Group Communications and APEX

Marc Stöcklin

Section Systèmes de Communication
Faculté Informatique et Communication
Ecole Polytechnique Fédérale Lausanne

Advisor :

Dr. Matthias Wiesmann
Distributed Systems Laboratory

EPFL-LSR

Lausanne, February 2004

Abstract

This document is the report of my semester project at the Distributed Systems Laboratory (LSR) at École Polytechnique Fédérale Lausanne (EPFL). The aim of the project was to explore the Application Exchange (APEX) protocol, an application layer datagram relaying service which recently reached RFC status, and to use it in context of group communications.

In a first, theoretical phase, I describe and analyze the APEX and its underlying protocol, the Blocks Extensible Exchange Protocol (BEEP). When looking for an appropriate implementation to use APEX in an application at the second stage, I realized the two implementations available did not fit my needs. Therefore I designed a new implementation of the APEX core in Java and integrated two services: the APEX Report Service defined in the specification and the Reliable Broadcast Service. In the second phase of this report, I present the design issues of my APEX implementation as well as the integrated services and the requirements. Examples of how to use APEX in an application and how to design new services are annexed in a tutorial.

Table of Contents

1 Introduction

Ever since the Internet and computer networking has existed, engineers and developers designed protocols to standardize data exchanges of applications. Hence, numerous application protocols have been drawn up and reached Request For Comment (RFC) status authorized by the Internet Engineering Task Force (IETF). Many of these protocols solve about the same problems but everyone adds or leaves out one or another feature and has its own principles for data transfer or encryption, which means that they often are not compatible.

In order to solve these problems in a general and efficient way, the Blocks Extensible Exchange Protocol (BEEP) has been designed. BEEP provides a common framework for application protocol designers and relieves them of basic connection management, message encoding and decoding, authentication or encryption, on creation of a new protocol. Thus, designers can fully devote themselves to the important part placed on top of BEEP: the development of an own application protocol which then is registered as a so called "profile" in the BEEP context.

One of these profiles reached RFC status in 2001, the Application Exchange (APEX) protocol, a best effort datagram relaying service on application level. The APEX provides, supported by the underlying BEEP, service discovery, application-layer addressing, presence information, and permission management. The APEX allows an application to exchange arbitrary MIME and XML messages by using a mesh of interconnected relays which transport these messages to the appropriate endpoints. These endpoints may either be applications or services; the latter allow the APEX even to support a complete client-server model.

As the APEX model is option driven, it is flexible and extensible for use in many kinds of data transfer or communication domains. Since the payload of its messages is arbitrary, the field of application is endless: APEX can for instance be applied in instant messaging, video streaming or to locate and transfer a file in a peer-to-peer mechanism.

The basic interest of this project is to explore the APEX standard in order to find out if it supports any mechanisms for group communications, basically unicast, multicast, and reliable broadcast messaging. It principally targets on finding or writing an implementation to be used in various distributed applications which evades programming directly onto sockets but to work with a comfortable interface.

1.1 Task Specification

The goal of this project is to explore the *Internet Engineering Task Force* (IETF) standard *Application Exchange* (APEX) protocol, to find an implementation and to use this implementation in a small application within the scope of group communications.

The main problems I aim to solve are

- exploring the APEX standard and explaining its mechanism
- finding an implementation of APEX or writing a new one
- designing an application using the implementation in context of reliable broadcast
- write a tutorial for a simple integration in existing applications

First of all it is a matter of giving an overview of the APEX protocol by presenting brief background information on its origin and detailed analyze of the mechanisms it provides.

While keeping a group communications perspective in mind, the purpose of the first phase is to focus on the specifications of the protocol. While doing so, it is a question of examining APEX concerning its architecture and semantic, as well as considering the interface to the underlying protocol BEEP. At the same time I am going to analyze and draw up schematically the so called operations of APEX such as channel establishment, profile identification and initialization, and message exchange, in the two modes of operation provided in the standard. Some important APEX options and APEX services which offer functionalities like access policies, presence service and data timing take another part of the theoretical presentation.

In a second phase of this project, it is about to find existing proof of concept implementations of the APEX protocol. I will study and compare these implementations to be able to pick out the best of them and to judge it in more details. If there are not any implementations we could use for this project, the goal of this phase is to write a new one by keeping the basic APEX core behavior while adding mechanisms for the use in a group communications context. With it, I am going to give in my report a detailed account of the installation or implementation as well as a toolkit description of its properties compared with the ones the specification gives. The outcome of this phase, supported by the

experiences gained while building an APEX environment, is to draw up a simple installation and implementation tutorial which enables programmers to quickly integrate APEX in their programs.

The third phase consists of writing a small messaging program which uses the APEX implementation chosen or written in the second phase. My experiences and descriptions are supposed to serve as a reference for practical applications in future projects in this environment. The program may show apart unicast and broadcast messages, a reliable broadcast messaging model running on several machines.

In addition to the four main problems specified above, I also try to answer the following questions and problems in the course of the project:

- How does the APEX protocol work and what useful tools does it provide?
- Which of these tools gives preference to APEX in a group communications application?
- How can reliable broadcast be used or integrated in APEX?
- Has the chosen or designed implementation any limitations?

Road map

October 20, 2003	start of project
November 13, 2003	examined BEEP and APEX specifications and set up a summary of the structure and essential operations
November 20, 2003	searched the Internet for APEX implementations, decided, based on the result, to design a implementation of the APEX core service
December 10, 2003	set up the basic implementation: structure / interfaces, connection management, MIME / XML parsing, mid-term presentation
December 18, 2003	implemented relaying and exception management
January 8, 2004	integrated the configuration model; designed and tested the `APEXReportService` and `APEXReliableBroadcastService`
January 15, 2004	completed javadoc documentation, report and the tutorial
January 22, 2004	finished and tested the implementation and applications

1.2 Overview

At the outset, I take a look at the Blocks Extensible Exchange Protocol (BEEP) and its basic mechanism to give a historical background and some notions of its architecture, syntax and mechanisms.

In Chapter 3, I analyze the Application Exchange (APEX) protocol by describing the general architecture, its operations and service model. At the end of the chapter, I discuss how APEX can be used in terms of group communication.

The results of my search for existing APEX implementation and its conclusion are discussed in Chapter 4.

Chapter 5 covers the implementation details, beginning with BEEP session establishment, package dependencies and interface description. In this chapter I also describe the internal architecture and some problems I solved in the course of the project as well as some assumptions made during implementation.

The design issues and integration of the Reliable Broadcast are discussed in Chapter 6 while the APEX Report Service integration is described in Chapter 7.

In Chapter 8 finally, the conclusion of my semester project can be found.

A tutorial, describing how to use the APEX implementation designed in this project in applications, and how to set up new APEX services, is presented in Appendix A.

2 Blocks Extensible Exchange Protocol (BEEP)

In this chapter, I would like to give an overview of the Blocks Extensible Exchange Protocol (BEEP) framework which brings along important features for APEX. I first start with an introductive part indicating the reason why BEEP has been created and what it is used for. Then I briefly explain the architecture of the protocol and in order to give a notion of the syntax, I give an example of a BEEP session in Section 2.3. Finally, I show an example where BEEP is used nowadays.

2.1 BEEP Specification

During the last twenty years, lots of protocols have been designed for data exchange between applications. Many of these protocols were solving the same problems like negotiating encryption, authenticating, transferring data, reporting errors, and so on. Simultaneously, programmers had to deal with new security aspects and an increasing complexity of networks, caused by firewalls, network address translation, and dynamic IP address assignment.

In 1998, Dr. MARSHALL T. ROSE and CARL MALAMUD, both highly experienced creators of several network protocols, united to make application protocol design more efficient and simple. The resulting *Blocks Extensible Exchange Protocol* (*BEEP*, former BXXP) has been standardized by the Internet Engineering Task Force in March 2001 mainly in the following two RFCs:

- **RFC 3080 The Blocks Extensible Exchange Protocol Core.** [1]
  This is the main document and describes the basic structure such as Session Establishment and Release, Channel Management, Message Syntax and Peer-to-Peer Behaviour.
- **RFC 3081 Mapping the BEEP Core onto TCP. [2]**
  This documents describes how BEEP sessions are mapped on TCP (Transport Control Protocol)

The BEEP core is a generic application protocol kernel for connection-oriented asynchronous interactions on networks. It is a peer-to-peer protocol that permits simultaneous and independent exchanges of messages between the endpoints on push and pull mode. These messages are arbitrary *Multipurpose Internet Mail Extensions* (MIME) content, but they are usually structured textual using a special *Document Type Definition*

(DTD) of XML, indicated by `Content-Type: application/beep+xml`. The use of MIME allows in addition the possibility to transport arbitrary MIME content as BEEP payload by using a MIME multipart message. BEEP is directly mapped onto the underlying transport service, e.g. on TCP as it is described in RFC 3081 [2].

To set up a BEEP connection, one process first is in *listening* mode while another process acts as a connection *initiator*. At the moment a BEEP session is established, the two of the processes are entirely symmetric in their use of the session properties. Each peer continuous by advertising the profiles it supports by exchanging the `<greeting>` message. A *profile* defines syntax and semantics of the messages exchanged on the specific channel, this enables for instance to use encrypted channels and authentication channels (e.g. SASL) at the same time in the same session.

Either peer can initiate new channels for the session by indicating the profiles it associates to that channel. The other one may accept the channel or decline its creation. Either peer can as well close any channel as well as closing the session at every moment.

The use of profiles for defining the syntax on a channel specifies a reusable solution for network programming. A programmer of a new application does not need to concern himself with handling connection establishment, encryption, or authentication, but simply can use the framework and define his own profile for the data exchange required by the application.

As mentioned above, transporting data with BEEP can be very useful as the network complexity increases: while normal HTTP requires a 3-way-handshake for every new connection, a BEEP connection remains preserved over a long time and does not need to be reinitiated.

2.2 BEEP Syntax and Architecture

To give a general notion of how BEEP works, I show and comment some exchanges during the establishment of a session including the creation of a channel. In order to understand the following example I briefly explain the frame syntax and its structure.

Every BEEP message begins with a so called *frame header* which consists of a three-character keyword followed by several parameters. These main keywords are:

- `MSG` start of a new message and with a unique message number
- `RPY` system reply to a message identified with the message number, such as a confirmation for a channel creation as well as decline of one
- `ANS` answer to a message in the associated channel
- `ERR` error message, e.g. if a requested profile does not exist
- `NUL` an empty message

The supplied parameters, each separated by a single space character, are structured in a special order as follows:

- `channel` channel number for this frame
- `msgno` number of the corresponding message sequence
- `more` takes either `*` or `.` which indicates if at least one frame follows or the frame completes the message
- `seqno` sequence number of the message which corresponds to the first Byte of the payload for the associated channel
- `size` number of Bytes in the payload
- `ansno` answer number to a message (only used in a ANS frame)

The payload of a BEEP message is structured according to the rules of the MIME standard and ends with *frame trailer* `END` followed by a `CRLF` pair. As mentioned above, the semantics of the message is specific to the profile associated to a channel – however, the specification says that a profile must define:

- initialization messages exchanged during channel creation if necessary
- messages exchanged in the payload
- semantics of these messages

2.3 Example: initiating a BEEP session

In order to illustrate some important operation in context of a BEEP session, I go straight through a normal establishment of a session and the creation of an APEX channel.

To initiate a new session, one process acts as a listener (L) while the initiator (I) opens a connection. The listener then takes the role of a server and, when the connection is established, advertises the profiles it supports. The initiator accepts the profile he is willing to use by an empty greeting tag.

```
L/S: <wait for incoming connection>
I/C: <open connection>
  S: RPY 0 0 . 0 111
  S: Content-Type: application/beep+xml
  S:
  S: <greeting>
  S:    <profile uri='http://iana.org/beep/APEX' />
  S: </greeting>
  S: END
  C: RPY 0 0 . 0 52
  C: Content-Type: application/beep+xml
  C:
  C: <greeting />
  C: END
```

Listing 1 Initiation of a BEEP Session

Now the initiator starts a new channel and sends a request containing a unique channel number and the profile he wants to associate to this channel. The listener accepts the channel and confirms. Note that the first number after the three-character keyword indicates the channel number. Channel number zero is used for channel management.

```
  C: MSG 0 1 . 52 116
  C: Content-Type: application/beep+xml
  C:
  C: <start number='1'>
  C:    <profile uri='http://iana.org/beep/APEX' />
  C: </start>
  C: END
  S: RPY 0 1 . 111 93
  S: Content-Type: application/beep+xml
  S:
  S: <profile uri='http://iana.org/beep/APEX' />
  S: END
```

Listing 2 Initiation of an APEX channel

Now that a channel is opened, profile-specific messages are sent over this channel (e.g. APEX attach or data messages).

To terminate a channel, again, a channel management message is sent over channel zero which contains the channel number to close and a reply code.

```
C: MSG 0 2 . 235 71
C: Content-Type: application/beep+xml
C:
C: <close number='1' code='200' />
C: END
S: RPY 0 2 . 392 46
S: Content-Type: application/beep+xml
S:
S: <ok />
S: END
```

Listing 3 Termination of channel '1'

Finally, the client closes the whole session (even if some channels could still be open), the server confirms with an `<ok />` reply and every peer closes its connection.

```
C: MSG 0 1 . 52 60
C: Content-Type: application/beep+xml
C:
C: <close code='200' />
C: END
S: RPY 0 1 . 264 46
S: Content-Type: application/beep+xml
S:
S: <ok />
S: END
I: <close connection>
L: <close connection>
L: <wait for next connection>
```

Listing 4 Termination of a BEEP session

2.4 BEEP Applications

The BEEP framework, due to its enormous advantages for application protocol setup and large field of application, is well established and used today. In the following paragraphs, I present some examples.

2.4.1 Apple's Xgrid

On January 6, 2004, Apple introduced *Xgrid* [3]. Xgrid allows to run a program on various machines in order to get the requested result faster. There are three different entities in an Xgrid calculation cluster:

- the *client* which wants to initiate a calculation,
- the *controller* which actually initiates the calculation, and
- a set of *agents* which perform the calculation.

Xgrid is designed for computations that take very long time. It provides the basic infrastructure for communication between computers: run commands remotely and obtain their results. Typical applications that may profit from Xgrid are Monte Carlo calculations, 3D rendering, or computations which can be split up in subunits to be treated separately.

In an Xgrid cluster, BEEP provides the underlying protocol between agent, controller, and clients for data and command transmission. More details to Xgrid can be found on [4].

2.4.2 Profile Development Training at Clipcode Knowledge Services

Clipcode Knowledge Services [5], a consulting and training service with headquarters in Ireland, offer several training courses in containing BEEP profile development as well work on BEEP profile specialist projects.

2.4.3 Intrusion Detection Working Group of the IETF

The *Intrusion Detection Working Group* (IDWG) of the Internet Engineering Task Force (IETF) uses BEEP as the protocol framework for exchanging intrusion detection messages between different systems. To do so, the *Intrusion Detection Exchange Protocol* [6] (IDXP) is specified as a BEEP profile and a Tunnel profile [7] is provided for different systems to exchange messages through firewalls.

3 Application Exchange Protocol (APEX)

In this chapter, I introduce the APEX protocol by discussing all tools it provides. First, I start with a brief introduction about the status of the protocol and its creators, then, I discuss the general architecture. In Section 3.3, I explain the profile's semantics by giving samples of messages exchanged in an APEX channel. The next section describes a set of options to alter the semantics of the relaying service and to expand its mechanisms. Finally I present useful services defined in the specification to complete the theoretical part.

3.1 APEX Specification

The *Application Exchange Protocol* (APEX) protocol is an extensible best effort datagram relaying service which transports each datagram over a mesh of relays from an originator endpoint to one or more recipient endpoints. Each of these so called endpoints is an application layer process dynamically attached to the relaying mesh. As APEX is a profile of BEEP, it profits of the full service provisioning (e.g. connection establishment, authentication) of BEEP. To complete BEEP to an entire messaging system, APEX provides additional services on top of the relaying mesh such as access control and report service.

APEX, like BEEP, has also been defined by MARSHALL T. ROSE together with DAVID CROCKER (who defined the e-mail formats in 1982) and GRAHAM KLYNE (a pioneer in use of e-mail to interface to telephone services). Its specifications are described in four RFCs:

- **RFC 3340 The Application Exchange Core.** [8]
  This is the main document and describes the architecture and basic operations and options including the correspondent algorithms such as data processing in relays and endpoints, status requests and initialisation.
- **RFC 3341 The Application Exchange (APEX) Access Service.** [9]
  This document describes the use and management of access restrictions.
- **RFC 3342 The Application Exchange (APEX) Option Party Pack, Part Deux!** [10]
  This RFC defines several options for advanced mechanisms such as improved report services, endpoint holding and data timing.
- **RFC 3343 The Application Exchange (APEX) Presence Service** [11]
  This document illustrates the presence service and related operations

3.2 APEX Architecture

As mentioned a several times, APEX is a profile of BEEP and is initiated when starting a channel. On top of the *APEX core* an *APEX process* – either a relay or an endpoint – handles the incoming and outgoing traffic. An endpoint may be an application that uses APEX for network communication. APEX services are also logically defined as endpoints but they do not need necessarily to be a single physical endpoint, instead they may for instance be co-resident with a relay within an administrative domain to provide services in this domain.

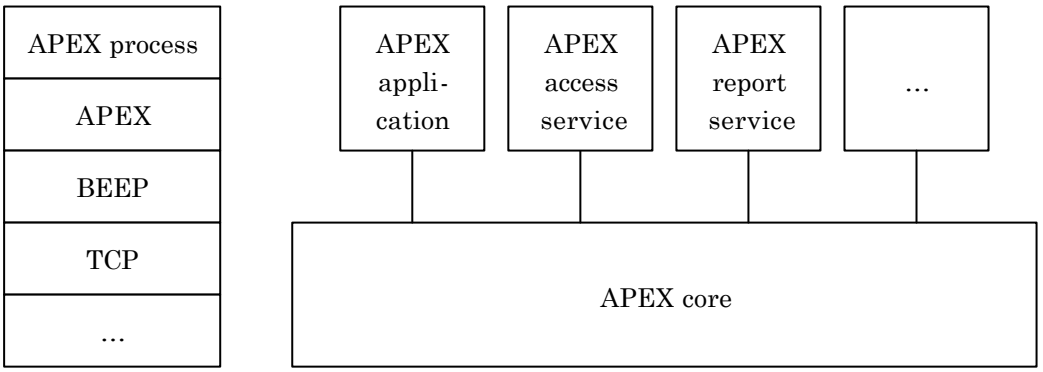

Fig. 1 The APEX stack

APEX can be used in two modes:

3.2.1 Endpoint-Relay mode: APEX edge connection

In endpoint-relay mode, a so called *edge connection*, an application (endpoint) initiates a BEEP connection to a relay. Endpoints are always initiators while relays are always listener in the BEEP context. Once an endpoint opened a BEEP connection to a relay it may attach as one or more endpoint, send data to other endpoints, receive data from other endpoints, or terminate any of its attachments. A relay may deliver data from other endpoints, terminate attachments, or indicate delivery status of data sent early by the endpoint.

3.2.2 Relay-Relay mode: APEX mesh connection

In a *mesh connection*, applications bind as relays to others relay which may be in different administrative domains. Once the BEEP connection is established, either relay may bind as one or more administrative domains, send data over the channel, receive data to deliver, or terminate any bindings.

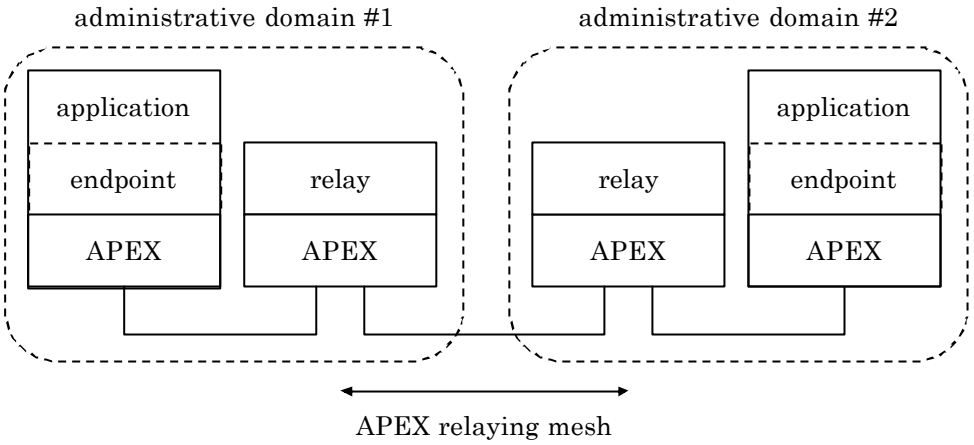

Fig. 2 The APEX entities

3.2.3 Well-Known TCP port numbers

For each of the two modes, a well-known TCP port number has been assigned by the Internet Assigned Number Association (IANA) [12].

