ANA Project
Autonomic Network Architecture

Sixth Framework Programme
Priority FP6-2004-IST-4
Situated and Autonomic Communications (SAC)

Project Number: FP6-IST-27489

Deliverable D.1.11

ANA Core Documentation

All you need to know
to use and develop ANA software

ANA Project

Autonomic Network Architecture

Project Number	FP6-IST-27489
Project Name	ANA - Autonomic Network Architecture
Document Number	FP6-IST-27489/WP1/D.1.11
Document Title	ANA Core documentation
Workpackage	WP1
Editor	Ghazi Bouabene (UBasel)
Authors	Ghazi Bouabene (UBasel) Christophe Jelger (UBasel) Ariane Keller (ETHZ) Daniel Rodriguez Fernandez (UiO)
Dissemination level	Public
Contractual delivery date	31st December 2008
Delivery date	No delivery date: document is updated frequently
Version	Version 0.3

Abstract:

This document is the companion of the software deliverable *D.1.11* due on month 36 of the project. It is a guideline helping ANA users and developers to bootstrap with the ANA Core software. This document contains instructions on how to run the ANA Core software as a simple user. It also contains instructions, in a "how to" fashion, helping developers in their first steps with the ANA Core libraries. Finally, this document contains some inner, low-level details about the ANA Core machinery.

Keywords:

ANA Core user manual, ANA development tutorial, ANA Core inner details.

Table of content

Chapter 1

Introduction

1.1 Where and how do I get the latest version of this document?

This document is included in the latest version of the ANA Core software. To download the latest version please check our download webpage :
`http://www.ana-project.org/web/software/start`

This document is also available via the main ANA code repository which is managed with the `subversion` software. To initially create/download your local (i.e. on your PC) ANA code repository use the following command:

```
'svn co https://subversion.cs.unibas.ch/repos/ana/ana-core/stable/'
```

The documentation can be found in the `stable/doc` directory. The document is updated as any other file in the repository. Use `'svn update'` to get the latest version.

1.2 Who should read this document?

This document is the main documentation of the ANA Core software. It is intended for both users and developers of the ANA software. A standard user probably only needs to read chapter 1 while a developer should typically read chapters related to his/her own development, however reading all the chapters and especially chapter 2 is probably a good idea before one starts developing new code for ANA.

Note that while the core developers will always try to maintain this manual updated, the documentation may evolve with some delay compared to the code. Hence the code itself (and the comments it contains), and the doxygen website (`http://www.ana-project.org/doxygen/doxygen/html/index.html`) are the most up-to-date documentation.

1.3 Files and directories: "who's who"

This section provides an overview of all the files and directories of the current code. If you are a developer and you are looking for the right place to put your files you should definitely read this section! We **strongly** advise you to create a directory for your own code in the `C/bricks` subdirectory if you are developing a new brick. In the very rare case where you need system specific code, you can create more subdirectories like e.g. the 3 "standard" sub-directories `common`, `kernel`, and `userspace` of the `MINMEX`, `AL0 API`, and `vlink` brick. Your main directory should also include the adapted template `Makefile(s)` to compile your code. Finally, any C header file (`*.h`) that is not shared with other bricks should remain within your brick's directory (i.e. Do not put in `C/include`).

The freshly checked out directory should contain the following file, sub-directories:

- `Makefile`, `config.txt.template` : the top-level makefile for compiling the code. See section 1.4 for details on how to use it.
- `README` : contains some basic indication to help you start with the core ANA software.
- `Makefile*.template` : templates to be used in case you are compiling your own developed brick
- `bin/` : after successful userspace compilation it will contain all the executable files.
- `so/`: after successful plugin userspace compilation it will contain all the .so plugin files.
- `C/` : the top-level directory for the development in C language.
- `doc/` : contains the code documentation i.e., what you are reading right now, plus the doxygen documentation
- `lib/` : after successful userspace compilation it will contain some function libraries (e.g. the level 0 and level 1 APIs for GATES mode)
- `modules/` : after successful kernel compilation it will contain the loadable Linux kernel modules.
- `log/` : directory where the log files of your excuted bricks will be stored
- `test_ana/` : contains scripts to test the ANA software.

1.4 Compiling the code

To be able to compile the core ANA software you need to have standard development tools installed such as `gcc` and `make`. The current code is developed for Linux systems only. For Linux kernel compilation, you also need to have the C sources of your running kernel: these are usually stored in `/usr/src/linux-???` where `???` is obtained by running `'uname -r'`. If you do not have the right kernel sources, try to install the appropriate package (depends on your Linux distribution) or compile a fresh kernel from the sources. You cannot compile the kernel version of ANA if you do not have the Linux kernel sources!

To make sure you start with a clean compilation, in the main directory, i.e. the directory containing the main `Makefile` and the `C/`, `bin/` directories, etc., run `'make clean'`. This will remove any previously compiled modules and binaries. To compile the ANA Core software :

1. in the main directory move the `config.txt.template` to `config.txt`
2. type: `'make'`

As a result, the compiled minmex and bricks, standalone programs, .so plugins and kernel modules will be respectively saved in the `bin/`, `so/` and `modules/` directories.

1.4.1 Tuning the compilation

If you don't require all the elements of the ANA Core software or are interested in compiling it only for user-space or kernel-space, you can tune the compilation process by modifying the `config.txt` file.

This file contains the following variables :

- `'COMPILE_USER_MINMEX = yes'` or `'COMPILE_USER_MINMEX = no'` to indicate wether you want the minmex to be compiled as a user-space application or not.
- `'COMPILE_KERNEL_MINMEX = yes'` or `'COMPILE_KERNEL_MINMEX = no'` to indicate wether you want the minmex to be compiled as a kernel module or not.
- `'USER_PROCESS_BRICKS'` : this variable contains the list of the bricks that will be compiled as standalone applications (i.e using the gates mechanisms). You can remove elements from the list or leave it empty (by putting nothing after the =) if you do not wish to have standalone bricks.
- `'USER_PLUGIN_BRICKS'` : this variable contains the list of the bricks that will be compiled as .so plugin that can be loaded by the minmex. You can remove elements from the list or leave it empty (by putting nothing after the =) if you do not wish to have plugin bricks.

- 'KERNEL_MODULE_BRICKS' : this variable contains the list of bricks to be compiled as kernel modules using the gates mechansisms
- 'KERNEL_PLUGIN_BRICKS' : this variable contains the list of bricks to be compiled as kernel modules directly interacting with the minmex

1.5 Running ANA

After a successful compilation, it is time to run your ANA software. Note that while the kernel version provides better performance in terms of execution time and processing power, and although the core development team is doing its best to avoid bugs, we cannot guarantee that the code is 100% safe. This means: in non-tested cases it might crash your running kernel. Please report any bugs to the ana-dev mailing list so that we decrease the probability of kernel crashes. If possible, check if the crash is repeatable and report your setup and sequence of events so that we can try to track down the problem and fix it. Thanks very much in advance!

1.5.1 MINMEX in user-space mode

To run the MINMEX as a userspace application the following steps need to be executed:

- set a shell environment variable '$ANA_BASE_DIR' with as value the path of the ANA core main directory, i.e the directory containing the main Makefile and the config.txt file.
- type './minmex' in the $ANA_BASE_DIR/bin directory.

The output of the minmex will be directed to the $ANA_BASE_DIR/log/ directory in the file : ??_anaMinmex where *??* stands for the *PID* of the minmex process.

1.5.1.1 attaching bricks to a user-space minmex :

The bricks that can attach to a user-space minmex are the ones compiled for user-space either as *.so* plugins or as standalone applications using the gates(IPCs).

attaching .so bricks to a user-space minmex To attach a .so plugin to the minmex, use the $ANA_BASE_DIR/bin/mxconfig command in the following way :

```
mxconfig load brick FULL_PATH_to_so_plugin
```

Where *FULL_PATH_to_so_plugin* is the complete path to the .so plugin file. For example, for the *vlink* brick, this path should be (unless you moved it)
$ANA_BASE_DIR/so/vlink.so.

On a successfull load, the minmex prints out in its running terminal standard output (NOT in its logfile), the full path to the log file in which the output(debug) messages of the loaded brick will be written. Otherwise an error message is printed out by the `mxconfig` command.

attaching standalone bricks to a user-space minmex In order to be able to attach standalone process bricks, you need first to load the *gates .so plugin* that implements the gates (IPC) functionality. To do so :

```
mxconfig load brick $ANA_BASE_DIR/so/gatesPlug.so
```

When launched without options, the default communication *gates* between the MINMEX and the bricks use unix sockets. To learn more about gates, please read chapter 2. The *gates plugin* then prints out, in its log file, the control socket it uses: by default, this is the unix socket `/tmp/anaControl_gatesPlug_??` with ?? standing for the PID of the minmex.

To run a brick, one then simply needs to give the gates plugin control gate as an argument of the brick with the -n option: for example, one would type `'./brick_name -n unix:///tmp/anaControl_gatesPlug_??'` (where ?? stands for the PID of the minmex). If needed, it is also possible to specify the control gate and data gate of either the gates plugin or a brick with respectively the `-c` and `-d` options. For example, this permits to run the MINMEX on one host and bricks on another host and have them communicate via UDP.

1.5.1.2 detaching bricks from a user-space minmex :

detaching .so bricks from a user-space minmex In order to detach (i.e. by user intervention) a .so plugin brick from the minmex, you can use the `'mxconfig'` command in the following way :

```
mxconfig unload brick brick_name
```

Where *brick_name* is the name of the brick you want to detach. This name can also be obtained by using the `'mxconfig'` command:

```
mxconfig show bricks
```

That displays the list of names of all the bricks attached to the minmex.

detaching standalone bricks from a user-space minmex In order to detach a standalone brick from the minmex, just type the `'CTRL-C'`(interrupt) key combination in the terminal running your brick. The interruption signal will be captured by the ANA library and your brick will exit correctly.

1.5.1.3 stopping a user-space MINMEX :

Stopping the user-space minmex is also done by typing the `'CTRL-C'` (interrupt) key combination in the terminal running the minmex. This should first trigger the attached plugin bricks to exit properly and alert the remote (standalone) bricks of minmex shutdown.

1.5.2 MINMEX in kernel-space mode

To run the MINMEX as a kernel module the following steps need to be executed:

- The "super module" *anaControl* needs to be loaded, this can be done by typing in `$ANA_BASE_DIR/modules/` as root : `'insmod anaControl.ko'`
- If the *anaControl* loaded succesfully, the minmex module can now be loaded by typing in `$ANA_BASE_DIR/modules/` as root : `'insmod anaMinmex.ko'`

The debug output of the minmex will be written either in `/var/log/messages` or `/var/log/kern.log` (depending on the linux distribution).

1.5.2.1 attaching bricks to a kernel-space minmex :

attaching kernel plugin bricks to a kernel-space minmex : To load a kernel plugin brick, one needs to type as root in `$ANA_BASE_DIR/modules/`:

```
insmod brick_module_file
```

Where *brick_module_file*, is the file of the brick kernel module (e.g. `vlink.ko`). When compiled as kernel code, communication between the MINMEX and the bricks uses by default direct functions calls. The function pointers are dynamically exchanged via the "super" module called the ANA Control (this was also developed to allow one to run multiple ANA nodes on a host).

attaching user-space standalone bricks to a kernel-space minmex : In order to attach standalone user-space bricks, two communication modes between the minmex and the bricks are possible that are *UDP* and *generic netlink*. To enable one of these modes, you first need to load the gates plugin kernel module by typing in `$ANA_BASE_DIR/modules/` as root:

```
insmod anaGates.ko a=c=udp://127.0.0.1:6666,d=udp://127.0.0.1:6667
```

In this case, we are indicating to the *gates plugin* that it should open a UDP control gate (via the option *c*=) on port *6666* and a UDP data gate (via the option *d*=) on port *6667*. For generic netlink gates, replace the UDP url with something similar to `'genericnetlink://arbitrary_name'`.

1.5.2.2 detaching bricks from a kernel-space minmex :

detaching kernel plugin bricks from a kernel-space minmex : To detach a kernel plugin brick use the system command as root :

```
rmmod brick_module_file_name
```

Where *brick_module_file_name* stands for the name of the kernel module file without the *.ko* extension (e.g. `'rmmod vlink'`).

detaching standalone user-space bricks from a kernel-space minmex : The same *CTRL-C* key combination described for the user-space case can be applied here too.

1.5.2.3 stopping a kernel MINMEX :

Stopping the kernel module minmex is also done by the use of the `'rmmod'` function. However due to module dependancy, you must first remove all kernel module bricks, then the MINMEX and finally the ANA Control module.

1.5.3 Debugging levels :

While running ANA, it is also possible to specify the debugging level. There are currently five debugging levels, ranging from no debug information to maximum debug: ANA_NONE, ANA_EMERG (emergency situations, fatal errors), ANA_ERR (error messages but can still work), ANA_NOTICE (important information), ANA_DEBUG (full debug). The default debugging level is ANA_DEBUG. To change the debuging level, use the `-D` option in userspace (e.g. `-D ANA_NONE`) and `debug` option with kernel modules (e.g. `debug="ANA_NONE"`).

Chapter 2

Overview

The ana core software is composed of the two following elements:

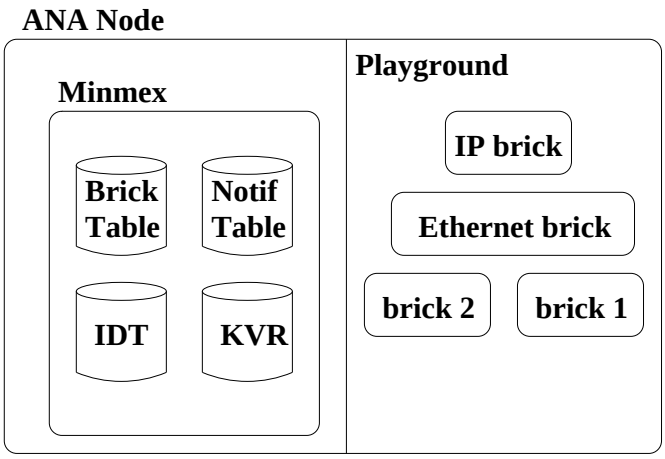

Figure 2.1: Overview of an example ANA node

- Bricks: a brick is the most atomic element providing and using ANA functionnalities. It is an implementation of an atomic funtional block, and can therefore be itself, or a member of, a bigger funtional block implementing a compartment, an information channel, etc. Together all the bricks form the *Playground* of an ANA node.
- MINMEX: the minmex is the element allowing the bricks to interact. It contains all the management units allowing a brick to discover other local bricks and to exchange messages and instructions.

In the current implementation, the bricks and the minmex **can run in user-space and kernel-space**. This is made possible through a set of wrapper functions that we will depict in a further chapter.

2.1 Plugin vs Gates mode

In the current implementation, the ANA bricks can be attached to the minmex either in *plugin mode* or in the *gates mode*.

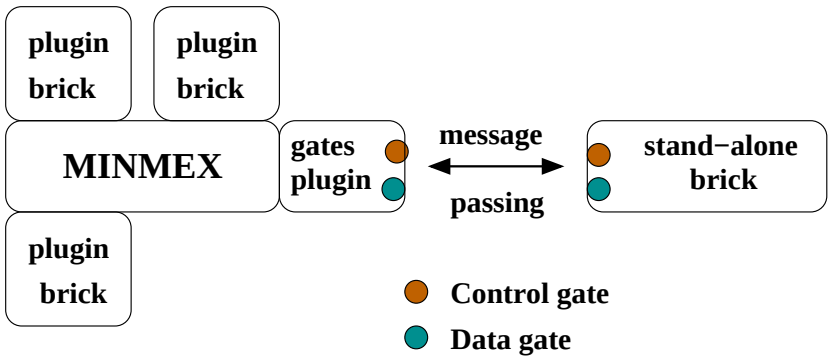

Figure 2.2: Brick attachment to minmex

2.1.1 Plugin mode

2.1.1.1 case of a user-space minmex

When bricks attach to the user-space minmex in plugin mode, their code (available in a .so file) is dynamically loaded by the minmex process. In such a configuration, the ANA software runs in a single process that is the MINMEX process. Each brick code runs however in a separate thread within the MINMEX context. Communication between the MINMEX and the bricks in such a configuration is done through direct function calls.

2.1.1.2 case of a kernel-space minmex

In this case, both the MINMEX and the brick are loaded as Linux kernel modules. Therefore they can communicate via direct function calls through a "super module" called *anaControl*.

2.1.2 Gates mode

2.1.2.1 case of a user-space minmex

In this mode, communication between the MINMEX and the stand-alone brick is done by message passing via IPCs. Therefore, we provide a plugin, called *gates plugin*, extending the functionality of the minmex with what we call *communication gates*.

Communication Gates paradigm For message reception, a brick or a minmex listens on a communication gate. We call communication gate any mean of message reception, be it a UDP or UNIX socket or any other kind of IPC. Basically each ANA Brick and *gates plugin* has two different communication channels, and therefore two communication gates: one for data and one for control information. The *gates plugin* however, is allowed to provide several communication channels for data and for control information simultaneously. This permits different types of Bricks to connect to the MINMEX.

Note that what is being described below is the "internal message exchange machinery" of the ANA core software. This is implemented in the MINMEX and in the ANA AL0 API. Brick developers **do not need to re-develop** this machinery: this is "invisible" to brick developers and is only described here such that developers better understand the underlying machinery of the ANA core software.

The available IPC support for communication gates between the brick and the user-space MINMEX are: UDP, named PIPES and UNIX sockets.

