TITLE	IDENTIFICATION
Antidote: Program Guide	Documentation for developers of applications based on Antidote IEEE 11073 library

DOCUMENT HISTORY

VERSION	COMMENTS	REALIZED BY	DATE
1.0	Version 1.0	Adrian Livio Aldenor Martins Diego Bezerra Elvis Pfützenreuter Fabrício Silva José Martins José Nascimento Marcos Fábio Mateus Lima Raul Herbster Walter Guerra	20/Apr/2011
2.0	Updated to Antidote 2.0	Elvis Pfützenreuter	26/Feb/2012
2.1	Content review and organization	Hyggo Almeida	05/Mar/2012
2.11	Solved some formatting issues	Elvis Pfützenreuter	07/Mar/2012
2.12	Fixed hyperlink Sync with final 2.0 version of Antidote	Elvis Pfützenreuter	21/Mar/2012

Contents

About this document

This document is a program guide for Antidote, an implementation of the IEEE 11073-20601 standard. As part of the SigHealth Platform, a connected health solution developed by Signove, Antidote provides the IEEE 11073 stack, plus developer-friendly access to IEEE 11073 features.

Who should read?

Developers of connected health applications should read this document to understand how to use the IEEE 11073 stack and how to write communication and transcoding plug-ins for Antidote.

Related Docs

The reader must refer to the following documents for a detailed description of the SigHealth platform and its components.

White Paper

- Signove - SigHealth Platform: A Connected Health Solution

Reference Documents

- SigBox Software: Reference Guide
- SigHealth Server: Reference Guide
- Care Program Applications: Reference Guide

Program Guides

- *Antidote: Program Guide*
- Qt Health Manager: Program Guide
- Java Script Health API: Program Guide
- SigHealth WS: Program Guide for CPA Developers
- SigHealth WS: Program Guide for SigBox Software Developers

1 SigHealth Platform

SigHealth is a platform for remote patient health monitoring and data management. It enables patient homecare by using wireless Personal Health Devices and seamless, automatic data synchronization with a safe and secure Internet server backend. The server provides information and services for health professionals, patient and family, allowing tracking health conditions and evolution. These services include alarms on critical situations, definition of targets to be achieved by patients, and graphical visualization of patient information, among others.

SigHealth is a completely customizable platform, allowing creating specific health and wellness programs, bringing together monitoring services and useful content related with specific groups of people. It is compatible with Continua Alliance[1] specifications, SBIS[3] (Brazilian Health Informatics Society) specifications, and existing Health Records solutions (Google Health[2], Microsoft Health Vault[3]). For a complete description of the SigHealth platform and their main components, the reader must refer to the white paper *SigHealth Platform: a Connected Health Solution*.

From the infrastructure point of view, SigHealth architecture has four main hardware components: Personal Health Devices, SigBox hubs, Internet server backend, and client devices. The general solution is based on gathering patient health data using Personal Health Devices; securely and automatically synchronizing the data with an Internet server backend through a SigBox hub; and providing information and services for health professionals, patient and family, anytime, anywhere, using the Internet.

As a software company, Signove provides software solutions for all SigHealth hardware components, establishing partnerships with hardware manufacturers to embed those solutions in their platforms, providing a complete connected health platform for customers. The software components available for customers and partners are:

- **Antidote:** Software stack running on Personal Health Devices and SigBox.
- **Health Manager:** Language-specific module for developing SigBox software.
- **JavaScript Health API:** JavaScript API for developing SigBox applications.
- **SigBox Software:** Reference implementation for SigBox.
- **SigHealth Server:** Server software running on the Internet server backend.
- **Care Program Applications:** Domain-specific applications accessed by client devices.

2 An IEEE 11073-20601 primer

The developer does not need to know the IEEE 11073-20601 standard[4] (from now on referred as “IEEE 11073”) to use Antidote productively.

On the other hand, it is highly desirable to have at least a basic understanding of the 11073 standard, in order to visualize how Antidote works and see the reasoning behind Antidote’s architectural and design decisions.

This section is not an exhaustive guide about 11073; the authoritative sources are the standard documents, and Continua Alliance makes excellent tutorials available.

2.1 Agents and Managers

There are two device types: agents and managers. Agents are data producers, typically sensor devices. Managers are the data collectors.

Connection may happen in either direction, but the agent normally takes the initiative because it “knows” when new data is available e.g. when a patient takes a blood pressure measurement.

IEEE 11073 is based upon the idea that agents, being sensor devices, are low-powered and have few processing resources, while the manager is typically a powerful device connected on mains. So, most of the burden of 11073 is put on managers.

Antidote has complete support for Manager applications and basic support for Agent applications. This is due to the fact that Antidote most common use case is the Manager role.

2.2 Communication model

IEEE 11073 is transport-agnostic, meaning that it can be carried by almost any packet-based technology, such as TCP/IP, Bluetooth, USB, etc. There are some formal requirements on transport layer, but they are fulfilled by any technology in current use.

IEEE 11073 fills the top three layers of OSI model for networking: presentation, session and application. Presentation is provided by MDER (see next item). Session is provided by communication model. Application is where useful data finally flows from agent to manager.

The IEEE 11073 session layer supports session persistence even upon disconnection at transport level. Transport may disconnect due to a temporary failure or to save power on inactivity. At IEEE 11073 level, an agent and a manager with a session established among them are said to be **associated**.

Since session persistence is optional, in practice many devices choose to terminate the session as soon as transport disconnects, a behavior that is much simpler to implement.

2.3 Bytes on the wire: ASN.1 and MDER

The IEEE 11073 protocol specification is written using the ASN.1 language. Actual encoding of an ASN.1 message to a stream of bytes is dictated by a set of rules, like BER, XER or MDER.

In case of health devices, MDER is the common-denominator rule, that every health agent and manager is expected to implement. MDER is optimized in order to accommodate low-end devices. For example, MDER makes it easy to store message templates pre-encoded on firmware, on which the device just replaces the variable octets and sends it out.

MDER supports a subset of ASN.1 primitive types, which are combined to form a well-defined and fixed set of Personal Health Device (PHD) types. Every 11073 message is ultimately a collection of PHD type elements, which in turn are based on a few ASN.1 primitive types.

2.4 APDUs

APDUs (Application Protocol Data Units) are the “packets” of IEEE 11073 communication. They can be as big as 65535 octets, even though they are typically much smaller, and maximum size (MTU) is further restricted by device specializations.

APDUs are analogous to network packets but an APDU can be much bigger (e.g. Ethernet has MTU of just 1500 octets). The transport layer must support someway to fragment and reassemble APDUs. Bluetooth’s L2CAP can do that.

