Arequipa: Design and Implementation

Werner Almesberger
werner.almesberger@lrc.di.epfl.ch

Laboratoire de Réseaux de Communication (LRC)
EPFL, CH-1015 Lausanne, Switzerland

November 14, 1996

Abstract

After briefly introducing the fundamental concepts of Arequipa, this paper first outlines an approach for implementing Arequipa on a typical Unix system and then gives a detailed description of its implementation in ATM on Linux.

1 Introduction

Arequipa [1] is a mechanism for allowing applications to establish direct connections on lower protocol layers that support such functionality. These connections are used exclusively by the applications that requested them. Direct connections are for instance useful for running Classical IP over ATM [2], because the Classical IP over ATM model assumes that IP packets are routed in the network, which is normally less efficient than switching in a direct ATM connection and which also makes it more difficult to use ATM's support for quality of service guarantees.

Although these problems are addressed and at least partially solved by NHRP [3] and RSVP [4], Arequipa has the advantages of being very easy to implement and of only requiring changes on the systems that want to use it, but not needing any modifications inside the network.

This document presents the general concept of Arequipa, describes how it can be implemented in a typical Unix kernel, and finally discusses the concrete solution that has been implemented on Linux [5].

Note that application issues are not addressed in this document. Please see [6, 1] for a case study. Also, only point-to-point end-to-end scenarios are considered.

2 General concept

In its broadest sense, Arequipa offers a means to use properties of a network technology that is used to transport another network technology (e.g. IP on ATM) without requiring the explicit design and deployment of sophisticated interworking mechanisms and protocols.

Traditional protocol layering typically only allows access to functionality of lower layers if upper layers provide their own means to express that functionality. This approach can introduce significant complexity if the semantics of the respective mechanism are dissimilar. Also, if the upper layer fails to provide that interface, no direct access is possible and the lower layer functionality may be wasted or used in an inefficient way (e.g. if using heuristics to decide on the use of extra features). By allowing applications to control the lower layer, Arequipa enables them to exploit those properties.

Note that Arequipa coexists with "normal" use of the networking stacks, i.e. applications not requiring Arequipa do not need to be modified and they will continue to use whatever other mechanisms are provided.

Example

A typical example is illustrated in figure 1: virtual connections between applications (e.g. TCP) are built by multiplexing their traffic over virtual connections at the upper layer (e.g. IP), which is in turn carried by a lower layer (e.g. Ethernet or ATM). Routers terminate lower layer segments in order to overcome scalability limitations of either layer or of the interface between the layers.

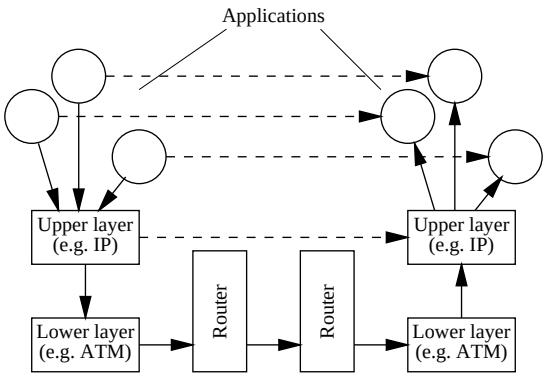

Figure 1: Communication without Arequipa.

Figure 2 shows the same scenario, but this time using only Arequipa. The applications still have their virtual connections, but there is one dedicated end-to-end connection at the lower layer for each of them. Furthermore, the virtual upper layer connection providing a general link between both end systems is no longer required (but may exist if needed for other traffic).

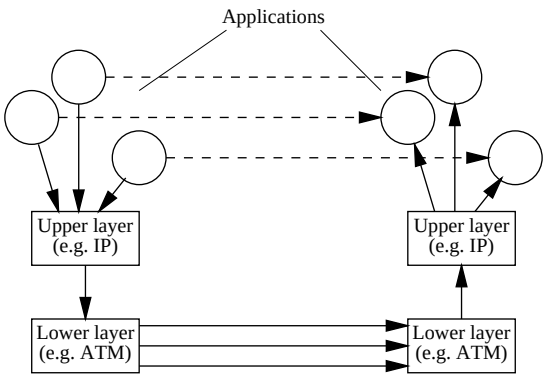

Figure 2: Communication with Arequipa.