When specifying communication gates arguments (via the `-d, -c, -n` arguments, one has to provide a URL description of the gate including the `mode` and the `name` of the gate. For example:

- UDP: `'udp://ip:port. Example: udp://127.0.0.1:6666'`
- PIPES: `'pipe://absolute_path_to_pipe. Example: pipe:///tmp/pipeFile'`
- UNIX: `'unix://absolute_path_to_sock. Example: unix:///tmp/sockFile'`

2.1.2.2 case of a kernel-space minmex

The same *gates plugin* depicted above can be compiled as a kernel module in order to extend the kernel MINMEX with communication gates support. The available communication gates in kernel space are *UDP* and *generic netlink*. The generic netlink URLs are of the form : `'genericnetlink://arbitrary_name'`.

2.2 Communication between the bricks

We would like to insist on the fact that all communication between bricks goes through the MINMEX which operates like a micro-kernel environment for bricks. In the current implementation of ANA, two bricks cannot communicate together directly. A brick can only send and receive messages on IDP labels.

Figure 2.3 shows a data message format sent from the brick to the minmex and from the minmex to the brick owning the IDP.

Figure 2.3: Data messages between bricks

2.2.1 Data reception in plugin mode

Both for user-space and kernel-space minmex, the data communication between the minmex and the bricks is done via message queues. The data sent from a brick A to a brick B is copied by the minmex and deposited on B's data message queue. B then pops the data copy out of its queue. Although an optimization might be possible where B pops the original data (since in user-space plugin mode, the memory between brick's code is shared), we think that such an optimization will introduce concurrence problems on the data access and prefer to adopt the copy approach that is more inline with usual message passing model.

2.2.2 Data reception in gates mode

In gates mode the data sent from a brick A to a brick B is copied by the minmex into a message buffer and sent via IPCs to brick B. It is then processed by B's *Shadow Dispatch Table* as described in subsection 3.6.1.1.

Chapter 3

MINMEX

This chapter provides some insights on the inner-details of the ANA Core software. Knowledge of such details is not mandatory for users nor for developers. However, a better understanding of the ANA Core inner mechanisms might prove useful to users and developers to deal with some "difficult" situation they might encounter.

The MINMEX implements the management units allowing all brick interactions: it is something like the *micro-kernel* of the ANA node. The components of the MINMEX are detailed in the following sections.

3.1 Brick Table

The brick table holds the information about each brick attached to the minmex. It is a hash table with a size defineable upon program startup. The brick table stores the pointers to the message queues of all the bricks attached to the minmex. The brick table also stores the number of permanent and volatile IDPs owned by each brick in order to insure that no brick exceeds the default allowed quotas. With the brick table, the minmex knows how to send data and notification messages to all the bricks attached to it. The bricks can change the content of this table only via 2 actions: attachment and detachment to the minmex available through the API level 0 that will be detailed in a further chapter. Figure 3.1 shows an example brick table.

3.1.1 Brick's heartbeat and Garbage collection

The age field in figure 3.1 is used for garbage collection. Periodically, each brick sends a "keep alive" message to the minmex. This is an automated task internal to the ANA library and brick developers don't have to care about that. Upon receiving this message, the minmex resets the brick's age to 0. If the brick gives no life sign to the minmex, its age keeps increasing periodically until it reaches a fixed (constant) threshold, at which point the brick will be considered unreachable. The minmex will then free all resources

allocated to this brick.

Name	Handle(id)	Message queue pointer	Age	Nb volatile IDP	Nb perm IDP	....
Brick X	random	0xAABBCC	0	15	2	....

Figure 3.1: Example Brick Table

3.2 Information Dispatch Table

The information dispatch table (IDT) holds the information on the registered information dispatch points (IDPs). At the minmex level, the IDT is used to map a label to a brick. Figure 3.2 shows an example of IDT usage. Upon the arrival of a labeled message, the IDT is consulted to know to which brick this IDP label belongs. With the help of the previously described brick table, the minmex knows then how to forward the message to this brick. In case of information channel IDPs, the IDT might also hold the information channel details (MTU, destination context, destination service, etc.)

3.2.1 IDT Views

The IDT has a concept of private and public views. When creating an IDT entry (IDP), a brick can decide wether the IDP is public (accessible by all other bricks), or private (restricted to a single brick). The bricks can modify the content of this table through some API level 0 functions that allow to create, delete, redirect and modify information about IDT entries and also to change their public/private status. These functions will be detailed in a further chapter.

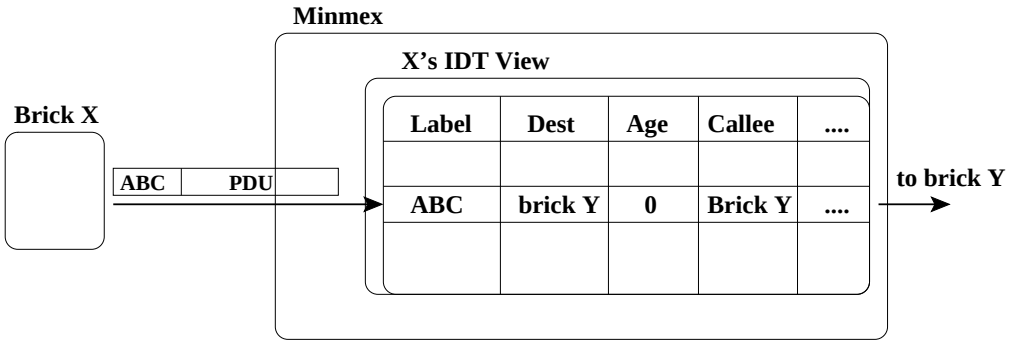

Figure 3.2: Example of IDT Usage

3.2.2 Garbage collection

The age field in figure 3.2 is used for garbage collection. Each time a volatile entry of the IDT is used (accessed), the minmex resets its age to 0. If a volatile entry is not used for a while, its age is increased periodically until it reaches a constant threshold at which point it will be considered as deprecated. The minmex will then free the entry from the IDT and notify its owner and viewer through their notification channel. There is a possibility for bricks to create permanent IDT entries through the API level 0 functions, i.e entries remaining in the IDT eventhough not used.

As an initial policing attempt, each brick attached to the minmex is authorized to create 128 permanent and 1024 volatile entries, over which creation of new IDt entries will be forbidden to the brick. Please note that these numbers are a rough estimate and can be easily increased if ANA core users estimate that they are too restrictive.

3.3 Key Value Repository

The key value repository (KVR), is a simple tool allowing bricks to discover each other locally on an ANA node. Using the KVR, a brick can publish an IDP which typically maps to a service the brick wants to offer. The KVR can be seen as a small database where an entry has an IDP as value and a set of keywords allowing to retrieve the value. An example entry is shown in figure 3.3 where an Ethernet brick called eth-brick has published an IDP labeled *0xABCDE* that can be retrieved by other bricks by using any of the keywords in the list.

Owner	**Value(IDP)**	**Keywords**	**Age**	
eth_brick	**0xABCDE**	**eth_send, Ethernet, interface**	**0**	**...**

Figure 3.3: Example Key Value Repository

The functions allowing interaction between the KVR and the bricks (publication and retrieval of entries) are detailed in a further chapter.

3.3.1 Garbage collection

The KVR has a garbage collection mechanism similar to the one of the IDT where unused entries are deleted after a timeout. However, since the KVR entries are related to IDT entries (that they consider as a value), the KVR has an extra garbage collec-

tion mechanism. That is, the minmex periodically checks for each KVR entry that the published IDP is still present in the MINMEX IDT. If not, the KVR entry is deleted.

3.4 Notification Table

This table stores all the subscriptions of bricks to ANA events. In the current implementation, the events concern only IDPs and bricks and express changes that happened regarding those IDPs or Bricks (e.g. Brick attachent/detachemnt event, IDP deletion, IDP redirection, etc.). When an event regarding a specific object (IDP, or Brick) is triggered, the MINMEX checks the notification table to forward the event notification to the bricks subscirbed to that particular event. Figure 3.4 shows a simplified view of the notification mechanism.

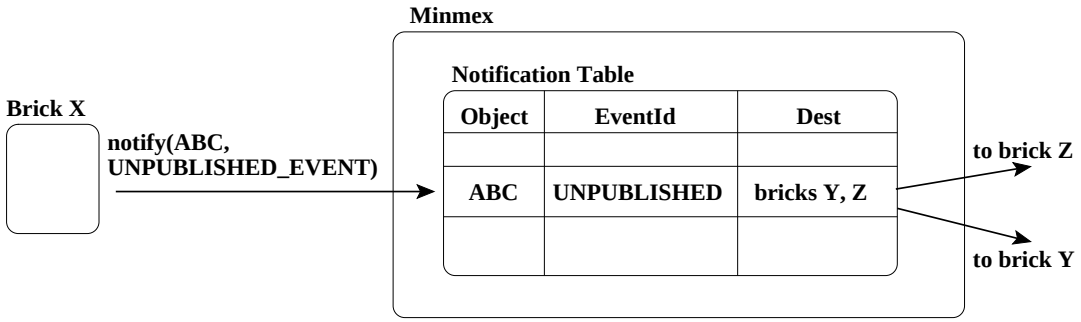

Figure 3.4: Simplified notification mechanism

3.4.1 Garbage Collection

Since subscriptions to an event are linked to a particular object (IDP or Brick), whenever the object disappears, the subscriptions become obsolete and should therefore be cleaned to save resources. Therefore whenever a brick is detached, or an IDP removed, all subscriptions regarding that particular brick or IDP are deleted from the notification table. Also whenever a brick detaches from the minmex, all its subscriptions regarding ohter bricks and IDP events are deleted from the notification table.

3.5 More details for running the minmex

While running the minmex via the command line (shell), one can tune multiple options that we list below :

- choose the hash mask and array size for the *Information Dispatch Table* with $-t$

flag for userspace and $t =$ for kernel. Note that the array allocated for the dispatch table will be 2^{mask} where *mask* is the value you tranmitted with $-t$.

- choose the hash mask and array size for the *Bricks Table* with $-b$ flag for userspace and $b =$ for kernel. Note that the array allocated for the brick table will be 2^{mask} where *mask* is the value you tranmitted with $-b$.
- choose which debug mode to run the minmex in with $-D$ flag for userspace and $D =$ for kernel

3.6 Extensibility of the minmex via plugins

The current design of the MINMEX allows to easily extend its functionalitiy by incorporating additional plugins. In the current version, we only provide the *gates plugin* that extends the minmex with communication gates support. More plugins might appear in future revisions.

3.6.1 Gates plugin

This plugin allows the minmex to offer its functionality to separate (i.e. not plugin) bricks. The *gates plugin* acts on behalf of the separate bricks by relaying data and control between the minmex and the bricks with message passing techniques via IPCs as depicted in Figure 2.2.

The Gates plugin maintains a list of the remote bricks as well as how to reach them (i.e. their data and control gate, notification label, etc.) as shown in Figure 3.5.

Brick handle	Control gate	Data gate	Notif label
0xDEFGH	**udp://127.0.0.1:6666**	**udp://...**	**0xABCDE**

Figure 3.5: remote brick's table at gates plugin

3.6.1.1 Shadow Dispatch Table

In order to inter-operate with the gates plugin, each **independant** ANA brick has a hidden management unit called the shadow dispatch table (SDT). The SDT's role is to map IDP labels to real actions i.e functions. Upon message reception from the gates plugin, ANA libraries will map the IDP label to an action, i.e. a callback function.

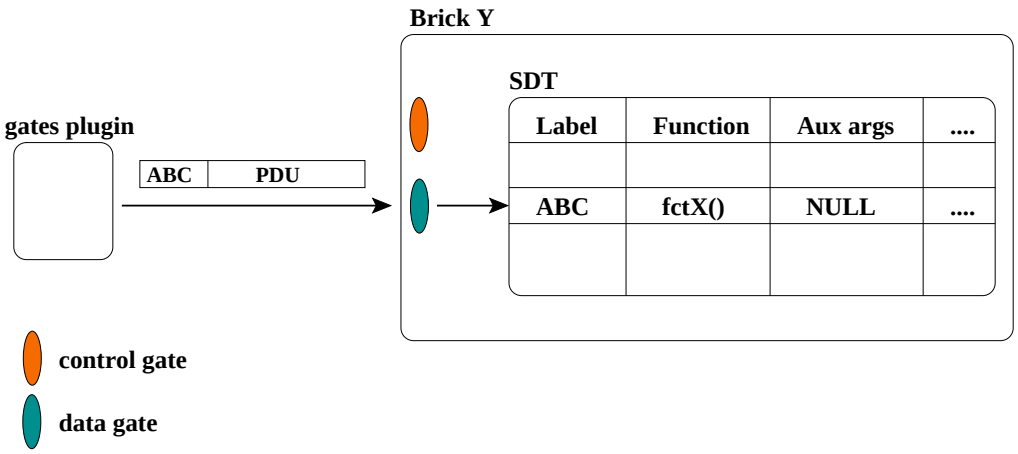

Figure 3.6: Example SDT usage

The function pointer and auxiliary argument fields are stored in the SDT when a new callback IDP is instanciated. Note that the API functions that allow to interact with the minmex's IDT also transparently do the necessary changes in the SDT. That is, the developer of a brick never interacts directly with the SDT.

Chapter 4

How to develop a brick

4.1 Recommended Skeleton for the brick's code

4.1.1 brick_template.h file

When creating a brick we recommend to include the file `C/include/brick_template.h` in the brick's code. The advantage of using `brick_template.h` is that it will hide all minmex attachment details from you. Also, another main advantage is that it permits to easily write bricks that are able to run both in user space, kernel(after being recompiled). Indeed, the file `brick_template.h` contains a *main* function for standalone userspace execution, an initialization function for .so plugin mode and an *__init* and *__exit* functions for kernel mode. All these initialization functions do the same following tasks:

1. Parsing the arguments passed at execution (data/control gates (in case of gates mode execution), auxiliary args).
2. In case of stand-alone (gates mode) execution, the *main()* function takes care of attaching to the minmex on the control gate specified in execution arguments (initialization of the data listening threads).
3. Calling the brick initialization function `brick_start()`.

Please note that if you include `brick_template.h`, your brick code **should not** contain any other `main` or `__init` functions. The system agnostic "main" function (i.e. the starting point) of each brick is `brick_start()`.

4.1.2 Required functions

Each brick using ANA libraries and including `brick_template.h` is expected to have at least the two following functions:

- `'int brick_start()'`: this function is the starting point of your brick. It is called at at the end of the "real" initialization function that is specified in `brick_template.h`. As you can see from the function prototype it does not take any arguments. Please keep in mind that when the *brick_start* function is called, your brick is already attached to the minmex and you can directly start "ANA business" (creating, deleting, publishing IDPs, sending data, etc).

- `'void brick_exit()'`: this function is called by the default ANA library exiting function. It is meant to allow you to execute some final tasks before your brick terminates. Although all your created IDPs and entries in the KVR should be collected by the garbage collection mechanism at the minmex, we recommend that you explicitely unpublish and delete them in the *brick_exit* function. Note that this is very critical for IDPs published in network compartments other than the node compartment as they might not necessarily not receive a notification of your brick termination.

As a general recommendation, we suggest that you try to keep your brick code the most generic possible by using the wrapper functions provided by the ANA library (these are detailed in a further chapter). Indeed, as soon as you start using mode specific functions (printf in user space for example), cross compilation of your brick to another execution mode (kernel) will fail.

4.2 How to change my brick's name in the minmex ?

In case you want to launch multiple similar bricks on the same minmex, each instance of the brick code should have a different name from the others. By name, we mean the brick identifier stored at the minmex's brick table, that is showed as a result of the `'mxconfig show bricks'` command.

Each brick code contains a hard-coded default name that can be redefined at launch time the following way :

- in case of a user-space independant (gates mode) brick : this can be done with the *-N* option followed by the new name value. Example : `'./brickBinary -n unix:///tmp/anaControl_gatesPlug_?? -N newName'`

- in case of a .so polugin: this can be done the *N=* option followed by the new name value. Example : `'mxconfig load brick PATH_TO_SO N=newName'`

4.3 How to pass auxiliary arguments to my brick ?

This section is specific for the cases where your brick is including the file `brick_template.h` (i.e using our predefined *main* or *init* function).

4.3.1 auxiliary arguments for stand-alone user-space bricks

In order to pass arguments to an independant user-space brick at execution time, you can specify them in the command line with the "-a" flag in front. Example:

```
'./brick_template -a arg1 -c unix:///tmp/anaControl_anaMinmex_4321 -a arg2'.
```

Note that you can specify the arguments anywhere in the command line, however the order in which you declare them matters when being retrieved inside the brick (in a similar way as argv[id] for common user space programs).

4.3.2 auxiliary arguments for .so plugin user-space bricks

In order to pass arguments to a .so plugin brick, you can specify them while loading the brick with `'mxconfig'`:

```
'$ANA_BASE_DIR/bin/mxconfig load brick FULL_PATH_TO_PLUGIN arg1 arg2 ..'.
```

4.3.3 auxiliary arguments for kernel module bricks

In order to pass arguments to a kernel module brick, you can specify them while loading the brick with `'insmod'`:

```
'insmod PATH_TO_KERNEL_MODULE_FILE a=arg1,arg2,arg3..'
```

Where "a=" indicates the start of the auxiliary arguments list. Please note the "," separator between the different arguments and the absence of any spaces between the arguments.
Important: For kernel module bricks, the number of auxiliary arguments that can be passed is limited to *10*.

4.3.4 access the auxiliary arguments within brick's code

To access an auxiliary argument you can use the following function:

```
char * getAuxArg(int index);
```

where *index* is the order of declaration of the argument you are interested in. The index count starts at 0.

Please note that the auxiliary arguments can be accessible from everywhere in your brick's code, meaning not only from the *brick_start* function. Indeed, they are allocated as global variables by the initialization function in *brick_template.h*. Once you are done with these arguments, you can free them by calling the function :

```
void freeAuxArgs();
```

4.4 How to terminate my brick from within the code

If you want to terminate the brick's execution from within the brick's code (i.e. this is different from brick unload via *mxconfig*), please use the following function for proper exit :

```
void anaL0_stopANA(int ret);
```

Where *ret* is an exit error code to be transmitted to the minmex in case of Plugin mode, or to the running shell in case f independant user-space bricks. This function, will first call your *brick_exit* function, detach itself properly from the minmex and then terminate all timers and threads of your brick.