TCP does not preserve message boundaries, but reassembly is possible by interpreting APDU length (third and fourth octets in header) and waiting for that amount of data before sending it to IEEE 11073 stack.

2.5 Boxes inside boxes inside boxes

IEEE 11073 defines numerous PHD types. Each type is quite simple by itself, but it may contain a list of children elements, which in turn contain their own children, and so on.

From APDU level to the actual “interesting” data, there may be half dozen encapsulations, and an 80-byte APDU is employed to send a mere 8-byte measurement. APDU decoding is heavily based on “choices”, that is, a length/choice value pair followed by a sequence of octets, more or less like the textbook TLV (type, length, value) structure.

Unknown choices may simply be skipped, because the length is known even if the sequence is opaque to the receiver.

Another recurring pattern is the attribute list. Each attribute is a pair of object identifier code (OID) and an octet string, whose format is defined by the OID. There are hundreds of possible OIDs defined on 10101[5] but fortunately each particular usage of AttributeList restricts the set of permissible OIDs to a handful. Attributes with unexpected OIDs for the situation may simply be skipped, allowing graceful degradation.

This characteristic of 11073 is also due to the fact that it borrows heavily from other standards (some of them quite old).

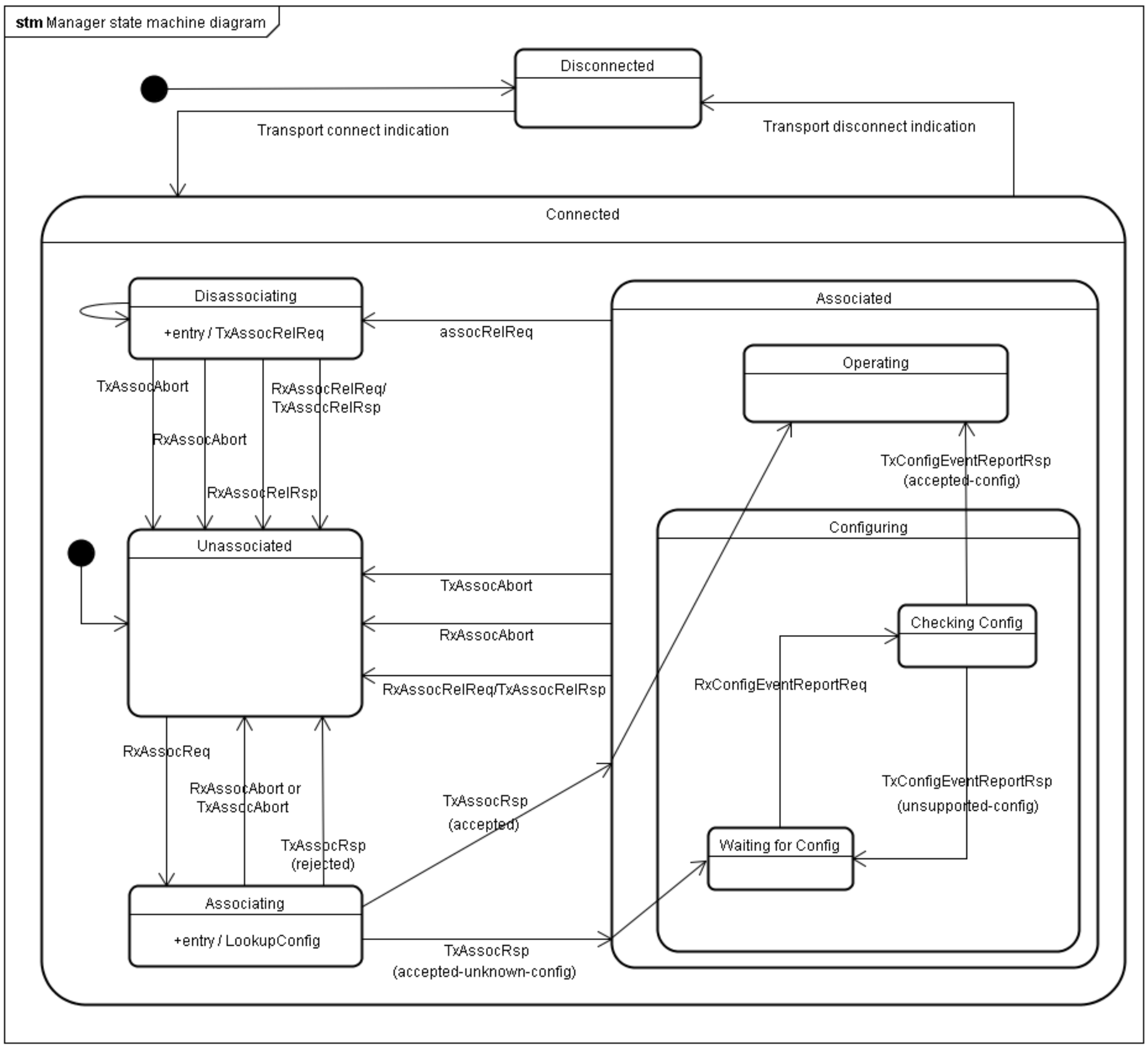
In Antidote, the developer only needs to have contact with these details if he is going to write new specializations or transcoding plug-ins. In most cases, he can build upon existing examples, and 11073 specializations supply Annexes with very handy examples.

2.6 State machine

Like most protocols, IEEE 11073 session layer is governed by a state machine. The basic states for the manager are:

- Disconnected
- Connected
  - Associating
  - Associated
    - Waiting for configuration
    - Checking configuration
    - Operating
  - Disassociating
  - Unassociated

As the indentation suggests, some states imply others. A device must be associated in order to be operational. The hierarchy is more didactical than true because, as mentioned before, associations may persist even upon transport disconnection. A more realistic state machine map can be found in the figure below.



2.7 APDU types

Considering the top-level APDU types, there are not many of them:

- AARQ: Association Request
- AARE: Association Response
- RLRQ: Association Release Request
- RLRE: Association Release Response
- ABRT: Association abort
- PRST: Presentation APDU
  - ROIV: Remote Invoke
  - RORS: Remote Invoke Response
  - ROER: Remote Invoke Error
  - RORJ: Remote Invoke Reject

Most types are concerned with association maintenance. ROIV is the workhorse that carries device configurations and actual medical data.

In Antidote, the developer normally does not need to know or care about APDU types and their internal structure.

2.8 Service model

ROIV APDUs carry a few basic commands: Get, Set, Action and Event Reports.

Each command admits confirmed and non-confirmed variants. Confirmed requests, as the name says, expect the target to send a confirmation APDU. If it does not happen in time, the request APDU is resent a number of times, and if a confirmation still does not arrive, the association is aborted.