Note that general upper layer connectivity may be necessary even if always using Arequipa, e.g. when running TCP/IP over Arequipa, ICMP messages would normally be sent using the default mechanisms.

Applicability

Arequipa is applicable if the following two conditions are met:

- applications can control "native" connections over the lower layer communication media
- the upper and the lower layer both allow communication between the same endpoints (or they share at least a useful common subset of reachable endpoints)

Arequipa could also be applied if the next two conditions are not met, but such an application would be of questionable use:

- the upper layer is multiplexed over the lower layer
- multiple lower layer connections are possible between a pair of endpoints

In order to simplify interaction with the protocol stack, Arequipa assumes that data sent to destinations for which no Arequipa lower layer connection has been established will be delivered by some default mechanism.

Motivation

The main motivations for using Arequipa are:

- if the "default" mechanism for transporting the upper layer over the lower layer does not use end-to-end lower layer connections, it may be desirable to overcome this limitation in order to reduce delays, improve availability, control privacy, etc.
- properties of the lower layer connection that cannot be exploited by the upper layer can be accessed directly if using Arequipa

Note that, despite its name (Application REQuested IP over ATM), Arequipa is not limited to IP and ATM only. The upper layer is typically IP or some similar protocol (e.g. IPX). The lower layer can be ATM, Frame Relay, N-ISDN, etc. Some of the advantages of using Arequipa in addition to the usual IP mechanisms are avoidance of routing overhead and the possibility of using dedicated connections with "hard" quality of service guarantees.

API summary

The following primitives are available to applications using Arequipa (see [6] and [7] for details):

`arequipa_preset` establishes a lower layer connection for a given upper layer socket and associates that connection with the socket

`arequipa_expect` enables or disables automatic use of an incoming Arequipa connection for outbound traffic, when data is delivered from this connection to the socket

`arequipa_close` terminates the association between the upper layer socket and the dedicated lower layer connection and closes the lower layer connection

3 Arequipa with IP over ATM

This section describes general aspects of implementing Arequipa for IP over ATM in a socket-based operating system kernel. The organization of kernel internal data structures is assumed to be similar to the one found in the networking part of the Linux kernel ([8]).

Kernel data structures without Arequipa

Figure 3 shows some of the kernel data structures that are typically associated with a TCP socket when not using Arequipa. Incoming data is demultiplexed by the protocol stack (in figure 3, the circle with TCP/IP) and queued on the socket. Outgoing data is multiplexed by the protocol stack and sent to the corresponding network interface.

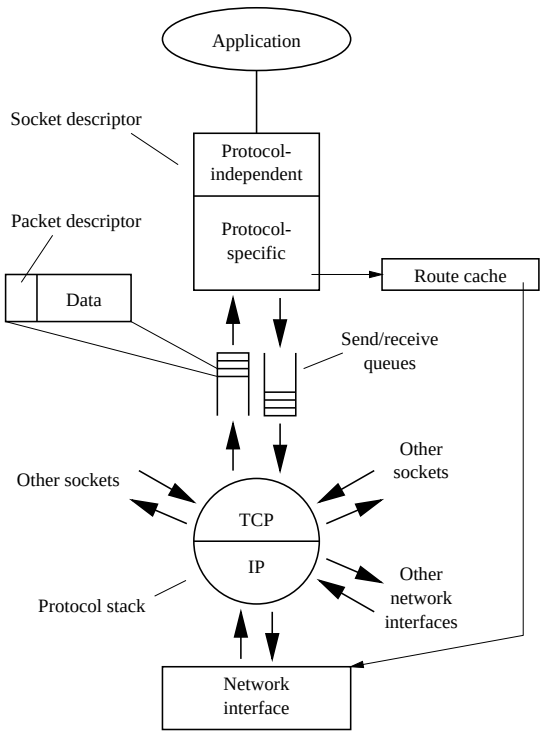

Figure 3: Kernel data structures of a TCP socket (simplified).