4.5 How to create IDPs

4.5.1 In AL2

Creation of IDPs, i.e the association of a callback function to an IDP label is done automatically by the AL2 library when you publish a service (a callback function) in a compartment (see section *How to publish a service in a compartment*). Apart from this, there is no other way in AL2 to create IDT and SDT entries except using AL0/1 techniques (using AL0/1 functions in your AL2 brick is possible since AL2 includes AL0 and AL1).

4.5.1.1 Callback functions for AL2

Since AL2 is based on AL0 and AL1, in the bricks using AL2, you can define your callback function in AL0/1 style as will be shown below.

However, in that case, when your callback function receives an XRP encoded data message (i.e that was encoded with *anaL1_encData* that can be seen in a further section), you would have to parse it to extract for e.g IDP labels, sender context, sender description, which goes against the user-friendliness that we want for AL2. Therefore in AL2, the library does the parsing for you (if the received message is encoded with *anaL1_encData*) before calling your callback function.

The callback function in AL2 must be of the following prototype:

```
void AL2CallbackExample( struct anaL2_message *msg);
```

Where *msg* is a pointer to the following structure :

```
struct anaL2_message{
       int dataLen;
       void *data;
       struct context_s *senderContext;
```

```
        struct service_s *senderService;
        anaLabel_t idp;
        anaLabel_t responseLabel;
        void *bufferPointer;
};
```

Where :

- *dataLen* is the length of the data in the received message (this is different from the length of the received message). This is equivalent to the size of the *PDU*.
- *data* points to the data included in the message. This is equivalent to the *PDU* of the received message.
- *senderContext* indicates the *context* of the sender, i.e. its name/address/identifier in the compartment. For example if we received a message from a chat brick running over Ethernet directly, the context **could** (the context field is not necessarily filled) be the MAC address of the sender.
- *senderService* is the description of the service who sent the message. For the previous example of chat over Ethernet, this would be the description that the Chat brick has published in the Ethernet compartment.
- *idp* is the IDP label on which the messasge was received.
- *responseLabel* is an IDP label in case the receiver is supposed to reply on a sepcific IDP.
- *bufferPointer* is the pointer to the unparsed buffer containing the fields above.

Please note that **not** all the fields in the above structure are necessarily filled when your callback function receives a message. Wether these fields are filled or not depends on the compartment brick forwarding the message to you, i.e for the example of the chat over Ethernet, your local Ethernet brick can decide to indicate to you the MAC address of the sender or not depending on local configuration or compartment policy.

4.5.2 In AL1

4.5.2.1 Callback functions for AL0 and AL1

As we saw previously, each IDP is associated to a callback function and the mapping is done hiddenly by the ANA library. In order for your AL0/AL1 callback functions to be called correctly by the library, they must follow the following C prototype:

```
typedef void (*anaCallback_t)(  void *data,
                                int len,
                                anaLabel_t input,
                                void *aux);
```

Where:

- *data* will be set by the library to point to the received message (PDU after IDP label).
- *dataLen* will be set by the library to indicate the size in bytes of the received *data*
- *input* will be set with the label of the IDP from which the message was received (useful in case a callBack function can be triggereed by multiple IDPs)
- *aux* is an auxiliary argument that can be set when a callBack function is created. It can be used to pass extra arguments to the function being called.

4.5.2.2 Registering Callback functions in AL0/1

By registering we mean creating a new entry in the minmex's Information Dispatch Table and, in case of stand-alone bricks, in the brick's Shadow Dispatch Table. To create IDPs in a brick using only API levels 0 and 1, use the following function:

```
anaLabel_t anaL0_registerCallback(anaCallback_t funct, void *aux,
                                  copyAuxFunction copyAux,
                                  freeAuxFunction freeAux,
                                  anaHandle_t viewer,
                                  int permanent, int thread);
```

Where:

- *funct* is a pointer to the callBack function you want to register. This function will handle the incoming messages destined to the newly created IDP.
- *aux* is an auxiliary argument that is passed to the callBack function each time it is triggered. This argument is **copied** in the IDT in case of plugin bricks and in the SDT in case of stand-alone bricks. Please note that since a copy of this argument is stored, the programmer can dispose of the original argument as he wishes, i.e it is possible to free it, the callback function will keep receiving the copy as an argument anyway. In case you need to pass multiple auxiliary arguments to your function, you will need to create a wrapper structure containing all those arguments.
- *copyAux* the function to be used to copy the *aux* argument in the SDT/IDT. This is useful in case the auxiliary argument you want to pass is a structure containing pointers to other memory slots (char * pointers for example). Your *copyAux* function should allocate and copy the *aux* argument correctly and return a pointer to the copy, i.e. its prototype is

  ```
  void * fct(void *);
  ```

- *freeAux* the function to be used to free the copy of *aux* from the SDT/IDT when the IDP is being deleted. Indeed since in some cases the auxiliary argument can be a structure containing pointers to other memory slots, you would need to provide a function to free the copy of *aux* correctly. In case your auxiliary argument does not contain any pointers, indicating the standard function *free* as *freeAux* would do the trick. The prototype of the freeAux function should be :

  ```
  void fct(void *);
  ```

- *viewer* allows to define a specific viewer brick for the IDP. If you want the new IDP to be public (visible by all bricks), set this argument to *0*. If you want it to be viewable only by a specific brick, you need to specify its *handle* i.e the brick's identifier in the minmex. The ANA Core software design, does not provide any mean to extract bricks identifiers (this is done on purpose). To know the handle (id) of a brick, you need to organize (i.e. program both bricks) so that the brick sends you its handle.

- *permanent* defines whether or not the IDP should be permanent in the IDT. Use the constant value *IDP_PERM* for permanent, 0 for volatile. Note that the number of permanent IDPs a brick can create is limited in this version to *128*. can create. Also, the delay of deletion of an unused volatile IDP is dependant on the minmex's configuration.

- *thread* defines whether the callback function *funct* should be executed in a dedicated thread. Recommendation: thread = 1, if your callback function executes "for a long time" or if it wants to receive further packets (e.g. with an AL2 function). thread = 0, if its execution is fast (e.g. non blocking, no loops, no sleep etc.). **Important :** For callback functions that block waiting for incoming messages, you must set the *thread* flag to 1. Indeed, callback functions are executed among the thread expecting data messages, therefore if this thread is blocked, further message reception is impossible and lead to a deadlock situation!

On success the function returns the label of the newly created IDP. On error, 0 is returned and the error value is set the *ANA errno equivalent* variable and can be retrieved via the function *anaPerror* that will be described in a further chapter.

Example :

```
void callBackFct( char *data, int len, anaLabel_t input, void *aux) {
   anaPrint(ANA_NOTICE, "callBack received: %s, \n", data);
}


int brick_start() {

   anaLabel_t callBackIDP;

   callBackIDP = anaL0_registerCallback(( anaCallback_t) callBackFct,
                                NULL, NULL, NULL,
                                0, IDP_PERM, 0);
```

```
    if(callBackIDP==0)
          anaPerror(ANA_NOTICE, "Register Failed");
}
```

Here, we ask the minmex to create a permanent (*IDP_PERM*), public (*viewer=0*) callBack IDP. Messages received at this IDP will locally trigger the callBack function *callBackFct* with a NULL auxiliary argument. Here since the *thread* flag is set to 0, the callBack function *callBackFct* will be triggered on the same thread waiting for messages.

4.6 How to delete IDPs from the minmex

By deleting here we mean removing an entry in the minmex's Information Dispatch Table and the brick's Shadow Dispatch Table. The procedure explained here is valid for the API levels 0 and 1 and 2. To delete an IDP, use the AL0 function:

```
int anaL0_unregisterCallback(anaLabel_t label, anaHandle_t viewer)
```

Where :

- *label* is the label of the IDP you want to delete
- *viewer* specifies the IDT view to which the IDP *label* belongs. If the IDP you want to delete is public, use 0 for this argument. To delete an IDP that was priorly created for a specific brick IDT view, use the handle (identifier at the minmex) of this brick for this argument.

Please note that you must be the **owner** of the IDP, i.e you priorly created it using the *anaL0_registerCallback* or *anaL2_publish* function with the same *viewer* argument, in order for the minmex to delete it. Otherwise, the deletion would fail.

On success, this function would result in the deletion of the corresponding remote entry in the IDT (minmex) and local SDT entry and 0 is returned. On failure a negative error value is returned.

4.7 How to receive messages on an IDP

If your brick is using only the API levels 0 and 1, reception of messages can only be done in "callback style". By "callback style" we mean that messages are always received in independant functions (the callback functions). Therefore, an instruction sequence as (pseudo-code):

```
message1 = receive(IDP1);
message2 = receive(IDP2);
someAction(message1, message2);
```

is not possible while using only API levels 0 and 1. To have the same result, using API levels 0 and 1, you will need global variables and two callback functions, as follows:

```
void *message1;

void callback1( char *data, int len, anaLabel_t input, void *aux){
        message1 = (void *) malloc(len);
        memcpy(message1, data, len);
}

void callback2( char *data, int len, anaLabel_t input, void *aux){
        someAction(message1, data);
}
```

As shown here, this style of execution (cascading callback functions) can be tricky to code sequential tasks. Therefore, in AL2 in addition to the usual "callback-style" reception, we propose the following function :

```
struct anaL2_message * anaL2_receive( const anaLabel_t input,
                                      struct timespec *timeout);
```

Where:

- *input* is the label of the IDP to receive data on.
- *timeout* is a pointer to a *timespec* allowing to specify the duration of the blocking period in seconds and nanoseconds.

In case the *timeout* argument is *NULL*, your execution thread will be blocked until a message is received on the *input* IDP.

On successfull message reception, the function returns a pointer to a *struct anaL2_message* (see *How to create IDPs* section for more details on this structure). On error (i.e time-out) a *NULL* pointer is returned.

Important:

- Please note that using the previous function on IDPs that you associated to some callback functions will always fail (return a *NULL* message). Indeed in order to be able to receive messages on IDPs using the *anaL2_receive* functions, the IDPs must not be priorly associated to a callback function of your own, but rather to a default library callback function that will queue the messages received on these IDPs. To create IDPs that are associted to this default queing function, you must indicate a *NULL* callback function pointer in your *anaL2_publish* call. See *How to publish a service in a compartment* section for more details.
- If you receive messages with *anaL2_receive* you **must** free the allocated memory for the message once you are done with it. To do so use the function :

  ```
  void anaL2_freeMsg(struct anaL2_message *msg);
  ```

Where:

– *msg* is the *anaL2_message* structure to free.

Please note that there is no need to use *anaL2_freeMsg* in your AL2 callback functions, since freeing the message is done for you by the library.

- As for any AL2 function call, this function should not be used in a Callback function **unless the callback is executed in a separate thread**. Otherwise the receive function will always timeout and reception always fails.

The sequential pseudo-code example shown above is then expressed in AL2 as follows:

```
int brick_start() {
        struct anaL2_message *message1 =  anaL2_receive(IDP1);
        struct anaL2_message *message2 =  anaL2_receive(IDP2);
        someAction(message1->data, message2->data);
        anaL2_freeMsg(message1);
        anaL2_freeMsg(message2);
}
```

4.8 How to temporarily block/unblock message reception

In case you need to temproarily block reception of messages on a callback IDP (without deleting the IDP), you can use the following function :

```
int anaL0_setIDPbusy( anaLabel_t label, anaHandle_t viewer, int busyflag);
```

Where :

- *label* : is the IDP label that you want to declare busy
- *viewer* : is the view to which the IDP belongs
- *busyflag* : is the status that you want the IDP to be in. 1 indicates that the IDP is busy i.e. message reception is blocked. 0 sets the IDP to be available again.

Note that changing the status (busy vs available) of an IDP will emit an *IDP_BUSY_EVENT* or *IDP_AVAIL_EVENT* event at the node compartment level in order to inform interested bricks of the change.

Also, as a result of setting an IDP be busy, the bricks sending messages on that IDP i.e. calling *anaL0_send() on the IDP*, will receive an error *ANA_IDP_BUSY_ERROR*.

4.9 How to send a message to an IDP:

In all API levels, to send data to an IDP you can use the AL0 function

```
int anaL0_send(anaLabel_t label, void *data, int dataLen);
```

Where:

- *label* is the identifier of the IDP to send data to
- *data* is the pointer to the data to send to the IDP
- *dataLen* is the size in bytes of the data to send

In case the message you want to send is an XRP encoded one, to which you expect a response, you can use the *requestReply* functionnality as follows:

For API level 2: Using the function *anaL1_requestReply* (shown in next parargraph) is possible in bricks using API level 2. However the reception of reponses in that case can only be done in "callback-style": this means that responses are received in a function different from the one where the request message was sent and therefore, sequential tasks are hard to code as it was previously shown in the *How to receive messages on an IDP* section. Therefore, in AL2 we propose a function that handles the response in the same function where the message is being sent. This function is the following:

```
char *anaL2_requestReply( const anaLabel_t dest,
                          xrpMsg_t msg, int reqlen,
                          int * respLen, struct timespec *timeout):
```

Where:

- *dest* is the label of the IDP to send the request(data) to
- *msg* is the message to send. Note that this **must be an XRP encoded message** for the anaL2_requestReply to work.
- *reqlen* is the size in bytes of the XRP message *msg*
- *respLen* is a pointer to an integer that will be filled with the size in bytes of the received response.
- *timeout* is a pointer to a *timespec* structure allowing to specify the duration of the blocking period in seconds and nanoseconds.

This function waits for an incoming response for the period indicated in *timeout*. In case *timeout* is NULL, a default timeout of *1* second will be used. On success a pointer to the received response is returned and the field pointed by *respLen* indicating the size in bytes of the response message is filled with the correct size. In case of error a *NULL* pointer is returned and the "ANA errno" equivalent variable is set to the proper error code.

For API level 1: To receive responses on a specific callback function, you can send your XRP encoded data, using the following function:

```
int anaL1_requestReply( const anaLabel_t label,
                        void* msg, int len,
                        anaCallback_t fct, void *aux,
                        copyAuxFunction copyAux,
                        freeAuxFunction freeAux, int thread);
```

Where :

- *label* is the label of the IDP to send data to
- *msg* is a pointer to the XRP encoded data to send
- *len* is the size in bytes of the data to send
- fct is the callback function that will handle the response
- *aux* is an auxiliary argument that will be passed along to the function fct
- *copyAux* is the function to be used to copy the *aux* to the IDT/SDT. This is useful in case the auxiliary argument you want to pass is a structure containing pointers to other memory slots (char * pointers for example). Your *copyAux* function should allocate and copy the *aux* argument correctly and return a pointer to the copy, i.e. its prototype is

  ```
  void * fct(void *);
  ```

- $freeAux$ the function to be used to free the copy of *aux* from the IDT/SDT. Indeed since in some cases the auxiliary argument can be a structure containing pointers to other memory slots, you would need to provide a function to free the copy of *aux* correctly.In case your auxiliary argument does not contain any pointers, indication the standard $free$ function would do the trick. freeAux prototype is :

  ```
  void fct(void *);
  ```

- *thread* defines whether the callback function fct should be executed in a dedicated thread.

The *anaL*1*_requestReply* function, first registers a callback IDP (through a call to *anaL*0*_registerCallback*) mapped to the fct callback function meant to handle the response. Upon reception of the newly created IDP's label, it adds it to the end of the XRP message *msg* and sends it.

The *anaL*1*_requestReply* function returns the number of bytes sent in case of success, or -1 if an error occurred. Errors can be for example due to the failure to create a new IDP for the response handler or inability to add the response handler's IDP to the message (not enough space in buffer).

4.10 How to publish a service in a compartment

4.10.1 In AL2

Contrarily to AL1, in AL2 the IDPs are created when the publish command is called. This means that the *anaL2_publish* call, that we will detail below, does the call to *anaL0_registerCallBack* for you.

To create and publish IDPs to compartments in AL2, use the following function:

```
anaLabel_t anaL2_publish( const anaLabel_t compIDP,
                          struct context_s *context,
                          struct service_s *service,
                          AL2Callback_t function,
                          int separateThread,
                          struct timespec *timeout );
```

Where:

- *compIDP* is the IDP label of the compartment provider functional block to whom we want to send the publish request.
- *context* defines a subset of the compartment in which you want to publish the IDP. Note that the values taken by this field are mostly compartment specific. For example, if you are addressing this publish command to an Ethernet compartment, the *context* can be a *MAC address*. However, there are 2 values common to all compartment provider functional blocks:
  - "*" : this character string containing a single '*' character indicates to the compartment provider functional block, that you want to publish the service associated to your IDP to the maximum scope this FB can reach (in some case like Ethernet this means the entire compartment).
  - "." : this indicates to the compartment provider functional block that you want to publish your service in its local repository only. This way only the compartment users that are running on the same ANA node as the compartment provider functional block can access your service..
- *service* is the description of the service provided by the IDP we want to publish. This will the used by remote peers to discover this published IDP.
- $function$ is the callback function providing the service we want to publish. Note you don't necessarily have to specify a callback function here. Indeed you can set this argument to NULL, in which case the messages that are received by the library on this IDP will be **queued**. You will be able to access them when needed, in order of their arrival, by calling the function *anaL2_receive* depicted in section *How to receive messages on an IDP*.
- $separateThread$ is a flag indicating wether you wish the callback function service to be launched in a separate thread or not.

Very Important: if your callback function uses AL2 function calls like *publish* or *lookup* (etc), if it is blocking, or also if it takes a long time to execute then your callback function **MUST** be launched in a **separate thread**. Indeed, if you set the *separateThread* flag to 0, the callback function will be executed by the thread expecting messages. Therefore, if the function blocks waiting for incoming messages (as required by AL2 publish, lookup etc), the thread receiving incoming messages is blocked and reception of messages becomes impossible ⇒ **deadlock !**

- *timeout* is a pointer to a *timespec* structure allowing to specify the duration of the blocking period waiting for publish response in seconds and nanoseconds. In case this argument is *NULL* a default timeout of 1 second is applied.