- 912 (tcp/udp) relay-relay mode, registered *apex-mesh*
- 913 (tcp/udp) endpoint-relay mode, registered as *apex-edge*

3.2.4 Naming conventions

Each relay is identified by a unique domain name. Domains can either be a *fully qualified domain name* (FQDN) or a domain-literal, e.g. "epfl.ch" or "[10.0.0.1]". Endpoint addresses, apart from the domain name, consist of a local-part formed by an address and zero or more sub addresses separated with a slash (/); an endpoint address has the form *local@domain*. Every administrative domain may provide a set of *well-known endpoints* (WKE) which stands for APEX services within this domain. These endpoints are addressed by using the prefix `apex=`.

Some examples show the correct use of the naming conventions:

- `user1@lsrwww.epfl.ch` user1 is an endpoint within the administrative domain 'lsrwww.epfl.ch'
- `apex=report@lsrwww.epfl.ch` APEX Report Service (WKE) for the administrative domain 'lsrwww.epfl.ch'
- `user1/appl=gc@lsrwww.epfl.ch` sub address (endpoint application) of the endpoint 'user1@lsrwww.epfl.ch'

3.3 Message Semantics

In this part I am dealing with the message syntax within an APEX channel to give an idea of how messages exchanged in an APEX channel are formed.

As mentioned in the description of BEEP, all messages in a session consist of an XML document and possibly an arbitrary MIME content. Thus, APEX uses the same conventions and offers advantage that the messages are easily readable.

3.3.1 Channel creation (BEEP)

During the APEX channel creation, the profile must be identified as `http://iana.org/beep/APEX`.

```
RPY 0 0 . 0 111
Content-Type: application/beep+xml

<start number='1'>
  <profile uri='http://iana.org/beep/APEX' />
</start>
END
```

Listing 5 Start of an APEX channel

3.3.2 Attach operation

After the creation of an APEX channel, an application may attach as an endpoint to the relaying mesh. To do so, it sends the `attach` element specifying an endpoint address that the application wants to attach as, the transaction identifier for this operation, and zero or more options. The relay replies either with an `ok` element upon success or an `error` element.

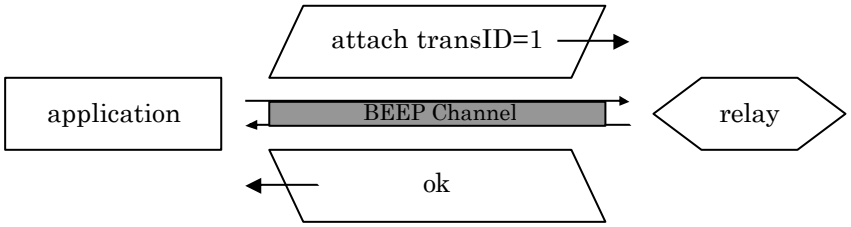

Fig. 3 An attach operation

```
A: <attach endpoint='user@lsrwww.epfl.ch' transID='1' />
R: <ok />
```

Listing 6 Attachment of 'user@lsrwww.epfl.ch'

To improve the legibility, I abandon BEEP frame header and trailer as well as MIME definitions in listings.

3.3.3 Bind operation

A relay which wants to connect to another relay sends a `bind` element specifying the relay administrative domain on whose the application wants to serve, the transaction identifier for this operation, and zero or more options. Again, the second relay confirms or declines the operation with the corresponding element.

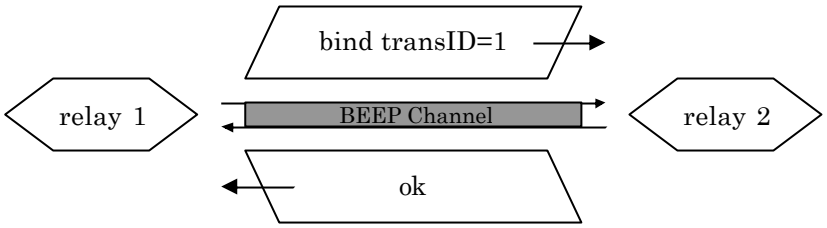

Fig. 4 A bind operation

```
R1: <bind relay='lsrwww.epfl.ch' transID='1' />
R2: <ok />
```

Listing 7 Binding of relay 'lsrwww.epfl.ch'

3.3.4 Terminate operation

To release an attachment or a binding to the mesh, an application or a relay sends a `terminate` element specifying the transaction identifier associated to this operation. Optional, this element may contain textual content and specify a three-digit reply code for diagnostic.

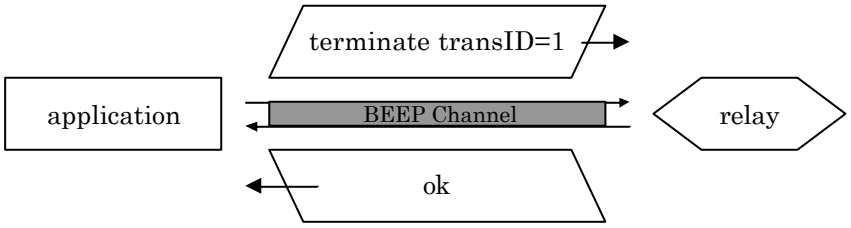

Fig. 5 A terminate operation

```
A: <terminate transID='1' />
R: <ok />
```

Listing 8 Termination of transaction 1

3.3.5 Data operation

To transmit data over the relaying mesh, an application and a relay respectively send a `data` element specifying the originator endpoint address, one or more recipients, and zero or more options. The content of the data is indicated as a URI-reference in the same MIME context.

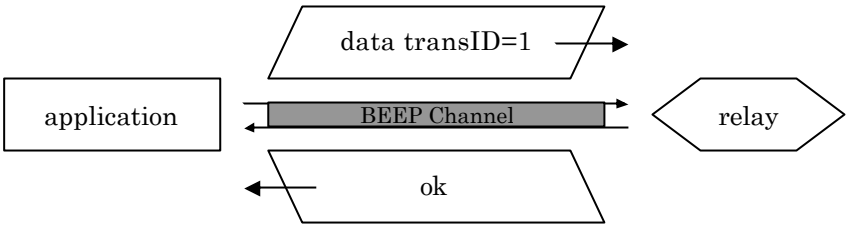

Fig. 6 A data operation

The following example shows data operation which contains XML content.

```
A: Content-Type: application/beep+xml
A:
A: <data content='#Content'>
A:   <originator identity='user1@lsrwww.epfl.ch' />
A:   <recipient  identity='user2@ltiwww.epfl.ch' />
A:   <data-content Name='Content'>
A:     <reply code='250' />
A:   </data-content>
A: </data>
R: <ok />
```

Listing 9 A data operation with XML content

The same content might as well be transported in a MIME Multipart structured data operation. Then, the content is linked by the "content" attribute of the `data` element.

```
A: Content-Type: multipart/related; boundary="boundary";
A:               start="<1@lsrwww.epfl.ch>";
A:               type="application/beep+xml"
A:
A: --boundary
A: Content-Type: application/beep+xml
A: Content-ID: <1@lsrwww.epfl.ch>
A:
A: <data content='cid:2@example.com'>
A:   <originator identity='user1@lsrwww.epfl.ch' />
A:   <recipient  identity='user2@ltiwww.epfl.ch' />
A: </data>
A:
A: --boundary
A: Content-Type: image/gif
A: Content-Transfer-Encoding: binary
A: Content-ID: <2@lsrwww.epfl.ch>
A:
A: ...
A: --boundary--
R: <ok />
```

Listing 10 A data operation with MIME Multipart structured content

3.4 Operation processing

RFC 3380 [8] specifies a set of rules of how relays and applications should process each operation. On behalf of treating every one individually, I prefer to take a closer look at the most important action: the process of a `data` element in a relay. In fact this processing algorithm covers most rules concerning every operation and gives a good notion of how they work. In Fig. 7 and Fig. 8, the steps of the algorithm are indicated in Arabic and Roman numerals respectively.

1. When a relay receives a `data` element (1 / I) it first verifies that the sending BEEP peer is authorized to originate or relay data according to the APEX access policies, otherwise an `error` element is returned.
2. Per-data options are processed, if they are present.
3. An `ok` (2 / II) element is returned.
4. Per-originator options are processed.
5. For each recipient, the relay performs the following step:
6. Per-recipient options are processed.
    a. If the recipient is not in the same administrative domain as the relay, an APEX channel is established to a relay that accepts data for the recipient's administrative domain – a new `data` element containing the respective `recipient` and `content-content` element is sent over this channel (III). If the correspondent relay returns an `ok` element (IV), the transmission for this recipient is successfully processed.
    b. If the recipient and the relay are in the same administrative domain, the relay verifies in the access entries that the originator is allowed to communicate with the recipient endpoint and that the recipient endpoint is currently attached – then, a new `data` element is sent to the corresponding endpoint (3 / V). Again, if it returns an `ok` element (4 / VI), the current recipient is successfully processed.

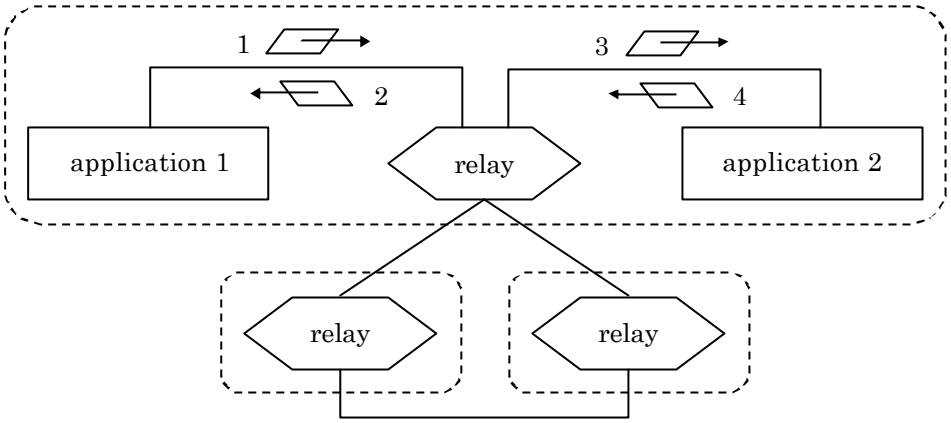

Fig. 7 Two APEX endpoints in the same administrative domain

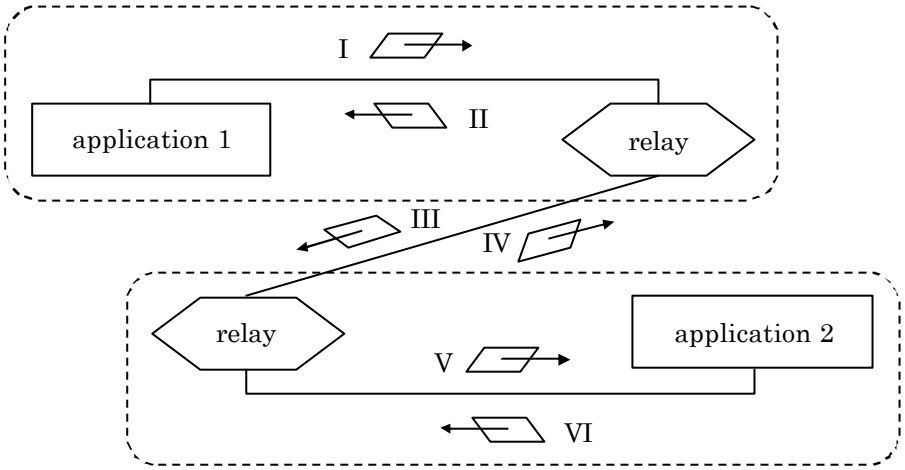

Fig. 8 Two APEX endpoints in different administrative domains

Notes to the algorithm

There are a few aspects that one has to be aware when using the `data` operation. First of all, it is obvious that there is an authorisation control and only authorized messages reach the recipients endpoint. Every relay on the trajectory of a message has to verify if it is allowed to pass to the next station and therefore – for a successful delivery – all relays must permit the transmission.

Another important point is that in the basic specification (without any option) no mechanism guarantees to the originator (or reports him) that the message successfully reached at least one of the specified recipients. An `ok` element is returned as soon as in the

first two steps of the algorithm no error is produced, say no access restrictions are violated and per-data options are successfully processed. No acknowledgement in the sense of TCP is returned on a successful transmission of the message – such replies can be force using the status report option `statusRequest` though.

If a relay receives a message for an endpoint which is not attached to the mesh at the moment, it discards the message without any notification to its originator; as well as if a relay cannot establish a connection to another relay or is not able to successfully send a message (due to access restrictions), the message is silently dropped.

If the originator and the recipient are not in the same administrative domain, a direct connection between their two relays is established in order to send the message.

The standard does not prescribe if a relay must or must not optimize its behaviour by grouping multiple recipients in the same administrative domain in a single message to a single `data` element that is subsequently transmitted. Therefore it depends on the implementation if such an optimization is applied.

3.5 APEX Options

The APEX model is option-driven, and therefore allows flexible control structures and extensible services. The APEX specification comes up with a large set of options to improve the behaviour of the basic algorithm to an efficient data transmission mechanism. The default mode of APEX is immediate delivery and best-effort relaying, but this behaviour can be modified by options to obtain for instance reliable or time-sensitive delivery.

I would like to give a brief overview of options provided in the standard and to present the most interesting ones.

- **statusRequest** invokes a status report in applicable relays
- **attachOverride** allows an endpoint to override a previously attached endpoint (terminates former attachments)
- **dataTiming** provides a set of attributes ("noLaterThan", "returnTrip", "reportAfter") which alter the best-effort behaviour of the APEX relaying mesh to control queuing delays
- **hold4Endpoint** forces relays to queue data if a receipients endpoint is not attached
- **dataHopping** detects misconfigurations caused by forwarding loops similar to the TTL mechanism of IP

3.5.1 Option element

An `option` element may be contained within a `data`, `originator`, `recipient`, or an `attach` element. It has several attributes which influence the processing of the element in a relay or an endpoint. The most important are:

The *internal* or *external* attribute defines the name of the option. While an "internal" option signifies a predefined IANA-registered option, an "external" option has the value of an URI for an additional option.

The *targetHop* attribute specifies the applicable relay(s) on a message's trajectory which should process the option. Possible values are "this" (the processed option must be removed before transmitting the containing element), "final" (only the relay that transmits the element containing the option to the final recipient is affected), and "all" (every relay must process the option and retain it for the next).

The *mustUnderstand* attribute indicates if whether the relay may ignore the option if it is not registered. This attribute has only effect provided that the processing relay is applicable according to the "targetHop" attribute.

3.5.2 The dataTiming option

The default behaviour of the APEX relaying mesh is that all relays and endpoints are expected to be able to process and transfer data without any delay. If no processing options are present and a relay or an endpoint is unavailable or ready to accept transfer due to queuing or a slow connection, the message is discarded without any report to its originator.

The *dataTiming* option contains a `dataTiming` element which allows the originator to specify three attributes:

The *noLaterThan* attribute hold a positive non-zero integer value which indicates an amount of time in milliseconds after that a message is discarded if not delivered. Each intermediate relay adjusts the amount according to its processing time. If the value becomes less or equal to zero, the message is discarded and if the "reportErrors" attribute is true, the APEX Report Service is invoked to send a timing error report. As well, if the relay does not receive an `ok` element after sending the element within the amount of time, an error has occurred.

```
<data content='cid:2@example.com'>
  <originator identity='user1@lsrwww.epfl.ch' />
  <recipient identity='user2@ltiwww.epfl.ch' />
  <option internal='dataTiming' targetHop='all'
          mustUnderstand='true' transID='86'>
    <dataTiming noLaterThan='60000' reportErrors='true' />
  </option>
</data>
```

Listing 11 A data operation containing a `dataTiming` element with a "noLaterThan" attribute

The *reportAfter* attribute allows the originator to invoke a notification if the message is not delivered after a specified time. Again, each intermediate relay adjusts this amount according to its processing time. If the value becomes less or equal to zero, the report service is invoked to send a transient timing report – however, the message continues its path.

```
<data content='cid:2@example.com'>
  <originator identity='user1@lsrwww.epfl.ch' />
  <recipient identity='user2@ltiwww.epfl.ch' />
  <option internal='dataTiming' targetHop='all'
          mustUnderstand='true' transID='86'>
    <dataTiming reportAfter='60000' />
  </option>
</data>
```

Listing 12 A data operation containing a `dataTiming` element with a "reportAfter" attribute

The *returnTrip* attribute sets an upper limit on the time for a "statusResponse" delivery, after which the originator supposes the message to be lost. If the "returnTrip" attribute is present, a "statusResponse" message containing a dataTiming option is generated. The value of the "noLaterThan" attribute is set to the value of the "returnTrip" attribute.

Listing 13 shows a final hop report: the "noLaterThan" bounds have not affected the transmission of the message, so the final relay R2 invokes the APEX Report Service.

```
R1: <data content='cid:2@example.com'>
        <originator identity='user1@lsrwww.epfl.ch' />
        <recipient identity='user2@ltiwww.epfl.ch' />
        <option internal='dataTiming' targetHop='all'
                mustUnderstand='true' transID='1'>
            <dataTiming noLaterThan='10000' returnTrip='20000' />
        </option>
    </data>
R2: <ok />

R2: <data content='#Content'>
        <originator identity='apex=report@ltiwww.epfl.ch' />
        <recipient identity='user1@lsrwww.epfl.ch' />
        <option internal='dataTiming' targetHop='all'
                mustUnderstand='true' transID='2'>
            <dataTiming noLaterThan='20000' />
        </option>
        <data-content Name='Content'>
            <statusResponse transID='1'>
                <destination identity='user2@ltiwww.epfl.ch'>
                    <reply code='250' />
                </destination>
            </statusResponse>
        </data-content>
    </data>
R1: <ok />
```

Listing 13 The "noLaterThan" bounds have not affected the transmission

3.5.3 The hold4Endpoint option

In absence of processing options, a relay silently drops a message if the recipient endpoint currently is not attached. The *hold4Endpoint* options alters this behaviour in order to queue the message either for a specified amount of time provided by the dataTiming option or until the recipients endpoint attaches to the APEX relaying mesh. In the absence of upper bounds on delivery, the data will be queued indefinitely.

The following example shows the transmission of a `data` element to 'user2@lti-www.epfl.ch' which is not currently attached to the relay in his administrative domain. As the hold4Endpoint option is set, the message is queued in the relay until the second endpoint attaches to it.

```
R1: <data content='cid:1@example.com'>
        <originator identity='user1@lsrwww.epfl.ch' />
        <recipient identity='user2@ltiwww.epfl.ch' />
        <option internal='hold4Endpoint' />
        <option internal='dataTiming' targetHop='all'
                mustUnderstand='true' transID='10'>
            <dataTiming noLaterThan='60000' />
        </option>
    </data>
R2: <ok />
```

Some time later, the requested endpoint attaches at the APEX mesh.

```
A2: <attach endpoint='user2@ltiwww.epfl.ch' transID='2'>
    </attach>
R2: <ok />

R2: <data content='cid:1@example.com'>
        <originator identity='user1@lsrwww.epfl.ch' />
        <recipient identity='user2@ltiwww.epfl.ch' />
        <option internal='hold4Endpoint' />
        <option internal='dataTiming' targetHop='all'
                mustUnderstand='true' transID='10'>
            <dataTiming noLaterThan='18000' />
        </option>
    </data>
A2: <ok />
```

Listing 14 'user2@ltiwww.epfl.ch' receives the message due to the hold4Endpoint option

3.6 APEX Services

The APEX specification provides three default services on which processes fall back during message exchanges:

- **Report Service** sends *statusResponse* messages which indicate the originator the transmission status of a message
- **Access Service** determines access policies, e.g. attachment to the mesh or message transmission
- **Presence Service** manages presence information for endpoints

As shown in Fig. 1, the services are placed in every administrative domain as separated endpoints, the so called *well-known endpoints* (WKE). An application addresses for instance the APEX access service by using "apex=access" for the local part in his administrative domain, e.g. 'apex=access@lsrwww.epfl.ch'; the APEX Report Service identifies with the WKE "apex=report".

In this section I would like to give a notion of these services. I will only expand on the APEX Report Service which will be important in view of an implementation in this project, while the other service may be implemented in a second phase.