Each request is labeled by an invoke-id, a simple integer that is used to correlate requests with responses. The invoke-id is also important in requests retransmissions: the target is expected to handle same-id requests as idempotent. (If two requests with invoke-id 0x1234 ask a robotic arm to move 45 degrees, the final result should be 45 degrees movement, not 90).

All communication between two associated devices is an exchange of requests, responses, events and confirmations.

2.9 DIM: Domain Information Model

DIM defines an idealized structure of data inside the agent device, which is mirrored at manager to describe it.

It is a kind of “object-oriented” structure, meaning that “classes” define it. The classes may be “extended”. Each class has a set of permissible attributes. Depending on class, some attributes are mandatory, others optional, and others forbidden and so on.

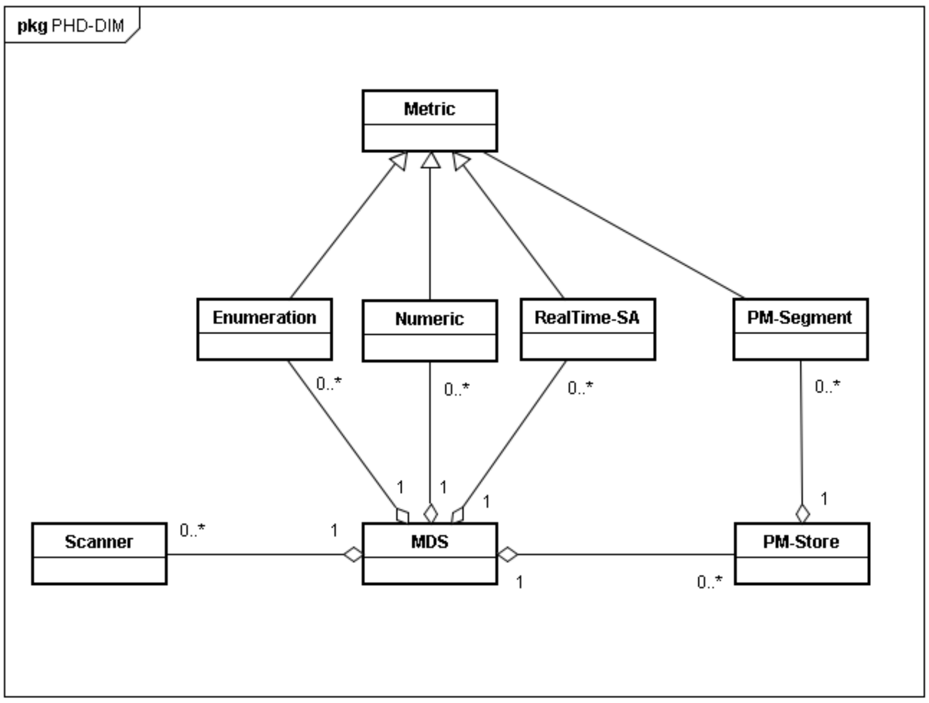
A real agent device has a number of objects that instantiate such “classes”: each object with a number of attributes. Each agent contains exactly one MDS (Medical Device System) object. MDS contains attributes like System ID, manufacturer information, specialization etc.

MDS can have a number of “children” objects, which will contain the actually interesting data. These objects may instantiate the following classes:

- Metric: an abstract class for several classes like Numeric and Enumeration.

- Numeric: as the name implies, it contains numeric data. It may be one number, of a small array of numbers. This is by far the most common in currently available agents.
- RT-SA: waveform measurements, in the form of arrays of numbers.
- Enumeration: status or annotation information. For example, the generic status of a patient when measurement is taken (1="healthy", 2="ill", 3="stressed", 4="menses").
- PM-Store: Storage of past measurements (for example, an oximeter connected to a patient while he sleeps, and night data is uploaded afterwards). Actual data is stored in PM-Segment children.
- PM-Segment: Chunks of stored measurement data, belonging to a PM-Store.
- Scanner: Configurable object that monitors and/or summarizes other objects, and emits events when configured conditions are met.

The UML diagram below shows the relationship between DIM classes.



Each object has a handle, which allows relating APDU messages with the respective MDS objects. MDS attributes can be retrieved using *Get* and sometimes written using *Set*.

The MDS "tree" is not transmitted in full over APDUs. The manager discovers the agent's MDS in several phases, mainly via configuration and measurement reports.

A few crucial MDS attributes are sent at association phase. The Manager as needed must request the rest of MDS attributes, as well as children object's attributes.

2.10 Configuration and Event reports

After association establishment, but before operating, the agent sends its Configuration to the manager (in an *Event Report* APDU). The configuration specifies how many child objects exist in MDS, their types, handles, units etc.

When the value of a MDS object is changed (e.g. the Pulse and Oximetry numeric objects of an oximeter are updated by sensor data), another *Event Report* with measurement data is sent to the Manager (this is when our application finally gets some sensor data, which is the whole point of 11073).

The event report packet cannot be interpreted by itself. It is a packed attribute list that depends on agent's configuration to be decoded (that's why the manager must discover configuration before entering operating state).

These mechanisms provide prompt extensibility. The manager does not need to have any preconceptions about agent's MDS; it learns everything at configuration phase.

As a matter of fact, Antidote sends "augmented" measurement reports to the client application, that is, with all ancillary information bundled. This spares the application from interpreting the configuration just to discover e.g. that unit is kg.

There are two types of configurations: standard and extended. The standard configurations are set in stone by specialization documents (see next section for more details). A manager may know some standard configurations in advance, so it does not need to learn it from agent upon association.

An extended configuration may have any format, so the manager must learn it from the device. Once it learns, it may save it locally, skipping this phase in future associations. The cached configuration is only valid for that particular device. If another device comes along, even being the same manufacturer and model, the manager has to learn again.

Standard configurations have uniform codes across all devices (e.g. 0x0190 is a standard configuration for oximeter). Extended configurations have codes above 0x4000, and they can't be correlated across devices (meaning that, if device A has *config-id* 0x5123 and device B has 0x5123 too, the manager can't assume they share the same configuration).

2.11 Specializations

Apart from the base IEEE 11073-20601 document, there are additional documents that define device specializations. Every specialization defines which objects are mandatory in agent's MDS, and what they mean.

For example, 11703-10404 is the oximeter specialization. It says that an oximeter must have at least two Numeric objects, one for oximetry (whose unit is %) and one for pulse (whose unit is bpm). The manufacturer is allowed to add more objects beyond the required two.