Each data packet consists of a packet descriptor and the actual data. The packet descriptor contains information like the socket the packet belongs to, the interface on which it was received, etc.

Most modern TCP/IP implementations also cache routing information (including the network interface) for each socket, so that route lookups only need to be done when a new connection is established or if the routing table is modified.

Data structures for incoming Arequipa

When using Arequipa, incoming packets are handled like when using Classical IP over ATM: after little or no ATM-specific processing, they're passed to the protocol stack, which then performs the usual demultiplexing, etc. The only significant difference is that they are marked in order to identify them as originating from Arequipa (and from

which VC) when they arrive at the socket.

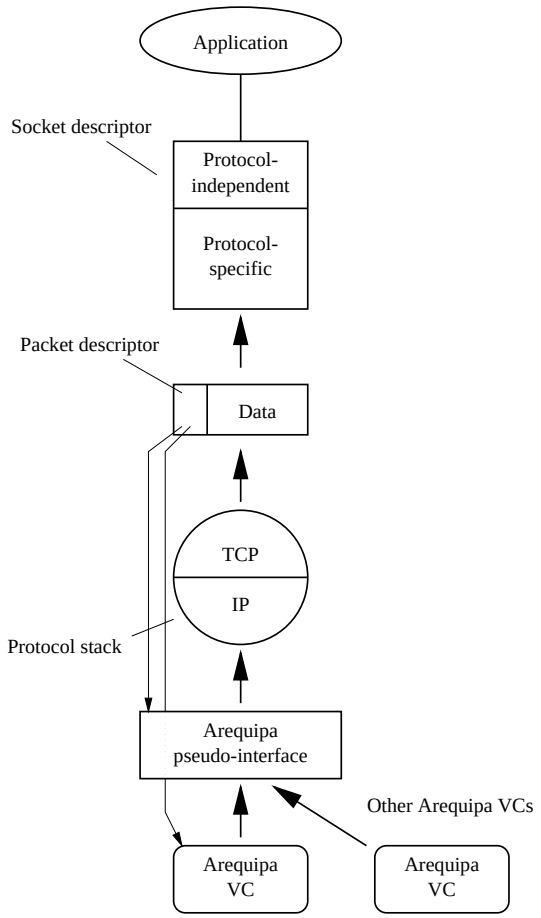

Figure 4: Arequipa for incoming data.

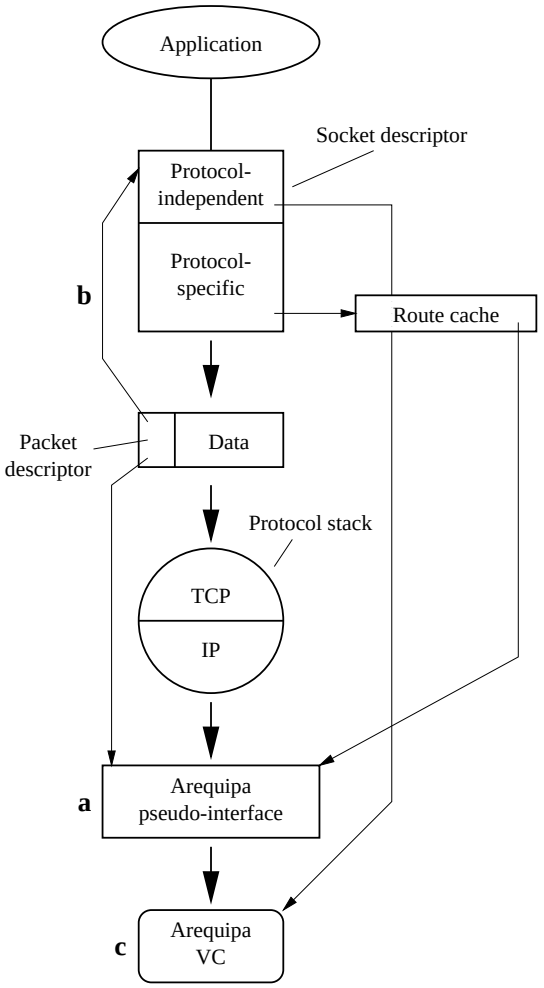

Figure 5: Arequipa for outgoing data.