3.6.1 APEX Report Service

The APEX Report Service is responsible for sending status reports in order to indicate transmission success or processing errors. If a relay processes a *statusRequest* option in a message, it invokes the report service which generates a statusResponse message. As well, if a processing error occurs, say an option is not registered and the "mustUnderstand" attribute is set to "true" as well as the status report is demanded, the report service handles the notification of the originator.

A *statusResponse* message is a `data` element which contains an `originator` element (e.g. "apex=report@lsrwww.epfl.ch"), a `recipient` element, zero or more `option` elements, and a `statusResponse` element as its content. The `statusResponse` element itself contains one or more `destination` elements, depending if the initial message has been addressed to one or multiple recipients. For each of the recipients' endpoints which are reported, a `reply` element indicates a three-digit reply code for the current recipient.

In the sample message of Listing 15, the APEX Report Service of 'ltiwww.epfl.ch' notifies the endpoint 'user1@lsrwww.epfl.ch' that 'user3' is an unknown within its domain.

```
<data content='#Content'>
  <originator identity='apex=report@ltiwww.epfl.ch' />
  <recipient identity='user1@lsrwww.epfl.ch' />
  <data-content Name='Content'>
    <statusResponse transID='10'>
      <destination identity='user3@ltiwww.epfl.ch'>
        <reply code='550'>unknown endpoint identity</reply>
      </destination>
    </statusResponse>
  </data-content>
</data>
```

Listing 15 Status response: 'user3' is unknown in 'ltiwww.epfl.ch'

The following example illustrates a status request contained in a `data` element with originator 'user1@lsrwww.epfl.ch' (appl. 1) to the recipient 'user2@ltiwww.epfl.ch' (appl. 2) over two intermediate relays for each administrative domain. Note that since the value of the "targetHop" attribute is "final", the option is only processed in relay 2 (r2) which sends the status response message as soon as it knows the status of the relayed message.

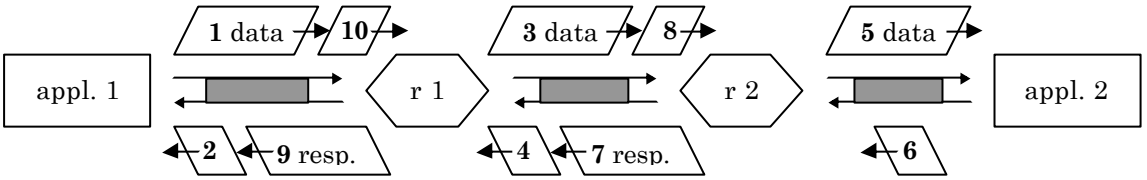

Fig. 9 Sequence of a final "statusRequest"

```
1  A1: <data content='cid:2@lsrwww.epfl.ch'>
         <originator identity='user1@lsrwww.epfl.ch' />
         <recipient identity='user2@ltiwww.epfl.ch' />
         <option internal='statusRequest' targetHop='final'
                 mustUnderstand='true' transID='2' />
       </data>
2  R1: <ok />
3  R1: <data content='cid:2@lsrwww.epfl.ch'>
         <originator identity='user1@lsrwww.epfl.ch' />
         <recipient identity='user2@ltiwww.epfl.ch' />
         <option internal='statusRequest' targetHop='final'
                 mustUnderstand='true' transID='2' />
       </data>
4  R2: <ok />
5  R1: <data content='cid:2@lsrwww.epfl.ch'>
         <originator identity='user1@lsrwww.epfl.ch' />
         <recipient identity='user2@ltiwww.epfl.ch' />
       </data>
6  A2: <ok />
7  R2: <data content='#Content'>
         <originator identity='apex=report@ltiwww.epfl.ch' />
         <recipient identity='user1@lsrwww.epfl.ch' />
         <data-content Name='Content'>
           <statusResponse transID='2'>
             <destination identity='user2@ltiwww.epfl.ch'>
               <reply code='250' />
             </destination>
           </statusResponse>
         </data-content>
       </data>
8  R1: <ok />
```

Listing 16 Delivery of a message containing a final "statusRequest" option

3.6.2 APEX Access Service

The APEX Access Service is a control mechanism for the relaying mesh as well as for APEX services. In every administrative domain, the well-known endpoint "apex=access" maintains "access entries" for each of its endpoints and services. These entries are consulted before the processing of a message continues.

An access entry is empty but has four attributes: the "owner" attribute specifies the address of the endpoint associated with the entry, the "actor" attribute specifies the address of a entity or a group of entities affected by this entry, the "actions" attribute indicates the permissions or restrictions for the actor(s) to the owner, and the "lastUpdate" attribute specifies the date and time of the last modification of the access entry.

The following two sample entries allow

1. all users within the 'lsrwww.epfl.ch' administrative domain to use all services and all operations in the context of 'user1@lsrwww.epfl.ch', and
2. all users in the APEX relaying mesh to use the data operation of the APEX core in the context of 'user1@lsrwww.epfl.ch'

```
<access owner='user1@lsrwww.epfl.ch'
        actor='*@lsrwww.epfl.ch'
        actions='all:all'
        lastUpdate='2003-11-11T11:11:00+01:00' />

<access owner='user1@lsrwww.epfl.ch'
        actor='*@*'
        actions='core:data'
        lastUpdate='2003-11-11T11:11:00+01:00' />
```

Listing 17 Access enties

3.6.3 APEX Presence Service

The APEX Presence Service provides presence information for attached endpoints. Applications communicate with this service over the well-known endpoint "apex=presence" in the respective administration domain. The APEX Presence Service maintains information about presence entries for every so called publisher.

To publish or modify presence information, an application sends a *publish* operation to the service, specifying a publisher address, one or more tuples (entity). Every tuple has a destination address (URI) and a time stamp which indicates the validity of the entry, e.g. the latest time the capable of receiving messages. In addition an attribute may specify the kinds of content the entity is capable to receive.

When another application wants to receive presence information associated to another endpoint, it sends a *subscribe* operation to the service, specifying the address of the target endpoint and a duration for the subscription. If the access entries agree, the service immediately responds with the current presence information of the endpoint and sends further publish operations, whenever the presence entries for the endpoint change.

To receive notices about endpoints that are subscribed, an application sends a *watch* operation, specifying his address and duration for the operation. For every modification of presence information (e.g. a new subscriber) in the APEX Presence Service entries, the watcher receives a notification if the respective access entries admit.

The following publish operation updates the presence information for the endpoint 'user1@lsrwww.epfl.ch' within its administrative domain:

```
A1: <data content='#Content'>
      <originator identity='user1@lsrwww.epfl.ch' />
      <recipient identity='apex=presence@lsrwww.epfl.ch' />
      <data-content Name='Content'>
        <publish publisher='user1@lsrwww.epfl.ch' transID='1'
                 timeStamp='2003-11-11T11:11:00-01:00'>
          <presence publisher='user1@lsrwww.epfl.ch'
                    lastUpdate='2003-11-11T11:00:00-01:00'
                    publisherInfo='http://lsrwww.epfl.ch/~user1/'>
            <tuple destination='user1/appl=gc@lsrwww.epfl.ch'
                   availableUntil='2003-11-11T11:21:00-01:00' />
            <tuple destination='mailto:user1@epfl.ch'
                   availableUntil='2011-11-11T23:59:59-01:00' />
          </presence>
        </publish>
      </data-content>
    </data>
R1: <ok />
```

Listing 18 Presence publication (publish operation) of 'user1@lsrwww.epfl.ch'

3.7 Auxiliary elements

In addition to the operations presented before, I would like to address to five definitions which I ignored completely up to now, but they will be important in this project as well: first the transaction-identifier, and then the `ok` element, the `error` element, the `reply` element and their reply codes.

3.7.1 Transaction identifiers

For every new transaction over a channel, a *transaction identifier* is necessary. The transaction identifier must be a *unique* integer number greater than zero for an originator endpoint address.

3.7.2 The ok element

The `ok` element is the positive reply on an APEX message. It indicates the correct receipt and per-data processing of the message. An `ok` element is empty and has no attributes.

```
<ok />
```

Listing 19 An `ok` element

3.7.3 The error element

The `error` element is the negative reply on an APEX message. It is sent as soon as an error occurs while an endpoint attaches to the mesh, a relay binds, a process verifies access policies, and so on. An `error` element has a "code" attribute which specifies a three-digit reply code meaningful to the recipient. In addition, it may contain arbitrary textual content as a diagnostic which is meaningful to implementers, perhaps administrators or even users and an optional "xml:lang" attribute which specifies the language that the textual content is written in.

```
<error code='537'>access denied</error>
```

Listing 20 An `error` element

3.7.4 The reply element

Many APEX services use the `reply` element for responses and acknowledgements in their responses. The APEX Access Service for instance uses the `reply` element to confirm or decline a set operation which updates the access entries, the APEX Report Service handles error and status messages using the `reply` element.

The `reply` element has a "code" attribute which again specifies a three-digit reply code and an "transID" attribute in the containing element which specifies the transaction-identifier corresponding to this reply.

```
<data content='#Content'>
  <originator identity='apex=presence@example.com' />
  <recipient identity='wilma@example.com' />
  <data-content Name='Content'>
    <reply code='250' transID='100' />
  </data-content>
</data>
```

Listing 21 A `reply` element in a successful transaction

```
<data-content Name='Content'>
  <statusResponse transID='86'>
    <destination identity='barney@example.com'>
      <reply code='537'>action not authorized
                        for user</reply>
    </destination>
  </statusResponse>
</data-content>
```

Listing 22 An access violiation indicated by an `reply` element

3.7.5 APEX reply codes

In Table 1, the default reply codes and their correspondent meanings are listed. These reply codes base on the common reply code model used as well in HTTP or FTP.

250	transaction successful
421	service not available
450	requested action not taken
451	requested action aborted
454	temporary authentication failure
500	general syntax error (e.g., poorly-formed XML)
501	syntax error in parameters (e.g., non-valid XML)
504	parameter not implemented
530	authentication required
534	authentication mechanism insufficient
535	authentication failure
537	action not authorized for user
538	authentication mechanism requires encryption
550	requested action not taken
553	parameter invalid
554	transaction failed (e.g., policy violation)
555	transaction already in progress

Table 1 Default reply codes

3.8 APEX and group communications

In this section, I take a closer look at the APEX protocol in order to use it in context of group communications. Hence, I discuss three communication models: unicast, multicast, and reliable broadcast messaging.

3.8.1 Unicast messaging

As it is specified in Paragraph 3.3.5 and RFC3340 [8], a `data` element, which stands for a data operation, contains at least one `recipient` element, specifying a recipient endpoint address. Thus, every message containing exactly one `recipient` element can be considered as a unicast message and is transported over the APEX mesh to the appropriate endpoint.

Unicast messaging is provided by APEX.

3.8.2 Multicast messaging

As seen above, multiple `recipient` elements are also allowed within a `data` element; such a message can be considered as a multicast message. For every specified recipient, a relay creates a new message and sends it to the address.

Multicast messaging is provided by APEX.

3.8.3 Reliable broadcast messaging

Reliable broadcast messaging is a model to send messages so much that if one recipient in a sequence of others receives a message, it guarantees that every one after him in the sequence receives the message as well – this mechanism is modelled by resending the message to all following recipients if it has not done that yet.

The reliable broadcast mechanism is not provided by the APEX and a way must be found to implement it. In my opinion, the most common way to do this is to *design a new service* which is invoked by an option specifying some information. This information may for instance be the sequence of endpoints to be supplied. The service could even be optimized in order to avoid redundant messages sent between relays and endpoints: since every relay knows what its attached endpoints receive, it may directly filter out messages already transported and take on the entire task in reliable broadcast messaging (c.f. Section 6.4).

4 Existing APEX implementations

When searching on the Internet for existing APEX implementation it turned out that at the moment only two implementations are available. In this chapter, I briefly summarize the results of my investigation on APEX and in Section 4.4 I present its conclusion.

4.1 APEX Implementation of the IMPP

The first APEX implementation found is the one of MICHAEL J. RIGGIO [13], Temple University NetLab, who has been working at the development of an APEX implementation of the IMPP (Instant Messaging and Presence Protocol), described in RFC 2779. The idea of this independent study was to write the APEX implementation using Java BEEP Core package provided by the official BEEP developers.

When analyzing and testing Mr. RIGGIO's code, I realized that it is incompatible with the standard and only implements the presence service in addition to the basic APEX core service which for this project is useless. I also tried to get in touch with Mr. RIGGIO to discuss some sections in his code and to find out if there are other limitations for his implementation yet. Unfortunately, Mr. RIGGIO did not answer my mails, and I finally gave up working on this implementation.

4.2 RRAPEX, an APEX implementation for the RoadRunner toolkit

The second implementation found is the master thesis work of JON HOLLSTRÖM and PER NORDLINDER [14], two students at the Umeå University, Sweden. It bases on the *RoadRunner* toolkit, an implementation of BEEP, which is written in object oriented C at *Codefactory AB*, Sweden.

This implementation has as well some limitations which principally impede this project in sending broadcast and reliable broadcast messages: it does not support multiple recipients in a single message. In addition the RRAPEX is not able to handle options as required by the specification, for instance an option specifying the attribute `mustUnderstand='true'` should be refused to be processes if it is not implemented. This mechanism is absolutely necessary for the principal aim of this project, the Reliable Broadcast Service, since all applicable entities have to know the options to reliably process these messages or to reject them if the service is not registered (c.f. Paragraph 6.2.1).

4.3 APEX working group newsgroup

When looking for further implementations, I found the APEX working group mailing list [15] which is not active anymore, and the APEX working group newsgroup [16] which can be accessed on *http://news.gmane.org/gmane.ietf.apex*. I was surprised when I realized that the last entry dated of the 18 April, 2003, say over half a year no one has shown any interest in this newsgroup. Nevertheless I began to browse through the entries and found a question by JENS ALFKE of 17 February, 2003 which pointed the way to the future:

[…] Is there actually real activity going on with APEX? Is it feasible for me to start designing a higher-level protocol that will use it, or will I have to wait a long time (or forever) for a real working APEX implementation? […]

MARSHALL T. ROSE responded the other day:

[…] (as a co-author of apex,) i think it's got a clean design and represents a very powerful evolution to store-and-forward semantics at the application layer. however, about a year ago it became clear to me that the only "killer app" on the horizon for apex was going to be instant messaging, and that field is regrettably too cluttered to support another entry […]

add stuff to jabber so that it could do everything apex could do is much smaller than the amount of energy required to: implement/deploy apex to the level that jabber is […]

if apex had come out 3 years earlier, or if there was no jabber, then i'd certainly be busy working on a couple of apex implementations […]

Note: Jabber is another set of protocols similar to APEX which enables two entities to exchange information over the Internet and is up to now principally used some instant messaging applications [17].

4.4 Conclusion

When reporting these results to my advisor, we decided to modify the task specification in order to add another task: **design a basic APEX implementation in Java**. We chose Java since there exists a stable and good integrated implementation of the underlying BEEP and it is portable to all usual platforms.

The idea is, after the design of the implementation, to integrate a *service* which handles **reliable broadcast messages**.

5 APEX Implementation

The principal goal of this chapter is to show how the Java APEX implementation, the `ch.epfl.lsr.apex` package, designed in the course of this project is structured, which the steps I performed were, and how it can be used and extended by other programmers in order to fit their own needs with specific services.

At the outset I give a notion of my first experiences and tests with the implementation of the BEEP implementation in Java, the BEEP Core. Then I show the basic structure of the API and class hierarchy I worked out according to the specification as well as the structure intended to integrate custom services later. At the same time, I indicate the package dependency and problems I had while choosing the packages. In Section 5.5, I present a brief the implementation details while in Section 5.6, I describe a relay example to show how the implementation works. Then, after configuration file definitions, I discuss some problems arisen during the implementation. Finally I conclude this chapter with some assumptions, a testing application and limitations the resulting implementation brings along.

5.1 Using the Blocks Extensible Exchange Protocol in Java

Since APEX is mapped on the Blocks Extensible Exchange Protocol it is necessary to find an implementation of this protocol which is considered as standard and proven. The choice of this implementation is not very difficult since on the official website of the BEEP standard the official Java package [18] is offered. When I started my project I worked on the two recent Java implementations of the BEEP Core:

- *beepcore-0.9.06*, 2001-07-11 which is simple to handle and offers a quick start without any other dependencies
- *beepcore-0.9.07*, 2002-09-21 needs two further packages to be installed in the class path (both packages are provided in the distribution available on the official BEEP website):
- the *jakarta-log4j* which is responsible for logging,
- the *edu.oswego.cs.dl.util.concurr*ent package whose `PooledExecutor` class is basically used for the thread management

In the course of the project another version of the BEEP Core was published, the

- *beepcore-0.9.08*, 2003-11-19 which as well depends on other packages (these are once more provided in the basic distribution):
- the *org.apache.commons.logging* package replaces the former logging package,
- the edu.oswego.cs.dl.util.concurrent package is reused

Further information about the package dependency of the APEX implementation is listed in Section 5.4.

5.2 A first BEEP example

To set up my first BEEP messaging application, I started with *beepcore-0.9.06* and followed the basic instructions of the provided notes to establish a message exchange over BEEP. This simple structure consists in a client and a server application which work on the same BEEP profile `MyProfile` identified by the URI

`http://lsrwww.epfl.ch/APEX/beepExample`.

The source of the applications is available within the distribution.

First, on the server side, as the application is started, a listener thread is initiated. Then, on the client side, one can start the application specifying a server address and a textual message to send. Now, the client establishes a connection (session) to the server specifying address, listener port and the profiles it supports,

```
session = AutomatedTCPSessionCreator.initiate(host, port, profileRegistry);
```

then, it starts a channel indicating the URI of the associated profile

```
channel = session.startChannel(MyProfile.URI);
```

and on success, it is able to send the message while specifying a reply listener.

```
channel.sendMSG(new StringDataStream(message),reply);
```

The `Reply` objects offers a mechanism that allows waiting for a reply by calling the blocking method `hasNext` of the corresponding reply instance until there is no reply left:

```
while (reply.hasNext()) {
  messageRPY = reply.getNextReply();
  ...
}
```

Now it can close the channel and the session and terminate the application.

The following listing shows the sequence of outputs of the two applications:

```
java beepExample.Server
----- working on beepcore-0.9.06 -----

Listening on port 7777
                                                java beepExample.Client 10.0.0.1 hello world
                                                      ----- working on beepcore-0.9.06 -----
                                                    initiating session ... session initiated
                                                        starting channel ... channel started
                                                                    sending "hello world" ...
new message: hello world
Listening on port 7777
                                                                                    message sent
                                                                                waiting for reply ...
sending reply: you sent [hello world]
                                                    reply received: "you sent [hello world]"
                                                          closing channel ... channel closed
                                                          closing session ... session closed
```

Listing 23 Sequence of outputs of two simple BEEP applications: Server.java and Client.java

5.3 Application Programming Interface (API)

As mentioned in Section 3.2, APEX is mapped on top of the Blocks Extensible Exchange Protocol. Moreover, it has to offer an interface to applications which use the APEX protocol. In this project the APEX is registered as a profile in the BEEP context and must be able to handle all incoming messages from the underlying layer. As it is intended by the model of the Java BEEP Core, so called handler classes exist for each profile (implementing the interfaces `Profile`, `StartChannelListener`, `RequestHandler`) which has among others a call-back methods `receiveMSG`. This method is called, as soon as a new message is received on a channel.

The intention of this project is to provide an API which offers a simple integration in either a relay or an endpoint application. Therefore it is necessary that neither connection (session) nor channel management must be done beyond the interface.

The APEX managers and APEX processes

Basically three functionalities must be offered to a programmer when using APEX in an *APEX relay process*:

- initiating the relay and specifying details (such as administrative domains it serves, corresponding mesh and edge ports, routing tables)
- send a message
- terminate bindings and attachments on this relay

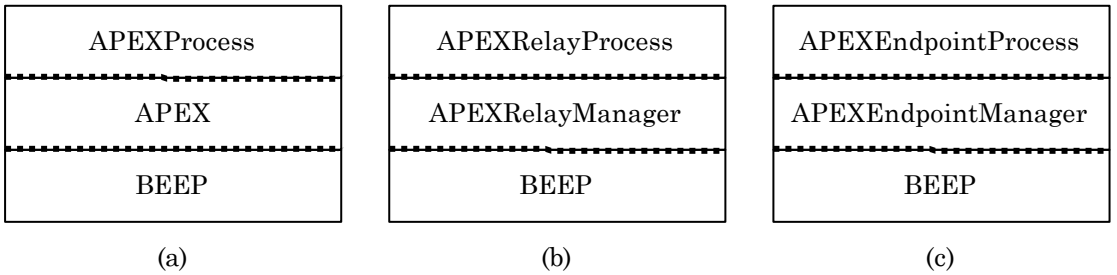

Fig. 10 Model of the APIs between the layers in (a) general, (b) a relay, and (c) an endpoint process

On the other hand, a relay process must have a method for notifications, e.g. if an endpoint attaches, a relay binds or an error occurs.