This guarantees that, even in face of flexible configurations and variable-format event reports, an oximeter will always produce that two crucial pieces of information, and the manager can count on them.

The IEEEE 11073-10404 document also specifies how PM-Store is laid out. PM-Store is optional in this specialization, but if an oximeter does implement storage, it must follow the spec.

In Antidote, four specializations are bundled in standard release. It is easy to add more, provided that developer has the specification document in hand. Note that Antidote can talk with devices whose specialization is unknown. The only difference is that configuration needs to be learnt upon the first association, and that happens automatically.

Each specialization document specifies one or more standard configurations. This does not mean that those are “preferred” configurations. They are rather bare-minimum, last-resort configurations. For example, none of oximeter standard configurations have a PM-Store, yet a good oximeter is expected to store measurements, so it must have a PM-Store and must use an extended configuration.

Finally, the specialization specifies the maximum transmission and reception sizes for APDUs, which allows building sensors with less memory, therefore simpler, cheaper and with longer battery life.

2.12 PM-Store

The PM-Store object is a bit complex, so it deserves a subsection of its own.

The PM-Store object does not contain stored data by itself. It is a top-level element, like MDS is for all objects. It does contain high-level attributes like operational state and sampling period.

Each PM-Store has a unique handle, like all MDS objects have. The manager mentions this handle to make requests to the store.

Each PM-Store has a variable number of PM-Segment children. They are essentially packed data arrays.

An agent may have more than one PM-Store in its MDS. This is useful when the agent stores data in different fashions (e.g. periodic x aperiodic measurements), or data from different sensors that don't work in sync.

The Manager "discovers" how many PM-Stores an Agent has, and their handles, via device configuration (the same way the other MDS objects like Numeric are discovered). This implies that the number of PM-Stores in device tends to be stable; otherwise the configuration would have to change.

Each PM-Segment belongs to a PM-Store and has a unique instance number, so the tuple (handle, instance) addresses uniquely a PM-Segment. Each segment has attributes, including its data format (PM-Segment-Entry-Map).

Data itself is sent in "blob" format to the manager. The manager must fetch PM-Segment-Entry-Map in advance in order to decode the blob.

Antidote performs this automatically. The PM-Segment data received by application is already decoded and labeled.

PM-Segments objects have no methods. All access is done through PM-Store. As stated before, PM-Store handles are discovered via configuration. Methods are:

- Clear-Segments: clears a range (or all) segments belonging to a store.
- Get-Segment-Info: gets information about a segment, including its instance numbers and data formats.
- Trig-Segment-Data-Xfer: request the transmission of a given segment.
- Segment-Data-Event: "callback" with requested segment's data blob. If segment data does not fit in one APDU, several events are sent back until completion.

Antidote PM-Store API is a simplified version of the PM-Store method set, but follows similar principles. First the user gets a list of segments, and then requests each segment's data. And data comes asynchronously.

2.13 Continua Alliance

Continua Alliance is a non-profit association that promotes the standardization of personal health devices, envisioning a market of standard, affordable and readily connectable sensors.

Continua chose IEEE 11073 as the main protocol stack, and also selected a number of transport technologies, e.g. Bluetooth (BT), Bluetooth Low Energy (BLE) and USB. These are technologies supported by almost every personal computer, so the user does not need to buy any kind of adapter, etc.

Moreover, a Continua-compliant manager (typically a software running on PC) has guaranteed compatibility to every Continua-certified device.

2.14 Transcoding

Not every Continua device implements IEEE 11073. Bluetooth Low Energy devices have very low power requirements, for which IEEE 11073 is inappropriate; and BLE model deviates quite a bit from the typical packet-based transport. ZigBee devices fall in the same situation.

Still, it is desirable to keep managers simple, which implies not having to support other protocols besides IEEE 11073. For those cases, a transcoding scheme[6] has been developed. The native device protocol is translated to IEEE 11073 messages by intermediate software, kind of a proxy that does the role of an Agent.

In the case of BLE devices, transcoding is actually a Continua standard, and at least the BLE devices still follow an open standard. Every BLE thermometer ‘talks’ the same protocol, so a single transcoder can handle all of them.

The idea can be extended to non-Continua devices, which employ non-standard and/or proprietary protocols. In that case, it takes a different transcoder for every device model. It is less convenient, but still useful because it spares the Manager software from non-11073 protocols.

Antidote supports transcoding via a plug-in architecture. Any developer can write new transcoding plug-ins. A very simple oximeter transcoding is bundled.

2.15 Bluetooth and HDP

Most Continua devices available at the market use HDP as transport. HDP is a Bluetooth profile for health devices.

In theory, HDP is application-neutral, but the only protocol currently supported is IEEE 11073, and HDP architecture is clearly aimed towards IEEE 11073 needs. HDP goes as far as having persistent channels (that survive link-layer disconnection), which goes hand in hand with IEEE 11073’s persistent sessions.

HDP has data Sources (related to Agents) and Sinks (related to Managers), and data specializations for each source and sink. Two HDP devices can establish a channel only if and only if they support the same specialization and are in opposite roles (one is source, the other is sink).

There are no "wildcard" specializations. The Manager must register all HDP specializations that it supports (even though IEEE 11073 allows a Manager to support unforeseen specializations).

All this information is added to SDP records, so a simple SDP query determines device role and specializations, all of that without the need of establishing an IEEE 11073 association.

Antidote is bundled with an HDP communication plug-in for Linux. In Android, HDP support is provided at Java level, and Antidote inherits it indirectly via Android plugin. HDP support depends on ERTM support, which few Bluetooth stacks have. Linux supports ERTM since kernel 2.6.36. Running a recent kernel (or backporting ERTM) is a prerequisite for HDP support.

2.16 USB and PHDC

Like Bluetooth has the HDP profile, USB has PHDC (Personal Health Device Class). The objective is the same as HDP: to provide suitable transport channels for IEEE 11073.

There is currently no PHDC driver in Linux Kernel. Antidote is bundled with a PHDC communication plug-in for Linux. It employs libusb to drive devices at user-level.

3 Antidote Architecture

This section is a high-level description of Antidote architecture. It is important to understand the general layout of source code, and to see the reasoning behind design decisions.

3.1 Design goals

Antidote was developed with very specific goals in mind. The most important goal is portability. This implies C language and C standard library.

Antidote has as few external dependencies as possible. Dependencies hurt portability. It is not tied to any particular event loop framework e.g. GLib or Qt or threads. This freedom makes Antidote easy to integrate with any application. Antidote will run on whatever event loop is employed by the application.

For example, Antidote implements its own linked list. While it could use Glib's GSList, that would prevent portability for platforms that don't support Glib, or would force the developer to port Glib too (not an easy task).

