indicates that an ATM socket is writable, this means that at least one write operation of up to `max_sdu` bytes[14] on it will succeed without blocking.[15] `select` also indicates writability if a non-blocking `accept` or `connect` has succeeded.

`select` indicates an exception if a non-blocking `connect` has failed.

3.2 Sending and receiving

The system calls `read`, `readv`, `recv`, `recvfrom`, `recvmsg`, `send`, `sendto`, `sendmsg`, `write`, and `writev` are supported with their usual semantics. (See [11] and [1].) Note that, however, the sockets have to be connected and that the arguments of `sendto`, `recvfrom`, `sendmsg`, and `recvmsg` specifying a source or destination address must be `NULL`.

3.2.1 AAL5

The use of AAL5 as defined in section 6 of [12] is requested by setting the `protocol` argument of the `socket` system call to `ATM_AAL5`. AAL5 supports message sizes from 1 to 65535 bytes.

3.2.2 "AAL0" or "raw" cells

The use of "AAL0" is requested by setting the `protocol` argument of the `socket` system call to `ATM_AAL0`. No AAL layer processing is performed on raw cells. Raw cells are presented to the application as four bytes containing the cell header, followed by `ATM_CELL_PAYLOAD` (48) bytes containing the cell payload. The header checksum (HEC) is not visible for the application. The constant `ATM_AAL0_SDU` (52) is defined in `linux/atm.h`.

The cell structure is described in section 3.3 of [3] or in [13].

The following set of macros is defined in `/usr/include/linux/atm.h` to access fields in the cell header if copied to an unsigned integer of appropriate size and if the byte order corresponds to host byte order (i.e. the kernel returns cell headers already with the correct byte order):

```
#define ATM_HDR_GFC_MASK        0xf0000000
#define ATM_HDR_GFC_SHIFT       28
#define ATM_HDR_VPI_MASK        0x0ff00000
```

[14] Or the respective default size if `max_sdu` is not defined or not applicable.

[15] This does not imply that multiple write operations with the same total number of bytes written would not block, since some buffer space may also be allocated per queued SDU.

```
#define ATM_HDR_VPI_SHIFT       20
#define ATM_HDR_VCI_MASK        0x000ffff0
#define ATM_HDR_VCI_SHIFT       4
#define ATM_HDR_PTI_MASK        0x0000000e
#define ATM_HDR_PTI_SHIFT       1
#define ATM_HDR_CLP             0x00000001
```

An additional set of macros defines possible values for the payload type identifier (PTI):

```
#define ATM_PTI_US0     0  /* user data cell, congestion not exp, SDU-type 0 */
#define ATM_PTI_US1     1  /* user data cell, congestion not exp, SDU-type 1 */
#define ATM_PTI_UCES0   2  /* user data cell, cong. experienced, SDU-type 0 */
#define ATM_PTI_UCES1   3  /* user data cell, cong. experienced, SDU-type 1 */
#define ATM_PTI_SEGF5   4  /* segment OAM F5 flow related cell */
#define ATM_PTI_E2EF5   5  /* end-to-end OAM F5 flow related cell */
#define ATM_PTI_RSV_RM  6  /* reserved for traffic control/resource mgmt */
#define ATM_PTI_RSV     7  /* reserved */
```

When sending or receiving AAL0 packets, the application must specify a buffer size of exactly `ATM_AAL0_SDU` (52) bytes. When sending, the application must set the VCI and VPI fields to the connection identifier the VC is bound to.

3.3 Alignment and size constraints

In order to optimize throughput, specific buffer alignment and size considerations may be necessary. This information can be used to adapt the send and receive procedures.

Buffer constraints can be obtained with the `getsockopt` system call:

```
int getsockopt(int s,int level,int optname,void *optval,int *optlen);
```

The `level` is `SOL_SOCKET`, the following values for `optname` are recognized (each parameter exists in the send and in the receive direction):

`SO_BCTXOPT` constraints for sending data with best throughput

`SO_BCRXOPT` constraints for receiving data with best throughput

`optval` is a pointer to a data structure of type `struct atm_buffconst` with the following fields:

```
struct atm_buffconst {
        unsigned long   buf_fac;
        unsigned long   buf_off;
        unsigned long   size_fac;
        unsigned long   size_off;
        unsigned long   min_size;
        unsigned long   max_size;
};
```

`buf_fac` is the factor which the buffer address (minus `buf_off`) should be an integer multiple of. `size_fac` times an integer plus `size_off` should yield the SDU size. `min_size` and `max_size` are the respective proposed limits for the SDU size.