This function does the following tasks for you:

1. Creation of the IDP (*anaL0_registerCallback* call)
2. Encoding and sending of an XRP publish message to the compartment provider functional block
3. Block until reception of publish response
4. Publish response parsing to determine if the publication went right or wrong

anaL2_publish returns the label of the created and published IDP in case of success. In case of error, the function returns 0 and sets the "ANA errno equivalent" to the appropriate negative value. The error can be displayed using the *anaPerror* function.

struct context_s : This structure is meant to hold the context of a service within a compartment (i.e. its MAC address in case of Ethernet, IP address in case of IP compartment). the structure is defined as follows :

```
struct context_s {
        void *value;
        int valueLen;
};
```

Where :

- *value*: points to the value of the context (i.e. the IP, MAC address, etc.)
- *valueLen*: is the size in bytes of the context value

struct service_s : This structure is meant to hold the description of a service published within a compartment (i.e. how the service wants to be called in the compartment context). the structure is defined as follows :

```
struct service_s {
        void *value;
        int valueLen;
};
```

Where :

- *value*: points to the value of the service description
- *valueLen*: is the size in bytes of the value

4.10.2 In AL1

If your brick uses only API levels 0 and 1, then the IDPs you want to publish must have been priorly created with the AL0 function *anaL0_registerCallback* described above. To publish a previously created IDP in a compartment, the following steps need to be filled:

- First, allocate memory to store the publish command message that will be sent to the compartment provider functional block.
- Second, encode and store in the previously allocated slot, an XRP encoded publication message containing the new entry's context, description and the IDP label.
- Third, send this encoded message to the compartment provider functional block while indicating a response handler function.

To allocate memory to store the publication message in (First step), use the AL1 function:

```
xrpMsg_t anaL1_allocateMessage(void);
```

This function takes no argument, allocates a memory slot with the convenient size and returns a pointer to it. The pointer is of type the *xrpMsg_t* that is a wrapper to *void* $*$. Do not forget to free this pointer once you're done with it (after sending the command to the compartment provider functional block).

To encode the publish message (Second Step), use the AL1 function:

```
int anaL1_encCompartmentPublish (xrpMsg_t msg, const anaLabel_t idp,
                                 struct context_s *ctxt,
                                 struct service_s *srv, int permanent)
```

Where:

- *msg* is the previously allocated memory slot for the message
- *idp* is the label of the previously created IDP we want to publish

- *ctxt* is the context in which to publish the service
- *srv* is the service description within the compartment context
- *permanent* is a flag indicating wether you want your published entry to be volatile i.e. removeable by garbage collection mechanisms, or permanent. Note that each compartment can have specific restrictions on permanent entries (for example, some might just forbid them). Set the value to 0 for volatile and *IDP_PERM* for permanent.

This function will fill the memory slot pointed by *msg* with the XRP encoded publish message. The function returns the size of the message in case of success and -1 if an error occurred. Errors can be due for example to a large description field that does not fit into the message buffer.

To fulfill the Third Step, i.e. send the encoded publish message to the compartment provider functional block, two ways are possible. The first option is to use the *anaL0_send* function shown above to send the message to the IDP of the compartment provider functional block. However by sending the publish command this way, you will not indicate any reply IDP to the target functional block. Therefore you won't be able to receive any confirmation/error message and you would have no way to determine if the publish was successful or not. Moreover, some compartment provider functional blocks might require that you expect a response and therefore drop the publish commands sent with *anaL0_send*.

Therefore we recommend to use the requestReply function for AL1 *anaL1_requestReply*, that was described in the *How to send data to an IDP* subsection, in order to know whether the publication succeeded or not.

We now show an example with all the three steps of a publish request, using the requestReply function:

```
int brick_start() {

    xrpMsg_t msg;              /* pointer to the publish message*/
    struct context_s ctxt;
    struct service_s srv;
    anaLabel_t IDP1;
    int lenMsg, ret;

    /* description of the service to publish */
    memset(&srv, 0, sizeof(struct service_s));
    srv.value = "service description";
    srv.valueLen = strlen("service description") + 1;

    /* we set the context to "*", i.e All  */
    memset(&ctxt, 0, sizeof(struct context_s));
    ctxt.value = "*";
    ctxt.valueLen = 2;

    /*we create the IDP first */
    IDP1 = anaL0_registerCallback( (anaCallback_t)
```

```
                                    callbackFct, NULL, NULL, NULL,
                                    0, IDP_PERM, 0);
/* Step 1 */
   msg = anaL1_allocateMessage();

/* Step 2 */
   lenMsg = anaL1_encCompartmentPublish( msg, IDP1,
                                &ctxt, &srv, IDP_PERM);
   if (lenMsg<0) {
        anaPrint(ANA_ERR, "problem encoding publish request!\n");
   }

/*Step 3*/
   /* CPE_IDP is the IDP label to reach the compartment
      provider functional block */
   ret =  anaL1_requestReply( CPE_IDP, msg, lenMsg,
                 (anaCallback_t) handlePublishReply,
                                    NULL, NULL, NULL, 0);

   if (ret<0) {
        anaPrint(ANA_ERR, "problem sending publish request!\n");
   }

   free(msg); /* Do not forget to free the encoded message buffer once
                      you're done with it */
```

Please note that we stored the publish message size returned by *anaL1_encCompartmentPublish* and we passed it as argument of message size to the *anaL1_requestReply* function. The code for the reponse handler function *handlePublishReply* will be shown in the next paragraph where we explain how to parse AL1 response messages.

What does the publish response look like : In case of success of your request, the compartment provider functional block will send back the exact same XRP request message as a response. In case of failure, it sends an error message containing all the arguments included in your request. An *anaL1_isError* call on this message would recognize that it is an error one (as we will show later).

How to parse AL1 XRP publish responses Let us consider the code to handle the compartment provider functional block's response to the publish command emitted in the example above:

```
void handlePublishReply(  char *data, int len, anaLabel_t input,
                          void *aux) {
   char *name=NULL;
   int nameLen = anaL1_decResponse(data, XRP_CLASS_SRC_SRV,
                                   0, (void**)&name);

   if(anaL1_isError(data)) {/*if ERROR command*/
      if(name!=NULL)
         anaPrint( ANA_ERR, "Publish of entry %s Failed\n",
                   description);
```

```
        else
            anaPrint(ANA_ERR, "Publish Error\n");

        return;
    }

    if(name != NULL)
        anaPrint( ANA_DEBUG, "Publish of entry %s succeeded\n",
                  description);
    else
        anaPrint(ANA_DEBUG, "Publish Success\n");
}
```

In this example we used two AL1 functions to parse the response. First there is :

```
int anaL1_isError(xrpMsg_t msg)
```

This function takes an XRP message, i.e. an ANA AL1 message as argument and determines if it is an error message or not. It returns 1 if the reply is an error message and 0 otherwise.

The second function is :

```
int anaL1_decResponse(xrpMsg_t msg, xrpClass_t className,
                      int occurence, void** ptPtval);
```

Where:

- *msg* is the received message to parse
- *className* is the XRP class of the argument you want to extract from the message. The XRP class can be seen as an equivalent to the variable *type* in programming languages. The most common XRP classes (that you would have to deal with frequently) are the following constants :
  - *XRP_CLASS_LABEL* : that is the class describing IDP labels
  - *XRP_CLASS_SRC_SRV* : that is the class describing source service descriptions passed along publish, lookup and data messages.
  - *XRP_CLASS_DST_SRV* : that is the class describing destination service descriptions passed along lookup and data messages.
  - *XRP_CLASS_SRC_CXT* : that is the class describing source context arguments passed along publish, lookup and data messages.
  - *XRP_CLASS_DST_CXT* : that is the class describing destination context arguments passed along lookup and data messages.

  For more details on XRP classes, we recommend you to see the dedicated section further in this document.

  If you are interested in extracting an IDP label from an AL1 response message the corresponding XRP class would then be the constant *XRP_CLASS_LABEL*. For the decription of the published service, *XRP_CLASS_SRC_SRV* can be used as shown in the example above.

- *occurence* indicates which occurrence of the XRP class argument we are interested in, in case many are present in the same message (for example many IDP labels in a AL1 message). The occurrence counting starts with 0.

- *ptPtval* is a **pointer to the pointer** where the value of the argument we are interested in will be stored. On successfull termination of the function, the field pointed by *ptPtval* should contain the pointer to the argument value.

On success, *ptPtval* is filled conveniently and the function returns the argument's size. On failure, i.e. when there is no argument of the corresponding XRP class and occurence in the AL1 message, -1 is returned.

IMPORTANT: the *anaL1_decResponse* function does not return a copy of the wanted value but rather a pointer to it inside the reponse message. If you need to use the value outside of the context of the function handling the response, you need to copy it elsewhere. Otherwise it will be crashed by the next incoming message.

4.10.3 Node compartment example

The two methods shown above can be applied to the node compartment to publish some IDP in the Key Value Repository. For both cases of AL1 and AL2, you just have to indicate the node compartment's IDP label as a destination for the publish request while following the same instructions above. The node compartment's IDP label can be obtained using the Macro : *NODE_LABEL*

As seen in a previous chapter, IDPs published to the node compartment are stored in the Key Value repository where they have a list of keywords to look them up with

In your AL1 or AL2 publish request to the node compartment you can indicate the list of keywords in the *service description* argument you pass.
Example: To publish an IDP in the node compartment with the keywords *service1*, *kw1* and *kw2*, the *service description* argument should be initialized the following way :

```
struct service_s srv;
memset(&srv, 0, sizeof(struct service_s));
srv.value = "service1+kw1+kw2";
srv.valueLen = strlen("service1+kw1+kw2") + 1;
```

Please note the + character that separates the different keywords. It is important to use + and no other character since the minmex will use + to parse the description field. Note that the first keyword of the list willbe considered as the name of the entry and **has to be unique** (publication will fail otherwise). The order of the rest of the keywords then does not matter. Please note also that for the node compartment, the *context* argument is not relevant since the scope of this compartment is local by definition.

4.11 How to lookup/resolve services in compartments

To access the services published in a compartment, two commands are available:

- resolve : constructs an Information Channel to the queried target service
- lookup : returns more information about the queried target service

4.11.1 Resolve request

In AL2: To send a resolve request in AL2 use the following function :

```
anaLabel_t anaL2_resolve( anaLabel_t compIDP,
                          struct context_s *context,
                          struct service_s *service, char chanType,
                          struct service_s *querierDescription,
                          struct timespec *timeout);
```

Where :

- *compIDP* is the IDP label of the compartment to which you want to address the resolve request.
- *context* delimits a subset of the compartment in which to look for the service
- *service* is the description of the service you want to resolve.
- *chanType* determines the type of channel you want to open to the service(s). The possible channel types are, unicast, anycast, multicast, broadcast. Indeed in case the request matches multiple different services in the compartment, this channel type will indicate to the compartment provider functional block what kind of channel to build in that case. The possible values are 'm' for Multicast, 'u' for Unicast, 'a' for Anycast, 'b' for Broadcast, 'c' for Concast.
- *querierDescription* is an optional argument. It corresponds to **your** description. This description might be passed along to the target service(s) so that they can reach you back.
- *timeout* is a pointer to a *timespec* structure allowing to specify a timeout in seconds and nanoseconds for this operation. In case this argument is set to NULL, a default timeout of 1 second will be applied.

This function encodes an XRP resolve message, sends it to the *compIDP* label, and then blocks for the indicated timeout period waiting for the compartment provider's response. When a response is received, the function parses it before returning.

anaL2_resolve returns an IDP label to contact the created information channel in case of success. In case of error, 0 is returned and the "ANA errno" equivalent value is set to the proper negative error value. The error message can be printed out using the *anaPerror* function as will be shown in a further section.

In AL1: The same 3 steps described in section *How to publish an IDP in a compartment*, for AL1 apply also here. Step 1 is exactly the same. Step2 is however different since it is a resolve request that we should encode this time.

To encode an XRP resolve request use the function :

```
int anaL1_encResolve( xrpMsg_t msg, struct context_s *ctxt,
                      struct service_s *srv, char chanType,
                      struct service_s *queryingService);
```

Where:

- *msg* is the previously allocated memory slot for the message
- *ctxt* delimits a subset of the compartment in which to look for the service
- *srv* is the description of the service you want to resolve.
- *chanType* determines the type of channel to the service(s) you want to open.
- *queryingService* is an optional argument. It corresponds to **your** description. This description might be passed along to the target service(s) so that they can reach you back.

This function returns the size of the encoded message in case of success or -1 in case of error.

In Step 3, since we expect a response with an IDP to reach the target, sending of the resolve request can then only be done with *anaL1_requestReply*.

What does the resolve response look like : In case of error in the resolve request, the Compartment Provider Functional Block (CPFB) sends back an error message with the same arguments that were included in the query. The function *anaL1_isError* would then determine in that case that the reply is an error message. In case of success of the resolve command, and once the information channel is built, the CPFB sends back a message containing the IDP label to reach that channel. The message is headed by the resolve XRP command *XRP_CMD_RESOLVE*.

Resolve response parsing in AL1: The resolve reply sent by the CPFB can be parsed using the same functions presented above (in parsing publish reply) i.e. *anaL1_isError* and *anaL1_decResponse*. The IDP label of the information channel can be extracted by using the function *anaL1_decResponse*, with the XRP class *XRP_CLASS_LABEL* at *occurence* 0.

Please note that similar to the *publish* request, the response here is received in a separate callback function (i.e. different from where the request was made).

4.11.2 lookup request

In AL2: to send a lookup request in AL2, use the following function :

```
int anaL2_lookup( anaLabel_t compIDP, struct context_s *context,
                  struct service_s *service,
                  struct anaL2_lkpResponse **result,
                  struct service_s *querierDescription,
                  struct timespec *timeout);
```

Where the arguments have the same semantics as for the *anaL2_resolve* command, except for the additional *result* argument. *result* is a pointer to a pointer to an *anaL2_lkpResponse* structure that will be allocated by the *anaL2_lookup* function. The structure *anaL2_lkpResponse* is as follows :

```
struct anaL2_lkpResponse{
        void *description;
        int descLen;
        anaLabel_t label;

        struct anaL2_lkpResponse *next;
};
```

Where :

- *description* is a complementary description of the service matching the query
- *descLen* is the size in bytes of the *description* field.
- *label* is an IDP label allowing to reach the service. This might not be always provided, depends on CPFB's will.
- *next* is a pointer to the next found service matching the lookup query

We will show later how to use this structure to interpret the node compartment's lookup response.
Important: This structure might not be suited for the lookup responses of different compartments. Therefore we introduce it here as a temporary solution for the node compartment's needs. This structure might then change in future.

The *anaL2_lookup* function returns the number of matching reponses in case of success (and fills the *result* pointer) or a negative error code in case of failure.

In AL1: The same steps detailed for the *resolve* case can be applied also here, except the encoding of the request that changes (step 2).

To encode an XRP lookup request, use the function:

```
int anaL1_encLookup( xrpMsg_t msg, struct context_s *ctxt,
                     struct service_s *srv,
                     struct service_s *queryingService);
```

Where the arguments have the exact same significance as for *anaL1_encResolve* shown above. This function returns the size of the encoded message in case of success or -1 in case of error.

What does the lookup response look like We do not propose any parsing strategy here since we did not fix yet a model for the lookup replies. We will later show the Lookup response of the node compartment, but that can not be applied to all compartments. This section will be filled after we do more experimentation.

Node compartment example

Lookup/Resolve query expressions : The lookup/resolve query addressed to the node compartment is a character string expressing keywords combination with the help of the following boolean operators:

- OR : expressed in the query string with the character '+'. This is a low priority operator. With this operator we can express that we are interested in entries matching either the first operand or the second. The order of operands is not relevant.
- AND : expressed in the query string with the character '*'. This is the highest priority operator. With this operator we can express that we are interested in entries matching both of the first and second operand. The order of operands is not relevant.
- MINUS : expressed in the query with the character '-'. This is a low priority operator. With this operator we can express that we are interested in entries matching the first operand **and not matching the second**. The order of operands in this case is relevant.

It is also possible to use parentheses to enforce a certain priority of operations.

Node compartment's Lookup response parsing in AL1: The lookup reply sent by the minmex can be parsed using the same functions presented before i.e *anaL1_isError* and *anaL1_decResponse*. For each entry matching the query, the minmex sends back the entry's first keyword (i.e. the first keyword passed in the publish request) and the IDP label stored in it. The matching results are listed in the reponse message the following way :

Where, first there is the first keyword and IDP of the first matching KVR entry, then first keyword and IDP of the second and so on. This order is important to know since we have to use the occurence argument correctly to extract the wanted aguments.

AL1 header	keyW 1	IDP 1	keyW 2	IDP 2	....

Figure 4.1: Lookup reply message

```
static void handleLookupReply( char *data, int len, anaLabel_t input,
                               void *aux) {
    char *name = NULL;
    anaLabel_t *labelP = NULL;
    anaLabel_t label;
    int labelSize,i;

    if(anaL1_isError(data)) {  /*if ERROR command*/
            anaPrint(ANA_ERR, "Lookup Failed\n");
            return;
    }
    i=0;
    while(anaL1_decResponse(data, XRP_CLASS_LKP_DESCR,
                            i, (void**)&name)>0)
    {
         labelP = NULL;
         labelSize =  anaL1_decResponse(data, XRP_CLASS_LABEL,
                                        i, (void**)&labelP);
         if(name!=NULL && labelP !=NULL){
                anaPrint(ANA_DEBUG, "found entry: %s\n", name);
                label = *labelP;
         }
         name = NULL;
         i++;
    }
}
```

Please note in the example above that we access the different results of the lookup query by incrementing the *occurence* argument that we pass to *anaL1_decResponse*

Node compartment's Lookup response parsing in AL2: In AL2 you do not need to do any XRP parsing since it is handled for you by the library. On success of the lookup query the *struct anaL2_lkpResponse *result* passed as argument to *anaL2_lookup* points now to a filled structure. The first matching KVR entry's keyword can then be accessed at $result \rightarrow description$ and the matching IDP label at $result \rightarrow label$. The next entry matching the lookup query can be accessed at $result \rightarrow next$ i.e. the lookup responses are stored in a chained list. To stop walking the chain, you can either check if $result \rightarrow next$ becomes *NULL* or use the number of responses that was returned by *anaL2_lookup* function.