Some plug-ins do use GSList as well as other Glib features, but they already depend on Glib for some other reason. For example, the Linux HDP plug-in depends on Glib and D-Bus, because the HDP API offered by BlueZ is based on D-Bus.

Dependence to a particular platform is concentrated on transport plug-ins. Almost any transport protocol may be used, such as Bluetooth, USB, TCP/IP, pipes, sockets etc.

Antidote itself is a library, free of platform dependencies by itself. The platform dependency is delegated to the executable, when Antidote is linked with application and the desired communication plug-ins.

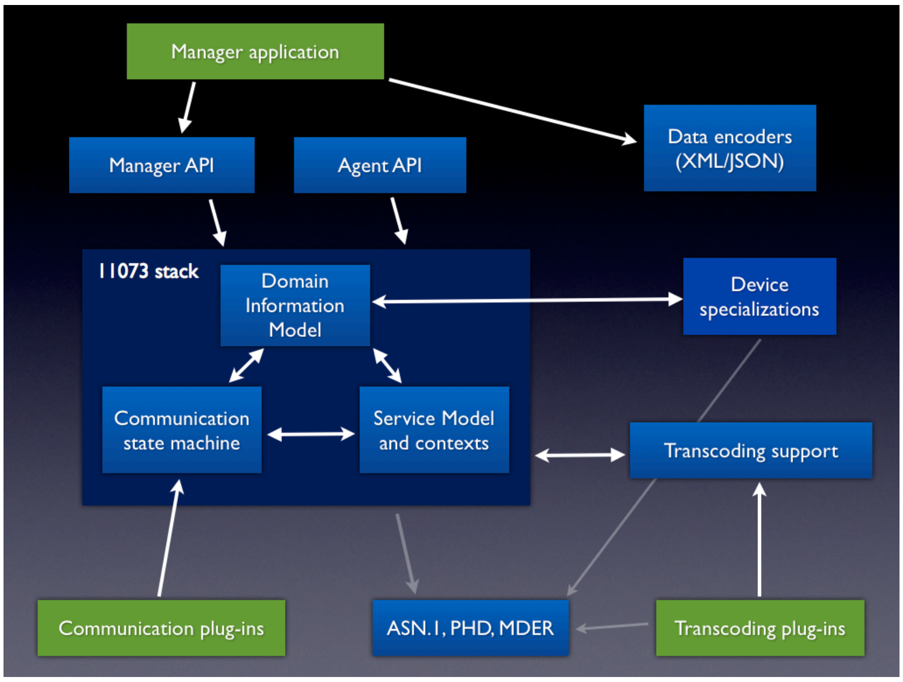
Antidote can work with asynchronous frameworks. It does not create threads, does not block waiting for data and does not spinlock. In short, it does not do anything that might hurt application's responsiveness.

Antidote praises modularity and endeavors to make each component independent from the others. This allows e.g. reusing some part like DIM or ASN.1 or service model even in a situation where the other parts are supplied by other products.

Another all-important goal is to insulate the client from IEEE 11073 complexities. This is explored in the next topic.

3.2 Layers and components

Antidote's internal organization follows loosely the IEEE 11073 layering of service model, communication model and DIM (figure below).



Data encoders and Transcoding do not belong to proper IEEE 11073. They are explained in other topics of this section.

3.3 Data encoding API

Antidote never forces the client application to deal directly with MDS types. It encapsulates every piece of data (measurements, configuration, MDS attributes, etc.) on *DataList* structures. They are easy to transverse and interpret.

Normally, the manager will be a service process separated from application UI. Each *DataList* is sent via IPC, and needs to be encoded. Anticipating this need, Antidote offers XML and JSON encoders. Those formats have decoding facilities in every major language and framework.

On the other hand, if the manager wants/needs to access MDS directly, there are no barriers (beyond the sheer complexity of IEEE 11073).

3.4 Development practices

Antidote is open-source, and development itself follows an open model. Source code can be found in the following Git repositories:

https://gitorious.org/antidote (project page in Gitorious)

https://gitorious.org/antidote/antidote (core Antidote)

https://gitorious.org/antidote/android-sample (sample Android service)

https://gitorious.org/antidote/antidote-sample-client (Android test client for service)

The one-stop project page, which contains all documentation, tarballs, mailing list address, etc., is:

http://oss.signove.com/index.php/Antidote:_IEEE_11073-20601_stack

Antidote has extensive unit-testing support. (If you are curious, see the *test.sh* script at topmost folder). It is regularly tested and "Valgrinded" for leaks.

Antidote has been tested with a wide range of real-world devices, and it is employed in commercial products.

Code is almost fully documented with Doxygen. Coding style follows K&R and Linux styles, except by line lengths (no point in limiting lines to 80 characters if any screen can show 110-column terminals, with big characters.)

3.5 Contexts and service requests

The single most important structure in Antidote is the communication context *(typedef Context)*. It represents a communication channel with a device. All device-related information is a direct or indirect attribute of *Context,* including MDS tree, pending requests and pending timeouts.

Most APIs pass a *ContextId* instead of *Context* Itself. A *ContextId* is passed by value and can be safely stored for future reference, while a *Context* can be released and any pointers to it become dangling.

ContextId is a tuple of two values: communication plug-in ID and connection ID (whose namespace is the plug-in). Generally, *ContextId* should be handled as opaque, but occasionally the constituent values are of interest to developer, in debugging sessions or when developing new plug-ins.

Another very important structure is the *Request*. As the name suggests, it contains the details of a request, and is returned to requestor when request is fulfilled (or failed).

3.6 Asynchronous operation

Antidote endeavors to meet two design goals that tend to conflict: framework neutrality and asynchronous (non-blocking) operation.

Some libraries achieve asynchronous operation either by using threads massively or by having its own event loop. Antidote does neither, and still can be asynchronous. However, there is a price to pay:

- The communication plug-ins must be capable of operating asynchronously, and carry the burden of choosing the techniques to achieve this;
- Workflow and causality relationships are sometimes difficult to grasp. Many APIs use the callback pattern to accommodate non-threaded asynchronous operation.

Antidote comes with a number of example plug-ins that the developer can study. Some use threading and some are Glib-based.

3.7 Synchronous operation

For very simple examples, or unit testing routines, it would be desirable not having to deal with event loops or asynchronous operation. For those cases, Antidote actually offers very simple event loops for managers and agents:

- *manager_connection_loop(ContextId)*
- *agent_connection_loop(ContextId)*

As the prototype suggests, they handle only one context (i.e. only one connection) at a time. When the context is killed (when connection is closed), the loop exits. They can be called again for new connections.