The contents of all fields are stored in host byte order and they must have non-negative values. A *maximum size* only limited by the general system architecture or by quotas is coded as zero. The following relations are true:

$$\texttt{buf_off} < \texttt{buf_fac}$$
$$\texttt{size_off} < \texttt{size_fac}$$
$$\texttt{min_size} \leq \texttt{max_size}^{16}$$
$$0 < \texttt{size_fac} \leq \texttt{max_size} - \texttt{min_size}$$
$$\texttt{min_size} = \texttt{size_off} \pmod{\texttt{size_fac}}$$
$$\texttt{max_size} = \texttt{size_off} \pmod{\texttt{size_fac}}^{17}$$

For "AAL0", the following parameters are returned:

size_fac	53
size_off	0
min_size	53
max_size	53

buf_fac and buf_off have implementation-specific values.

3.4 Asynchronous I/O

Support for asynchronous I/O is left for further study.

3.5 Example

```
const char msg[] = "Hello, world !\n";
char *buffer,*start;
struct atm_buffconst bc;
ptrdiff_t pos;
size_t length,buf_len;
ssize_t size;

length = sizeof(bc);
if (getsockopt(s,SOL_SOCKET,SO_BCTXOPT,(char *) &bc,&length) < 0) {
    perror("getsockopt");
    exit(1);
}
buf_len = sizeof(msg)-bc.size_off+bc.size_fac-1;
buf_len = buf_len-(buf_len % bc.size_fac)+bc.size_off;
if (buf_len < bc.min_size) buf_len = bc.min_size;
if (!(buffer = malloc(buf_len+bc.size_fac-1))) {
    perror("malloc");
    exit(1);
}
pos = (ptrdiff_t) (buffer-bc.buf_off+bc.buf_fac-1);
start = (char *) (pos-(pos % bc.buf_fac)+bc.buf_off);
if (sizeof(msg) != buf_len)
    memset(start+sizeof(msg),0,buf_len-sizeof(msg));
if ((size = write(s,start,buf_len)) < 0) {
    perror("write");
    exit(1);
}
if (size != buf_len)
```

[16] Unless the maximum size is coded as zero, in which case the minimum size can have any value.
[17] Unless the maximum size is coded as zero, in which case the size alignment is not constrained by the maximum size.

```
fprintf(stderr,"Wrote only %d of %d bytes\n",size,
  sizeof(msg));
```

4 Administrative functions

The section is still under construction ...

Certain interface and low-layer parameters can be modified in some implementations. In addition to that, statistics of important system events can be queried.

4.1 ioctl arguments

Ioctls that exchange a variable amount of data with the kernel but which don't apply to a specific ATM interface use `struct atm_iobuf` to describe the data area. `struct atm_iobuf` is defined in `/usr/include/linux/atmdev.h`:

```
struct atm_iobuf {
        int length;
        void *buffer;
};
```

When sending data to the kernel or when returning data from the kernel, `length` is set to the length of the actual data. When requesting data from the kernel, `length` is set to the size of the buffer. `buffer` points to the data area.

All ioctls that apply to a specific ATM interface use `struct atmif_sioc` to exchange parameters with the kernel. `struct atmif_sioc` is defined in `/usr/include/linux/atm.h`:

```
struct atmif_sioc {
    int number;
    int length;
    void *arg;
};
```

`number` contains the interface number. `length` is used in the same way as in `struct atm_iobuf`. `arg` points to the data area.

4.2 Interface configuration

Unless indicated otherwise, interface configuration ioctls can be applied to PVC and to SVC sockets. All the ioctls and related data structures are defined in `/usr/include/linux/atmdev.h`.

4.2.1 List of available ATM interfaces

The list of available ATM interfaces can be obtained with the `ATM_GETNAMES` ioctl. In its argument of type `struct atm_iobuf`, it returns a list of integers with the interface numbers. The return value is the number of interface numbers that have been returned, or a negative number in case of an error.

4.2.2 Interface attributes

The type name of a specific interface can be queried using the `ATM_GETTYPE` ioctl. This ioctl takes an argument of type `struct atmif_sioc` and returns the NUL-terminated interface type name. The maximum length of a type name (without the terminating NUL byte) is defined to be `ATM_ITFTYP_LEN`.

The end system identifier (ESI) of an interface can be queried using the `ATM_GETESI` ioctl. This ioctl takes an argument of type `struct atmif_sioc` and returns the ESI value. The length of an ESI is defined to be `ESI_LEN` (six bytes).

4.2.3 Connection identifier ranges

The number of bits that are available for VPI and VCI values on an interface can be queried and set with the ioctls `ATM_GETCIRANGE` and `ATM_SETCIRANGE`, respectively. Both ioctls operate on a `struct atm_cirange`:

```
struct atm_cirange {
        char    vpi_bits;
        char    vci_bits;
};
```

Setting either value to `ATM_CI_MAX` when invoking `ATM_SETCIRANGE` allocates the maximum number of bits available. If both fields are set to `ATM_CI_MAX`, the maximum number of VCI bits is allocated and the remaining bits are used for VPIs.