Figure 4 shows the data structures used when receiving from Arequipa. Note that all Arequipa VCs on a system can use the same Arequipa pseudo-interface.[1]

If the socket is not yet using Arequipa for sending (i.e. if it has no associated Arequipa VC) and if it expects incoming Arequipa traffic (i.e. if `arequipa_expect` has been invoked with the `on` argument set to a non-zero value), the Arequipa VC on which the packet has been received is attached to the socket, so that outbound traffic uses the VC.

[1] The term "pseudo-interface" is used to make it clear that the Arequipa interface does not correspond to a physical network interface (i.e. hardware) although the protocol stack interacts with it as if it did.

The following subtle race condition needs to be considered: if a packet is received over an Arequipa VC, that VC may no longer exist at the time the data is delivered to the socket. It is therefore necessary to verify the validity of the incoming Arequipa VC before attaching it to the upper layer socket (the normal closing procedures only ensure that both layers are synchronized *after* establishing the association).

This can be implemented as follows:

- all incoming Arequipa VCs are registered in a list (while they're dangling, i.e. while not attached)

- there is a global generation number, which is incremented whenever a new Arequipa VC is created. The generation number at the time of VC creation is stored in the VC descriptor.

- the generation number of the Arequipa VC is recorded in the descriptor of each data packet arriving on that VC

A VC is still valid when the packet is delivered to the socket only if that VC is still in the list and if its generation number matches the one stored in the packet descriptor.

Data structures for outgoing Arequipa

When sending from an Arequipa socket, outbound packets must be associated with the corresponding Arequipa VC. As illustrated in figure 5, this is done by sending them all through an Arequipa pseudo-interface (a) which then looks up a back pointer (b) to the originating socket in the packet descriptor. The originating socket contains a pointer to the descriptor (c) of the VC over which the data has to be sent.

Note that an Arequipa connection may be removed (e.g. because the remote party has closed it, because of a network failure, etc.) without notification at the socket. In this case, the Arequipa route is removed and all outbound traffic is sent with the "normal" IP mechanisms again.

Networking code changes

If using the approach outlined in the previous sections, the networking code has to be modified at least at the following places:

- when creating a socket, the Arequipa information (i.e. if Arequipa is in use on that socket, if the socket expects incoming Arequipa traffic, etc.) needs to be initialized

- when connecting a UDP or TCP socket, a cached Arequipa route exists and must be reinstated if `arequipa_preset` was invoked before the `connect` system call

- when delivering data from Arequipa to a socket, the Arequipa VC is attached to the socket if
  - the socket expects incoming Arequipa traffic, and
  - the socket does not currently use Arequipa, and
  - the Arequipa VC is not already attached to a different socket

- if an incoming TCP connection is received on a listening socket which expects incoming Arequipa traffic, the new socket (the one returned by `accept`) is also set to expect incoming Arequipa traffic and, if the packet has arrived via Arequipa and if the constraints listed above are met, the Arequipa VC is attached to the new socket

- when an upper layer socket is closed, the underlying Arequipa connection has to be closed too

- when forwarding IP packets, packets received over an Arequipa connection must be discarded (see [6], section 6)

Additional modifications may be necessary depending on how per-socket route caches are invalidated. Also, socket destruction may be interrupt-driven and may therefore need special care.

TCP issues

The use of TCP over Arequipa raises two specific problems: (1) if the Arequipa connection is attached after establishing the TCP connection, the maximum segment size (MSS) of TCP may be very small, typically increasing processing overhead. (2) there are no generally useful semantics for listening on a socket for which an Arequipa connection has already been set up.