4.12 How to unpublish services from compartments

If you want your service to be removed from a given compartment, you can send an unpublish command to the compartment provider functional block (CPFB). This operation is different for API levels 1 and 2.

In AL2: To unpublish a service in AL2, use the following function :

```
int anaL2_unpublish( anaLabel_t compIDP, anaLabel_t label,
                     struct context_s *ctxt,
                     struct service_s *service,
                     struct timespec *timeout);
```

Where :

- *compIDP* is the IDP label of the compartment provider functional block you want to unpublish the service from
- *label* is the IDP label associated to the service you want to unpublish
- *context* is the compartment context in which the service was published
- *service* is the description of the service to unpublish
- *timeout* indicates the timeout value for this operation. In case set to NULL, a default timeout of 1 second will be used.

This function encodes an XRP unpublish message, sends it to the CPFB, waits for a response and parses it before returning. In case of success it returns 0 otherwise a negative error code is returned.

Note that this function does not remove the IDP from the minmex (i.e the callback function could still be activated). Therefore, if you want to deactivate a service, you need to follow the instructions in previous subsection *How to delete IDPs from the minmex*

In AL1: The same 3 steps depicted for other AL1 commands are also needed here, i.e., allocating an AL1 message buffer, encoding an AL1 unpublish command and sending it to the CPFB.

To encode an AL1 unpublish command (Step 2), use the AL1 function:

```
int anaL1_encCompartmentUnpublish( xrpMsg_t msg, anaLabel_t idp,
                                   struct context_s *ctxt,
                                   struct service_s *srv);
```

Where:

- *msg* is the pointer to the allocated memory slot where to store the XRP encoded message
- *label* is the IDP label corresponding to the service you want to unpublish
- *ctxt* is the compartment context in which the service was published
- *srv* is the description of the service to unpublish

This function returns the size of the XRP encoded message in case of success and -1 if an error occurred.

To send the unpublish command to the CPFB, the two ways described for the publish command are possible i.e *anaL*1*_requestReply* by specifying a response handler function or *anaL*0*_send* in which case the CPFB won't be able to send a confirmation/error report.

The response parsing is done in a similiar way than the previously described AL1 functions (lookup, resolve, publish), with the help of the two functions *anaL1_decResponse* and *anaL1_isError*.

Please note that unpublishing a service does not delete the IDP that is associated to it. Indeed other bricks priorly knowing the IDP can still activate the associated callback function even though the IDP has been unpublished. Therefore if you want to deactivate a service, you need to follow the instructions depicted in *How to delete IDPs from the minmex* subsection.

4.13 How to release an IDP

This functionality concerns IDPs used by your brick and belonging to (owned by) other bricks as for example Information Channels IDPs that your brick is using. In case you are done with such an IDP, you can inform its owner brick that you no longer require the services of the IDP. This will allow the owner brick to free some resources in case the IDP is no longer needed by any other bricks.

4.13.1 In AL2

To release an IDP, you can use the following function :

```
int anaL2_release(anaLabel_t IDP);
```

Where *IDP* is the label of the IDP to release. This function returns 0 on success or a negative error value otherwise.

4.13.2 In AL1

The same 3 steps depicted for other AL1 commands are also needed here, i.e., allocating an AL1 message buffer, encoding an AL1 release command and sending it to the interested brick.

To encode an AL1 release command (Step 2), use the AL1 function:

```
int anaL1_encRelease(xrpMsg_t msg, anaLabel_t label);
```

Where:

- *msg* is the pointer to the allocated memory slot where to store the XRP encoded message
- *label* is the label of the IDP to release

This function returns the size of the XRP encoded message in case of success and -1 if an error occurred.

To send the release command to the interested brick, use the *anaL*0*_send* function. Note that in this case the *anaL*1*_requestReply* function is not needed since there is no reponse to be expected.

4.14 How to get/set information about an IDP

In oder to allow for ”intelligent“ composition of functionnalities (notably through a Functional composition Brick), it is possible in ANA to set and obtain information about existing IDPs.

The IDP information is stored in a structure *IDPinfo_s* defined as follows :

```
struct IDPinfo_s {

      anaLabel_t label;
      anaHandle_t viewer;
      anaHandle_t owner;
      char permFlag; /* 1 for permanent */

      char ICflag; /* 1 for IC*/
      struct context_s destContext;
      struct service_s destService;
      uint16_t MTU;
      char chanType;
      anaLabel_t controlIDP;

      char IDPtype;
      union destination_u {struct calleeBrick_s brick;
                           struct nextIDP_s *IDPs;} binding;
};
```

Where:

- *label* is the label of the IDP concerned by the info
- *viewer* is the view to which this IDP belongs. If equal to 0, than the IDP is public.
- *owner* indicates the handle (i.e. brick identifier) of the brick that created the IDP
- *permFlag* is a flag indicating if the IDP is permanent or volatile. 0 for volatile, 1 for permanent
- *ICflag* is a flag indicating wether the IDP is the start of an information channel or not. 1 for yes, 0 for no
- *destContext* in case the IDP is the start of an IC, this might (optional) indicate the destination context within the compartment
- *destService* in case the IDP is the start of an IC, this might (optional) indicate the destination service description within the compartment
- *MTU* in case the IDP is the start of an IC, this might (optional) indicate the Maximum Transmission Unit authorized on this channel
- *chanType* in case the IDP is the start of an IC, this might (optional) indicate the type of the channel. 'm' for Multicast, 'b' for Broadcast, 'a' for Anycast, 'u' for Unicast and 'c' for Concast.
- *controlIDP* indicates the control IDP (API calls handler) of the brick owning this IDP
- *IDPtype* indicates the type of the IDP, i.e either it is attached to a brick or it is a redirected IDP or a forked IDP. This field takes the values on an enumeration containing the following constants: *BRICK*=0, *REDIRECT*=1, *FORK*=2
- *binding* this union indicates the next step within the ANA node. Depending on the IDP type, one of the union elements is filled.

4.14.1 More details on the binding union

Case of an IDP attached to a brick in case the IDP is of type *BRICK*, the field *brick* in the *binding* union is filled. This field is of *struct calleeBrick_s* structure defined as follows :

```
struct calleeBrick_s {
        anaHandle_t handle;
        anaLabel_t outputIDP;
};
```

Where :

- *handle* is the identifier of the brick at which this IDP triggers a callback function
- *outputIDP* might(optional) indicate to which IDP the brick identified with *handle* forwards the data after treating it

Case of a redirected or forked IDP In case the IDP is redirected or forked (i.e. if *IDPtype* equals *REDIRECT* or *FORK*) the union field *IDPs* is filled. This field is a pointer to a structure *struct nextIDP_s* defined as follows :

```
struct nextIDP_s {
        anaLabel_t label;
        anaHandle_t viewer;
        struct nextIDP_s *next; /* in case it is a fork */
};
```

Where :

- *label* is the IDP label of the destination of the fork or redirection
- *viewer* is the view to which belongs the destination
- *next* is a pointer to another element of the same structure. This pointer is filled in case the IDP is of type *FORK* in which case the IDP leads to multiple other IDPs that are stored in a chained list. In case the IDP is of type *REDIRECT*, this pointer is *NULL*.

4.14.2 Getting information about an IDP

To get information about a specific IDP, you can use the following function :

```
int anaL0_getIDPinfo(  anaLabel_t label,  anaHandle_t viewer,
                       struct IDPinfo_s *result);
```

Where :

- *label* is the label of the IDP to extract information about
- *viewer* is the IDT view to which the IDP belongs
- *result* is a pointer to the *IDPinfo_s* structure to be filled by the function.
  Important : this structure **must be allocated prior to the call** to *anaL0_getIDPinfo*

This function returns 0 in case of success and fills the pointed *result* structure appropriately. Note that on success, all the fields of the IDPinfo structure are not necessarily filled (example if the IDP is not an IC, the destContext, destService and MTU fields are empty). In case of error a negative error code is returned.

Important : In case the IDP you got information about is of type *REDIRECT* or *FORK*, the *binding.IDPs* field of the *IDPinfo_s* structure has been allocated by the library to hold the chained list of IDPs. Therefore, you must free these properly when you are done with them, by calling :

```
int anaL0_clearIDPinfo(struct IDPinfo_s *info);
```

Where *info* points to the *struct IDPinfo_s* that was filled by *anaL0_getIDPinfo*. Note that this fuction frees the allocated *binding.IDPs* fields but does not free the structure pointed by *info*.

4.14.3 Setting information about an IDP

Setting some information about an IDP is done in 2 steps :

1. prepare the *struct IDPinfo_s* containing the information you want to set
2. submit this structure to the minmex via the *anaL0_setIDPinfo* function.

Step 1, preparing the information structure : The information that you can set about a certain IDP(belonging to your brick) is :

- Indicating the label and viewer of the IDP you are setting information about by filling the corrsponding fields in the *struct IDPinfo_s*
- Indicating the control IDP of the brick owning the IDP you are setting information about by filling the corrsponding field in the *struct IDPinfo_s*
- Indicating wether it is an Information channel or not. This can be done by setting ICflag of the *struct IDPinfo_s* to 1. If your IDP is a start of an IC, you can choose (not mandatory) to set the following information too :
  - The destination context of the IC: by filling the *struct context_s* structure that is part of the *struct IDPinfo_s*
  - The destinaton service of the IC: by filling the *struct service_s* structure that is part of the *struct IDPinfo_s*
  - The Channel type (Unicast, Multicast. etc.) of your IC by filling the *chanType* field of the *struct IDPinfo_s* with one of the values depicted above ('u', 'a', 'm', 'b', 'c')
  - The MTU supported by your IC by filling the *MTU* field of the *struct IDPinfo_s* with the appropriate value
- Indicating to which IDP within the ANA node you forward the data (after treatment) received on the IDP you are setting information about. This is very useful for Functional composition and can be done by filling the *binding.brick.outputIDP* field of the *struct IDPinfo_s* with the label of the next IDP.

Step 2, submitting the information to the minmex Once you have prepared the *struct IDPinfo_s* structure, you can submit it to the minmex (i.e. do the information setting) using the following function:

```
int anaL0_setIDPinfo( struct IDPinfo_s *info)
```

This function returns 0 in case of success or a negative error code in case of failure.

Important : you need to be the owner of the IDP in order to be able to set information regarding it, otherwise this function will return an ”Unauthorized“ error.

Example of setting an IDP information In this example, we first publish an IDP and then set some information about it :

```
int brick_start() {

        struct service_s service;
        struct context_s context;
        struct timespec timeout;

        struct IDPinfo_s info;

        int ret ;

        memset(&service, 0, sizeof(struct service_s));
        memset(&context, 0, sizeof(struct context_s));


        anaL2_initDefault();

        context.value = ".";
        context.valueLen = 2;

        service.value = "service1";
        service.valueLen = strlen("service1")+1;

        memset(&timeout, 0, sizeof(struct timespec));
        timeout.tv_sec = 5;


        /* we publish then a service with a callback function */
        serviceIDP = anaL2_publish( NODE_LABEL, &context,
                        &service, (AL2Callback_t) &recvData2,
                                        1, &timeout);

        if(serviceIDP == 0)
                anaPerror(0,"---> publish failed, error ");
        else
                anaPrint(ANA_NOTICE, "---> publish ok IDP:%d\n",
                        serviceIDP);


        /* we set Some info about the IDP */
```

```
        memset(&info, 0, sizeof(struct IDPinfo_s));

        info.label = serviceIDP;
        info.viewer = 0;

        info.ICflag = 1;
        info.destContext.value = "192.168.0.1";
        info.destContext.valueLen = strlen("192.168.0.1") + 1;

        info.destService.value = "chat";
        info.destService.valueLen = strlen("chat") + 1;

        info.MTU = 1500;
        info.chanType = 'u'; /* Unicast channel */

        /* here we suppose that nextIDP is the label to which
           our brick forwards data (after treatment) received
           on serviceIDP
        */
        info.binding.brick.outputIDP = nextIDP;

        ret =  anaL0_setIDPinfo( &info);
        if(ret < 0)
                anaPrint(ANA_DEBUG, "*** setIDPinfo failed \n");

        return 1;

}
```

Note that in this example the AL0 library will hiddenly fill some other fields of the *struct IDPinfo_s*, notably setting the *IDPtype* to *BRICK* and the *binding.brick.callee* with the handle of our brick.

4.15 How to subscribe/notify about events

In ANA, in order to allow bricks to be reactive to some events regarding IDPs and other bricks, we put in place a notification system that can alert about the following events:

- Deletion of an IDP : this event is identified by the constant *IDP_DELETED_EVENT*
- IDP is temporarily busy : this event is identified by the constant *IDP_BUSY_EVENT*
- IDP is back to be available : this event is identified by *IDP_AVAIL_EVENT*
- Redirection of an IDP : this event can be triggered whenever an IDP is redirected (via *anaL0_redirect* function) towards another IDP. This event is identified by the constant *IDP_REDIRECTED_EVENT*
- Unpublication of an IDP : whenever a compartment provider brick unpublishes an IDP from its compartment, it can alert the IDP owner of this fact by triggering this event. This event is identified by the constant *IDP_UNPUBLISHED_EVENT*

- Information update about an IDP : in case a brick changes the information regarding one of its IDPs (via the *anaL0_setIDPinfo* function), it can alert the other bricks interested in this information by triggering this event so that they fetch the new info. This event is identified by the constant *IDP_INFOUPDATED_EVENT*
- Attachment of a brick : this event can be triggered whenever a brick is attached to the minmex. This event is identified by the constant *BRICK_ATTACHED_EVENT*.
- Detachement of a brick : this event can be triggered whenever a brick is detached from the minmex. This event is identified by the constant *BRICK_DETACHED_EVENT*.

4.15.1 Subscription to events

In order to subscribe to events, 2 steps need to be filled :

1. preparing the subscription structure
2. submitting it to the minmex

Step 1, preparing the subscription structure This structure is defined as follows :

```
struct anaEventSub_s {
        uint32_t eventMask;

        anaLabel_t label;
        anaHandle_t viewer;

        char *name;
};
```

Where :

- *eventMask* indicates to which events the subscription is being done. The subscription can be done to multiple events as long as they concern the same object (Brick or IDP). Indeed, events can be combined with the binary OR operator in a similar way flags are combined in file system calls. For example `'IDP_DELETED_EVENT|IDP_UNPUBLISHED_EVENT|IDP_REDIRECTED_EVENT'` is a valid mask for subscription to the three events at the same time.
  Important : As mentionned priorly , the combination of events **must** concern the same object (IDP or Brick). Therefore event masks susch as `'IDP_DELETED_EVENT|BRICK_ATTACHED_EVENT'` do not make sense and will be rejected by the minmex when submitted.
- *label*, *viewer* should be filled if you are subscribing to an event regarding a specific IDP. If you are subscribing to a Brick event, or to events regarding all IDPs (e.g. you want to know about every IDP being deleted), these fields should be left empty.

- *name* is the name of the Brick you are interested in, in case you are subscribing about an event concerning a specific Brick. In case of an IDP event, or an event regarding all Bricks (e.g. you want to know about every Brick attached or detached), this field should be left empty (NULL).

Step 2, submitting the subscription To submit the subscription, you can use the function :

```
int anaL0_eventSubscribe( struct anaEventSub_s* sub,
                          eventHandler_t handler, int threadFlag);
```

Where:

- *sub* points to the subscription structure filled in Step 1
- *handler* is a function pointer of the function that will be triggered upon reception of this event. we describe these handlers in more details in the next paragraph
- *threadFlag* indicates wether your event handler should be triggered in a separate thread or not.
  Very Important: if your event handler function uses AL2 function calls like *anaL2_publish* or *anaL2_resolve* (etc), if it is blocking, or also if it takes a long time to execute then your event handler function **MUST** be launched in a **separate thread**. Indeed, if you set the *separateThread* flag to 0, the function will be executed by the thread expecting messages. Therefore, if the function blocks waiting for incoming messages (as required by AL2 publish, lookup etc), the thread receiving incoming messages is blocked and reception of messages becomes impossible $\Rightarrow$ **deadlock !**

This function returns 0 in case of successfull subscription

Event handler functions The event notification handler functions must be of the following prototype :

```
void exampleHandler(struct anaEventNotif_s *notif)
```

Where *notif* points to a structure containing the notification info, defined as follows :

```
struct anaEventNotif_s {
     uint32_t eventID;

     anaLabel_t label;
     anaHandle_t viewer;

     anaLabel_t destLabel;
     anaHandle_t destViewer;

     char *name;
};
```

Where:

- *eventID* is the constant identifying the event that was triggered. Not that in this case this field is not a mask (i.e. a combination of events) but rather a unique event value.
- *label*, *viewer* these fields are filled in case the received notification is regarding an IDP event and indicate which specific IDP it concerns. In case of an event regarding a brick, these fields are empty.
- *destLabel*, *destViewer* are filled in case the notification is about a redirection of an IDP (i.e. *eventID = IDP_REDIRECTED_EVENT*). These fields indicate towards which destination IDP the redirection was done. In case of an event regardint a brick, these fields are empty.
- *name* indicates the name of the brick being attached or detached in case of a notification regarding a brick event. Also, in case of *IDP_UNPUBLISHED_EVENT* this field might contain the name of the brick that unpublished the IDP.