While the loop is running, it is blocking. Doing any concurrent work (including timer handling) will need threads or signals.

When these loops are used, the communication plug-in must implement *network_wait_for_data ()* that must block until there is an APDU ready to receive.

An asynchronous plug-in does not need to implement this, but then it cannot work with {manager,agent}_connection_loop() APIs.

3.8 Communication plug-ins

Antidote itself is portable but it is deaf and blind. In order to do its job, it needs to communicate with the outside world. This communication is provided by (surprise!) communication plug-ins.

A communication plug-in must provide a handful of services to Antidote stack, through a standard C interface:

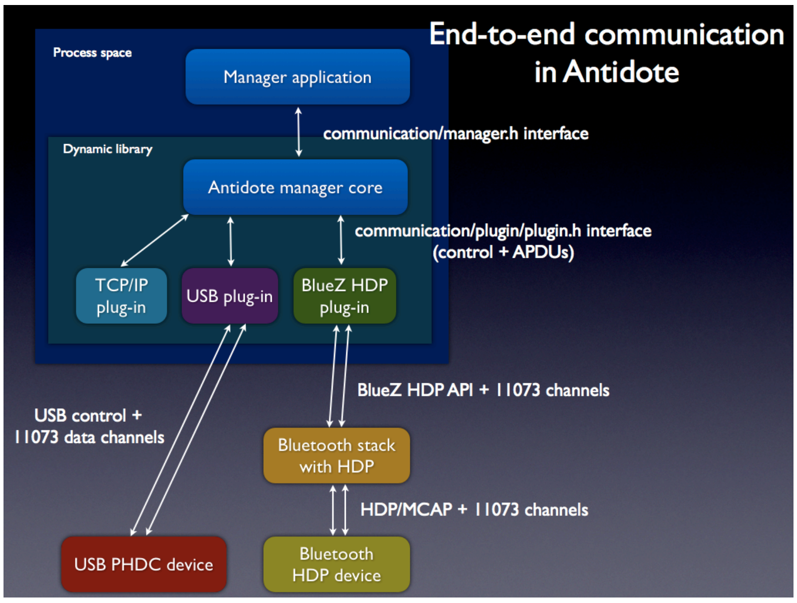
- Notify stack when a connection is opened or closed;
- Create and maintain unique IDs for each device/connection;
- Finish a connection when requested by stack;
- Notify stack when a new APDU arrives;
- Accept an APDU from stack and send it;
- Start and cancel asynchronous timers in behalf of stack;
- Protect a context if the plug-in uses multiple threads.

Looks simple, but dealing with protocols is never simple. Moreover, dealing with transport and timers brings in platform dependencies.

The developer must decide between asynchronous calls and threads when dealing with multiple connections. The decision will depend on the platform and the framework adopted by application.

Fortunately, many plug-ins are bundled with Antidote, so the developer can study the examples and/or borrow an existing plug-in design.

Since the top-level application that links with Antidote “knows” which communication plug-ins it is going to use (it needs to load and initiate them anyway), direct data exchange between plug-in and application is allowed. Antidote stack does not interfere on this.


End-to-end communication in Antidote
Process space
Manager application
communication/manager.h interface
Dynamic library
Antidote manager core
communication/plugin/plugin.h interface (control + APDUs)
TCP/IP plug-in
USB plug-in
BlueZ HDP plug-in
BlueZ HDP API + 11073 channels
USB control + 11073 data channels
Bluetooth stack with HDP
HDP/MCAP + 11073 channels
USB PHDC device
Bluetooth HDP device

For example, the HDP plug-in for Linux notifies the top-level application when a Bluetooth device connects, and supplies the low-level MAC address. It means nothing to IEEE 11073 (it has the System-ID attribute in MDS anyway), but *healthd* wants to know the MAC.

Finally, a plug-in code does not have to be written inside Antidote source tree *(src/communication/plugin)*. A plug-in is just a list of callbacks that is registered with the stack. It works regardless where the callback functions are implemented, provided they follow the API.

3.9 Timers

IEEE 11073 state machine prescribes timeouts for some cases, e.g. resending requests when a response is not heard in time. IEEE 11073 timeouts have one-second resolution. Timeout handling and asynchronous communication are related issues; they need to be considered as a bundle.

For example, if the communication plug-in is single-threaded, you cannot use threads for timeouts, because the plug-in could break when two threads ask simultaneously to send APDUs -- each APDU the reaction to a given timeout.

In order to reflect this, Antidote puts the burden of timeouts into communication plug-ins. Each plug-in must implement two timeout-related functions:

- *timer_count_timeout:* Register a timeout in behalf of a Context. The function must return an opaque handle.
- *timer_reset_timeout:* Cancel a timeout. The function is given the opaque handle in order to know which timer is to be cancelled.

If timeout is reached and not cancelled, the plug-in calls Context's callback. At least the Context related to each pending timer must be stored in some fashion (using a table or map).

There is an additional consideration: if an application uses several communication plug-ins, it might be wasteful implementing timeout handling in each one. The option is to implement application-wide timeout handlers and fill each plug-in attribute with them, after plug-in creation but before registration. This approach is actually used in sample managers bundled with Antidote source code.

3.10 Transcoding and transcoding plug-ins

Antidote supports transcoding. It means that non-Continua and non-11073 devices can communicate with Antidote, yet the client application thinks they are all 11073.

Support to any kind of device can be added without changing a single line of code at client side. All it takes is a transcoding plug-in that “understands” that device.

This is the downside of transcoding: every device model demands a different plug-in. There is potential for code sharing or even plug-in sharing if e.g. a manufacturer adopts the same base protocol for its entire product line.

Transcoding plug-ins are **not** communication plug-ins; they have different APIs and live in different folders of Antidote source tree.

Transcoding plug-ins tend to be more complicated than communication plug-ins, because the latter just need to send APDUs back and forth, while the first must in addition parse the supported protocol and re-encode messages in IEEE 11073 structures.

The developer must have the relevant IEEE 11073 specialization documents in hand to write a transcoding plug-in. It is one of the very few situations where IEEE 11073 knowledge is required in Antidote.

3.11 Specializations

Antidote comes with a number of specializations. They are kind of plug-ins as well. Each one contains the respective standard configurations.

This way, Antidote already knows those standard configurations in advance and does not need to learn them from devices with such configurations.

Optionally, a specialization contains event report builders: one for each configuration. This is necessary only if you plan to use Antidote in Agent’s role (Only agents emit event reports). Currently, only the oximeter specialization implements the event report.