TCP implementations frequently limit the MSS to a value which is based on the MTU of the IP interface on which the connection is started. If connections are set up over a media with an MTU size that is small compared to the default IP over ATM MTU size ([9]), that MSS will have to be kept even if Arequipa is later used for that socket (see RFC1122

[10], section 4.2.2.6). It is therefore recommended to invoke `arequipa_preset` before `connect` and to invoke `arequipa_expect` before `listen`.

Note that this is the only way to ensure that the use of Arequipa is known at both sides when exchanging the initial SYN segments. Applications that require the TCP listener to set up the Arequipa connection are therefore not able to ensure the use of a larger MSS.

Although the API could allow associating an Arequipa connection with a socket that is used to listen for incoming connections, the usefulness of such an operation is questionable.

Therefore, attempts to execute `arequipa_preset` on a listening socket or to `listen` on a socket for which an Arequipa connection already exists yield an error.

4 Arequipa on Linux

This section describes the changes that had to be made to Linux with the ATM on Linux extensions in order to support Arequipa. This description is based on version 0.22 of ATM on Linux and the 2.0.14 kernel.

The explanations in the following sections are meant to be read along with the corresponding source code portions. They may be difficult to understand if read alone.[2]

Data structures

Small changes to several kernel data structures are required (see also figure 6):

- the ATM VC descriptor (`struct atm_vcc` in `include/linux/atmdev.h`) is extended with fields to store a pointer to the upper layer socket (`upper`), a pointer to the own socket (`sock`; for closing), pointers for the list of dangling Arequipa VCs (`aq_next` and `aq_prev`), and the generation number (`generation`). These fields are initialized in `net/atm/common.c:atm_create` and in `net/atm/arequipa.c:make_aq_vcc`.
- two new ATM VC flags are added: `ATM_VF_AQREL` is set if third party closing is in progress. `ATM_VF_AQDANG` is set while the VC descriptor is in the dangling Arequipa VC list.
- a field for the generation number (`generation`) is added to the socket buffer descriptor (`struct sk_buff` in `include/linux/skbuff.h`).
- the protocol-specific socket descriptor (`struct sock` in `include/net/sock.h`) is extended with a field to store a pointer to that socket's copy of the Arequipa route (`aq_route`) and a pointer to the lower layer socket (`arequipa`). These fields are initialized in `net/ipv4/af_inet.c:inet_create`.

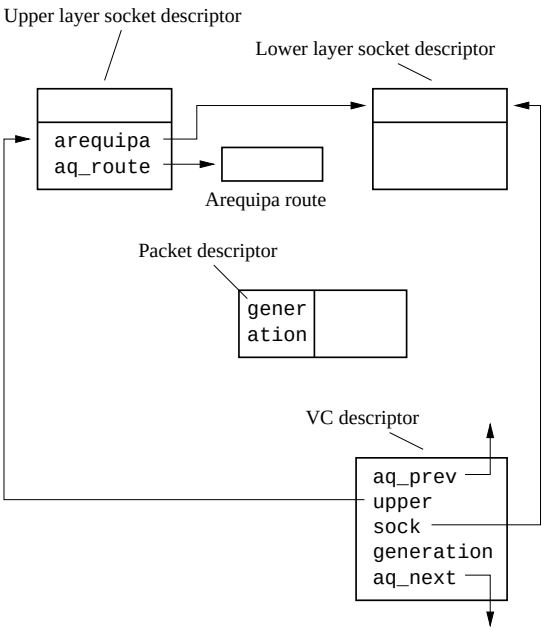

Figure 6: Arequipa-specific fields in kernel data structures.

`sk->arequipa` is `NULL` if no Arequipa VC is attached. `aq_route` is `NULL` if the socket has no attached Arequipa VC and if it doesn't expect incoming Arequipa traffic. If `aq_route` is

[2] The ATM on Linux distribution can be obtained from `ftp://lrcftp.epfl.ch/pub/linux/atm/dist/`. The kernel source tree can be obtained from `ftp://ftp.funet.fi/pub/Linux/kernel/src/v2.0/`.

non-NULL, sk->aq_route == sk->ip_route_cache means that an Arequipa VC is attached. Otherwise, incoming Arequipa traffic is expected, but no VC is attached yet.