The interface to an *APEX endpoint process* is similar to the one of the APEX relay:

- attach with an arbitrary endpoint address
- send a message
- terminate an attachment

However, an endpoint process needs to handle an incoming data message as well as notifications on terminations or errors.

Since there are these two different behaviours it was evident to set up two interfaces for each an APEX relay process (`APEXRelayProcess`) and an APEX endpoint process (`APEXEndpointProcess`) which have the general property of a common APEX process (`APEXProcess`) which is an abstract class.

On the other hand, the interface to the programmer implies that there are also two

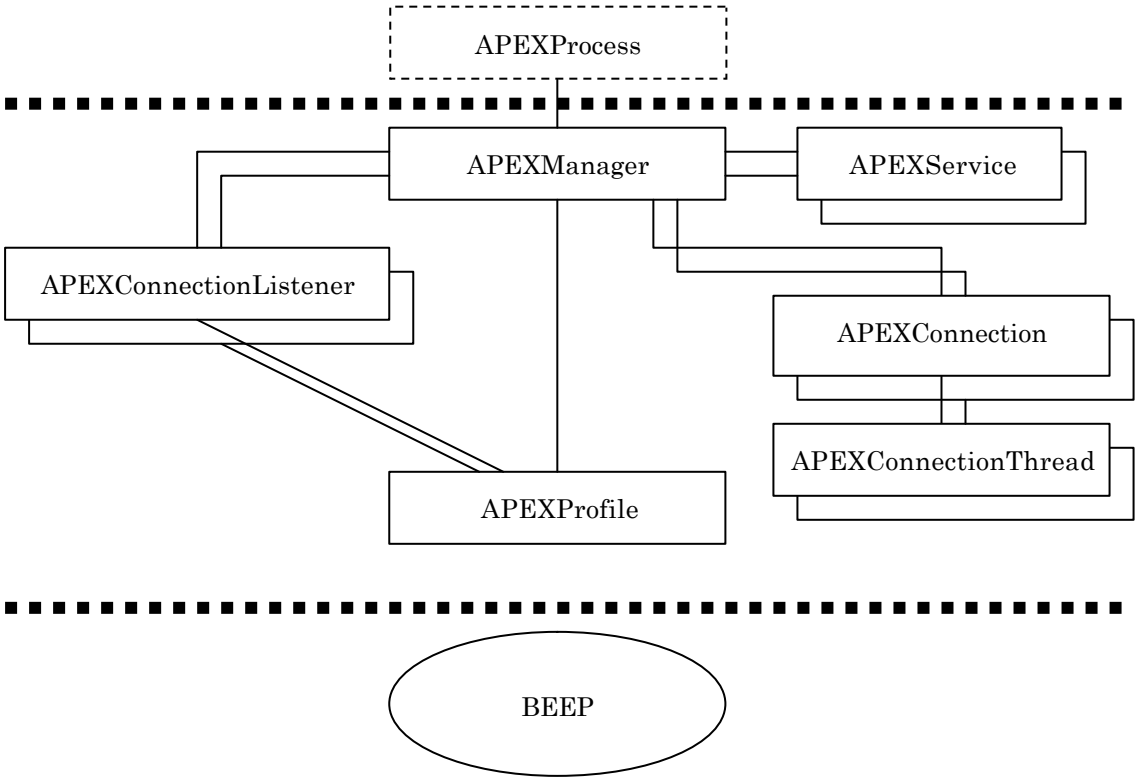

Fig. 11 Simplified structure of the class dependency within the APEX layer

managers which offer the functionalities for a relay (`APEXRelayManager`) and for a endpoint (`APEXEndpointManager`) – both extending the common class `APEX` which defines the main syntax of the APEX profile.

To use APEX, a relay process instantiates a new `APEXRelayManager` specifying the call-back class for notifications, an XML relay configuration file, and an optional Boolean for debugging. From now on, the `APEXRelayManager` parses the configuration file and opens the specified ports– the relay process only needs to wait for notification.

An endpoint process on the other hand instantiates an `APEXEndpointManager` specifying at least the class accepting call-backs. Henceforth, it is able to call the `attachAs`, `sendMessage`, and `terminate` method at every moment. For every incoming data message received at the endpoint manager, the call-back method `receiveData` is called which is free to handle the message as it deserves. Since the `receiveData` method is called from an manager independent thread, the endpoint process cannot slow down the performance of the endpoint manager by blocking or generating an exception.

5.4 Package dependency

The BEEP implementation used for this project is as mentioned the *beepcore-0.9.08* [18] which has been published on Novermber 19, 2003. It has been written by the *Invisible Worlds Inc.* and its content is subject to the *Blocks Public License* which can be found on [19]. When using the APEX implementation, the only package of *beepcore-0.9.08* necessary to put in the class path is actually the *core.jar* which contains the main mechanisms of BEEP such as `Session`, `Channel`, `Message`, `TCPSession` (Transport) and `Profile` classes. These classes are the basic utilities to write an application using a BEEP connection.

Now, some of these classes depend as well on two other packages, as well provided in the standard distribution:

- *edu.oswego.cs.dl.util.concurrent* (*concurrent.jar*) is used for a pooled thread management of the call-back queue in the `ThreadedMessageListener`
- *org.apache.commons.logging* (*commons-logging.jar*) is used to log events in a connection, principally needed on exceptions

Since APEX messages can be based on the RFC822 MIME standard, it is necessary to parse them in an appropriate way. There were several packages I analyzed in order to find the one that fits best the needs for this implementation:

- the *com.jscape.inet.mime* package of the *JSCAPE iNet Factory 5.2* [20] was the first package I used but I encountered lots of problems when building and parsing MIME multipart messages, principally when transferring a message containing binary encoded contents, the MIME implementations throws a `NullPointerException`. This can be avoided by using Base64 or UUEncoding in all messages but this restriction can not be applied because «BEEP provides an 8bit-wide path, a "transformative" Content-Transfer-Encoding (e.g., "base64" or "quoted-printable") should not be used» [8]. Thus, another package which it capable parsing correctly binary encoded contents was needed.
- the second package I used was the *netscape.messaging.mime* from the *Netscape Messaging SDK 3.51* [21] which actually fitted pretty well the requirements for the task. There was only one drawback which I could not solve: When building a MIME multipart message in APEX, for each body part a so called Content-ID header must be set as well as in the `Content-Type` header of the envelop a

`start="`*starting Content-ID*`"` attribute must be specified (c.f. Paragraph 3.3.5). Now, when parsing a well formed APEX multipart message with the net-scape.messaging.mime package, everything works well except that when addressing the `getContentTypeParams` method of the `MIMEBodyPart` class, an empty `String` is return and thus the "start" attribute is lost.

- finally I found an implementation which fitted my needs at best. The *javamail-1.3.1* [22] (JavaMail™ API 1.3.1 release), an official distribution to build, parse, send and receive E-Mail messages for Sun protocol providers. A complete description for this distribution can be found on the Sun's Java website, http://java.sun.com/products/javamail. The main classes needed of the javamail distribution are located in the *javax.mail.internet* package (*mailapi.jar*), principally `MimeMessage`, `MimeBodyPart` and `MimeMultipart`. The only drawback I realized when using this package is that it also depends on a further package, the *activation framework*. I downloaded and tested successfully with the JavaBeans™ Activation Framework 1.0.2 Release [23] (jaf-1.0.2 / *activation.jar*) which can be found as well on the Sun's Java website, http://java.sun.com/products/javabeans/jaf. Both packages run under the Sun Microsystems, Inc. Binary Code License Agreement

So summarizing briefly the jar-files needed to be accessed within the class path for using the APEX implementation:

- for BEEP: beepcore.jar, concurrent.jar, and commons-logging.jar
- for APEX: mailapi.jar and activation.jar

For XML message parsing, the *org.xml* package's document builder is used together with the *org.w3c.dom* package which offers all necessary elements. Both packages are available in recent Java SE (J2SE).

5.5 Implementation details

In this part I introduce some of the main classes of this APEX implementation in order to illustrate how the implementation works and on which ideas it bases. I begin at the upper part of the APEX layer, close to the interface to the APEX process, and dive more and more into the layer until the interface to the BEEP package.

The description of the classes is not intended to be a complete reference but to give a notion of the ideas and reasons of introducing several methods and variables or even classes. For a complete reference, please refer to the usual javadoc API documentation.

5.5.1 APEXProcess, APEXEndpointProcess, APEXRelayProcess

As mentioned above, an `APEXProcess` defines a general interface for a call-back and handler class in the context of this implementation. It has two methods to be redefined ba a programmer:

- **notification** takes two arguments, an integer `code` and an `Object` specification: on every event, e.g. attachment, binding or termination, the notification method is called and a report about the event (specified by the three-digit code) and a detailed information is given, e.g. if an endpoint attaches at a relay, `APEX.NOTIFICATION_ATTACH` and an `APEXEndpointAddress` object of the attaching endpoint is passed to a relay manager, as well in the endpoint manager almost the same happens for this event (c.f. Paragraph A.2.1).
- **debug** can be used by a programmer to receive detailed debug messages generated by all internal classes. The method above all useful when testing new services or changes in service priorities.

An `APEXEndpointProcess` needs a third method to be redefined for endpoints:

- **receiveData** is called if a data operation is incident in the endpoint manager, a `APEXDataMessage` is passed to the process.

5.5.2 APEX

As mentioned above, the `APEX` class defines the main syntax of the APEX core. The class contains all elements defined in the APEX core DTD as static final variables as well as the common error codes.

5.5.3 APEXManager, APEXRelayManager, APEXEndpointManager

The `APEXManager` class, which defines a general manager, principally provides a hash table for all classes which are called in a static way and need to save data for later events. This is principally the case for APEX services. In addition to this, it contains a transaction table and associated methods to manage incoming and outgoing transaction for each channel.

The relay and endpoint manager need to redefine principally three methods:

- **handleIncomingMessage** is called when a new message arrives: the relay and endpoint manager first find out the type of the message (attach, bind, terminate, or data) and then process it.
- **messageSent** is called when a message is sent to an entity as well as a reply is received; some services may need this information, e.g. the report service.
- **messageDiscarded** is called when a message could not be sent to an entity after several attempts.

Further information concerning relaying algorithms can be found in Section 3.4.

Both, the APEXRelayManager and APEXEndpointManager, work independent from its applications, e.g. if they realize a disconnection, they first try to reconnect and authenticate; if they do not succeed despite several attempts, they finally inform their application.

5.5.4 APEXConnection, APEXEdgeConnection, APEXMeshConnection

As specified by the RFC3340, there exist two similar types of connections, one between an endpoint and a relay, the APEX edge connection, and one between two relays, the APEX mesh connection. Now it is obvious to implement these two types again with two subclasses of a general class for any connection which defines common properties in methods and variables. This class, the `APEXConnection`, is a thread which actually supervises the general state of the connection, accepts commands, and returns status messages. The two subclasses, called `APEXEdgeConnection` and `APEXMeshConnection` respectively, implement the behaviour of an edge or mesh connection, each by having there own state machine.

To guarantee that always the same connections for an endpoint or relay is used, the construction of an APEX connection object as well as calls to its methods are managed by a factory within the `APEXEdgeConnection` and `APEXMeshConnection` respectively.

5.5.5 APEXConnectionThread

The real BEEP connection is managed by the `APEXConnectionThread` which calls back the blocking method `getMessage` in its owner class to obtain a new message to be sent. Therefore the `APEXConnection` and its two subclasses, which own an `APEXConnectionThread`, can be considered as a sort of buffer where the managers add messages and the corresponding connection thread fetches its tasks. Now as the basic connection establishment of both, the edge and the mesh connection, is the same, the `APEXConnectionThread` and its state machine can be used by the two of them. The main states are "session establishment", "channel establishment", "wait for messages", "message transfer", "wait for reply", where the last three cycle after the channel establishment. If an error occurs during processing one of the states, an appropriate handler in the `APEXConnection` is called which decides what is to do, e.g. report if no connection can be established, adjust the general state of the connection if disconnected, etc.

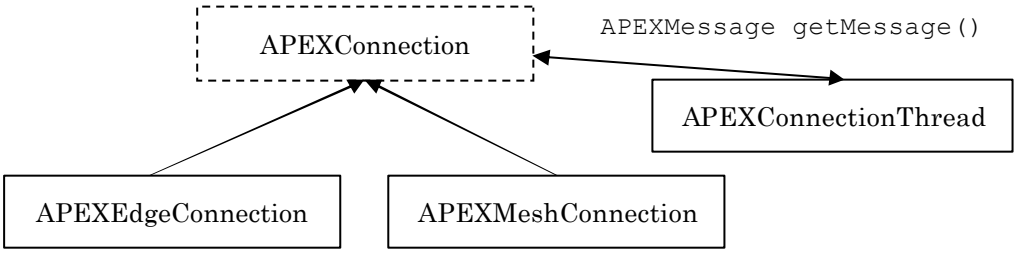

Fig. 12 The connection thread retrieves all messages from the connection it belongs to

5.5.6 APEXMessage

Each message object passed to the different handlers is an instance of `APEXMessage`. This class defines the general type of message for the four APEX core operations "data", "attach", "bind", and "terminate". An incoming BEEP message is first translated in an `APEXMessage` by the `APEXMessageParser` class where it is important to distinguish the type and content of the message. Of course it is pretty easy to recognize and to parse an attach, bind or terminate operation since there is generally only a single XML element to be analyzed, but the `data` operation has two different models: on the one hand it may be a single XML document, on the other hand a data operation can as well be nested in a MIME structured message. Therefore it first has to be verified if the message is a single body part or a multipart message – if the latter is the case the XML part (defined by the "start" attribute) first must be extracted and parsed, while the data content must be stored.

Since for every kind of APEX message common structures such as an XML string or a byte array of its content are needed, the *abstract* `APEXMessage` class offers these mechanisms with

- **getXMLMessage** abstract method which returns the XML part of an operation in a String
- **getDataStream** returns a fully generated operation (including the complete MIME structure) as a org.beepcore.beep.core.OutputDataStream which is used for the payload of a BEEP message

A exact redefinition of these methods is made by the subclasses `APEXDataMessage`, `APEXAttachMessage`, `APEXBindMessage`, and `APEXTerminateMessage` respectively. It is important to know that each of these classes has its own class variables dependent on their content and assignment. A more detailed description of the mechanisms of the most important operation, the `APEXDataMessage`, is given in the application part in this report in Paragraph A.1.3.

5.5.7 APEXProfile

The `APEXProfile` can be considered as the interface to the BEEP layer since it specifies the call-back function for incoming messages as well as for channel tasks. As mentioned in the BEEP example in Section 5.2, the `APEXProfile` associated to an APEX channel implements three interfaces:

- **org.beepcore.beep.profile.Profile** is used to be able to register as a valid profile implementation for a BEEP channel
- **org.beepcore.beep.core.StartChannelListener** defines basic methods used for channel management (`advertiseProfile`, `startChannel`, `closeChannel`)
- **org.beepcore.beep.core.MessageListener** provides the permission to register as a message listener of a channel (`receiveMessage`)

The task of the `APEXProfile` consists of two parts: on its instantiation, it has to parse the service configuration file (specified in Paragraph 5.7.2) in order to register the known services, and, as soon as it is registered with a BEEP channel it has to accept BEEP messages, parse them by means of the `APEXMessageParser`, and pass the resulting `APEXMessage` to the manager. If an error in parsing occurs, the `APEXProfile` catches it and introduces appropriate action, say respond immediately with an error reply.

5.5.8 APEXService

The `APEXService` interface defines the main methods to implement for APEX services. The model provided by the standard and the algorithms 4.4.4.1 and 4.4.4.2 of RFC3340 demand that the options invoking services are parsed before sending a message but to act as well after (for instance in report service). In addition, messages can explicitly be sent to services by addressing its well-known endpoint address, say "apex=wke". Therefore it is necessary to offer three different types of methods for a complete processing of an `option` element:

- **handleMessage** handles an incoming message destined explicitly for a service.
- **handleOption** handles an `APEXOption` for all applicable endpoints, e.g. all recipients if "targetHop" attribute of an option is "all" or "this", final recipients if the "targetHop" attribute is "final".
- **handleSent / handleDiscarded** is called by the manager if either a message as been sent to an entity, say a reply is received, or if a message has been discarded after several attempts to send it.

With this given structure it should be simple to write own APEX services each by implementing the `APEXService` interface and redefining the three abstract methods above. A complete example for setting up an APEX service is given in the appendix in Section A.3.

5.5.9 APEXStatus

To every transaction and operation an `APEXStatus` can be associated. This object acts as a status listener, call-back initiator and can be blocking at the same time. This means that, to obtain the status of a transaction or operation, a programmer either can:

- call the **blocking methods** `getStatusCode` or `getStatusReason`,
- set an **ActionListener** for the `APEXStatus`, or
- **redefine** the `receivedStatus` method within the `APEXStatus`.

5.6 In a relay: example of fully processing a data operation

In this section we are going to take a closer look at how a data operation actually is processed in a relay, from the moment it is fetched in a BEEP connection until it is removed of the message table and therefore considered as processed. While studying this order, it is useful to have a brief overview of how the classes are related to each other.

On the instantiation of an `APEXRelayManager`, automatically the configuration file of this relay is parsed and the relay is initiated. That means that according to the configuration a set of `APEXConnectionListeners` is instantiated as well as some `APEXRoutingPoints` are defined which result in the routing table of the relay. When the connection listeners are instantiated, they are registered with the `APEXProfile` as their message listener. Now, the relay is set up and ready to receive messages.

To illustrate which the path of an operation within a relay, we assume that the considered relay has been initialized with the following configuration file:

```
<config>
  <defaultPort port='912' />
  <relay mesh='912' edge='913' />
  <administrativeDomain identity='lsrwww.epfl.ch' />
  <routingPoint identity='ltiwww.epfl.ch' gateway='lcawww.epfl.ch' />
</config>
```

Listing 24 Relay configuration file

The relay therefore acts in the administrative domain 'lsrwww.epfl.ch'; a mesh connection listener is initiated on port 912 and an edge connection listener on port 913. The routing table's entry says that every message addressed to 'ltiwww.epfl.ch' is routed to 'lcawww.epfl.ch' on (default) mesh port 912.

We assume as well that two endpoints 'user1@lsrwww.epfl.ch' and 'user2@lsrwww.epfl.ch' are already attached at the relay and the relays in the administrative domains 'ltiwww.epfl.ch' and 'lcawww.epfl.ch' are set up. Finally we assume that an endpoint 'user4@ltiwww.epfl.ch' is attached at its relay.