Note that an ATM device driver may put arbitrary restrictions on what values it accepts for `ATM_SETCIRANGE`.

4.2.4 AAL level statistics

Per-interface AAL level statistics can be obtained with the `ATM_GETSTAT` and `ATM_GETSTATZ` ioctls. They differ in that `ATM_GETSTATZ` zeroes the statistics counters after obtaining their current values. It therefore is only available to privileged users.

Both ioctls take a `struct atmif_sioc` as their argument. The ioctls return a `struct atm_dev_stats`:

```
struct atm_dev_stats {
        struct atm_aal_stats aal0;
        struct atm_aal_stats aal34;
        struct atm_aal_stats aal5;
};
```

This structure contains one block of statistics counter values for each AAL that is supported by the implementation. Each block contains the following fields:

```
struct atm_aal_stats {
        long            tx,tx_err;
        long            rx,rx_err;
        long            rx_drop;
};
```

`tx` and `rx` count the number of successfully transmitted PDUs. `tx_err` and `rx_err` count the number of PDU whose transmission had failed due to some error condition. `rx_drop` counts the number of PDUs that were dropped because no memory was available for storing them on the local host.

4.2.5 Address configuration

A variable-length list of ATM addresses can be configured per interface. This list can be manipulated with the following ioctls. They all take a `struct atmif_sioc` as their argument and they can only be applied to SVC sockets. ATM addresses are always expressed as a `struct sockaddr_atmsvc`.

`ATM_GETADDR` returns the list of addresses of the specified interface. The return value of `ATM_GETADDR` is the number of bytes that have been written, or a negative integer in case of an error.

`ATM_RSTADDR` resets the list of ATM addresses, i.e. it removes all addresses.

`ATM_ADDADDR` adds the specified ATM address at the end of the list of local addresses. The address is checked for validity. Addition of addresses that are already in the list is refused.

`ATM_DELADDR` deletes the specified ATM address from the list of local addresses. Note that the order of the remaining addresses is preserved.

All address configuration ioctls but `ATM_GETADDR` are only available to privileged users.

4.3 Connection configuration

Various lower-layer attributes can be set or queried on a per-connection basis.

4.3.1 AAL type

The AAL type of a socket can be queried with the `SO_AALTYPE` socket option at level `SOL_AAL`. All the necessary definitions are in `/usr/include/linux/atm.h`. The option value is an integer containing the AAL type number. Note that the AAL type can only be read, not set.

4.3.2 Cell loss priority

The cell loss priority of cells being sent on a VC can be set with the `SO_SETCLP` socket option at level `SOL_ATM`. The option value is an integer which can be either 0 or 1, indicating the value used in the cell loss priority field of ATM cells. `SO_SETCLP` is defined in `/usr/include/linux/atm.h`.

This option is not yet implemented.

4.3.3 Receive timestamp

The arrival time of the SDU that has most recently been read from the socket can be obtained with the `SIOCGSTAMP` ioctl. It returns the time in a `struct timeval`. An error is returned if no time measurements are available.

Note that the accuracy of timestamps depends significantly on the capabilities of the ATM adapter hardware and the device driver.

4.4 Physical layer

Not yet documented

```
SONET_GETSTAT
SONET_GETSTATZ
SONET_SETDIAG
SONET_CLRDIAG
SONET_GETDIAG
SONET_SETFRAMING
SONET_GETFRAMING
SONET_GETFRSENSE
```

4.5 /proc

Not yet documented

5 Related services

The section is still under construction ...

This section describes support of services not defined in [3], which are commonly associated with ATM.

5.1 IP over ATM

This section is incomplete

API details for IP over ATM support (as specified in [14], [15], [16], and [17]) are for further study.

```
CLIP_NULENCAP
CLIP_LLCENCAP
```

5.2 Arequipa

The following library functions are provided in `/usr/include/arequipa.h` to support Arequipa (see [21]):

`int arequipa_preset(int sd,const struct sockaddr_atmsvc *addr,const struct atm_qos *qos);`

Presets the specified INET domain socket to use a direct ATM connection to `addr` with the QOS parameters specified in `qos`. If the socket is already connected, the ATM connection is set up immediately and data is redirected to flow over that connection.

`int arequipa_expect(int sd,int on);`

Enables (if `on` is non-zero) or disables (if `on` is zero) the use of Arequipa for return traffic on the specified INET domain socket. When enabling the use of Arequipa for return traffic, the Arequipa connection on which the next data packet or incoming connection for the socket is received is attached to that socket.