Important : the *struct anaEventNotif_s* structure is freed hiddenly by the library, so there is no need to free it at the end of your handler functions.

4.15.2 Unsubscription from events

Unsubscription is also done in 2 steps as for the subscription. The first step is exactly identical to the subscription case. The second step (unsubscription submission) can be performed by using the following function :

```
int anaL0_eventUnsubscribe(struct anaEventSub_s* sub);
```

Where *sub* points to the structure filled in step 1. This function returns 0 in case of success or a negative error code in case of error.

4.15.3 Notify subscribers about occurence of an event

Your brick can notify other bricks about the following events :
IDP_REDIRECTED_EVENT, *IDP_UNPUBLISHED_EVENT*,
IDP_INFOUPDATED_EVENT. The rest of the events
(*IDP_DELETED_EVENT* and brick events) are triggered automatically by the minmex whenever they occur.

The notification is also done in 2 steps:

1. filling the *struct anaEventNotif_s* notification structure
2. submitting it to the minmex

Step 1, filling the notification structure Step 1 is done in a symetric way to notification reception by the handler, i.e. you fill a *struct anaEventNotif_s* structure with the fields regarding your event.

- in case of *IDP_REDIRECTED_EVENT*, set the *eventID* field of the structure to *IDP_REDIRECTED_EVENT*, and fill in the *label* and *viewer* of both the IDP being redirected and the destination IDP.
- in case of *IDP_UNPUBLISHED_EVENT*, set the *eventID* field of the structure to *IDP_UNPUBLISHED_EVENT*, and fill in the *label* and *viewer* of the IDP being unpublished. you can also indicate the name of your brick in the field *name* of the *struct anaEventNotif_s* structure.
- in case of *IDP_INFOUPDATED_EVENT*, set the *eventID* field of the structure to *IDP_INFOUPDATED_EVENT*, and fill in the *label* and *viewer* of the IDP of which the info is being updated.

Step 2, submitting the notification to the minmex This can be done using the follwing function :

```
int anaL0_eventNotify( struct anaEventNotif_s *notif);
```

Where *notif* points to the structure that was filled in Step 1.

4.16 How to interpret error codes

In case of functions returning negative error codes (such as *anaL0_eventSubscribe*, *anaL0_redirect*, *anaL0_setIDPinfo*, etc.) the corresponding error message can be printed out using the function :

```
char *getErrorString(int ret)
```

Where *ret* is the error code to which you want to map a ”human“ understandable error message.

There are some functions however that do not return a negative error code since their prototype does not allow it. For example, *anaL0_registerCallback*, *anaL2_publish*, *anaL2_resolve*, etc., all return an IDP label that is by definition a positive non null value. For error report in these function, we defined in ANA, a system similar to the standard C *errno* + *perror* one. Following this system, functions like (*anaL0_registerCallback*) can set the ANA *errno* equivalent for their threads. This error code can then be retrieved and its error message printed using the ANA *perror* equivalent :

```
void anaPerror(int level, char * str)
```

Where :

- *level* indicates the debugging level (e.g. *ANA_DEBUG*, *ANA_NOTICE*, etc.) to be passed to the *anaPrint* function
- *str* is the character string you want to print out before the error message (i.e. the output is ”str : error message“)

4.17 XRP details

4.17.1 Format of XRP messages

Figure 4.2 shows an XRP encoded message where :

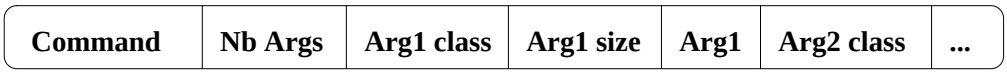

Figure 4.2: XRP message

- *Command* determines the purpose of the message (For example if it is a Publish or Lookup command). We will give later the list of the default most common command constants.
- *Nb Args* is the number of XRP arguments attached to this command.
- *Arg*1*class* is the XRP class of the first argument attached to this command. By class we mean a known description (meta-data) about the nature of the argument. This can be seen as an equivalent to *types* in programming languages. We will also give later an enumeration of the default class constants.
- *Arg*1*size* is the size in bytes of the first argument value.
- *Arg*1 is the value of the first argument associated to the XRP command.

4.17.2 Defined constants

XRP Commands The most common XRP commands you could have to handle are the following:

- *XRP_CMD_PUBLISH* : this constant is put in front of the XRP encoded publish commands (encoded usig *anaL1_encPublish*)
- *XRP_CMD_RESOLVE* : for resolve commands (*anaL1_encResolve*)
- *XRP_CMD_LOOKUP* : for lookup commands (*anaL1_encLookup*)

- *XRP_CMD_UNPUBLISH* : for unpublish commands (*anaL1_encUnpublish*)
- *XRP_CMD_DATA* : for data forwarding commands (*anaL1_encData*)
- *XRP_CMD_ERROR* : for error messages

See `C/include/xrp.h` for a full list of XRP COMMAND constants.

XRP Classes As said previously, XRP classes are the type of the arguments passed along with XRP commands. The most common XRP classes you would have to deal with are the following:

- *XRP_CLASS_LABEL* : this constant is the type of the IDP label arguments that could be for example passed in publish commands, resolve responses, etc.
- *XRP_CLASS_SRC_CXT* : for the source *context* argument passed along with publish commands, resolve and lookup commands and responses.
- *XRP_CLASS_DST_CXT* : for the destination *context* argument passed along with publish commands, resolve and lookup commands and responses.
- *XRP_CLASS_SRC_SRV* : for the source *service* description argument passed along with resolve, lookup and forward data commands.
- *XRP_CLASS_DST_SRV* : for the destination *service* description argument passed along with resolve, lookup and forward data commands.
- *XRP_CLASS_LKP_DESCR* : for the complementary service description passed along in *lookup* responses.
- *XRP_CLASS_MESSAGE* : for the *data* argument you pass along the forward data command.
- *XRP_CLASS_CHANTYPE* : for the *chanType* argument you pass along the resolve command (*anaL1_encResolve*)
- *XRP_CLASS_PERMANENT* : for the *permanent* flag argument you pass to the publish command (*anaL1_encPublish*)

See `trunk/C/include/anaCommon.h` for a full list of XRP CLASS constants.

4.17.3 XRP API

The XRP API is composed of the following functions :

- `int allocateXRPMsg(xrpMsg_t *msg);`

Where : *msg* is a pointer to a *xrpMsg_t*. The *xrpMsg_t* type being itself a pointer to *char* *. This function allocates a memory slot for an XRP message (with a default size). On sucess, the pointer pointed by *msg* now points to an allocated memory slots, and 0 is returned. on failure -1 is returned.

- `int fillXRPCommand (xrpMsg_t msg, xrpCmd_t cmd);`

  Where :

  - *msg* points to a memory slot allocated by *allocateXRPMsg*
  - *cmd* is the XRP COMMAND constant corresponding to the command you want to encode.

  This function will store the constant value at the head of the allocated buffer.

- ```
  int addXRPArg(  xrpMsg_t msg, xrpClass_t xrp_class,
                  void *content, unsigned short int len);
  ```

  Where :

  - *msg* is the allocated buffer for the message
  - *class* is the XRP class constant of your argument
  - *content* is the value of the argument
  - *len* is the size in bytes of the argument value (size of *content*)

  This function adds the argument at the tail of the XRP message (if possible) and updates the total number of arguments in the message. In case the argument does not fit in the buffer -1 is returned, otherwise the new size of the XRP message is returned.

- `int getXRPnbArgs(xrpMsg_t msg);`

  Where *msg* is the received XRP message. This function returns the number of arguments contained in *msg* (by reading the second field of the XRP message).

- ```
  void *getXRPArg( xrpMsg_t msg, xrpClass_t xrp_class,
                   int *len, int occurence);
  ```

  Where :

  - *msg* points to the received XRP message
  - *class* is the XRP class constant of the argument you are interested in
  - *len* is a pointer to an allocated integer. On success the pointed field will contain the size in bytes of the wanted argument.
  - *occurence* is the occurence of the XRP argument of class *class* that we are interested in. For example if the received XRP message contains 2 argument of class *XRP_CLASS_LABEL*, setting *occurence* at 1 means that we are interested in the second IDP label contained in the message (i.e the occurence count starts at 0).

```
int equalXRPCommands(xrpCmd_t cmdA,  xrpCmd_t cmdB );
```

This function compares the two pointed XRP COMMAND values. It returns 1 in case they are equal, 0 otherwise.

- `int equalXRPClasses(xrpClass_t classA,  xrpClass_t classB );`

  This function compares the two pointed XRP CLASS values. It returns 1 in case they are equal, 0 otherwise.

4.18 I'm building a compartment provider brick, what requirements should I follow ?

4.18.1 Visibility in the node Compartment

In order for other bricks to use your compartment services, you should make your brick visible in the node compartment. You can do so by following the *node compartment example* subsection in section *How to publish an IDP in a compartment*.
Important : To ease the self-association of other bricks to your compartment, **please add the keyword "compartment" to the description you publish to the Node Compartment**.
This way, a brick querying the node compartment for all bricks matching the keyword "compartment" will be able to know all the networking capabilities of its ANA node.

4.18.2 Handling the common generic commands

The callback function associated to the IDP you published to the node compartment must be able to parse the incoming generic XRP commands (publish, lookup, etc ..) and then execute the corresponding function and send an answer.

The command can be identified with the *equalXRPCommands* function as shown previously, while the arguments can be extracted with the function *anaL1_decResponse* also shown previously. Note that there is no specific way for handling this in AL2, only AL1 style is available.

```
void publishedCallBack( char *data, int len,
                        anaLabel_t input, void *aux) {

/*if your are using AL2 The previous header can be replaced by :*/
/* static void publishedCallBack(struct anaL2_message *msg) */


        xrpCmd_t command = data;
```

```
        /* xrpCmd_t command = msg->data (for AL2)*/

        if (equalXRPCommands(command, XRP_CMD_PUBLISH)) {
        /* we extract the description of the service to publish*/
                void *description = NULL;
                int descLen =  anaL1_decResponse( command,
                                XRP_CLASS_SRC_SRV, 0, &description);
                ...
        }
}
```

Note that where *anaL1_decResponse* was used *getXRPArg* depicted in the XRP dedicated section could also be used.

Abstract layout of received generic common commands: Here we will show the order of appearance of arguments for the common commands your compartment brick will receive.

pub	label	service	context	permanent	reply label

Figure 4.3: Arguments order in received publish message

Publish command Figure 4.3 shows the layout of a publish command, where :

- *pub* is the *XRP_CMD_PUBLISH* header
- *label* is the IDP label of the service to publish. Its XRP class is *XRP_CLASS_LABEL*. The variable type to store this argument in is *anaLabel_t*
- *service* is the description of the service to publish. Its XRP class is *XRP_CLASS_SRC_SRV* The variable type to store this argument in is *void* *
- *context* is the context where the service is to be published. Its XRP class is *XRP_CLASS_SRC_CXT*. The variable type to store this argument in is *void* *. Please do not forget to handle the cases of context "*" and ".". Context "*" means that the user wants you to publish the service with the largest possible compartment scope. Context "." means that the user wants you to publish the service in your local repository only.
- *permanent* is the flag to say wether the user wants the publication to be permanent or not. Its XRP class is *XRP_CLASS_PERMANENT*. The presence of the class *XRP_CLASS_PERMANENT* indicates that the flag is on. In case the publication is not meant to be permanent, there will be no *XRP_CLASS_PERMANENT* in the message.

- *reply label* is the IDP label to send the publish response to. Its XRP class is *XRP_CLASS_LABEL*. The variable type to store this argument in is *anaLabel_t*. Pay attention to indicate *occurence* equal to 1 to *anaL1_decResponse* in order to extract this label and not the service IDP Label

Figure 4.4: Arguments order in received resolve message

Resolve command Figure 4.4 shows the layout of a received resolve command, Where :

- *rsv* is the *XRP_CMD_RESOLVE* header
- *service* is the description of the service the user is looking for. Its XRP class is *XRP_CLASS_DST_SRV*. The variable type to store this argument in is *void **
- *context* is the subset of the compartment we will have to look in for the service. Its XRP class is *XRP_CLASS_DST_CXT*. The variable type to store this argument in is *void **. Please do not forget to handle the cases of context "*" and ".". Context "*" means that the user wants you to look for the service as far as you can. Context "." means that the user wants you to look for the service in your local repository only.
- *chanType* is the type of Information Channel the user wants to build. Its XRP class is *XRP_CLASS_CHANTYPE*. The variable type to store this argument in is *char*
- *descrip* is the description of the querying user.
  Its XRP class is *XRP_CLASS_SRC_SRV*. The variable type to store this argument in is *void **. Note that this will be included in the message only if the querier indicated a description. Otherwise there would be no *XRP_CLASS_SRC_SRV* in the message.
- *reply label* is the IDP label to send the publish response to. Its XRP class is *XRP_CLASS_LABEL*. The variable type to store this argument in is *anaLabel_t*. In this case, you can indicate *occurence* equal to 0 to *anaL1_decResponse* since there is no other argument of class *XRP_CLASS_LABEL* in the message.

Lookup command The lookup command is similar to the resolve one except that the command header changes and there is no *chanType* argument in *lookup*

Figure 4.5: Arguments order in received unpublish message

Unpublish command Figure 4.5 shows the layout of a received unpublish command, Where :

- *unp* is the *XRP_CMD_UNPUBLISH* header
- *label* is the IDP label of the service to unpublish. Its XRP class is *XRP_CLASS_LABEL*. The variable type to store this argument in is *anaLabel_t*
- *context* is the compartment context from which to unpublish the service. Its XRP class is *XRP_CLASS_SRC_CXT*. The variable type to store this argument in is *void* *
- *service* is the description of the service the user wants to unpublish. Its XRP class is *XRP_CLASS_SRC_SRV*. The variable type to store this argument in is *void* *
- *reply label* is the IDP label to send the unpublish response to. Its XRP class is *XRP_CLASS_LABEL*. The variable type to store this argument in is *anaLabel_t*. In this case, you can indicate *occurence* equal to 0 to *anaL1_decResponse* (or *getXRPArg*) since there is no other argument of class *XRP_CLASS_LABEL* in the message.

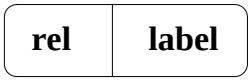

Figure 4.6: Arguments order in received release message

Release command Figure 4.6 shows the layout of a received release command, Where :

- *rel* is the *XRP_CMD_RELEASE* header
- *label* is the IDP label of the service to unpublish. Its XRP class is *XRP_CLASS_LABEL*. The variable type to store this argument in is *anaLabel_t*

Command answering strategy:

Error report If the command queried by the user did not for some reason succeed, and you want to report the error back, all you have to do is replace the previous command header with the error command header *XRP_CMD_ERROR*, and send the message back to the user on the *reply IDP* he indicated in the command, as shown in the Example below:

```
void publishedCallBack( char *data, int len,
                        anaLabel_t input, void *aux) {
        ...
        /* first we extract the IDP label to reply on */
        /* we suppose this is a publish command, therefore */
        /* replyTo label is at second occurence in message*/
        /* note that the occurence count starts at 0 */

        anaLabel_t *labelP= NULL;
        anaLabel_t replyTo;
        int labelSize =  anaL1_decResponse( data,
                              XRP_CLASS_LABEL, 1, &labelP);
        replyTo = *labelP;

        /* Now we will replace the header of the received */
        /* message with XRP_CMD_ERROR */

        fillXRPCommand((xrpMsg_t) data, XRP_CMD_ERROR);

        /* and we send it back to the user */

        anaL0_send(replyTo, data, len);
}
```

Success report

- For *publish* and *unpublish* : you just have to send the command message sent by the user back to him on the reply label he indicated.

```
void publishedCallBack( char *data, int len,
                        anaLabel_t input, void *aux) {
        ...
    anaLabel_t *labelP = NULL;
    anaLabel_t replyTo;
    int labelSize =  anaL1_decResponse( data,
                                XRP_CLASS_LABEL, 1, &labelP);
    replyTo = *labelP;
    /*pay attention to use occurence = 0 for unpublish */
    anaL0_send(replyTo, data, len);
}
```

- For *resolve* command: the user is expecting back one IDP label of the Information Channel you constructed for him. The resolve response message is then composed of an *XRP_CMD_RESOLVE* and one argument of class *XRP_CLASS_LABEL*. An example of building such a reponse message is given below.

```
void publishedCallBack( char *data, int len,
```

```
                        anaLabel_t input, void *aux) {
    ...
    anaLabel_t *labelP = NULL;
    anaLabel_t replyTo;
    int msgSize;
    xrpMsg_t msg;
    int labelSize =  anaL1_decResponse( data,
                                XRP_CLASS_LABEL, 0, &replyTo);
    replyTo = *labelP;
    /* we allocate a response message */
    allocateXRPMsg(&msg);

    /* we first fill the command header */
    fillXRPCommand(msg, XRP_CMD_RESOLVE);

    /* we then fill the IDP label argument */
    /* resLabel is the label you created for the Info Channel */
    msgSize = addXRPArg(msg,XRP_CLASS_LABEL,resLabel,labelSize);

    /* and we send the message back to the user */
    anaL0_send(replyTo, msg, msgSize);
    free(msg); /* do not forget to free the message once sent */
}
```

- For *lookup* command : we did not determine a response format yet.

4.18.3 Forwarding incoming messages to a "higher layer"

On reception, from other compartment peers, of a message destined to a service that published itself in your compartment brick, you can forward this message to the user brick ("higher layer" brick) using the following functions:

- For AL2 :

```
int anaL2_forwardData(  anaLabel_t destIDP, void *data ,
                        int lenData,
                        struct context_s *senderContext,
                        struct service_s *senderService,
                        anaLabel_t responseLabel)
```

  Where :

  - *destIDP* is the IDP to reach the "higher layer" brick.
  - *data* is a pointer to the received data
  - *lenData* is the length of the received data
  - *senderContext* is an optional argument. It corresponds to the subset of the compartment from which the message came. For example in an Ethernet compartment this could be the *MAC address* of the sender. It is up to you (compartment specific) to decide wether to trasmit or not this information to the upper "layers"

- *senderService* is the service description of the data sender. This argument is optional too
- *responseLabel* is also an optional argument. It is meant to be filled in case you built an information channel back to the sender and want to indicate to the user brick that it could answer back using that IDP's label.