Now, 'user1@lsrwww.epfl.ch' has sent the following message:

```
<data content='#Content'>
  <originator identity='user1@lsrwww.epfl.ch' />
  <recipient identity='user2@lsrwww.epfl.ch' />
  <recipient identity='user3@lsrwww.epfl.ch' />
  <recipient identity='user4@ltiwww.epfl.ch' />
  <option internal='statusRequest' targetHop='final' mustUnderstand='true'
          transID='5'/>
  <data-content Name='Content'>hello world</data-content>
</data>
```

Listing 25 Message sent by 'user1@lsrwww.epfl.ch'

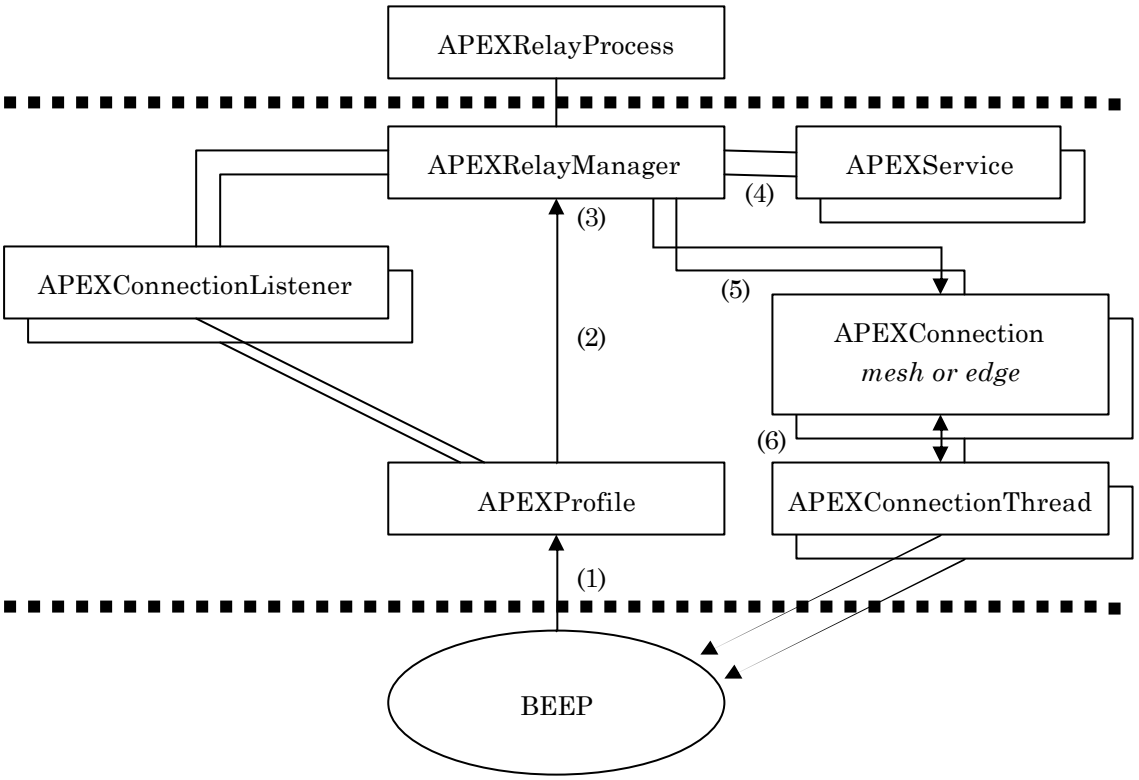

Fig. 13 Processing of a data operation in a relay

1. As soon as the BEEP peer receives the first message segment, it calls the `receiveMessage` method in the profile registered for the appropriate channel, in this case the `APEXProfile`, and passes a BEEP `MessageMSG`.

2. As discussed in the class overview section, this method parses the message and if no error occurs, passes the resulting `APEXMessage` to the manager by using the method `handleIncomingMessage`.

The syntax of the XML code of the sample operation is correct, so the `APEXMessage` *can be parsed without an exception.*

3. It is the task of the `handleIncomingMessage` to separate the four operation types of an APEX message and either to pass on the message to `attachEndpoint` for an attach operation, to `bindRelay` for a bind operation, `terminate` for a terminate operation, or *finally, as it is the case of this example,* `processData` for a data operation.

4. The `processData` method actually performs the first four steps of the "4.4.4.1 Relay Processing of Data" algorithm of RFC3340 although skipping the first point as we are not using access policies. What it in fact does is first to look up

which options are known. If there is an option which is unknown, say not registered in the service table, and the "mustUnderstand" attribute has "true" as its value, a so called error in processing occurred, an error reply is generated and the message is discarded. The next step the method performs is to sort the options according to the service priorities specified in the service configuration file and to process the `options` element. To do so, for every known option, the `handleOption` method of the appropriate service is called. At this moment, it has to be known as well for which endpoints the option is applicable (specified by the "targetHop" attribute). Finally, if all options are parsed correctly, an `ok` element is sent as reply to the sending hop and the message is passed to the method `sendMessage`.

In the case of the example, since we consider the relay 'lsrwww.epfl.ch' and the attribute of the status report option is set to "final", all endpoints within the domain 'lsrwww.epfl.ch' are applicable and are passed to the `APEXReport-Service`. *The service remembers these two endpoints (user2 and user3) and stores them in the main hash table of the* `APEXRelayManager`. *Since the XML code is well-formed and the option is known, no error occurred during processing and an* `ok` *element is returned to 'user1@lsrwww.epfl.ch'.*

5. The task of the `sendMessage` method is the most complex: first of all it has to determine which recipients are in the same administrative domain as the relay itself and which are in other domains. Then it has to regroup recipients by administrative domain to avoid sending multiple messages to the same relay consecutively but to optimize the behaviour. What follows is that first the messages to the endpoints within the same administrative domain as the relay are processed, which means to look up if they really are attached, to generate a new data operation and to add it to the message file in the appropriate `APEXEdgeConnection`. If the endpoint is not attached, the `discardMessage` method is called for this message. Then the regrouped endpoints in other administrative domains are processed, first by translating the domain according to the routing table and then by passing the generated message to the `APEXMeshConnection` obtained by calling the connection factory. The factory either returns an existing connection or instantiates a new one.

   Two of the sample recipients are in the same administrative domain as the relay. Since 'user2' has already opened a connection to the relay, the connection factory for edge connections returns a non-null connection and the message can

be passed to the obtained `APEXEdgeConnection`*. Please note that now the* `APEXEdgeConnection` *must verify if 'user2' is really attached (which is the case). The message generated for 'user2' is added to the message file within the* `APEXEdgeConnection` *for 'user2'.*

'user3' in contrast has not an entry in the connection table of the factory which directly means that it is not attached – the message is passed to the `discardMessage` *method which itself passes the message in this case to the* `handleDiscardedMessage` *method in the* `APEXReportService` *(the first endpoint status for reporting is defined and stored).*

Last but not least, the regrouped endpoints of the administrative domain 'ltiwww.epfl.ch' are processed (containing 'user4@ltiwww.epfl.ch'). Now as the routing table defines the gateway of this administrative domain to be 'lcawww.epfl.ch', the edge connection factory is polled with the translated domain and the generated message is added to the message file of the resulting connection. Please note that an `APEXMeshConnection` *automatically sends* `bind` *operations if a relay is not yet bound at the other relay.*

6. As the `APEXConnectionThread` continuously calls the blocking method `getMessage` of either the mesh or the edge connection, every message in the message file authorized to be sent (if and only if the endpoint is attached / the relay is bound) is transferred to the so called launching pad. As soon as the message is sent to the BEEP channel, a message status immediately indicates if either the message is sent or it is not. If it has been sent successfully, a new reply listener thread `APEXReplyListener` is instantiated to wait for the reply generated by the next hop. This reply listener calls back to `APEXConnection`'s method `handleReply` to report the fetched reply. If the message is not sent, say the hop is not responding or doesn't accept the message, the resend policies are activated which have different behaviours for mesh and edge connections. If the message must be discarded after several attempts to send it, again, the `discardMessage` method of the manager is called.

   For this example the message to 'user2' is sent over an existent `APEXEdge-Connection`*, so the message is added to the message file. Now the* `APEXConnectionThread` *calls the* `getMessage` *method or is blocked in its wait and finally released – however, it receives this message to send. Since the connection is established and everything works well, the message can be sent successfully to the endpoint as well as the reply arrives in the*

`APEXReplyListener`. *Therefore the* `handleReply` *method in the* `APEXEdgeConnection` *is called which itself calls the* `messageSent` *method in the manager. At this moment, the second endpoint for the status report is defined and only the last one (user4) still misses.*

Since we assume that no other error occurs during the transferring of the message to the relay in 'lcawww.epfl.ch', the `APEXReportService` *is invoked for the last time by the* `messageSent` *method. It finally can finish the status report by generating a message addressed to 'user1' and again call the* `sendMessage` *method of the manager with transfers the message to the appropriate connection for user1.*

```
<data content='#Content'>
  <originator identity='apex=report@lsrwww.epfl.ch' />
  <recipient identity='user1@lsrwww.epfl.ch' />
  <data-content Name='Content'>
    <statusResponse transID='2'>
      <destination identity='user2@lsrwww.epfl.ch'>
        <reply code='250' />
      </destination>
      <destination identity='user3@lsrwww.epfl.ch'>
        <reply code='550'>unknown endpoint identity</reply>
      </destination>
      <destination identity='user4@ltiwww.epfl.ch'>
        <reply code='250' />
      </destination>
    </statusResponse>
  </data-content>
</data>
```

Listing 26 The statusResponse message generated by the report service of 'lsrwww.epfl.ch'

5.7 Configuration Files

There are two configuration files necessary to initiate an APEX relay and one for an APEX endpoint. The main configuration file specifies the services which are provided and should be used in the APEX profile. The configuration file for the APEX relay specifies its environment, administrative domains and optional routing points.

5.7.1 Service configuration DTD

A *service configuration file* specifies the APEXServices to be registered in the APEX profile as well as their priority in service processing. Therefore it is necessary, to indicate for each service its *unique* priority in relation with all other services used. If no service configuration file is indicated or the service configuration cannot be accessed on runtime, no service is registered a priori.

```
<!--  DTD for the APEX service configuration -->

    <!ELEMENT config ((apexservice)*)>

    <!ELEMENT apexservice EMPTY>
    <!ATTLIST apexservice
              name      CDATA     #REQUIRED
              class     CDATA     #REQUIRED
              priority  UNIQUEID  #REQUIRED
    >

<!--
    DTD data types

    Name of APEX Service
        NAME         CDATA           APEX Report Service

    Class name of the APEX Service
        CLASSNAME    CDATA           pcapex.services.APEXReportService

    Unique-identifier for priorities
        UNIQID       0..2147483647   1
-->
```

Listing 27 Service configuration DTD

5.7.2 Relay configuration DTD

In order to configure an APEXRelayManager, a *relay configuration file* must be explicitly indicated. It is necessary in order to instantiate a relay – if no file or an invalid file is indicated, an `APEXException` or `APEXParsingException` respectively is thrown. The file contains the following specifications:

- a **default mesh port** number, say the port number on which the relay tries to connect when binding to other relays – if no port is specified, the IANA well known port number for APEX mesh connections (912) is set as default port
- **relay** types, for which it instantiates a connection listener on the specified port, either a mesh-connection listener with **mesh** port or a edge-connection listener with **edge** port – if no relay type is specified, no port is opened and connection listener initiated
- **administrativeDomain**s to which it serves and which may contain proper routing points (which override general routing points) – if no administrative domain is specified, the relay cannot be instantiated
- some general **routingPoint**s in the sense of a routing table specifying an **identity** address to be routed over a **gateway** and an optional **mesh** port

```
<!-- DTD for the APEX relay configuration -->

    <!ELEMENT config ( (defaultPort)?
                       (relay)+
                       (administrativeDomain)+
                       (routingPoint)* )>

    <!ELEMENT defaultPort EMPTY>
    <!ATTLIST defaultPort
              mesh      PORTNB    #REQUIRED
    >

    <!ELEMENT relay EMPTY>
    <!ATTLIST relay
              edge      PORTNB    ""
              mesh      PORTNB    ""
    >

    <!ELEMENT administrativeDomain ((routingPoint)*>
    <!ATTLIST administrativeDomain
              identity  ADMDOM    #REQUIRED
    >

    <!ELEMENT routingPoint EMPTY>
    <!ATTLIST routingPoint
              identity  ADMDOM    #REQUIRED
              gateway   ADMDOM    #REQUIRED
              mesh      PORTNB    #REQUIRED
    >

<!--
    DTD data types

    Administrative domain
        ADMDOM         CDATA         lsrwww.epfl.ch
                                     10.3.3.33

    Port number
        PORTNB         1..65535         912
-->
```

Listing 28 Relay configuration DTD

5.8 Selected problems arisen on implementation

While designing and writing the implementation described above, I faced some of essential questions and problems in implementation, e.g. MIME parsing (already described in Section 5.4), XML parsing, or connection management.

In this section, I do not want to give a complete list of all problems I encountered and how I solved them, but I would like to pick out three main difficulties to manage: first I explain how the connection control within a manager works, and then I take a look at notification call-backs. Finally I present how I treat with transaction identifiers.

5.8.1 Connection control

An important problem to solve was the connection control for each the mesh and the edge connection. The main question was: what happens on a channel if the opposite entity closes the connection – and how can this be detected. Now, if we consider the connection thread which is the most time blocked in the `getMessage` method waiting for a new message to be sent, and returns as soon as it fetches one, the next it has to do is to verify that the connection is still alive. When reading the documentation of the BEEP Session and Channel and their related classes, I did not find any method receive a connection status – the only thing which appears is that an exception is generated if a message is sent on a yet closed connection. The main problem was that this exception cannot be caught since it is an internal one and is displayed directly in the console. The next step to solve this problem was to analyze the Session class in order to find out how one can find out the status without sending a message.

The only solution, which actually is someway odd, I found, was to get a String expression of the connection before sending a new message. If the connection is down, a `NullPointerException` is thrown which can be caught and correspondent connection management initiated. The following listing gives an incomplete example of how one can prevent sending a message on a close connection:

```
try {

    channel.getSession().toString();
    MessageStatus ms = channel.sendMSG(ods, reply);

} catch (BEEPException e) {
    connection.messageNotSent(currentMessage);
} catch (NullPointerException e) {
    connection.messageNotSent(currentMessage);
}
```

Listing 29 Mechanism to verify the status of a connection

A second approach to this problem was to verify regularly if the connection is still up, since with the procedure sketched above, the `APEXConnection` (and thus the manager and the process) discern a disconnection *only* if a message is sent. This implies that if no message is sent over a long period, the other peer could have closed the connection but the `APEXConnection` is not informed.

In order to solve this problem I introduced a polling mechanism which works on the `APEXConnection` layer. Remember that as well an `APEXConnection` is a thread which manages the behaviour of the `APEXConnectionThread`. Now, as soon as the connection is established, the `APEXConnection` verifies the status of the connection in intervals, the same way as described above: obtaining a String and handle the possible exception.

5.8.2 APEXStatus notification

When sending an attach operation for instance, an endpoint process normally wants to know the resulting status of this attachment, say if the relay could have been contacted and what he responded. The basic problem I encountered is that on the one hand, an endpoint programmer desires to wait until the status becomes available, but on the other hand, during this waiting time, the process should not be blocked and work on something else.

Therefore I introduced three different models for the APEXStatus notification; a blocking one and two thread independent:

- a programmer, which wants to wait and block its process until a status becomes available, calls the blocking method **`getStatusCode`** or **`getStatusReason`** (for a textual reply) to wait. These two methods, verifying when they are called that no status code or reason has yet been set, wait for a notify call. If a status code or reason is already set, they immediately return it. Of course, one can also start a new listening thread for the status but this solution is not very sensible in my opinion.
- a common solution for the second, non blocking model I used, is that the programmer may redefine the **`actionPerformed`** method of an `Action-Listener` and to register it in the status object. Within this method, which returns him the `APEXStatus` as action source, he can now call the non blocking `getStatusCode` or `getStatusReason` as well as call-back to the main process. The status object offers particularly the possibility to add an object which can be retrieved later to work with.

- the last model offered to a programmer is that he can redefine the internal **`receiveStatus`** method of the `APEXStatus` object he instantiated to execute the code he would like to. In this case, he simply adds the new method in braces right behind the newly instantiated status object for redefinition. Again, using this model, it can be useful to add an object to the status for a later reuse.

A complete example for each model is presented in the appendix in Paragraph A.1.6

Now let us take a look behind the `APEXStatus` and see what happens when the `setStatus` method is called from an internal mechanism, say the status is updated. The first thing to be done is to set the internal status code and textual status reason. Then it checks if an action listener is set and if necessary it calls its `actionPerformed` method. At next, the internal (probably redefined) `receiveStatus` method which executes its code (by default this method is empty). Finally the waiting elements are woken up by a `notifyAll` call.

5.8.3 Transaction identifiers

As it is specified by RFC3340 [8], every transaction on a channel is identified by a *unique* transaction identifier. Thus, for each incoming operation with an associated transaction identifier, the manager has to verify that it does not exist yet and afterwards has to store it. The main problem I encountered for this operation was to link the channel on which a message was incident with messages sent earlier, e.g. if the manager attaches as an endpoint and at a given moment, the relay terminates this attachment. This does actually not seem difficult but when a user creates a message or attaches, it happens that a unique transaction identifier must be known before the channel is initiated. Therefore, I had to introduce a `channelID` which is unique within the APEX manager and can be known before a channel exists.

In order to manage the transaction identifiers, the `APEXManager` class offers some methods which add, verify, and remove transactions for each `channelID` and which also completely remove `channelID` from the internal transaction table.

5.9 Assumptions

In the course of implementing the APEX protocol two assumptions concerning the underlying BEEP implementation and structure of BEEP messages have been made.

5.9.1 Sequence of incoming message segments

The *beepcore-0.9.08* provides the class `MessageMSG` which stands for an incoming BEEP message. In order to retrieve the payload of the message, the BEEP API documentation offers the method `getDataStream` which returns an `InputDataStream`. Following the documentation, to retrieve the final byte content, one has to use the methods `getNextBufferSegment` or `waitForNextBufferSegment` (blocking) until the `InputDataStream` is complete. The question now is, if the returned `BufferSegments` are in succession or if segments arrived out of order are returned before others. This behaviour is actually not important if we use BEEP on top of a TCP connection (which is the case for this project) but the goal is to use the APEX implementation also on top of another transport layer which may not reorder segments on their arriving, if they are out of order.

Unfortunately this demand is neither described in the `BufferSegment` nor in the `InputDataStream` documentation. Analyzing the methods offered by the `BufferSegment` class, I realized that there is a method called `getOffset` without any description what kind of offset actually is meant (e.g. offset of the segment or offset in the byte array). While running several tests on the `getOffset` method, I realize that despite the message is sent in segments over the network, the returned offset is always zero – and obviously is not affected by the segmentation.

My assumption for this problem therefore is that, when retrieving the byte content of a message payload from the underlying BEEP implementation (TCP, UDP, ...), it returns the segments *in order* and *blocks until the required segment is received at the peer*.

5.9.2 MIME Multipart messages

As it is not specified particularly in the BEEP specification [1], a MIME Multipart structured message may contain more than the basic two body parts. Now, if a URI within the actual content (in XML part or another `MIMEBodyPart`) points on another part of the Multipart message, the content of the URI must be extracted separately, c.f. 5.11.3.

Thus, I recommend structuring the content as a multipart message and this one as a `MIMEBodyPart`. For details, please refer to the documentation of the JavaMail™ API [22].

5.10 Testing the APEX implementation

Despite testing continuously each new integrated mechanism during designing the APEX implementation, a realistic environment to test the APEX implementation and its services was needed. Thus, I have written a primitive application called *EndpointGUI* and on the other hand, I used the relay developed in Paragraph A.2.2, the relay application part of this project. These tools allow a user to attach as different endpoints on arbitrary relays as well to send XML and MIME structured messages.

To guarantee a good overview in the MDI window, as one is attached as many endpoints though, incoming and outgoing messages are represented within an internal frame in a tree as shown in Fig. 14 on the left. As soon as a user is attached as an endpoint, he may initiate messages specifying for each message the recipient or multiple recipients, the XML content or a file and its content-type, and optionally some options. The options integrated in the EndpointGUI are the APEX Report Service (Chapter 7) once in "final" and once in "all" targetHop mode and the APEX Reliable Broadcast Service (Chapter 6) which is always used in "final" targetHop mode.

For presentation purposes, I additionally wrote a *RelayGUI* indicating all events in a console frame. Both applications are placed in the ch.epfl.lsr.apex.gui package.

I tested the EndpointGUI and the RelayGUI on Microsoft Windows XP, SunOS 5.8 (UNIX), Red Hat Linux 9, and Debian Linux, as well as mutually, to ensure the capability of an environment independent package.

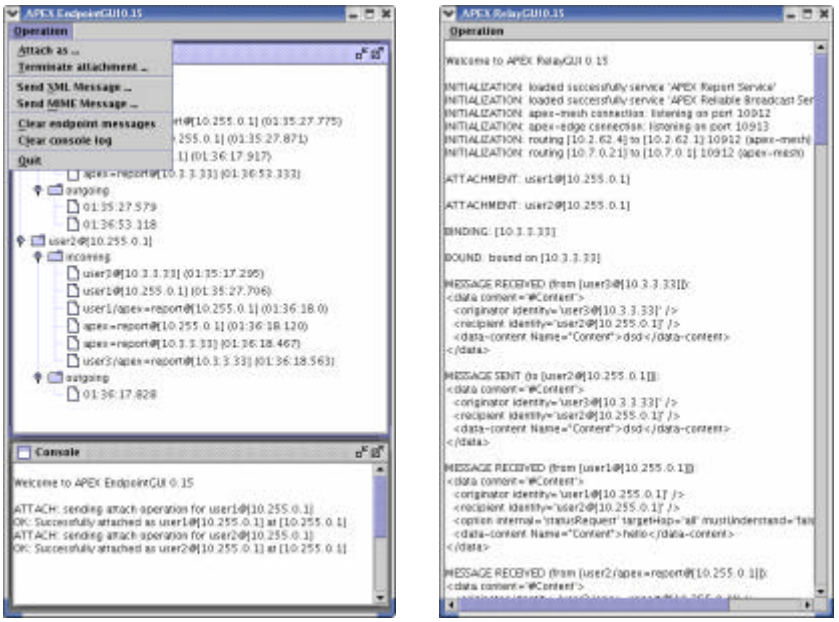

Fig. 14 The EndpointGUI (left) attached the RelayGUI (right) as two endpoints

5.11 Limitations

Since time of this project is limited and the task specification aims chiefly on the APEX core and the Reliable Broadcast Service, some parts of the complete APEX specification are not implemented in the implementation described above.