Route cache handling

The route cache is handled in the following way:

- when an Arequipa VC is attached to a socket, the route cache is initialized with a fake route that points to the Arequipa pseudo-interface
- the Arequipa route is re-established after any change to the route cache
- unnecessary route cache changes (e.g. after routing table updates) are prevented
- when destroying a socket with an Arequipa route cache entry, that entry is deallocated too

A template for the Arequipa route is contained in net/atm/arequipa.c:arequipa_rt. Its rt_dev entry it set to net/atm/arequipa.c:arequipa_dev when Arequipa is initialized. In each copy of the route, the source address (rt_src) is set to the source address that would be used with the corresponding non-Arequipa route.

Places where the route cache is initialized (sk->ip_route_cache = rt;) are changed to call a new function include/net/route.h:set_rt_cache, which copies and adapts the route cache template, and then releases the non-Arequipa route (rt).

If connect is invoked after arequipa_preset, the Arequipa route will not contain a valid source address and therefore needs to be updated. This is done in net/ipv4/tcp.c:tcp_connect.

include/net/route.h:ip_check_route checks if a given route is still valid and computes a new route if it isn't. For Arequipa, a test is added that always returns the old route if it points to the Arequipa pseudo-interface.

Arequipa routes are removed in net/ipv4/af_inet.c:destroy_sock. Note that instead of decrementing the reference count, as done for non-Arequipa routes, the buffer space is freed.

Arequipa socket operations

Arequipa sockets are created by the application with the library function arequipa_preset or by the Arequipa demon (a demon process that accepts incoming Arequipa calls and hands them to the kernel) with the AREQUIPA_INCOMING ioctl. INET domain sockets are prepared for using Arequipa with the library function arequipa_expect. arequipa_preset and arequipa_expect invoke their corresponding counterpart in net/atm/arequipa.c. AREQUIPA_INCOMING invokes net/atm/arequipa.c:arequipa_incoming.

net/atm/arequipa.c:arequipa_expect either allocates space for the Arequipa route (if enabling) or it frees that space (if disabling). In both cases, additional sanity checks are performed.

net/atm/arequipa.c:arequipa_preset performs the usual sanity checks and then invokes net/atm/arequipa.c:arequipa_expect to allocate the route cache space. After that, it tries to attach the VC to the upper layer socket using net/atm/arequipa.c:arequipa_attach_unchecked and finally enables Arequipa use (i.e. LLC/SNAP encapsulation) by invoking net/atm/arequipa.c:make_aq_vcc.

net/atm/arequipa.c:arequipa_incoming performs some sanity checks and enables Arequipa use by calling net/atm/arequipa.c:make_aq_vcc.

One special action of make_aq_vcc is to increment the usage count of the file descriptor. This allows the calling process to close the socket after indicating Arequipa use. The internal data structures of that socket will continue to exist until the VC is closed using fs/open.c:close_fp (see below).

Attaching Arequipa

Arequipa sockets are attached either explicitly by calling arequipa_preset (see the previous section) or implicitly by receiving a packet on a socket that expects incoming Arequipa traffic.

Implicit attaching can occur when data is delivered by TCP or by UDP (in net/ipv4/tcp_input.c:tcp_rcv and net/ipv4/udp.c:udp_deliver, respectively). In either case, net/atm/arequipa.c:arequipa_attach is called, which first verifies that

the Arequipa VC is still valid (see section 3) and then proceeds by invoking `net/atm/arequipa.c:arequipa_attach_unchecked` (see above).

Another situation where implicit attaching occurs is when a listening socket on which incoming Arequipa traffic is expected receives a SYN segment that opens a new TCP connection (`net/ipv4/tcp_input.c:tcp_conn_request`). In this case, the buffer for the Arequipa route is allocated for the new socket and `net/atm/arequipa.c:arequipa_attach` is called.