This function encodes an *XRP_CMD_DATA* and sends it to the IDP *destIDP*. On success the number of sent bytes is returned, a negative error code is returned otherwise.

- For AL1 : forwarding data is done by encoding an *XRP_CMD_DATA* command and sending it to the user brick's IDP. To encode the *data* command in AL1 use :

```
int anaL1_encData( xrpMsg_t msg, void *data , int lenData,
                   struct context_s *senderContext,
                   struct service_s *senderService,
                   anaLabel_t responseLabel);
```

Where :

- *msg* is a buffer previously allocated with *anaL1_allocateMessage()*.
- *data* points to the message you want to forward to the user brick (the message you received from the compartment peer).
- *lenData* is the size in bytes of *data*
- *senderContext* is the subset of the compartment from which the message came.
- *senderService* is the description that the remote (sender) compartment user published to the compartment.
- *responseLabel* is meant to be filled in case you built an information channel back tot he sender and want to indicate to the user brick that it could answer back using that IDP's label.

This function returns the size of the XRP message encoded in *msg* on success or -1 in case of error.

4.19 How to compile my brick

4.19.1 User space compilation

In order to compile your brick for user-space execution, 2 steps need to be done :

1. Prepare the user-space specific Makefile at the root of your brick's directory to correctly compile your brick
2. Modify the config.txt file (i.e. main compilation system) so that it compiles your brick too

Step 1, preparing the user-space Makefile This step is done by copying the right user-space Makefile template to the root of your brick's directory. In case your brick consists of a single "executable" (i.e. one .so and one binary only), you can copy the `Makefile-user-singlebrick.template` file at the ANA root directory to your brick's directory and rename it to `Makefile-user`.

If you wish to build a functional block consisting of multiple bricks (i.e. multiple "executables" as it is the case for the *ip* brick), you can copy the `Makefile-user-multiplebricks.template` file at the ANA root directory to your brick's directory and rename it to `Makefile-user`.

Note that this Makefile will be responsible for compiling your brick in both PLUGIN (as a .so) and GATES (as a binary) modes. Instructions how to modify the Makefiles properly are indicated in the templates.

Step 2, modifying config.txt file to compile your brick In order for your brick to be compiled for user-space when you type `'make'` from the ANA root directory, you need to modify the `config.txt` file in the following way :

- to compile your brick as a .so plugin : add your brick's (FB's) directory name (contained in `C/bricks/`) to the list of the variable *USER_PLUGIN_BRICKS*
- to compile your brick as a standalone application : add your brick's (FB's) directory name contained in `C/bricks/`) to the list of the variable *USER_PROCESS_BRICKS*

4.19.2 Kernel compilation

The same steps for user-space compilation need to be done here.

Step 1, preparing the kernel-space Makefile For kernel compilation of your code, you can copy the `Makefile-kernel.template`, present at the ANA root directory, to your brick's directory and rename it to `Makefile`. Note that, on the contrary to the user-space case, this template Makefile works for both single brick and multiple bricks Functional Blocks. Instructions how to modify the Makefile properly are indicated in the template.

Step 2, modifying config.txt file to compile your brick

- to compile your brick as Kernel module in PLUGIN mode (using direct function calls), add your brick's directory name to the variable *KERNEL_PLUGIN_BRICKS* in `config.txt`
- to compile your brick as Kernel module in GATES mode, add your brick's directory name to the variable *KERNEL_MODULE_BRICKS* in `config.txt`

4.20 How to instantiate a brick from wihtin the code

In case your brick requires the services of an additonal brick that is not currently running on the local node, you can instantiate the brick if the plugin or binary file is available on your file system. Note that for security reasons, this functionality is available only for user-space minmex.

4.20.1 To instatiate a .so plugin brick

You can use the following function :

```
int anaL0_launchSoBrick(char *path, char *auxArgs);
```

Where :

- *path* : is the full path to the .so file
- *auxArgs* : is the character string containing the auxiliary arguments to be passed to the launched brick. e.g. if you need to pass 2 arguments then

  ```
  char *auxArgs = "arg1 arg2";
  ```

4.20.2 To instatiate a standalone brick

```
int anaL0_launchGatesBrick(const char *path, char *args, char *envp);
```

Where :

- *path* : is the full path to the brick's binary file.
- *args*: is the character string containing the auxiliary arguments to be passed to the launched brick.
- *envp* : is the character string containing the SHELL environment variables to be passed to the brick. See *man execve* for more details.

Note that this functionality might be de-activated in future versions since it allows to launch any executable (not only bricks) and constitutes therefore a major security threat.

Chapter 5

Virtual link support

5.1 What is it?

Strictly speaking, the `vlink` subsystem does not belong to the "ANA world" in the sense that it does not provide any autonomic networking functionality and is not an abstraction of ANA. Actually, the `vlink` brick is one (possible) implementation of the abstraction layer of the ANA Node (as described in the Blueprint): it basically decouples the real physical world of network interface cards and physical links from the connectivity seen by ANA entities. The main motivation for developing this brick is to have a flexible and dynamically re-configurable subsystem for emulating connectivity between ANA nodes without limitations and constraints imposed by technical issues such as manual cabling.

A key objective was to "virtualize" network connectivity in order to be able to e.g. emulate link failures without having to physically interfere with network equipments and without having to physically connect or disconnect network devices. The ability to virtualize physical links makes it possible to create multiple and "parallel" arbitrary virtual network topologies on top of the real underlying physical world: it hence becomes possible to simultaneously run multiple experimentations which, while sharing the same physical devices and links, do not interfere with each other in the emulated environment.

Note that the configuration of vlinks is not automatic nor autonomic: human users must configure vlinks. In order to perform this task, we have developed a `'vlconfig'` command with which one can easily configure vlinks locally on a given ANA host; which means so far one has to have local access to an ANA host in order to locally configure vlinks. In the future, some partners of WP4 should develop testbed tools that will allow the configuration of vlinks remotely, possibly via a remote access to the `'vlconfig'` command tool.

5.2 Files, compilation

Starting from the ANA root directory, the directories and files that form the `vlink` implementation are:

- `C/include/ana_vlink.h` : header file for `vlink`.
- `C/include/ana_vlinkAPI.h` : header file for `vlink` API: public functions.
- `C/bricks/vlink/` : contains all the code for `vlink`.
- `C/bricks/vlink/Makefile` : the makefile for Linux kernel 2.6 compilation.
- `C/bricks/vlink/Makefile-user` : the makefile for userspace compilation.
- `C/bricks/vlink/common/ana_vlink.c` : the basic common code for creating and manipulating `vlink` objects.
- `C/bricks/vlink/kernel/kernel_vlink.c` : the code for sending and receiving packets to/from interfaces inside the Linux kernel.
- `C/bricks/vlink/userspace/user_vlink.c` : the code for sending and receiving packets to/from interfaces via standard interfaces with RAW sockets.

The `vlink` brick is compiled by default by the current ANA build system. Note that when compiled as a .so plugin, **the minmex process must be exectued with root priviledges** as it uses a RAW socket (PF_PACKET) to send and receive data. Also when compiled as a userspace application, **the vlink brick must be executed with root priviledges** for the same reasons.

5.3 How does it work?

The `vlink` brick has direct access to the real interfaces and the legacy networking layers (e.g. UDP) of the host operating system. To distinguish "`vlink` packets" from other packets (e.g. IP, ARP, etc), the `vlink` brick uses a specific Ethernet type field (ETH_P_ANAVLINK defined in `ana_vlink.h`) over Ethernet and GRE (Generic Routing Encapsulation) interfaces. It is also possible to use UDP "tunnels" to interconnect distant nodes: at the moment the UDP port is fixed (hard-coded as UDP_VLINKTUN in `ana_vlink.h`) but in the future we will allow users to configure this.

A given `vlink` can be configured to send and receive data via multiple interfaces. For example, one can configure a `vlink` to use an Ethernet interface and a GRE or UDP tunnel in order to emulate an Ethernet link with a subset of the nodes connected to the real Ethernet link (i.e. those with a `vlink` configured with the same vlink_id) and with a remote host (via the GRE or UDP tunnel).

Each vlink is identified by a unique 32-bits ID which is manually configured by a human user (see below): when sending a packet, this ID is added immediately after the layer-2 or UDP header and before the ANA data payload. Upon reception of a packet, the vlink brick checks the value of the vlink_id inside the packet against the vlinks configured locally: if there is a match, the payload is sent to the (higher level) brick that is bound to the matching vlink. If there is no match, the packet is discarded. Note that when receiving a packet that contains a valid vlink ID, the vlink brick checks that the data is coming from an interface or tunnel that is configured in the corresponding vlink object.

When (manually) configuring vlinks, one has to ensure that the assigned vlink_ids are locally unique among all the ANA nodes running on the host. This is to prevent unwanted data to be sent to multiple ANA nodes. However, all the ANA nodes that use a given vlink have to "coordinate" to find a vlink_id that is not yet assigned in all the nodes: while this requirement can be manually solved by the human users configuring the vlinks, the future (distributed) testbed management system developed in WP4 should eventually solve this issue in an automatic way.

5.4 The vlconfig command

The 'vlconfig' command permits to configure the vlinks that are configured on a system. In the current version, the command connects to the vlink component via a UDP socket (for both the kernel and userspace versions of the vlink brick). The syntax of the command is very simple:

```
Usage: vlconfig { COMMAND } [ argument(s) ]

where: COMMAND := { help | list | create | delete | up | down |
                    add_if | rem_if | add_udp | rem_udp }
```

where

- 'vlconfig list' : displays all vlinks
- 'vlconfig create vlink_id' : create vlink with the given ID
- 'vlconfig delete vlink_name' : delete vlink with given name
- 'vlconfig up vlink_name' : activate vlink with given name
- 'vlconfig down vlink_name' : de-activate vlink with given name
- 'vlconfig add_if vlink_name if_name' : add (real) interface to vlink
- 'vlconfig rem_if vlink_name if_name' : remove (real) interface from vlink

- `'vlconfig add_udp vlink_name ip:port'` : add remote UDP peer to vlink
- `'vlconfig rem_udp vlink_name ip:port'` : remove remote UDP peer from vlink

The following example creates a vlink that sends and receives data via two interfaces and via a UDP tunnel.

```
'vlconfig list'
'vlconfig create 1234' // returns say, vlink0
'vlconfig add_if vlink0 eth0'
'vlconfig add_if vlink0 eth1'
'vlconfig add_udp vlink0 10.1.2.3:6789'
'vlconfig up vlink0'
'vlconfig list'
```

5.5 The vlink API

An important design point of the `vlink` subsystem is that only one brick can send and receive data via a given `vlink`. The typical scenario is that an Ethernet brick (or another layer 2 brick) registers with a given `vlink` and is granted exclusive access to this resource. This is somehow similar to having an Ethernet network interface card having exclusive access to the cable linking it to a switch or another Ethernet card. In other words, the `vlink` brick "provides the cable" which connects two NICs ("bricks").

Because ANA provides an autonomic environment, the binding between bricks and `vlinks` is dynamic. Typically, each newly created `vlink` is published in the MINMEX key-val repository (KVR) with some well-known keywords such as "vlink". This means interested bricks can periodically check the KVR for resources published with the "vlink" keyword and, upon a match, start the binding procedure to get exclusive access to the `vlink`. The functions used to bind and unbind with a `vlink` are provided "inline" in `ana_vlinkAPI.h`: note that these functions are experimental and the format of messages may (slightly) change in the future.

To bind with a given `vlink`, one has first to obtain the IDP that each `vlink` publishes in the node's key-val repository. Note that this IDP cannot be used to send data: its sole use is to allow a brick to bind with the `vlink`. "Binding" means that the `vlink` will send all received data to the brick bound to it, so only one brick can bind with a given `vlink` at any time. Before binding with a `vlink`, a brick has to register with the MINMEX the callback function (IDP `datarecv`) that will be used for receiving data, and another function (IDP `replyto`) used to treat the reply message sent back during the binding procedure. The primitive used to bind is:

```
int vlinkBind( anaLabel_t vlink_idp, anaLabel_t replyto,
               anaLabel_t datarecv )
```

The function receiving the reply message should call the following function which

1. changes the view of the `datarecv` IDP such that only the `vlink` brick can call it
2. returns the IDP that must be used to send data to the `vlink`
3. sets the `unbind` IDP which can be used later to unbind from the `vlink`
4. saves the virtual MAC address attributed by the vlink in the *vl_mac* variable. Note that the *vl_mac* pointer must point to a 6 byte allocated field.

```
anaLabel_t vlinkBindResponse(void *msg, int len,
                             anaLabel_t dataRecv, anaLabel_t *unbind,
                             unsigned char *vl_mac);
```

To unbind from a given `vlink`, a brick simply has to send its `datarecv` IDP to the `unbind` IDP. The function performing the unbind inside the `vlink` brick indeed checks that the `datarecv` IDP corrresponds to the `vlink`. If the check is successful, the `vlink` object is reset and re-published in the KVR. To be able to reuse it, one has first to re-perform the binding procedure.

5.6 Virtual MAC address

As said previously, the bricks directly using `vlinks` will typically be link-layer bricks like e.g. Ethernet. Since in a sense, a `vlink` "emulates a physical device", each `vlink` has to provide device-specific parameters such as hardware address and MTU. How a brick bound to a given `vlink` can obtain these parameters is still an open issue and has not yet been implemented. One solution is to provide some kind of `ioctl` functions or to provide such information during the binding procedure.

Because the vast majority of link-layer devices are nowadays Ethernet cards, each `vlink` object is configured with a virtual MAC address (vMAC) of length 6 bytes. Note that among all the nodes using a given `vlink`, each vMAC must be unique in order to be able to uniquely identify all destinations. Upon creation of a `vlink` object, the first byte of the vMAC is configured as being the internal (locally unique on the node) index of the `vlink` object (i.e. the number following `vlink` names: `1` for `vlink1`). Hence for `vlink1`, the initial vMAC is set to `01:00:00:00:00:00`. Note that this encoding restricts the number of `vlinks` on a given node as being maximum 255 (which is by far sufficient).

The next 5 bytes of the vMAC of a `vlink` object are then generated when the first real interface or UDP tunnel is configured: in the first case, the bytes are taken from the MAC address of the real interface and in the second case, the remaining 5 bytes are randomly generated. If the first interface configured is the loopback interface, the vMAC is not further configured. Note that while in the two cases the probability that the vMAC is unique is very high, this is not guaranteed but we believe that vMAC collisions are very unlikely to occur. However in the future we might change the way vMACs are generated (and used).

5.7 Example

```
/* For simplicity, the code does not contain any error checking */
/* Read code/functions from bottom */

#include "ana_vlinkAPI.h"
anaLabel_t datarecv;   /* IDP for receiving data */
anaLabel_t sendtoV;   /* IDP to send data via vlink */
anaLabel_t unbind;   /* IDP to unbind with vlink */

unsigned char my_vMAC[6]; /* array storing the allocated virtual MAC */


void do_unbind()
{
   /* call this function to unbind with vlink */
   if (!unbind)
        return;

   anaL0_send(unbind, &datarecv, sizeof(anaLabel_t));
}


void sendto_vlink(void *data, unsigned int len)
{
   /* call this function to send data to vlink */

   if (!sendtoV || !data || !len)
        return;

   anaL0_send(sendtoV, data, len);
}


void handle_bindreply(char *data, int len, anaLabel_t input, void *aux)
{
   /* sendto can now be used to send data via the vlink */
   sendtoV = vlinkBindResponse(data, len, datarecv, &unbind, my_vMAC);
}


void recv_from_vlink(char *data, int len, anaLabel_t input, void *aux)
{
   anaPrint(ANA_NOTICE, "Received %d bytes of data via vlink\n", len);
}


void brick_exit() {

        anaPrint(ANA_DEBUG, "Brick EXIT\n");
}


int brick_start()
{
```

```
        struct service_s service;
        struct context_s context;
        struct timespec timeout;
        anaLabel_t label;
        anaLabel_t bindreply;

        memset(&service, 0, sizeof(struct service_s));
        memset(&context, 0, sizeof(struct context_s));

        anaL2_initDefault();

        context.value = ".";
        context.valueLen = 2;

        service.value = "vlink";
        service.valueLen = strlen("vlink")+1;

        memset(&timeout, 0, sizeof(struct timespec));
        timeout.tv_sec = 5;

        label = anaL2_resolve( NODE_LABEL, &context,
                               &service, 'u', NULL, &timeout);

        /* bind with vlink found in key-val repository */
        datarecv = anaL0_registerCallback(
                               (anaCallback_t)recv_from_vlink,
                               NULL, NULL, NULL, 0, IDP_PERM, 0);

        bindreply = anaL0_registerCallback(
                               (anaCallback_t)handle_bindreply,
                               NULL, NULL, NULL, 0, 0, 0);

        vlinkBind(label, bindreply, datarecv);

        return 1;
}
```

Chapter 6

Generic ANA Threads

6.1 What is it?

The ANA core provides a generic thread library. It provides one and the same application interface for user space and kernel space programming. Up to now, the ANA thread library provides a basic interface to start a thread. There is no advanced facility like it is well known from the posix thread library. Note that kernel threads and userspace threads are different to handle. Therefore the anaThread library differs from the standard posix thread library.

6.2 Why should I use ANA threads?

ANA threads should be used whenever a callback function may block for a "long time". Blocking in a callback function means that no other message can be processed from that brick, which is probably not what you want.