5.11.1 Additional APEX services

As mentioned above, the main focus of this project was the APEX protocol in context of group communications, thus only the APEX core and the general service model of the APEX specification is implemented. On the other hand, all mechanisms are prepared to integrate easily the remaining and new services, for instance:

- APEX Access Service (RFC3341), the mechanisms and structure for a simple integration is given (it only has to be accessed in an if-clause) but the source must be recompiled.
- APEX Presence Service (RFC3343), the structure is given by the `APEXService` interface which has to be implemented by the `APEXPresenceService` class.
- APEX Options according to RFC3342 (attachOverride, dataTiming, …), the structure as well allows a simple integration by using the `APEXService` interface and the main hash table of the `APEXManager`.

5.11.2 Option processing

Another thing, which is left for an advanced implementation, is the processing of options nested within other elements than the `data` element. RFC3340 specifies that options may also be per-originator and per-recipient, as well as they are allowed to be present in `attach`, `bind` and `terminate` elements. Although the integration of these features would not be too hard, it takes a lot of work to write the needed code.

5.11.3 MIME Multipart messages

When an endpoint receives a MIME multipart message, it is recommended to access its content by calling the method `getMimeContent` of the `APEXDataMessage` object. The obtained `MimeBodyPart` object is the content specified within the "content" attribute of the

`data` element in the XML part of the message. Now it is also possible that the multipart message contains other related contents – at least it is not prohibited by the BEEP specification [1]. So if a URI points on another disposition within the same Multipart, it can be obtained by calling the `getMultipartContent` of the `APEXDataMessage`, c.f. Paragraph 5.9.2.

5.11.4 XML content in messages

As described in Section 5.4, I use the *org.xml*'s document builder to parse an incoming operation and to obtain its elements. I specified the document builder to validate the XML structure, so it throws an exception if there are invalid elements or the document is not complete. Therefore it is necessary that even in the payload, nested in the `data-content` element, the XML structure is valid. If nonconformity is detected, the document builder aborts and the processing relay or endpoint returns an error reply specifying code 500 which stands for *general syntax error*. Now, when someone needs to transmit an invalid or even a complete XML document (containing processing instructions, start elements or inlaid document type definitions, DTD), it is recommended either to send it in a MIME Multipart message or to encapsulate the content with `<![CDATA[ ... ]]>`; thus it is not decoded.

6 Reliable Broadcast Service

The second goal of this project is to integrate a reliable broadcast mechanism in the APEX implementation of Chapter 5. As it is obvious, the simplest way in connection with the APEX protocol to integrate such a mechanism is to register a new service, the so called *Reliable Broadcast Service*, and to set up a correspondent option.

In the first section, I show how the model of the reliable broadcast is defined and works. In Section 6.2, I explain how I built the corresponding service and its corresponding option. In order to illustrate the designed Reliable Broadcast Service, an example of messages exchanged with a reliable broadcast option is presented in Section 6.3. Finally, I discuss the designed Reliable Broadcast Service in context of the APEX protocol and other services.

6.1 Reliable broadcast

The following definitions originate from the PhD thesis of my advisor, Dr. MATTHIAS WIESMANN [24]:

Reliable broadcast is a primitive that ensures that all processes in a set get a message even in the case of failure. Reliable broadcast defines two primitives R-broadcast(m) and R-deliver(m), specified as follows:

Validity	If a process R-delivers *m* then it was R-broadcast by some process.
Uniform Agreement	If a non-red process R-delivers a message m, then all non-red processes eventually R-deliver m.
Uniform Integrity	For every message m, every process R-delivers m at most once, and only if it was previously R-broadcast by sender(m).

Note: In this context, a non-red process is defined as a process which does not crash during the period of observation.

The basic behaviour of the reliable broadcast guarantees that if one process received a reliable broadcast message, it has to assure that all other affected processes receive this message as well.

One implementation of the model uses some sort of a sequence table which indicates the order of delivering the messages to the individual processes. This means that a process which received a message, sends this message to all processes after him in the sequence, say on a lower order. If one of the processes fails or crashes, the message is still sent to processes after him by processes located before the crashed process in the sequence table.

As represented in Fig. 15, although process 2 crashes and the initial message to process 3 gets lost, process 3 and process 4 receive the message at least once.

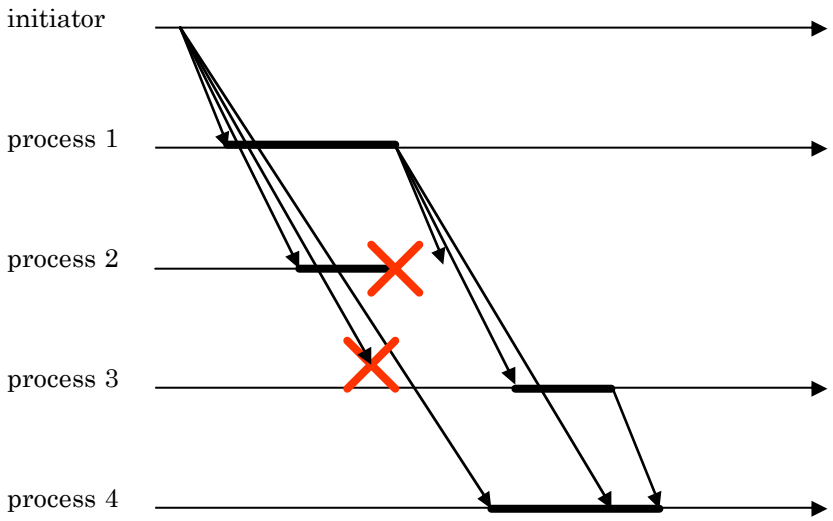

Fig. 15 A reliable broadcast message sent to multiple recipients

6.2 Implementing the Reliable Broadcast Service

In this section, I give an overview of the ideas how I introduced the notion of reliable broadcast in context of APEX. The first part shows the algorithm and functionalities I implemented in the `ch.epfl.lsr.apex.services.APEXReliableBroadcastService` while the second part defines the option invoking the service.

6.2.1 The basic behaviour

In Paragraph 3.8.3, I decided that the best way to integrate the reliable broadcast mechanism in APEX is to design a service. To do so, I defined the following properties:

- an endpoint adds a reliable broadcast option indicating the applicable end-points and their correspondent sequence. The "targetHop" attribute must be set to "final" – so only the delivering relay is applicable.
- an applicable relay (the message contains at least one recipient endpoint in the relay's domain) processing a reliable broadcast option extracts all endpoints of its administrative domain. According to these endpoints, it defines its position in the sequence, generates and sends messages to the endpoints of other administrative domains in a later position in the sequence by removing all endpoints before the first attached endpoint in the sequence. Finally, the relay sends the message to the attached recipient endpoints and on success, say an

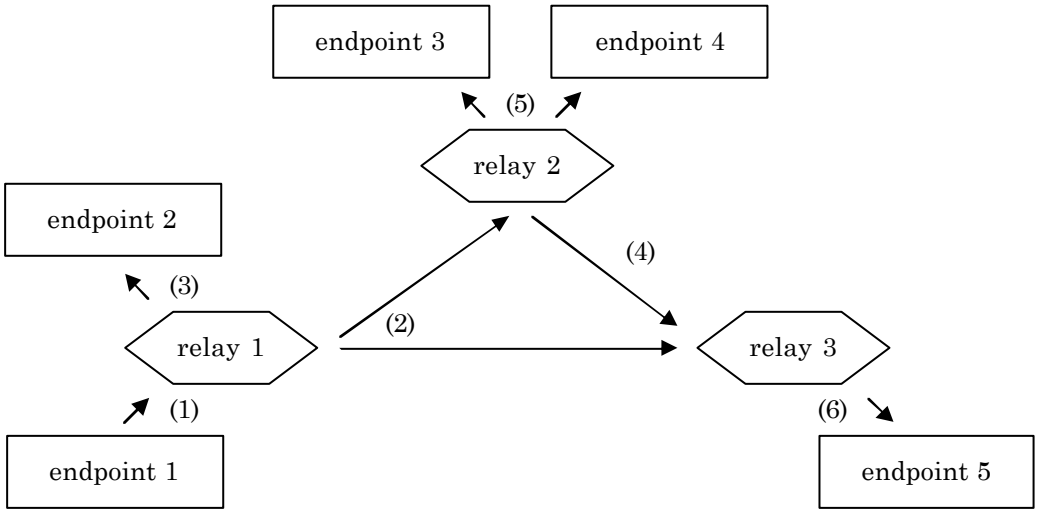

Fig. 16 Sequence of messages sent to the entities if a reliable broadcast option is present

`ok` operation is returned from all attached endpoints, considers the transaction as successfully processed.

- A relay processing a reliable broadcast option whose transaction is (already) successfully processed, silently discards the message while returning an `ok` element to the sender.

This way to set up the Reliable Broadcast Service benefits form the fact that only the relays exchange messages and each endpoint receives the message only once; hence, redundant messages between relays and endpoints are avoided (see Paragraph 3.8.3 as well).

A transaction in context of the reliable broadcast option is identified by a unique APEX transaction identifier defined in Paragraph 3.7.1.

In order to optimize the behaviour of the mechanism, the Reliable Broadcast Service picks out all endpoints attached at the processing relay, even if in the sequence they are placed after the first applicable endpoint. All endpoints in this administrative domain are added to the message as recipients and therefore they receive it instantly. This algorithm offers two main advantages, c.f. Fig. 16:

- the number of messages sent between relays is reduced since all endpoints of a domain are treated at once, and
- all services or options are directly informed about all recipients to manage, e.g. the APEX Report Service must generate only one statusResponse message for all applicable recipients.

Please note that to entirely profit from this functionality, the reliable broadcast option *must* be one of the first options to be processed and thus it is recommended to adjust the service configuration file specified in Paragraph 5.7.2 correctly.

6.2.2 The Reliable Broadcast option

An APEX option contains either an internal or an external identifier. In the case of the reliable broadcast option, the identifier is external because it is not a part of the IANA authorized specification in the RFCs. I defined the identifier to be an URI of the Distributed Systems Laboratory at EPFL:

http://lsrwww.epfl.ch/APEX/ReliableBroadcast

As mentioned before, the "targetHop" attribute must be set to "final" because only the delivering relay must process the option in order to guarantee a minimum of messages

exchanged over the APEX mesh. The "mustUnderstand" attribute is also defined to be "true" to assure that all applicable relays are able to interpret the option.

The reliable broadcast option contains a number of `endpoint` elements which indicate the endpoints to be registered in the Reliable Broadcast Service as well their position in the sequence. The first position in the sequence takes zero and all subsequent recipients take higher numbers which should be regularly ascending. If two or more endpoints have the same number, the processing service considers the sequence as the elements are nested within the `option` element as second criteria of order.

In order to obtain a reliable broadcast option, the `getReliableBroadcastOption` method can be used. It returns a well-formed `APEXOption` if the following parameters are correctly processed:

- `APEXEndpointManager`, the manager where a unique transaction identifier can be obtained, and
- `APEXEndpointAddress[]`, an array of destinations for the reliable broadcast (the index of the endpoints in the array is significant for the order)

If either the manager is not defined or the destinations array is empty, an `APEX-OptionException` is thrown.

6.2.3 DTD of a Reliable Broadcast option

```
<!--  DTD for the APEX relay configuration -->

    <!ELEMENT option ( (endpoint)+ )>
    <!ATTLIST option
              external  "http://lsrwww.epfl.ch/APEX/ReliableBroadcast"
                                    #REQUIRED
              targetHop       "final"   #REQUIRED
              mustUnderstand  "true"    #REQUIRED
              transID         INTNB     #REQUIRED
    >

    <!ELEMENT endpoint EMPTY>
    <!ATTLIST endpoint
              identity        ENDPOINT  #REQUIRED
              order           INTNB     #REQUIRED
    >
<!--
    DTD data types

    Endpoint address
        ENDPOINT    entity              user1@lsrwww.epfl.ch

    Integer number
        INTNB       1..2147483647       5
-->
```

Listing 30 DTD of a Reliable Broadcast option

6.3 An example

The most effective way to illustrate the behaviour described above, is to go along a simple example. Consider 'user1@lsrwww.epfl.ch' wants to send a reliable broadcast message to 'user2@lsrwww.epfl.ch', 'user3@ltiwww.epfl.ch', and 'user4@icawww.epfl.ch'. Therefore he instantiates a new `APEXDataMessage`, adds the option built by `APEXReliableBroadcastService`, and finally sends the message.

Fig. 16 may illustrate the example assuming "endpoint 1" to be 'user1' and "endpoint 2" to be 'user2', both attached at "relay 1" alias 'lsrwww.epfl.ch'; "endpoint 3" to be 'user3' attached at "relay 2" alias 'ltiwww.epfl.ch'; and "endpoint 5" to be 'user4' attached at "relay 3" alias 'icawww.epfl.ch'.

The (hard coded) source code for this operation would look like this (where the `APEXManager` is already instantiated and the originator is considered as attached):

```
APEXEndpointAddress[] recipients = new APEXEndpointAddress[3];
recipients[0] = new APEXEndpointAddress("user2@lsrwww.epfl.ch");
recipients[1] = new APEXEndpointAddress("user3@ltiwww.epfl.ch");
recipients[2] = new APEXEndpointAddress("user4@icawww.epfl.ch");

APEXDataMessage message = new APEXDataMessage(originator,
                                              recipients,
                                              content);

APEXOption reliableBroadcastOption =
    APEXReliableBroadcastService.getReliableBroadcastOption(apexManager,
                                                             recipients);

message.addActiveOption(reliableBroadcastOption);

APEXStatus status = new APEXStatus();

apexManager.sendAPEXDataMessage(message, status);
```

Listing 31 Adding a Reliable Broadcast option for multiple recipients to a data message

Assuming that the content is XML, the resulting message sent to the relay, is:

```
<data content='#Content'>
  <originator identity='user1@lsrwww.epfl.ch' />
  <recipient identity='user2@lsrwww.epfl.ch' />
  <recipient identity='user3@ltiwww.epfl.ch' />
  <recipient identity='user4@icawww.epfl.ch' />
  <option external='http://lsrwww.epfl.ch/APEX/ReliableBroadcast'
          targetHop='final' mustUnderstand='true' transID='2'>
    <endpoint identity='user2@lsrwww.epfl.ch' order='0' />
    <endpoint identity='user3@ltiwww.epfl.ch' order='1' />
    <endpoint identity='user4@icawww.epfl.ch' order='2' />
  </option>
  <data-content Name='Content'>...</data-content>
</data>
```

Listing 32 A message containing a reliable broadcast option

Now, incoming in relay 'lsrwww.epfl.ch', the message is analyzed and the options are processed:

- since the `APEXReliableBroadcastService` knows from its transaction table that it has not yet processed this option, it immediately sends a new message to all endpoints in the sequence list (sorted by ascending order) which are not in the same administrative domain (and removes them from the initial message) – a new option with a new sequence list is generated for each message and only relevant `endpoint` elements are added.

  One message is sent to relay 'ltiwww.epfl.ch' addressed at 'user3@ltiwww.epfl.ch'. It contains a reliable broadcast option which includes only the endpoints the Reliable Broadcast Service on the next applicable relay has to treat.

```
<data content='#Content'>
  <originator identity='user1@lsrwww.epfl.ch' />
  <recipient identity='user3@ltiwww.epfl.ch' />
  <option external='http://lsrwww.epfl.ch/APEX/ReliableBroadcast'
          targetHop='final' mustUnderstand='true' transID='2'>
    <endpoint identity='user3@ltiwww.epfl.ch' order='0' />
    <endpoint identity='user4@icawww.epfl.ch' order='1' />
  </option>
  <data-content Name='Content'>...</data-content>
</data>
```

Listing 33 The message sent to 'user3@ltiwww.epfl.ch'

  Since 'user4@icawww.epfl.ch' is the last endpoint in the sequence, it is the only endpoint to be processed by the Reliable Broadcast Service on relay 'icawww.epfl.ch'. However, this relay receives this message twice and discards it once; the second is sent by the Reliable Broadcast Service of relay 'ltiwww.epfl.ch'.

```
<data content='#Content'>
  <originator identity='user1@lsrwww.epfl.ch' />
  <recipient identity='user4@icawww.epfl.ch' />
  <option external='http://lsrwww.epfl.ch/APEX/ReliableBroadcast'
          targetHop='final' mustUnderstand='true' transID='2'>
    <endpoint identity='user4@icawww.epfl.ch' order='0' />
  </option>
  <data-content Name='Content'>...</data-content>
</data>
```

Listing 34 The message sent to 'user4@icawww.epfl.ch'

- back to relay 'lsrwww.epfl.ch': the initial message containing the remaining endpoints as recipients (endpoints in the administrative domain and endpoints which are not in the sequence list) is passed to the `sendMessage` method and sent as well.

The message directly addressed at the attached endpoint of the 'lsrwww.epfl.ch' relay, 'user2@lsrwww.epfl.ch', is sent over the correspondent edge connection.

```
<data content='#Content'>
  <originator identity='user1@lsrwww.epfl.ch' />
  <recipient identity='user2@lsrwww.epfl.ch' />
  <data-content Name='Content'>...</data-content>
</data>
```

Listing 35 The message finally delivered to each recipient

6.4 Discussion: Reliable Broadcast and APEX

The Reliable Broadcast Service in APEX brings along some restrictions to guarantee an optimal behaviour but offers on the other hand a intermediate mesh configuration independent reliable messaging mechanism. In this section, I discuss these restrictions and their effects if they are not kept.

6.4.1 The targetHop attribute

As mentioned in Paragraph 6.2.2, the "targetHop" attribute's value of the reliable broadcast option must be "final". This property, on the one hand, effects that only relays which are *delivering* the message directly to one or more attached endpoints, process the option. If the "targetHop" attribute's value is "all", all intermediate relays on the messages' trajectory and all endpoints process the option and initiate new messages. By setting the value to "final", the number of messages sent between the relays is thus reduced to a minimum

On the other hand, this restriction offers an enormous advantage: the Reliable Broadcast Service needs only to be registered on delivering relays at the edge of the APEX mesh, whereas intermediate relays are not forced not to know it to guarantee the mechanism. Hence, a reliable broadcast message can be sent over an *arbitrary* APEX mesh where the Reliable Broadcast Service may not be registered on intermediate relays.

6.4.2 The mustUnderstand attribute

As mentioned as well in Paragraph 6.2.2, the "mustUnderstand" attribute's value of the reliable broadcast option must be "true". This is necessary to guarantee that the mechanism is entirely working. However, if on one applicable relay the Reliable Broadcast Service is not registered and the "mustUnderstand" attribute's value is "false", the relay silently ignores the option and sends the message only to the present message recipients – *not* to all recipients indicated in the sequence table, though, since the reliable broadcast option is *not* processed: reliable messaging is *not* guaranteed.

If the attribute's value is "true" on the other hand, it rejects the processing of the message and returns an error reply.

6.4.3 APEX Report Service

By default, the APEX protocol does not return status responses on successful delivering or discarding a message. As presented in Paragraph 3.6.1, if a statusRequest option is present in message, the APEX Report Service invoked. In the case of the Reliable Broadcast Service, when a statusRequest option is processed, the resulting statusResponse message is not sent until the status of the messages to all attached recipient endpoints is known. Status of messages to other relays are not reported as long as to error in sending occurs.

The Reliable Broadcast Service combined with the APEX Report Service results in a reliable messaging service with *reports on delivery* of all messages to correspondent recipient endpoints.

6.4.4 hold4Endpoint option

The hold4Endpoint option, which is not implemented in the APEX core implementation of Chapter 5 but presented in Paragraph 3.5.3, transforms the Reliable Broadcast Service into an entirely reliable service. By default, if a message is addressed to an endpoint which is not attached at the APEX mesh, the message is discarded – if the statusRequest option is present in addition, a negative statusResponse reply is sent. If the hold4Endpoint option is present as well, the message is queued and sent as soon as the demanded recipient endpoint attaches.

Combining the reliable broadcast and the hold4Endpoint option ensures that the message *is received* at each specified endpoint at some time.