Closing Arequipa

Closing an Arequipa VC may seem simple, but it is in fact a rather tricky operation. Arequipa VCs are closed in the following situations:

- the upper layer protocol destroys the socket
- the Arequipa VC is explicitly closed with `arequipa_close`
- the Arequipa VC is closed by the remote party or by the network

Closing by the upper layer is the most interesting case, because it may be initiated by an interrupt, e.g. when a TCP socket is closed by a timer event after waiting 2·MSL in state TIME_WAIT. From an interrupt, the Arequipa VC can be detached from the upper-layer socket, but it can't be closed, because 1) the SVC close operation may have to wait for a confirmation from the signaling demon (see also [11]) and 2) the ATM device driver's `close` function is allowed to sleep too (see [12]).

The approach chosen is therefore to only detach the Arequipa VC from the upper layer socket but to delegate the actual closing to a user-mode process. The obvious choice for this process is the Arequipa demon `arequipad`. The close request message can be sent without sleeping, so this operation is possible even from interrupts.

The only remaining problem is that ordinary processes can't just close arbitrary sockets (which may belong to other processes, etc.). This was solved by adding an ioctl (`AREQUIPA_CLS3RD`) for closing an ATM VC by the pointer to its descriptor. Invocation of this ioctl is restricted to privileged users. `arequipad` performs this third-party close procedure whenever it receives a close request from the kernel.

With these mechanisms in place, the three close operations can be implemented as follows:

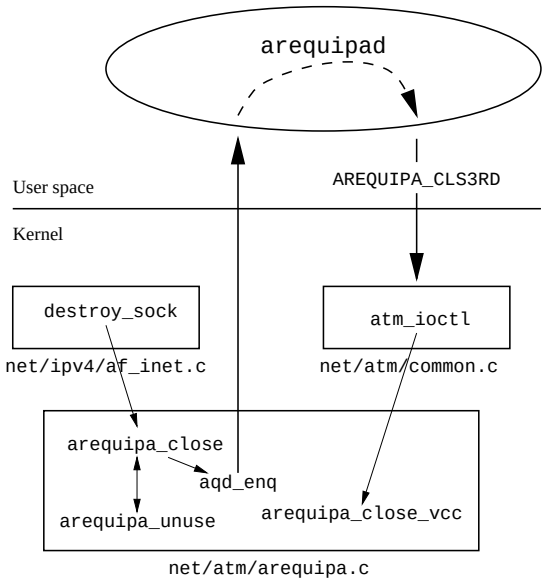

Figure 7: Closing of Arequipa sockets via a third party.

When an upper layer socket is destroyed (`net/ipv4/af_inet.c:destroy_sock`, which calls `net/atm/arequipa.c:arequipa_close`), the Arequipa VC is detached and marked as no longer in use (`net/atm/arequipa.c:arequipa_unuse`). Unless closing has already been initiated for some other reason, a close request is sent to the Arequipa demon (`net/atm/arequipa.c:aqd_enq`). Eventually, `arequipad` will receive the message and close the VC (`net/atm/arequipa.c:arequipa_close_vcc`). The control flow is illustrated in figure 7.

The `AREQUIPA_CLOSE` ioctl (which is called from the library function `arequipa_close`) invokes `net/atm/arequipa.c:arequipa_close` too, so the same procedure as described above is followed.

If the remote party closes the VC (`net/atm/arequipa.c:arequipa_callback`), we're already in user mode (called from `net/atm/signaling.c:sigd_send`). It is therefore possible to close the VC immediately, unless a close operation is

already in progress. As usual, the Arequipa VC is marked as unused (`net/atm/arequipa.c:arequipa_unuse`) and detached from the upper layer socket. If a close request is pending, we return and wait for the third-party close. Otherwise, `net/atm/arequipa.c:arequipa_close_vcc` is invoked directly.

The actual close operation is performed by calling `fs/open.c:close_fp`, which in turn invokes all the protocol-specific functions.

Data forwarding

The data forwarding functions are very similar to the ones used with Classical IP over ATM. In fact, the functions for allocating memory for incoming packets (`net/atm/ipcommon.c:atm_peek_clip`) and for freeing packets after sending them (`net/atm/ipcommon.c:atm_pop_clip`) are identical.