6.3 Files

Starting from the ANA root directory, the directories and files that form the `anaThread` implementation are:

- `C/include/anaThread.h` : header file for `anaThread`.
- `C/shared/anaThread.c` : contains all the code for managing anaThreads.

6.4 How shall I use anaThreads?

Since the ANA threads are already used in the default libraries (AL0, AL2), all the necessary files to use them are included in the default libraries (i.e. you don't have to do anything special).

6.5 The anaThread API

The current API consists of the following functions:

```
unsigned long int anaThread( void * (*start_routine)(void *),
                             void *arg, char *name);

int anaThreadShouldStop(void);
```

`void anaThread(int (*start_routine(void *), void * arg, char * name)`
Starts the function `start_routine` in a separate thread with the argument `arg`.
The *name* argument is useful in case of PLUGIN mode execution of your brick. We strongly encourage you to give significant names to your thread that will help you for debugging in case of problems.

If `name == NULL`, the default name "anaThread" is assigned.
On successs `anaThread()` returns the pid (a non null unsigned integer) of the newly generated thread. On error 0 is returned.

`int anaThreadShouldStop(void)`
Returns 1 if the current thread should stop, 0 otherwise.

6.6 Example

```
int loopingThread(void *data){
        while (!anaThreadShouldStop()){
                //do some processing;
        }
        return 0;
}

static int value;
static int brick_start() {
        value = 0;
        if(anaThread(loopingThread, &value, "thread1") == 0)
                anaPrint(ANA_DEBUG, "Error launching thread 1");
}
```

Chapter 7

Generic ANA timers

7.1 What is it?

Probably all algorithms and protocols in networking need timers. This of course holds for ANA and we quickly realised that the ANA core also needed timers. Actually Linux already provides 2 different implementations of timers in userspace and at least 2 different ways of scheduling defered work in the kernel. So why bother? Well since one objective of ANA is that the same code can be used in userspace and kernel, it is clear that, in order to ease programmers' life, we needed something to abstract the timer internal details of the "running platform" upon which ANA is executed. We hence decided to provide a generic abstraction for handling timers in ANA: how the "real" timer is implemented is hidden and can hence be changed or extended to support new platforms or future changes. Another advantage of having an extra abstraction layer is that it is possible to add new functionalities on-demand without having to modify existing system code.

7.2 Files

Starting from the ANA root directory directory, the directories and files that form the `anatimer` implementation are:

- `C/include/anatimer.h` : header file for `anatimer`.
- `C/shared/anatimer.c` : contains all the code for managing anatimers.

7.3 How shall I use anatimers?

Same as for the ANA threads, ANA timers are already used in the AL0, AL2 libraries, therefore you can directly use the anaTimer API.

7.4 The anatimer API

The current API is very simple and consists of only two initialization functions and three functions for creating, removing, and restarting timers:

```
int anatimer_add(long msec, unsigned char period,
                 unsigned char fix_clock,
                 fct_t action, void *fct_data, int datalen);
int anatimer_del(unsigned int id);
int anatimer_restart(unsigned int id, long msec);
int anatimer_change(unsigned int id, long msec);
```

To add a timer, one must call `anatimer_add` and provide the delay (`msec`) after which the function `action` must be called with `datalen` bytes of arguments. A copy of the function data pointed to by `fct_data` is kept in the anatimer structure (so one can free the initial memory after calling `anatimer_add`). Calling `anatimer_add` returns a positive timer ID if successful or -1 otherwise. The field `period`, when set to ANATIMER_PERIODIC, is used to indicate that the timer should be re-started after it expires; in that case the ID remains the same. The field `fix_clock`, when set to ANATIMER_ABSOLUTE, is used to indicate that a periodic timer should be re-scheduled exactly `msec` milliseconds after the timer was supposed to expire, even if the timer expired with some delay (Linux timers can expire with some delay). When `fix_clock` is set to 0, the periodic timer is re-scheduled `msec` milliseconds after it expired (whatever previous time it was supposed to expire).

To remove a timer, one must simply call `anatimer_del` with the ID of the timer to stop. The function `anatimer_restart` permits to restart the (yet unexpired and still active) timer `id` after `msec` from the current time. The function returns the timer ID upon success or -1. This is similar to deleting the timer `id` and adding a new timer except that with `anatimer_restart` the ID remains the same.

Finally `anatimer_change` allows to change the period of the periodic timer identified by *id*. This function returns the timer ID upon success or -1.

Note that when a timer expires, the function `action` is executed as a separate thread. This ensures that the anatimer subsystem can resume its execution immediately (without having to wait for `action` to return).

7.5 Example

```
void run_once(void)
{
        // do something
        // this runs in a separate thread
        // handled by the anatimer framework
}

void run_periodic(void *data)
{
        anaPrint(ANA_NOTICE, "Data is integer %d\n", *(int *)(data));
}

int brick_start()
{
        int tmp = 123;

        anatimer_add(2500, 0, 0, run_once, NULL, 0);
        anatimer_add(5000, ANATIMER_PERIODIC, ANATIMER_ABSOLUTE,
                     run_periodic, &tmp, sizeof(int));
}
```

Chapter 8

Quick Repository

8.1 What is it?

The quick repository is a tool that we offer to quickly implement a repository in your Brick's code. It is an option that we provide to simplify the programmer's life in case he needs to quickly implement a storage repository accepting boolean queries. A programmer can of course choose to develop his own repository for his brick.

8.2 Files

Starting from the `trunk` directory, the directories and files that form the `Qrep` implementation are:

1. `C/bricks/tools/quickRep/QRepos.c` : implements the quick repository's core functions
2. `C/bricks/tools/quickRep/QREPParser.c` : implements the boolean queries parser
3. `C/include/quickRepository.h` : header file for the quick repository

8.3 How shall I use Quickrep?

To be able to use Quickrep, one simply has to include the `quickRepository.h` header file in his code and add the QRepos.{c,o} and QREPParser{c,o} files in the Makefile of the brick in which Quickrep is to be used. An easy way to add those files to the Makefile is to use the *$(QREP_FILES)* variable as indicated in the Makefile template `'Makefile-user-singlebrick.template'`.

8.4 Entries structure

the Qrep (Quick repository) looks like this:

Name	Value	Key words
example	**void ***	**kw1, kw2, kw3**

Figure 8.1: Quick Repository

Each Qrep entry has a unique name and an unlimited list of keywords with which it can be looked up. Differences with the KVR are the following:

- The most important difference is the type of the values stored in the QREP. Unlike KVR, they can be of any kind (the QREP expects a void * pointer). Therefore the entry values can point to complicated strucutres and are not restricred to IDP labels.
- There is no owner field for the QREP. Since the QREP is intented to general purpose use, we did not include any ownership strategy since this can vary according to the Brick(compartment) using the quickRep. If you need to use such strategies, use a structure as a value with an owner field in it.
- There is no age field since there is no garbage collection strategy (generic purpose use). Same as for the previous bullet, if you need to have a garbage collection strategy, add the needed field to the strucure that you use as a value for the QREP entries.

8.5 The Quickrep API

Internally, the Qrep has two management hash tables, one for storing the entries and the other to associate keywords to entries. The sizes of the two tables are tuneable at initialization and may influence on the performance of the Qrep.

Repository initialization : First of all, a QRep is of the following type (defined in `quickRepository.h`)

```
struct QREP exampleQrep;
```

This structure contains pointers to the entries table and keyword table mentionned above. Those tables need to be initialized i.e allocated. To initialize the repository use the function :

```
int initQRepository(   struct QREP *rep, int hashMaskEntryTab,
                       int hashMaskKeyTab);
```

Where:

- *rep* points to the *struct QREP* to be initialized.
- *hashMaskEntryTab* will define the size of the entries hash table. This size is $2^{hashMaskEntryTab}$.
- *hashMaskKeyTab* will define the size of the keywords to entries mapping hash table. This size is $2^{hashMaskKeyTab}$.

Please note that the two hashTables have a hash collision management mechanism using chained lists. This implies that even if the size of the hash tables is choosed to be small, we still can store an unlimited number of entries in the QRep. However, a small hash table size implies low performance since we would be "travelling" through huge chained lists.

Repository destruction :

```
void freeQRepository(   struct QREP *exampleRep,
                        freeValFunction_t freeValFct);
```

This function frees the quickREP, i,e the two allocated hash tables, and the entries they contain. Since the values stored can be in some cases structures, the user must provide the function to free them correctly. This function must take only one *void* $*$ argument that is the pointer to the Entry's value. This function $freeValFct$ will be applied to each value member of every QREP entry in the *exampleRep* QRep. The $freeQRepository$ function will then free the two allocated hash Tables.

Repository purge :

```
void purgeQRepository(  struct QREP *exampleRep,
                        freeValFunction_t freeValFct);
```

This function does the same as $freeQRepository$ except that is does not deallocate the hash tables. It just frees all the entries they contain. We provide this function in case a Brick wants to periodically delete all entries as a data refresh strategy.

Adding Qrep entries : To add Qrep entries, first you need to allocate them using :

```
struct QREPEntry *allocateQREPEntry(    char *name, int nameLen,
                                        void *value, int valueSize);
```

Where:

- *name* is the unique name for the Qrep entry. The character string will be **copied** to the QRep entry structure, i.e you can later on free the *name* pointer without affecting the Qrep entry
- *nameLen* is the size in bytes of the entry name
- *value* points to the value associated to this entry. The pointed value is also **copied** to the QRep entry structure, i.e you can free the *value* pointer without affecting the Qrep entry
- *valuesize* is the size in bytes of the memory slot pointed by *value*. This is needed since we do a copy of the the value to store

This function returns an allocated *struct QREPEntry* containnig a copy of the names and value passed as arguments. A NULL pointer is returned in case of error.

Before adding the entry to the QREP, we need to associate a set of keywords to it that would be used as a way to look the entry up in the QREP. Please note again that this has to be done **before** adding the entry to the QREP that we will show later.

To associate a keyword to an entry use the function

```
int addKeywordToEntry(struct QREPEntry *entry, char *keyWord);
```

Where:

- *entry* is a previously allocated QREPEntry to which we want to associate the keyword
- *keyWord* is the character string of the keyword

In case you have multiple keywords to associate to the entry, use the *addKeywordToEntry* function in a loop. This function returns 0 on success and -1 in case error occurred.

Now that we have a full QREP entry (name+value+keywords), we add it to the repository using:

```
int addQREPEntry(struct QREP *rep, struct QREPEntry *entry);
```

Where:

- *rep* is the Qrep to add the entry to.

- *entry* is the QRep entry to add.

This function returns 0 on success and -1 if error.

Removal of a single entry: To correctly remove an entry from the QREP (i.e with updates of the keywords to entry mapping) and free the *QREPEntry* structure use:

```
int freeQREPEntry(struct QREP *rep, struct QREPEntry *entry,
                  freeValFunction_t freeValFct);
```

Where:

- *rep* is the Qrep to remove the entry from.
- *entry* is the QRep entry to free.
- $freeValFct$ points to the function to correctly free the entry's value.

QREP querying: There are two ways extract entries from the QREP. If you know the unique name of the entry you're interested in, use:

```
struct QREPEntry * getQREPEntryByName(struct QREP *rep, char *name);
```

On success (found the entry), this function returns a pointer to the entry in the QREP. To access the value, of this entry, use the field *value* of the *QREPEntry* strucutre. For example, if you stored the returned pointer in a variable called *returnedEntry*, the value would be acessible using $returnedEntry->value$. On error (entry not found), the function returns a NULL pointer.

Another way to get entries from the QREP is to use the boolean queries with this function:

```
int QREPfindByQuery(struct QREP *rep, char *query,
                    struct QREPKeyObjectList **res);
```

Where:

- *rep* is the target QREP
- *query* is the character string of the boolean query
- *res* is a pointer to the pointer that will contain the matching entrie's list

This function returns 0 on success and -1 if error (bad query). The boolean query strings are expressed in a similar way to those of the KVR with the same operators. On success, the *res* pointer now points to the list of entries in the KVR matching this query.

The *struct QREPKeyObjectList* defined in `quickRepository.h` simply a chained list of QREP entry pointers as you can see below:

```
struct QREPKeyObjectList{
        struct QREPEntry *entry;
        struct QREPKeyObjectList *next;
};
```

Walk through this chained list using the *next* pointer (untill hitting a NULL pointer) and you can access all QREP entries.
Important: Once you are done with the results of the query, do not forget to free the chain of results, i.e the chained list of *QREPKeyObjectList*. To do so , you can use the following function:

```
void QREPfreeQueryRes(struct QREPKeyObjectList *list);
```

this function does not free any entry in the QREP (in fact it receives no pointer to a QREP), it just frees the chained list of structres used to store pointers to query matching QREP entries.

Apply a function to all entries values: This is a miscellanous functionnality we provide in case you need to frequently apply a function to all the QREP entries you stored. This might be useful for example in case you want to implement a garbage collection strategy and need to check the "age" of each entry. To do so use the function:

```
void applyToAllQREPEntries(struct QREP *rep,
                           valueFunction_t valueFunction, void *aux )
```

Where:

- *rep* is the target QREP
- *valueFunction* is the function to apply to all Entries. To be called correctly, this function needs to be of the prototype :

  ```
  typedef void (*valueFunction_t) (void *value, void *aux);
  ```

  Where *value* would be the pointer to the Entry and *aux* an auxillary argument.
- *aux* is the auxillary argument that will be passed to the called function *valueFunction*

Chapter 9

Generic ANA Locks

9.1 What is it?

Since callback functions, threads and timers provide many opportunities for concurrent data manipulation, the ANA-core provides a generic locking mechanism. It is important to protect all data structures which may be manipulated by more than one function. E.g. one function may add an element to a list, whereas another function may delete one. Without locks, this situation may lead to a race condition, which could end in illegal memory access or in a destroyed list.

9.2 Files

ANA locks are implemented in a single header file:

- `C/include/analock.h` : Implementation of ANA locks

9.3 How shall I use ANA locks?

To be able to use ANA locks, one simply has to include the `analock.h` header file in its code.

9.4 The ANA lock API

The current API consists of the following functions:

```
int analock_init(analock_t *mylock);

int analock_lock(analock_t *mylock);

int analock_unlock(analock_t *mylock);
```

`int analock_init(analock_t *mylock)`
Initializes the lock `mylock`. To be called before the lock `mylock` is used the first time.
returns 0 on success, -1 otherwise;

`int analock_lock(analock_t *mylock)`
analock_lock blocks until it is safe to enter the critical section.
returns 0 on success, -1 otherwise (e.g. if the lock was not initilized).

`int analock_unlock(analock_t *mylock)`
analock_unlock has to be called upon leaving a critical section.
returns 0 on success, -1 otherwise (e.g if the lock was not initialized).

9.5 Example

```
analock_t lock;

static int brick_start(struct anaMinmexSpecs_s *myminmex) {
        analock_init(&lock);
        analock_lock(&lock);
        //critical section
        analock_unlock(&lock);
}
```

Chapter 10

Miscellanous functions of the API

10.1 Wrapper functions

All the functions defined in this section can be found in the files `C/include/anaCommon.h`, `C/include/gatesCommon.h` ,`C/shared/gatesCommon.c`, `bricks/API/AL0/AL0GatesInternal.c/h` and `C/shared/anaCommon.c`. They are used by both the MINMEX and the bricks.

Because we want to offer to the programmer the possibility to develop a code that is indifferent to the execution mode (userspace or kernel), we provide a set of wrapper functions to some of the most basic C functions. Also because most programmers are more familiar with userspace programming, we provide the necessary code to support userspace functions for kernel compilation. For example, one can simply use the `malloc` function in a brick: shall it be compiled for the kernel, the function `malloc` will be replaced by its kernel equivalent (`kmalloc`). The functions currently supported are: `malloc, free, random, atoi, inet_addr, sleep, gettimeofday`.

In addition, some more complex primitives are currently wrapped in functions that have the 'Wrapper' suffix in their names. These functions are mostly used for the gates communication between the gates plugin and the bricks. Note that bricks typically use the ANA API to communicate so one should not directly use the standard socket API in ANA.

- `int sockCreateWrapper(void *toReturnSock, int domain, int type, int protocol);`

  Wrapper to *socket* function in user space and *sock_create* in kernel mode. *toReturnSock* is the pointer to the socket descriptor to be filled (for userspace, this would a *int* $*$ pointer to the integer where to store the descriptor of the new socket), i.e after calling this function the newly created socket is accessible with the descriptor *toReturnSock*. The returned integer is just an error code (not a result descriptor). The function returns 0 on success and -1 in case of error (bad arguments);

- `void closeWrapper(void * sock);`

  Wrapper to userspace *close* function. Due to the fact that in kernel mode this function calls *sock_release* function, it should be used only to close sockets.

- `int bindWrapper(void * sock, struct sockaddr * address, int size);`

  Wrapper to userspace *bind* function. Assigns a name to a socket. For user space, on success, zero is returned , on error, -1 and errno is set ppropriately.

- `int setSockOptWrapper(void *sock, int level,int op, char * optval, int optlen);`

  Wrapper to userspace *setsockopt* function and kernel *sock_setsockopt*. Manipulates the options associated with a socket. Returns the called native function error code.

- `int readWrapper(void *sock, char * data, int len);`

  Wrapper to the function *read* in user space and *sock_recvmsg* for kernel. Attempts to read up to *len* bytes from socket *sock* into the buffer starting at *data*. Returns the effetive number of read bytes.

- `anaPrint(level, fmt, args...);`

  Wrapper to the *printf* and *printk* functions. The level indicates a debugging level you choose for your message. This would decide wether your message gets printed or not according to the debugging level your code will be runned in. The debug levels are the following :

  1. *ANA_EMERG* : emergency, something really bad happend
  2. *ANA_ERR* : error condition e.g system drops a packet, a lookup fails
  3. *ANA_NOTICE* : normal situation, worthy to note
  4. *ANA_DEBUG* : debuging information

  The other *anaPrint* argument are similar to *printf*, first the format string then the needed variables according to the format.