7 APEX Report Service

In the course of the implementation, I developed the `ch.epfl.lsr.apex.services.APEXReportService`, an implementation of the APEX Report Service specified in RFC3340 and presented in Paragraph 3.6.1. This service is basically needed for status responses; say to obtain information if an initiated message is received by its destination endpoint(s). In Section 7.1, I describe the way I implemented the APEX Report Service and the most important algorithm I developed. Section 7.2 gives a brief application example of this service.

7.1 Implementing the APEX Report Service

In order to implement the APEX Report Service, like seen for the Reliable Broadcast Service, one needs to create a new class extending the general `APEXService` class. The next step to be determined is how the main abstract methods `handleMessage`, `handleOption`, `handleSent` and `handleDiscarded` should be redefined.

As RFC3340 already specifies the name and type of the option, I refer to Paragraph 3.6.1 for more details.

7.1.1 handleMessage method

The `handleMessage` method is actually the easiest method to be solve, since the standard specifies that no message should be addressed at the APEX Report Service – this method can be left empty.

7.1.2 handleOption method

As specified in Paragraph 5.5.8, if an APEX Report Service option is processed in a relay, the `handleOption` method is called. This method must now extract all applicable endpoints for which it has to gather the message status and report in the statusResponse message. As it turns out of the processing algorithm discussed in Section 3.4, the option is processed before the messages are sent and their status is known. Therefore the Report Service uses the hash table of its manager to store the information about the endpoints to be treated and adds for each applicable endpoint an empty status. If a message addressed

to an endpoint is successfully sent or if it is discarded, this status must be updated.

It is important to note that if the service is run on an endpoint, it can immediately respond with a statusResponse message since the message has been captured successfully and does not need to be relayed to another entity within the APEX mesh.

7.1.3 handleSent / handleDiscarded methods

The `handleSent` as well as the `handleDiscarded` method are called if the status of a message is known; either the next hop returned an ok operation or an error in sending the message arose – therefore these two methods are only relevant on a relay. Now, the status of the addressed endpoint or endpoints (if it is a multicast message) in the above mentioned hash table can be updated according to the event. After the updates of this "status table", the service needs to check if at the moments all states are defined and if this is true, it can send the requested statusRespond message. If some states still remain unresolved, the "status table" is stored for further processing.

While the `handleSent` message is straight forward, the `handleDiscarded` method is a bit more delicate. Considering for instance the event that a relay cannot send a message to another relay after several attempts and the message contains a reliable broadcast option having a "targetHop" attribute's value "final". It turns out at this moment that, since this attribute is present, the option may not have been processed but nevertheless, the report service has to respond with an error message. Therefore the algorithm of updating the status elements can be described like the following:

1. if the option has applicable endpoints for the option
    a. if the discarded message is addressed to at least one of these, the status the endpoint(s) must be updated
    b. if the destination endpoint is an not applicable, a new status object is instantiated and sent to the originator
2. if the option has no applicable endpoints (the case of a discarded message in an immediate relay and "final" valued "targetHop" attribute), status objects for all destinations are instantiated and returned to the originator.

Please note that this algorithm need not to be integrated in the `handleSent` method since no reporting is needed if the message is sent successfully and the endpoint is not applicable on this relay.

7.1.4 getStatusRequestOption method

In order to facilitate the instantiation of an APEX status request option for programmers of endpoint application, the `APEXReportService` class offers some static methods `getStatusRequestOption` which return an `APEXOption` object for given arguments.

7.2 A Status Request testing example

Consider in this example 'user1@lsrwww.epfl.ch' would like to send a message to multiple recipients. At the same time, he would like to know if the message has been received correctly. Therefore he adds a statusRequest `APEXOption`, generated by the static method `getStatusRequestOption`, to this message – naturally he could also instantiate a new `APEXOption` specifying all arguments himself.

The hard coded instructions to instantiate and send this message would look like the following listing:

```
APEXDataMessage adm = new APEXDataMessage (this.endpointAddress,
                                           recipients,
                                           content);

APEXOption request = APEXReportService.getStatusRequestOption(
                                           apexManager,
                                           this.endpointAddress,
                                           APEX.TARGETHOP FINAL);

adm.addOption(request);
apexManager.sendMessage(adm, status);
```

Listing 36 Adding a final statusRequest option to a data message

Now for this example assuming the following setup for the specified recipients:

- 'user2@lsrww.epfl.ch' is attached at its relay
- 'user3@lsrwww.epfl.ch' is not set up or attached at its relay
- 'user4@ltiwww.epfl.ch' is set up and works correctly
- 'user5@lcawww.epfl.ch' no relay is even working on its domain

The message sent by 'user1@lsrwww.epfl.ch' to its relay looks like:

```
<data content='#Content'>
  <originator identity='user1@lsrwww.epfl.ch' />
  <recipient identity='user2@lsrwww.epfl.ch' />
  <recipient identity='user3@lsrwww.epfl.ch' />
  <recipient identity='user4@ltiwww.epfl.ch' />
  <recipient identity='user5@lcawww.epfl.ch' />
  <option internal='statusRequest'
          targetHop='final'
          mustUnderstand='false'
          transID='2' />
  <data-content Name='Content'>hello world</data-content>
</data>
```

Listing 37 A message containing a final statusRequest option

The endpoint receives the following messages generated by the `APEReportService` on the respective relays:

- lsrwww.epfl.ch reports that the message has been sent successfully to 'user2@lsrwww.epfl.ch' but the 'user3@lsrwww.epfl.ch' is not known

```
<data content='#Content'>
  <originator identity='apex=report@lsrwww.epfl.ch' />
  <recipient identity='user1@lsrwww.epfl.ch' />
  <data-content Name='Content'>
    <statusResponse transID='2'>
      <destination identity='user2@lsrwww.epfl.ch'>
        <reply code='250' />
      </destination>
      <destination identity='user3@lsrwww.epfl.ch'>
        <reply code='550'>unknown endpoint identity</reply>
      </destination>
    </statusResponse>
  </data-content>
</data>
```

Listing 38 A statusRespose message for final recipients from 'lsrwww.epfl.ch'

- some time later, 'lsrwww.epfl.ch' reports as well that it was not able to transfer the message addressed for 'user5@lcawww.epfl.ch' to relay 'lcawww.epfl.ch' after several attempts and therefore has been discarded

```
<data content='#Content'>
  <originator identity='apex=report@lsrwww.epfl.ch' />
  <recipient identity='user1@lsrwww.epfl.ch' />
  <data-content Name='Content'>
    <statusResponse transID='2'>
      <destination identity='user5@lcawww.epfl.ch'>
        <reply code='450'>an intermediate relay does not respond</reply>
      </destination>
    </statusResponse>
  </data-content>
</data>
```

Listing 39 A statusRespose message for recpients not delivered to the next hop

- finally, 'ltiwww.epfl.ch' reports upon the successful relaying operation to 'user4@ltiww.epfl.ch'

```
<data content='#Content'>
  <originator identity='apex=report@ltiwww.epfl.ch' />
  <recipient identity='user1@lsrwww.epfl.ch' />
  <data-content Name='Content'>
    <statusResponse transID='2'>
      <destination identity='user4@ltiwww.epfl.ch'>
        <reply code='250' />
      </destination>
    </statusResponse>
  </data-content>
</data>
```

Listing 40 A statusRespose message for final recipients from 'ltiwww.epfl.ch'

8 Conclusion

While analyzing and implementing the APEX core, for the first time I met the task of implementing a recent standard that up to now is known only by a few people. In the course of implementation, there were a lot of things to be determined and interpreted out of the specification. In many cases, I had to search on the Internet for detailed information concerning the underlying BEEP, which I basically found in the official newsgroups of BEEP and APEX. During the coding phase, I also experienced how important it is to first blue-print a general structure, and then to proceed on and on deeper into details while documenting every step. Hence, I kept track when modifying parts or writing the final documentation presented in this report as well as when completing the javadoc comments in the end of the project.

The BEEP in my opinion is a very powerful messaging framework to design robust application protocols, in the BEEP context called channels, and ease enormously the designer from basic socket programming or encryption. I think I really profited to get to know this protocol and to intensively work with it; I am sure to cross it some day later again.

As well the APEX, as far as I can judge the protocol, is a very clean and well structured protocol and due to its flexibility can be used in many domains. In my opinion, it is a real pity that APEX has not persuaded and has not overcome the resistance every new protocol has to fight against. I hope, by providing the Java implementation of this project, APEX can be at least used in context of group communications.

For my part, I enjoyed working on the APEX and its implementation, although it took me a large amount of time to design and write the implementation, the services and correspondent applications. However, I gained enormous knowledge in Java, especially in synchrony and parsing issues, and of course some models of group communications, basically the mechanism of reliable broadcast.

9 Acknowledgements

I would like to thank the following persons and institutions which helped me on my work in this project, principally developing the APEX implementation:

- Dr. MATTHIAS WIESMANN
  my project advisor, he was always there when I needed him and helped me on every problem I had.
- Prof. ANDRÉ SCHIPER and the EPFL-LSR
  for offering this project in which I experienced work on protocol analysis and implementation.
- PATRIK BLESS AND MARTIN RUBLI
  for some Java advices and testing on Debian Linux.
- Invisible Inc.: beepcore-java
  for providing the Java BEEP implementation which is for the most part very well documented.
- IBM Corp.: The Eclipse Platform
  for developing this powerful programming tool – I really love it!
- JAMES GOSLING, Java Technology inventor, and related developers
  for giving rise to this marvellous technology.

A APEX applications – a short tutorial

In this chapter we are going to set up two tiny processes in order to illustrate how to use the APEX implementation introduced in Chapter 5. In the first section we present an APEX endpoint process. This endpoint will work together with the APEX relay set up in the second section. The complete source code of each process is given in the last parapgraph of each section.

The third section of this chapter gives a notion of how an APEX service should be designed by giving a simple example.

A.1 An APEX endpoint process

To initiate an APEX endpoint, it is necessary to define a class which is able to receive call-backs from the APEX endpoint manager, e.g. to receive incoming data messages. This call-back class must implement the `APEXEndpointProcess` interface and define its abstract methods `receiveData`, `notification` and `debug` (c.f. Paragraph A.2.1).

A.1.1 The APEX endpoint manager

The next thing to be done is to instantiate the `APEXEndpointManager` which, from now on, is the drop-in centre for all tasks to be accomplished. Basically the `APEXEndpointManager` needs to know the call-back class (`APEXEndpointProcess`) and the service configuration file defined in Paragraph 5.7.1. In addition, the default edge port number for connections to relays can be modified by passing an integer port number as argument.

A.1.2 Attach as an endpoint

Now the endpoint is set up and the first operations can be started: the first thing we need to do is to attach as an endpoint to be able to send messages. To do so, the `APEXEndpointManager` offers the `attachAs` method which takes an `APEXEndpointAddress` and an `APEXStatus`. It is not recommended to pass a `null` value as status in order to handle an exception on attaching (say if the relay refuses to attach the endpoint or even the relay cannot be contacted). Since the `APEXStatus` offers a blocking method `getStatusCode` which returns a three-digit status code (250 for a successful transaction,

c.f. Paragraph 3.7.5) and a textual reason with `getStatusReason`, one can either wait for the status code in the current thread. For simplicity, we use in the example the `getStatusCode` method directly which blocks the process. For a more flexible solution, please refer to Paragraph A.1.6, which presents examples for thread independent models.

A.1.3 Send a data operation

Now the process is attached as at least one endpoint at a relay, and therefore it is able to send an `APEXDataMessage` which may have several properties and different forms. A programmer using the APEX implementation is mostly dealing with the `APEXDataMessage` since he needs to instantiate it in order to use the `sendDataMessage` method as well as he needs to extract incoming data in the `receiveData` call-back method. We now take a closer look at this class. In addition to this intro it has to be mentioned that the `APEXEndpointManager` offers four different `sendMessage` methods which allows a programmer to by-pass the instantiation of an `APEXDataMessage` by directly passing the relevant arguments to one of these methods.

APEXDataMessage

To build an `APEXDataMessage` basically three parameters must be given: the originator, the destination, and what kind of data should be sent. There are two different types of the data content, textual content (e.g. XML code) or binary content (e.g. an image file). In addition, to modify the basic behaviour of the APEX model, one needs also to initialize some options. Please note that for multiple recipients, a `HashSet` or an array containing the `APEXEndpointAddresses` of the recipients is required.

In order to send a simple textual (or XML) content data operation, the following example shows how to use the `APEXDataMessage` class (note that `manager` is the `APEXEndpointManager` and the originator has already sent an attach operation and the `getStatusCode` method is blocking until the message has left the endpoint). Please refer to Paragraph 5.11.4 for limitations in context with XML content.

```
APEXEndpointAddress originator = new APEXEndpointAddress("user1@lsrwww.epfl.ch");
APEXEndpointAddress recipient = new APEXEndpointAddress("user2@lsrwww.epfl.ch");

String content = "hello";

APEXDataMessage message = new APEXDataMessage(originator, recipient, content);

APEXStatus status = new APEXStatus();
manager.sendAPEXDataMessage(message, status);
int statusCode = status.getStatusCode();
```

Listing 41 Instantiating and sending an data message with XML content

To send a more complex MIME content data operation to multiple recipients (multicast), one needs on the one hand to add the recipients to a `HashSet` and, on the other hand, instantiate a `MIMEBodyPart` (javax.mail.internet.MimeBodyPart) object containing the content. The *javamail* distribution [22] is well documented and therefore thus not discussed here.

We assume that an image "image.gif" should be the content of the data operation, encoded as binary.

```
APEXEndpointAddress originator =
    new APEXEndpointAddress("user1@lsrwww.epfl.ch");

   HashSet recipients = new HashSet();
recipients.add(new APEXEndpointAddress("user2@lsrwww.epfl.ch"));
recipients.add(new APEXEndpointAddress("user3@ltiwww.epfl.ch"));

InputStream is = new FileInputStream("image.gif");
byte[] byteContent = getBytes(is);

InternetHeaders headers = new InternetHeaders();
MimeBodyPart bodyPart = new MimeBodyPart(headers, byteContent);

bodyPart.addHeader(APEXMessage.FILENAME,entry);
ContentType ct = new ContentType();
ct.setPrimaryType("image");
ct.setSubType("gif");

bodyPart.addHeader(APEXMessage.CONTENTTYPE,ct.toString());
bodyPart.addHeader(APEXMessage.CONTENTTRANSFERENCODING,
                   APEXMessage.ENC BINARY);

APEXDataMessage message = new APEXDataMessage(originator,
    recipient, content);

APEXStatus status = new APEXStatus();

apexManager.sendAPEXDataMessage(message, status);
```

Listing 42 Instantiating and sending an data message with MIME Multipart structured content

This code will generate a `MIMEBodyPart` object which contains the following content:

```
Content-Type: image/gif
Content-Transfer-Encoding: binary
Filename: image.gif

GIF89a...
```

Listing 43 The content of `MIMEBodyPart bodyPart` in Listing 42

One could as well work with "base64" or "quoted-printable" but needs to encode it by the auxiliary encoder classes provided by the *javamail* distribution [22], namely for instance the `BASE64EncoderStream` of the *com.sun.mail.util* package in *mailapi.jar* (for complemented information concerning encoding, please refer to Section 5.4 and Section 4.1 Use of MIME and XML in RFC3340 [8]).

A.1.4 Receiving a data operation

As mentioned above, a call-back method called `receiveData` is required to be redefined in when implementing the `APEXEndpointProcess` interface. The following instructions briefly show how the received `APEXDataMessage` is analyzed and the content is correctly extracted.

First of all, it must be found out if the data message received at the peer contains MIME or XML content. To do so, `APEXDataMessage` offers the method `getContentType` which returns an integer. This integer can be compared to two final constant values

- `APEXDataMessage.MIMECONTENT` for MIME content, or
- `APEXDataMessage.XMLCONTENT` for XML content.

Once the content type is determined either the `getMimeContent` or the `getXML-Content` method can be called in order to retrieve a `MimeBodyPart` or an `Element` (org.w3c.dom.Element) respectively. The `MimeBodyPart` represents the whole MIME part of the multipart identified by the content URI attribute of the XML part while the `Element` represents the `data-content` element containing the XML elements of the content. In order to treat MIME Multipart structured messages please, refer also to Paragraph 5.9.2 and Paragraph 5.11.3 which indicates possible limitations.

It may be important to know that the `receiveData` method is called from a manager independent thread, thus the endpoint process cannot affect or slow down the performance of the endpoint manager blocking while processing.

A.1.5 A complete APEX endpoint

The following listing is a complete APEX endpoint capable to attach with a single endpoint address passed as the first argument, to send XML content to multiple users, and to receive and display XML and MIME content messages. The source of "Endpoint.java" is available in the ch.epfl.lsr. apex.example package of the distribution.

```
public class Endpoint implements APEXEndpointProcess {

    APEXEndpointAddress endpointAddress;
    APEXEndpointManager apexManager;

    public static void main(String[] args) {
        if (args.length > 0)
            try {
                String svcCfgFile = "";
                int port = 0;
                if (args.length > 1) {
                    try {
```

```
                    port = Integer.parseInt(args[1]);
                }
                catch (NullPointerException e) {
                    svcCfgFile = args[1];
                }
            }
            if (args.length > 2)
                port = Integer.parseInt(args[2]);
            new Endpoint(new APEXEndpointAddress(args[0]),svcCfgFile,port);
        } catch (Exception e) {
            System.out.println("Error: "+e.getMessage());
            System.exit(1);
        }
    else
        System.out.println("Usage: java Endpoint endpointAddress [svcCfg] [port]");
}

public Endpoint(APEXEndpointAddress aea, String svcCfg, int port) throws Exception {
    this.endpointAddress = aea;
    if (port == 0)
        this.apexManager = new APEXEndpointManager(this, svcCfg);
    else
        this.apexManager = new APEXEndpointManager(this, svcCfg, port);

    System.out.print("\nAttaching as "+aea.getEndpointAddress()+" ... ");
    APEXStatus attachStatus = new APEXStatus();
    apexManager.attachAs(aea, attachStatus);
    int attachStatusCode = attachStatus.getStatusCode();
    String attachReason = attachStatus.getStatusReason();
    if (attachStatusCode == APEX.STATUS TRANSACTION SUCCESSFUL)
        System.out.println("ok! ("+attachReason+")");
    else {
        System.out.println("Error: "+attachReason+" (code "+attachStatusCode+")");
        throw new Exception("could not attach!");
    }

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String entry = null;
    String content = "";
    boolean run = true;
    HashSet recipients = new HashSet();
    int state = 0;
    do {
        switch(state) {
            case 0:
                System.out.print("\nEnter recipient address: ");
                state = 1;
                break;
            case 1:
                try {
                    APEXEndpointAddress r = new APEXEndpointAddress(entry);
                    recipients.add(r);
                    System.out.print("Add another recipient (y or n): ");
                    state = 2;
                } catch (APEXParsingException e) {
                    System.out.println("Error: "+e.getMessage());
                    System.out.print("\nEnter recipient address: ");
                    state = 1;
                }
                break;
            case 2:
                if (entry.equalsIgnoreCase("y")) {
                    System.out.print("\nEnter recipient address: ");
                    state = 1;
                }
                else if (entry.equalsIgnoreCase("n")) {
                    System.out.println("\nEnter XML content (terminate with .):");
                    content = "";
                    state = 3;
                }
                else System.out.print("Add another recipient (y or n): ");
                break;
            case 3:
```

```
                    content += "\r\n";
                    if (!entry.equalsIgnoreCase(".")) {
                        content += entry;
                        break;
                    }
                case 4:
                    if (content.length() <= 2) {
                        System.out.println("\nEnter XML content (terminate with .):");
                        state = 3;
                    }
                    else {
                        APEXStatus status = new APEXStatus();
                        System.out.print("\nSending message to relay ... ");

                        APEXDataMessage adm = new APEXDataMessage (this.endpointAddress,
                                                   recipients,
                                                   content);
                        apexManager.sendMessage(adm, status);
                        int statusCode = status.getStatusCode();
                        String reason = status.getStatusReason();
                        if (statusCode == APEX.STATUS_TRANSACTION_SUCCESSFUL)
                            System.out.println("ok! ("+reason+")");
                        else
                            System.out.println(reason+" (code "+statusCode);
                        recipients.clear();
                        System.out.print("\nEnter recipient address: ");
                        state = 1;
                    }
            }
            do {
                try {
                    entry = in.readLine();
                    entry = entry.trim();
                } catch (NullPointerException e) {
                    run = false;
                } catch (Exception e) {
                    System.err.println("Reading error: " + e);
                }
            } while (entry.equals(""));
        } while (run);
    }

    public void receiveData(APEXDataMessage message) {
        System.out.println("\n-- Received data from " +
            message.getOriginator().getEndpointAddress() + ":");

        if (message.getContentType() == APEXMessage.MIMECONTENT) {
            MimeBodyPart mbp = message.getMimeContent();
            if (mbp != null) {
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                try {
                    mbp.writeTo(baos);
                    String mimeContent = new String(baos.toByteArray());
                    System.out.println("-- MIME content: \n"+mimeContent);
                } catch (Exception e) {
                    System.out.println("Exception while reading MIME content");
                }
            }
        }
        else if (message.getContentType() == APEXMessage.XMLCONTENT) {
            System.out.println("-- XMLContent: \n"+message.getXMLContent());
        }
        System.out.println("-- end of data");
    }

    public void notification(int code, Object specification) {
    }

    public void debug(String debugMessage) {
    }
}
```

Listing 44 A complete endpoint application

Using the application,

1. attaching as 'user1@lsrwww.epfl.ch'
2. sending a message to itself ('user1@lsrwww.epfl.ch')
3. receiving the same message

gives the following result:

```
Attaching as user1@lsrwww.epfl.ch ... ok! (transaction successful)

Enter recipient address: user1@lsrwww.epfl.ch
Add another recipient (y or n): n

Enter XML content (terminate with .):
hello world
.

Sending message to relay ... ok! (transaction successful)

-- received data from user1@lsrwww.epfl.ch: (XMLContent)
<data-content Name="Content">
hello world
</data-content>
-- end of data
```

Listing 45 Demonstration of the endpoint application of Listing 44

A.1.6 An APEXStatus example

As mentioned several times, it is not often the best solution to block the process when waiting for a status code of an `APEXStatus`. As indicated in Paragraph 5.8.2, there are two other possibilities which we present here in contrast to the blocking example.