`int arequipa_close(int sd);`

Dissociates an Arequipa VC from the specified socket. After that, traffic uses normal IP routing. Note that the Arequipa connection is automatically closed when the INET socket is closed.

A Acronyms

This appendix lists some of the acronyms appearing in this document:

AAL ATM Adaption Layer

ABR Available Bit Rate

ABT ATM Block Transfer

API Application Program Interface

ATM Asynchronous Transfer Mode

BSD Berkeley Software Distribution

CBR Constant Bit Rate

CDV Cell Delay Variation

NNI Network Node Interface

NSAP Network Service Access Point

PCR Peak Cell Rate

PVC Permanent Virtual Circuit (see below)

QOS Quality Of Service

SAP Service Access Point

SDU Service Data Unit

SVC Switched Virtual Circuit (see below)

UBR Unassigned Bit Rate

UNI User-Network Interface

VBR Variable Bit Rate

VC Virtual Channel

VCI Virtual Channel Identifier

VPI Virtual Path Identifier

Note that the terms "PVC" and "SVC" are not official ATM terminology as used by ITU. They originate from Frame Relay terminology and it is common practice by ATM Forum and other groups to use them to describe the corresponding concepts in ATM. Unfortunately, ITU-T has re-used the abbreviation "SVC" for ATM to mean "Signalling Virtual Channel", which is the VC used to carry signaling messages.

References

[1] Stevens, W. Richard. *UNIX Network Programming*, Prentice-Hall, 1990.

[2] The ATM Forum. *ATM User-Network Interface Specification, Version 3.0*, Prentice Hall, 1993.

[3] The ATM Forum. *ATM User-Network Interface (UNI) Specification, Version 3.1*, `ftp://ftp.atmforum.com/pub/UNI/ver3.1`, Prentice Hall, 1994.

[4] The ATM Forum, SAA API Ad-hoc Work Group. *Native ATM Services: Semantic Description Version 1.0*, ATM Forum contribution 95-0008, January 1996.

[5] The ATM Forum, SAA/Directory Work group. *ATM Name Service (ANS) Specification Version 1.0*, ATM Forum contribution 95-1532R2, April 1996.

[6] Gilligan, Robert E.; Thomson, Susan; Bound, Jim. *IPv6 Program Interfaces for BSD Systems* (work in progress), Internet Draft `draft-ietf-ipngwg-bsd-api-05.txt`, April 1996.

[7] Almesberger, Werner. *Linux ATM device driver interface*, `ftp://lrcftp.epfl.ch/pub/linux/atm/docs/`, January 1996.

[8] ITU-T Recommendation E.164/I.331. *Numbering plan for the ISDN era*, ITU, 1991.

[9] ITU-T Recommendation I.371. *Traffic control and congestion control in B-ISDN*, ITU, 1993.

[10] ANSI/ISO 9899-1990; Schildt, Herbert. *The Annotated ANSI C Standard*, Osborne McGraw-Hill, 1990.

[11] IEEE Std 1003.1b-1993; IEEE Standard for Information Technology. *Portable Operating System Interface (POSIX). Part 1: System Application Program Interface (API)*, IEEE, 1994.

[12] ITU-T Recommendation I.363. *B-ISDN ATM adaptation layer (AAL) specification*, ITU, 1993.

[13] ITU-T Recommendation I.361. *B-ISDN ATM layer specification*, ITU, 1993.

[14] RFC1483; Heinanen, Juha. *Multiprotocol Encapsulation over ATM Adaptation Layer 5*, IETF, 1993.

[15] RFC1577; Laubach, Mark. *Classical IP and ARP over ATM*, IETF, 1994.

[16] RFC1626; Atkinson, Randall J. *Default IP MTU for use over ATM AAL5*, IETF, 1994.

[17] RFC1755; Perez, Maryann; Liaw, Fong-Ching; Mankin, Allison; Hoffman, Eric; Grossman, Dan; Malis, Andrew. *ATM Signaling Support for IP over ATM*, IETF, 1995.

[18] ITU-T Recommendation Q.2931. *Broadband Integrated Services Digital Network (B-ISDN) – Digital subscriber signalling system no. 2 (DSS 2) – User-network interface (UNI) – Layer 3 specification for basic call/connection control*, ITU, 1995.

[19] Le Boudec, Jean-Yves. *The Asynchronous Transfer Mode: a tutorial*, Computer Networks and ISDN Systems, Volume 24, Number 4, 1992.

[20] Almesberger, Werner. *ATM on Linux, magic numbers*, `http://lrcwww.epfl.ch/linux-atm/magic.html`

[21] Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Application REQuested IP over ATM (AREQUIPA)* (work in progress), Internet Draft `draft-almesberger-arequipa-01.txt`, June 1996.