Antidote reuses the same standard configuration for Manager and Agent roles. This indirectly means that, currently, Antidote-based agents are limited to standard configurations.

3.12 Healthd

Antidote itself is an IEEE 11073 library, but providing only that would leave the prospective developer with no means of making quick tests, or build upon a working sample.

Because of this, Antidote is bundled with *healthd*, a full-featured manager application. It serves multiple purposes:

- It is a very complete example of Antidote-based application;

- It has many command-line arguments suitable for testing;
- It exports a D-Bus health API.

The last item is more important than it looks. It allows writing health applications that don't link with Antidote; they just need to talk with *healthd* via D-Bus. That is, you can use Antidote and still have **zero** contact with its C API, if you want.

D-Bus was chosen because, at least in Linux, it is supported by all major programming languages. Every D-Bus API is effectively neutral in this aspect. Of course, nothing prevents a developer from using another IPC technology.

The standard Antidote distribution builds in Linux, using autotools (configure, make). Being a full application for Linux, *healthd* brings dependencies:

- BlueZ version 4.90 or better (with HDP support);
- Kernel 2.6.36 or better (ERTM support). Some 3.0.x kernels have problems with ERTM (Ubuntu Oneiric is a case);
- D-Bus 1.4 or better (needed by BlueZ HDP);
- LibUSB 1.0 (Or comment USB plug-in support from source);
- D-Bus Glib;
- Glib 2.0.

Every reasonably recent Linux distribution meets these requirements. There is a bug in ERTM, fixed in very development branch of 3.x kernel, which prevents connection with A&D devices in SSP mode. Current workaround is to disable SSP mode:

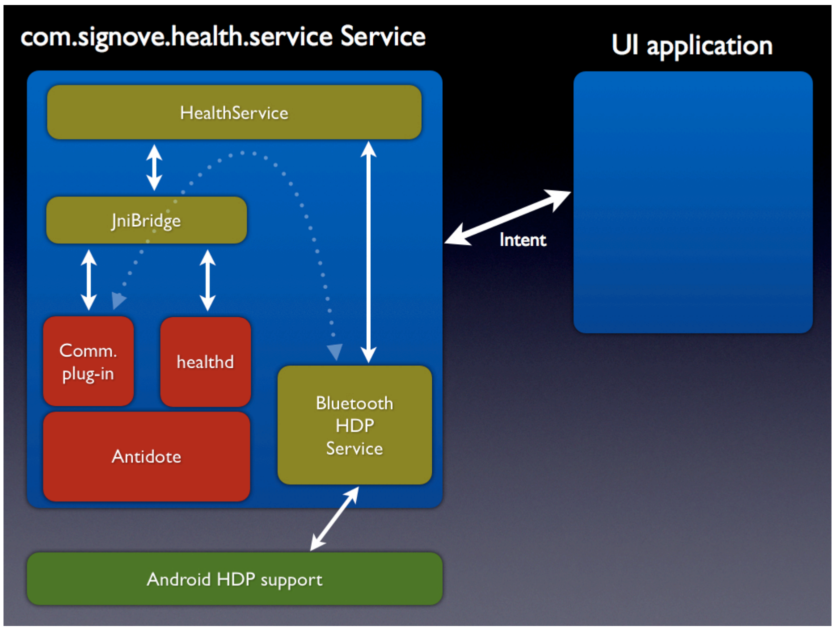
```
$ sudo hciconfig hci0 sspmode 0
```

3.13 Android

Antidote distribution builds readily in Android, using NDK (Native Development Kit). The build files (Android.mk) are in place.

Low-level speaking, Android is Linux but the platform is very different. Applications don't have native access to D-Bus, and therefore no access to BlueZ. No root access, so no direct hardware access (necessary for USB). Android uses Intents instead of D-Bus to implement IPC. Native applications can implement native activities, but not native services.

All these factors preclude using *healthd* "as is" in Android. A separate source code *healthd_android.c* is employed in Android build. It does not do everything by itself; it depends on Java-level support to communicate via HDP and provide a service to UI applications.



The communication plug-in for Android is basically a proxy that communicates with Java to send and receive APDUs. Even the connection ID must be generated at Java level.

The Android communication plug-in is very simple and not tied to HDP in any way. It can be reused as it is for other transport protocols; it is just a matter of adding Java-level support and generating unique IDs for every connection.

Antidote project offers a sample *HealthService* project that implements the needed Java support. In theory, a Java application could support Antidote in ways completely dissimilar from *HealthService*. In practice, most developers will appreciate to build upon a working and tested example.

3.14 Code structure

Antidote follows the code structure described below.

src/	Library and sample apps folder
asn1/	ASN.1 and PHD types
util/	Utility modules: MDER codecs of basic ASN.1 types, logging, reader and writer streams
communication/	Communication and service models implementation
parser/	MDER codecs for PHD, error detection
plugin/	Communication plug-ins
android/	Android plug-in
bluez/	Linux/Bluetooth/HDP plug-in
trans/	Dummy transcoding plug-in (boilerplate and debugging purposes)

usb/	USB PHDC plug-in (Linux)
dim/	Domain information model implementation
trans/	Transcoding support engine
plugin/	Transcoding plug-ins
specializations/	Device specializations
resources/	Non-code resources e.g. D-Bus service descriptor
api/	XML and JSON data encoders
tests/	Unit tests
doc/	Doxygen generated documentation goes here
debian/	Debian package build files

All source code in *src/* folder belongs either to Antidote library or to plug-ins, except by:

Source	Description	Related binary
src/healthd_android.c	Manager for Android, main source of native part	N/A
src/healthd_service.c	Healthd manager service main source	*Healthd*
src/test_healthd.py	Test client for Healthd manager, written in Python, CLI	N/A
src/sample_agent.c	Sample and testing agent, emulates an oximeter, supports TCP and FIFO	*ieee_agent*
src/sample_bt_agent.c	Sample agent, emulates a Bluetooth HDP oximeter	*sample_bt_agent*
src/sample_manager.c	Sample and testing manager, supports TCP and FIFO	*ieee_manager*

3.15 ASN.1 and byte streams

Normally, the developer does not have to know anything about IEEE 11073, ASN.1 or PHD types when working with Antidote. But it is necessary to possess this knowledge when writing a new specialization or a new transcoding plug-in. In both cases, the developer creates a number of 11073 structures. The developer also needs to use the PHD API, normally used only within Antidote.