The function to push incoming packets to upper layers (`net/atm/arequipa.c:atm_push_arequipa`) sets the originating interface of the packet to the Arequipa pseudo-interface (this is the marking mentioned in section 3) and copies the generation number from the VC descriptor to the packet descriptor. After that, it proceeds with `net/atm/ipcommon.h:ipcom_push`.

The function that transmits outgoing packets (`net/atm/arequipa.c:arequipa_xmit`) first performs some sanity checks to discard packets from sockets that no longer use Arequipa and packets sent to a VC that is no longer available, determines the VC by looking in the upper layer socket, and sends the packet using `net/atm/ipcommon.h:ipcom_xmit`.

Miscellaneous kernel changes

Various minor changes are required to interface with the Arequipa code. Among the changes are:

- the Arequipa part of the kernel is initialized at boot time. This is done by calling `net/atm/arequipa.c:atm_init_arequipa` from `net/atm/pvc.c:atmpvc_proto_init` (`atmpvc_proto_init` already performs similar initialization tasks)
- the Arequipa-specific ioctls are added to `net/atm/common.c:atm_ioctl`
- the `AREQUIPA_PRESET` ioctl needs to translate the file descriptor to a pointer to the socket descriptor. This is done with `sockfd_lookup`, which also exists as an inline function in `net/socket.c`
- Arequipa ioctls that are applied to the upper-layer socket are handled in `net/ipv4/af_inet.c:inet_ioctl`
- various additions to `net/atm/proc.c` for status monitoring via `/proc/atm/arequipa`
- `net/ipv4/ip_forward.c:ip_forward` is changed to reject IP packets received from Arequipa and to send a "time exceeded" ICMP
- a check to prevent listening on a socket with an attached Arequipa VC is added to `net/ipv4/af_inet.c:inet_listen`

User mode changes

The only changes in user mode are the creation of a library `libarequipa` containing the Arequipa-specific functions and the new demon process `arequipad`.

The library functions are essentially wrappers around ioctls which hide this "raw" interface. They also hide non-obvious procedures like "closing" of the Arequipa VC socket when the VC is established.

`arequipad` has two purposes: it has to register a SAP for incoming Arequipa connections and hand them to the kernel, and it has to process third party close requests. In addition to that, it could also enforce policy restrictions on incoming Arequipa calls.

5 Conclusion

This paper briefly introduced the general concepts of Arequipa, outlined one possible approach for implementing Arequipa for IP over ATM on a Unix system, and finally gave a detailed description of what actually needed to be changed in ATM on Linux when adding Arequipa support.

References

[1] Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Application Requested IP over ATM (AREQUIPA) and its Use in the Web*, Global Information Infrastructure (GII) Evolution, pp. 252–260, IOS Press, 1996.

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

[3] Luciani, James V.; Katz, Dave; Piscitello, David; Cole, Bruce. *NBMA Next Hop Resolution Protocol (NHRP)* (work in progress), Internet Draft `draft-ietf-rolc-nhrp-10.txt`, October 1996.

[4] Braden, Bob; Zhang, Lixia; Berson, Steve; Herzog, Shai; Jamin, Sugih. *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification* (work in progress), Internet Draft `draft-ietf-rsvp-spec-13.ps`, August 1996.

[5] Almesberger, Werner. *ATM on Linux*, `ftp://lrcftp.epfl.ch/pub/linux/atm/papers/atm_on_linux.ps.gz`, EPFL, March 1996.

[6] RFC *to appear*; Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Application REQuested IP over ATM (AREQUIPA)*, IETF, October 1996.

[7] Almesberger, Werner. *Linux ATM API*, `ftp://lrcftp.epfl.ch/pub/linux/atm/api/`, EPFL, July 1996.

[8] Cox, Alan. *Network Buffers and Memory Management*, Linux Journal, issue 30, October 1996.

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

[10] RFC1122; Braden, R. *Requirements for Internet Hosts – Communication Layers*, IETF, October 1989.

[11] Almesberger, Werner. *Linux ATM internal signaling protocol*, `ftp://lrcftp.epfl.ch/pub/linux/atm/docs/`, September 1996.

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