In the following examples, we assume that we want to attach as an endpoint and give a notification (in the console) on success or failure, and indicate a code and textual reason.

When using the **blocking** method of `APEXStatus`, `getStatusCode`, we simply can wait until the status code becomes available, and display it:

```
APEXEndpointAddress aea = new APEXEndpointAddress("user1@lsrwww.epfl.ch");

APEXStatus attachStatus = new APEXStatus();
apexManager.attachAs(aea, attachStatus);

int attachStatusCode = attachStatus.getStatusCode();
String attachReason = attachStatus.getStatusReason();
if (attachStatusCode == APEX.STATUS_TRANSACTION_SUCCESSFUL)
    System.out.println("Attached as "+aea+" ("+attachStatusReason+")");
else
    System.out.println("Not attached as "+aea+": "+attachStatusReason+
                       " (code "+attachStatusCode+")");
```

Listing 46 Obtaining the status code by the blocking methods `getStatusCode` and `getStatusReason`

In the case of the **ActionListener** we can either use a call-back method to update all internal variables or directly update them within the `actionPerformed` method. Please note that in following example we add the endpoint address as an object to the status, so we can use it later:

```
APEXStatus attachStatus = new APEXStatus();
 ActionListener attachListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        APEXStatus status = (APEXStatus)e.getSource();
        int internalStatus = status.getStatusCode();
        APEXEndpointAddress aea = (APEXEndpointAddress)status.getObject();

        int attachStatusCode = internalStatus.getStatusCode();
        String attachStatusReason = internalStatus.getStatusReason();
        if (attachStatusCode == APEX.STATUS TRANSACTION SUCCESSFUL)
            System.out.println("Attached as "+aea+" ("+attachStatusReason+")");
        else
            System.out.println("Not attached as "+aea+": "+attachStatusReason+
                               " (code "+attachStatusCode+")");
   }
}
attachStatus.setActionListener(attachListener);
attachStatus.setObject(aea);
apexManager.attachAs(aea, attachStatus);
```

Listing 47 Obtaining the status code by adding an `ActionListener` to the `APEXStatus` object

The last model, **redefining** the `receivedStatus` method within the new instance of the `APEXStatus` object, is even easier to initialize:

```
APEXStatus attachStatus = new APEXStatus() {
    public void receivedStatus(APEXStatus status) {
        int internalStatus = status.getStatusCode();
        APEXEndpointAddress aea = (APEXEndpointAddress)status.getObject();

        int attachStatusCode = internalStatus.getStatusCode();
        String attachStatusReason = internalStatus.getStatusReason();
        if (attachStatusCode == APEX.STATUS TRANSACTION SUCCESSFUL)
            System.out.println("Attached as "+aea+" ("+attachStatusReason+")");
        else
            System.out.println("Not attached as "+aea+": "+attachStatusReason+
                               " (code "+attachStatusCode+")");
    }
};
attachStatus.setObject(aea);
apexManager.attachAs(aea, attachStatus);
```

Listing 48 Obtaining the status code by redefining the receivedStatus method of the `APEXStatus` object

Note: since we normally instantiate the `APEXStatus` object within the main class, we can easily call-back to its methods (e.g. `attachedAs` or `notAttachedAs`):

```
if (attachStatusCode == APEX.STATUS TRANSACTION SUCCESSFUL)
    attachedAs(aea);
else
    notAttachedAs(aea, attachStatusCode, attachStatusReason);
```

Listing 49 Accessing methods of the parent class from `actionPerformed` and `receivedStatus`

A.2 An APEX relay process

Similar to the previous section, we are going to set up an APEX relay using the APEX implementation. In contrast to the APEX endpoint, the APEX relay is ways easier set up because actually no interaction on a terminal is required and the implementation, after its initiation, works on its own.

There is only one thing a programmer of an APEX relay process has to define: a class implementing the `APEXRelayProcess` interface which offers the call-back method for possible notifications and debug functionalities.

A.2.1 The APEX relay manager

To initiate a set of relays on a machine working in an administrative domains, an `APEXRelayManager` object has to be instantiated, passing at least the call-back class and the relay configuration file name, and optionally a service configuration file name. The details and the DTDs of both configuration files are given in Section 5.7.

Again, we must redefine two methods:

- in `notification` method we can show initialization issues, attachments, bindings and so one, made by the endpoints and relays managers respectively. There are two parameters passed to the method: an integer code and an `Object`. The integer code can be resolved by using static definitions placed in the APEX class (e.g. `APEX.NOTIFICATION_INIT` or `APEX.NOTIFICATION_ATTACH`) which are documented in the javadoc. For every code number, an special object is associated, e.g. the attachment code number signifies that the object is an `APEXEndpointAddress` which is attaching at the relay or (in an endpoint process) has successfully attached at a relay, e.g:

```
if (code == APEX.NOTIFICATION ATTACH)
   System.out.println("[ ATTACHMENT: "+specification+" ]");
```

Listing 50 A `notification` method sample redefinition to indicate attachments

- the `debug` method we leave blank, although we could add the following line to show detailed debug messages received from the `APEXRelayManager`, e.g.:

```
if (debug) System.out.println("DEBUG: " + debugMessage);
```

Listing 51 A `debug` method sample redefinition

A.2.2 A complete APEX relay

The following listing is completely sufficient to set up a working APEX relay, configured by a relay configuration file and an optional service configuration file. The source of "Relay.java" available in the ch.epfl.lsr.apex.example package of the distribution.

```
public class Relay implements APEXRelayProcess {

    public Relay(String relayCfgFile, String svcCfgFile) {
        try {
            new APEXRelayManager(this, relayCfgFile, svcCfgFile);
        } catch (APEXException e) {}
    }

    public static void main(String[] args) {
        if (args.length > 0) {
            String svcCfgFile = "";
            if (args.length > 1)
                svcCfgFile = args[1];
            new Relay(args[0],svcCfgFile);
        } else
            System.err.println(" Usage: java ch.epfl.lsr.apex.example.Relay"+
                               " relayCfg [svcCfg]")
    }

    public void debug(String debugMessage) {}

    public void notification(int code, Object specification) {
        if (code == APEX.NOTIFICATION INIT)
            System.out.println("[ INITIALIZATION: "+specification+" ]");
        if (code == APEX.NOTIFICATION ATTACH)
            System.out.println("[ ATTACHMENT: "+specification+" ]");
        if (code == APEX.NOTIFICATION REMOVEATTACHMENT)
            System.out.println("[ REMOVING ATTACHMENT: "+specification+" ]");
        if (code == APEX.NOTIFICATION BIND)
            System.out.println("[ NEW BINDING: from "+specification+" ]");
        if (code == APEX.NOTIFICATION BOUND)
            System.out.println("[ RELAY BOUND: on "+specification+" ]");
        if (code == APEX.NOTIFICATION TERMINATE)
            System.out.println("[ TERMINATION: "+specification+" ]");
    }
}
```

Listing 52 A complete relay application

When using the relay application above, the result of arbitrary attachments and bindings may look like this:

```
[ INITIALIZATION: apex-mesh connection: listening on port 912 ]
[ INITIALIZATION: apex-edge connection: listening on port 913 ]
[ ATTACHMENT: user1@lsrwww.epfl.ch ]
[ ATTACHMENT: user2@lsrwww.epfl.ch ]
[ RELAY BOUND: on ltiwww.epfl.ch ]
[ NEW BINDING: from ltiwww.epfl.ch ]
[ TERMINATION: user1@lsrwww.epfl.ch ]
[ ATTACHMENT: user3@lsrwww.epfl.ch ]
```

Listing 53 Demonstration of the relay application of Listing 52

A.3 An APEX service

This section is thought to give a notion on how an APEX service can be designed in order to integrate it easily in the discussed implementation. The general properties of the `APEXService` class which defines an APEX service, are given in Paragraph 5.5.8.

In this application part we are going to set up a simple service which can be considered as a *Traceroute* service, indicating all hops on the course of an operation.

A.3.1 Setting up an APEX service

First of all, we need to define an URI, e.g. **http://lsrwww.epfl.ch/APEX/traceroute**

Then, we can choose the behaviour of the option which, in this case, is very simple: if a traceroute option is found and the hop is applicable, it returns a message to the originator containing a `traceroute` element. This `traceroute` element, which of course we can define ourselves, is empty but has a "transID" attribute and an "identity" attribute which specifies the name of the entity:

```
<traceroute transID='7' identity='lsrwww.epfl.ch' />
```

Listing 54 A sample `traceroute` element

So let us consider how we could implement this service: at the moment a traceroute option is processed for the first time, a correspondent reply message must be generated. Since the `handleOption` method is the first method called to process the option, we need to focus on it. The other three methods (`handleMessage`, `handleSuccessfullySent` and `handleDiscarded`) need not to be defined since no message should be addressed directly to the service as well as no action has to be taken when a message containing a traceroute option is sent or discarded. What finally remains, is to redefine the `getOptions` method to indicate the option identifiers handled by this service (the URI defined above). That is done easily by adding the URI as key of the handling object to a hash table and returning this.

To process the option, one needs to get the originator of the message containing the option. Since we restrict that the option is only processed in an APEX data operation, we can, after casting the general `APEXMessage` to a `APEXDataMessage`, access the originator by calling the `getOriginator` method. Then we can set up the content of the reply message by adding the transaction identifier of the initial option and the name of the

processing hop. This name is accessible thanks to the argument (`administrativeDomain`) passed to the `handleOption` method. The next thing to do is to generate the originator address of the reply message, e.g.

- "appl=traceroute@lsrwww.epfl.ch" for a relay, or
- "user1/appl=traceroute@lsrwww.epfl.ch" for an endpoint handling the option.

Now, the parameters to instantiate and send a new data operation are ready and the message can be sent! Please note in the following listing that for simplicity no imports are specified as well as the try / catch clause is empty – this stems of the fact that we can be sure that the endpoint address is well-formed and no exception is thrown. The source of "TracerouteService.java" is available in the ch.epfl.lsr.apex.example package of the distribution.

```
public class TracerouteService extends APEXService {

    public static final String SERVICEADDRESS = "appl=traceroute";
    public static final String TRACEROUTEURI = "http://lsrwww.epfl.ch/APEX/traceroute";
    public static final String TRACEROUTE = "traceroute";

    public TracerouteService() { }

    public Hashtable getOptions() {
        Hashtable options = new Hashtable();
        options.put(TRACEROUTEURI, this);
        return options;
    }

    public void handleMessage(APEXMessage message) { }

    public void handleOption(APEXOption option, APEXMessage message, String hopName)
                                                                    throws APEXOptionException {
        if (message instanceof APEXDataMessage) {
            APEXDataMessage adm = (APEXDataMessage)message;
            APEXEndpointAddress destination = adm.getOriginator();
            int transID = option.getTransID();
            String messageContent = "<"+TRACEROUTE+" " +
                                        APEX.TRANSID+"='"+transID+"' " +
                                        APEX.IDENTITY+"='"+hopName+"' />";
            try {
                APEXEndpointAddress originator = null;
                if (manager instanceof APEXEndpointManager) {
                    originator = new APEXEndpointAddress(hopName);
                    originator.setApplication(SERVICEADDRESS);
                }
                else
                    originator = new APEXEndpointAddress(SERVICEADDRESS+"@"+hopName);
                APEXDataMessage reply = new APEXDataMessage(originator,
                                                            destination,
                                                            messageContent);
                manager.sendAPEXDataMessage(reply);
            } catch (APEXParsingException e) { }
        }
    }

    public void handleSuccessfullySent(APEXOption option, APEXMessage message,
                                       String administrativeDomain) { }

    public void handleDiscarded(APEXOption option, APEXMessage message, int code,
                                reason, String administrativeDomain) { }
}
```

Listing 55 A complete APEX service: the APEX traceroute service

A.3.2 Integration of the service

Putting the APEX relay, APEX endpoint and the Traceroute service together, means to modify two parts. First, a service configuration file for the two processes must be established and then the option must be added to the messages sent by the APEX endpoint.

The following service configuration file integrates the Traceroute service which is placed in the ch.epfl.lsr.apex.example package:

```
<config>
  <apexservice name='APEX Traceroute Service'
               class='ch.epfl.lsr.apex.example.TracerouteService'
               priority='0' />
</config>
```

Listing 56 A sample service configuration file to integrate the APEX traceroute service

The following snippet adds to every data operation of the endpoint the traceroute option; the targetHop attribute has the value "all", so every hop is applicable:

```
APEXDataMessage adm = new APEXDataMessage (originator,
                                           recipients,
                                           content);

adm.addActiveOption(new APEXOption(APEXOption.EXTERNAL,
                                   TracerouteService.TRACEROUTEURI,
                                   APEXOption.TARGETHOP ALL,
                                   apexManager.getUniqueTransID(null)));

apexManager.sendAPEXDataMessage(adm, status);
```

Listing 57 Adding a "allhop" traceroute option to a data message

When initializing the relay with the correspondent configuration file, it indicates that it has loaded the Traceroute service:

```
[ INITIALIZATION: loaded successfully service 'APEX Traceroute Service' ]
```

Listing 58 Initalization of a relay with the service configuration file of Listing 56

From the point of view of an APEX endpoint user, the result of these modifications looks like this ("EndpointTraceroute.java" is as well available in the ch.epfl.lsr.apex.example package of the distribution):

```
Attaching as user1@lsrwww.epfl.ch ... ok! (transaction successful)

Enter recipient address: user1@lsrwww.epfl.ch
Add another recipient (y or n): n

Enter XML content (terminate with single .):
hello world
.

Sending message to relay ... ok! (transaction successful)

-- Received data from appl=traceroute@lsrwww.epfl.ch: (XMLContent)
<data-content Name="Content">
  <traceroute transID="2" identity="lsrwww.epfl.ch" />
</data-content>
-- end of data

-- Received data from user1@lsrwww.epfl.ch: (XMLContent)
<data-content Name="Content">
hello world
</data-content>
-- end of data

-- Received data from user1/appl=traceroute@lsrwww.epfl.ch: (XMLContent)
<data-content Name="Content">
  <traceroute transID="2" identity="user1@lsrwww.epfl.ch" />
</data-content>
-- end of data
```

Listing 59 Result of sending a message containing a traceroute option

One can clearly see the two hops the data message passed: the 'lsrwww.epfl.ch' relay and the 'user1@lsrwww.epfl.ch' endpoint.

B Bibliography

[1] **RFC 3080 The Block Extensible Exchange Protocol Core**
ROSE, M. T., 2001-03-30
http://www.ietf.org/rfc/rfc3080.txt

[2] **RFC 3081 Mapping the BEEP Core onto TCP**
ROSE, M. T., 2001-03-30
http://www.ietf.org/rfc/rfc3080.txt

[3] **Apple Releases Xgrid 1.0 Technology Preview**
Apple press release, 2004-01-06
http://www.apple.com/pr/library/2004/jan/06xgrid.html

[4] **Simple: Xgrid**
CÔTÉ, D., A weblog discussing computing tools in science.
http://unu.novajo.ca/simple/archives/000022.html

[5] **Clipcode Knowledge Services**
http://www.clipcode.biz

[6] **The Intrusion Detection Exchange Protocol (IDXP)**
FEINSTEIN, B. S., MATTHEWS, G. A., Internet-Draft, 2001-08-21
http://www.cs.hmc.edu/clinic/projects/2000/aerospace/internet-drafts/draft-ietf-idwg-beep-idxp-01.html

[7] **The Tunnel Profile Registration**
NEW, D., Internet-Draft, 2001-02
http://www.cs.hmc.edu/clinic/projects/2000/aerospace/internet-drafts/draft-ietf-idwg-beep-tunnel-01

[8] **RFC 3340 The Application Exchange Core**
ROSE, M.T., KLYNE, G., CROCKER, D., 2002-07-30
http://www.ietf.org/rfc/rfc3340.txt

[9] **RFC 3341 The Application Exchange (APEX) Access Service**
ROSE, M.T., KLYNE, G., CROCKER, D., 2002-07-30
http://www.ietf.org/rfc/rfc3341.txt

[10] **RFC 3342 The Application Exchange (APEX) Option Party Pack, Part Deux!**
ROSE, M.T., KLYNE, G., CROCKER, D., 2002-07-30
http://www.ietf.org/rfc/rfc3342.txt

[11] **RFC 3343 The Application Exchange (APEX) Presence Service**
ROSE, M.T., KLYNE, G., CROCKER, D., 2003-04-29
http://www.ietf.org/rfc/rfc3343.txt

[12] **IANA assigned port numbers**, 2004-01-08
http://www.iana.org/assignments/port-numbers

[13] **APEX implementation of the IMPP**
RIGGIO, M.J., Independent study at Netlab, Temple University, 2003-09-04
http://netlab.cis.temple.edu/apex/

[14] **An APEX implementation for the RoadRunner toolkit**
HOLLSTRÖM, J., NORDLINDER, P., Master Thesis at Department of Computing Science, Umeå University, 2003-01-13
http://www.cs.umu.se/education/examina/Rapporter/437.pdf

[15] **APEXwg – Mailing list for the IETF's APEX working group**
Website: http://lists.beepcore.org/mailman/listinfo/apexwg/
APEXwg Archives are not on the server anymore

[16] **APEX working group newsgroup**
Website: http://news.gmane.org/gmane.ietf.apex

[17] **Jabber Software Foundation**
Official Website: http://www.jabber.org/

[18] **beepcore-java 0.9 release**
implementation of beep core RFC 3080 and beep mapping for TCP RFC 3081.
http://www.beepcore.org/

[19] **The Blocks Public License**
http://www.beepcore.org/beepcore/about_publiclicense.jsp

[20] **JSCAPE iNet Factory 5.2**
a robust suite of TCP/IP networking components for the Java platform
http://www.jscape.com/

[21] **Netscape Messaging SDK 3.51**
provides a set of Protocol Level APIs that the developer can use to write messaging applications and extend applications with messaging services
Guide: http://developer.netscape.com/docs/manuals/messaging/msdkj/contents.htm
Download: http://developer.netscape.com/software/sdks/messaging/downloads.html

[22] **JavaMail™ API 1.3.1 release**
a platform-independent and protocol-independent framework to build mail and messaging applications
http://java.sun.com/products/javamail/

[23] **JavaBeans™ Activation Framework (JAF) 1.0.2 Release**
standard extension to the Java platform, allows services to: *determine the type of an arbitrary piece of data; encapsulate access to it; discover the operations available on it; and instantiate the appropriate bean to perform the operation(s)*
http://java.sun.com/products/javabeans/jaf/index.jsp

[24] **Group Communications and Database Replication: Techniques, Issues and Performance.**
WIESMANN, Dr. M., PhD thesis at Faculté informatique et communications, École Polytechnique Fédérale de Lausanne, Switzerland, May 2002.
http://lsewww.epfl.ch/Documents/acrobat/Wie02.pdf

C Index of Figures, Tables and Listings

C.1 List of Figures

C.2 List of Tables

C.3 List of Listings