Further sections of this document, as well as the source code itself, provide practical examples of composing IEEE 11073 structures. We will restrict ourselves to the key points:

- Every 11073 type (ASN.1 primitive or PHD) has a C *typedef* counterpart in Antidote.
- For every PHD type e.g. *AttributeList*, there are three related functions: *encode_attributelist(), decode_attributelist()* and *del_attributelist()*.
- For every primitive type, e.g. INT-U16, there are two related functions: *write_intu16()* and *read_intu16()*.
- Encoding a type always sends octets to a *ByteStreamWriter*.
- Decoding a type always reads octets from a *ByteStreamReader*.
- Byte streams are "safe" buffers used in conjunction with encoders and decoders to detect buffer overflows and missing octets.
- Encoders do NOT automatically fill octet counts. It is caller's responsibility to fill counts at all children levels, as well as providing a *ByteStreamWriter* with the exact total size. (These will be optional in version 2.1.)
- Never use *sizeof ()* to find the encoded octet size of a PHD structure. It happens to work for primitive types but not for compound ones because the *structs* are not packed.

Each type "knows" its children and encodes/decodes them implicitly. For example, *encode_attributelist()* encodes all children *AVA-Type's* too. The byte stream must be large enough to hold everyone.

Thanks to this, encoding can be delayed at many instances. For example, the specialization's configuration returns a *ConfigReport*, not a byte stream. The stack only needs to encode it when sending it to another device (and that happens only when Antidote is playing Agent role).

There are two important exceptions to implicit encoding and decoding: *Any* type and *Octet-string* type.

Any is an opaque octet string whose meaning is context-dependent. For example, every *AVA-Type* element contains a 16-bit OID identifier and an *Any* member. The actual meaning of *Any* contents depends on OID value.

Octet-string is normally a "true" string e.g. manufacturer's name. But sometimes it is used to hold encoded data, normally when format is defined outside 11073. Example: *ProdSpecEntry*.

When decoder finds *Any* or *octet-string* attributes, it just copies the octets without interpreting them. Likewise, the encoding process expects *Any* to contain a pre-encoded buffer and copies it into the stream. For example, to describe an *AttributeList* with a single *MDC_ATTR_ID_TYPE* attribute element, it is necessary to:

- Have the *AttributeList* variable *(t)*.
- Have one *AVA-Type* variable *(a)*.
- *a.OID* = *MDC_ATTR_ID_TYPE*;
- Create a writer byte *stream*.
- Encode two 16-bit values (e.g. *MDC_PART_SCADA* and *MDC_PULS_OXIM_SAT_O2*) into the stream, as implied by *MDC_ATTR_ID_TYPE*.
- *a.Any.length* = *4;* (octets in stream)
- *a.Any.buffer* = *stream->buffer;*
- *t.count* = *1*;
- *t.length* = *8*; (4 octets to *Any* buffer, 2 for *Any* length value, 2 for *OID* value)
- *t.value* = *malloc(sizeof(AVA-Type));* (space for *AttributeList* only child)
- *t.value[0]* = *a;*
- *free(stream)* (the descriptor, but not its buffer)

In the end, *(t)* is still a raw *AttributeList*, unencoded, but it contains a pre-encoded fragment (the ID_TYPE attribute) held by *t.value[0].Any.buffer*.

The heap-allocated elements (*stream->buffer, t.value*) are owned by *AttributeList* and properly released by *del_attributelist(t)*.

In theory, *Any* could be an OID-based choice and implemented with *union*, but this approach was not adopted in Antidote because:

- OID namespace is huge, that would translate to a huge *union*;
- OID namespace can be extended by future documents, and there is a private range. In the other hand, there are no easy ways to extend a *union* dynamically.
- In every instance where *Any* is employed (configuration, MDS, event report, etc.) only a subset of OIDs is actually acceptable. It is more efficient to handle each subset “near” where data is consumed.

These peculiarities make the PHD API look difficult at first sight. Fortunately, there are many examples bundled in Antidote’s source code, so the developer never needs to begin from scratch.

Future versions are expected to add helper functions to insulate the developer from these bureaucratic details to some extent.

4 How to Use

This section shows Antidote in practical usage, from installation to program development. Except when explicitly mentioned, all examples use standard Linux operating system.

4.1 Installing Antidote

Antidote is installed like most open source software – with the *configure/make* idiom:

```
$ ./autogen.sh # optional
$ ./configure
$ make
$ make install
$ # sudo make install if needed
```

Sample applications like *healthd* can be run from the source tree itself, without having to be installed in your system. This is convenient when developing on Antidote itself or for a quick test.

All executables are in src/ folder. Don't forget the prepend ./ when running them in src/. Note that healthd needs the file src/healthd.conf to be copied to /etc/dbus-1/system.d/ because it exports a D-Bus service to other apps. If you don't make install, you need to copy this file manually and restart D-Bus.

Build system uses libtool, which puts binary objects in `src/.libs` before installation; the executables in *src/* are libtool scripts. If you want to debug some Antidote application, it is better to have the actual binaries straight on *src/*. This is achieved by a configuration parameter:

```
$ ./configure –disable-shared
```

Another option to install Antidote is to generate packages and install the packages. Some may prefer this because it allows for a cleaner uninstall.

Currently, Antidote supplies Debian packaging support, as well as a handy *create_deb.sh* script. There is some vestigial support for RPM format (see *create_rpm.sh* script), but given the variety of RPM-based environments, it is not guaranteed to work out of the box.

4.2 Running test manager/agent

The test binaries are *ieee_manager* and *ieee_agent*, respectively. They are used by unit tests but can be run by the developer as well.

Both can communicate via TCP/IP or FIFO, by passing --tcp or --fifo respectively. For eventual usage, TCP/IP is more convenient.

The default is FIFO transport. FIFO expects two files to exist in current working directory: *read_fifo* and *write_fifo*. They can be created with *mkfifo*.

They can made to connect to each other or to any other 11073 device that uses TCP/IP transport. In TCP/IP mode, the manager listens at port 6024.

For a quick test, run the manager in a terminal:

```
$ ieee_manager --tcp
```

In another terminal, run the agent:

```
$ ieee_agent --tcp
```

The manager quits after three agent connections. The agent just connects, sends some measurement data and quits. The agent simulates an oximeter.

4.3 Running Healthd and a test client

Just run it. No root permissions necessary:

```
$ healthd
```

If it is running, the debug console should show something like:

```
DEBUG   <manager_start in manager.c:406> Manager starting...
DEBUG   <init in plugin_bluez.c:1211> Starting BlueZ link...
```

If you want to run *healthd* in source tree, please mind the notes regarding *healthd.conf* installation, mentioned at the beginning of this chapter.

The health service does not do much by itself. It needs a client to establish an “end-to-end” path to health data. You can run the test client in another terminal:

```
$ # assuming that CWD is antidote/src
$ ./test_healthd.py
```

At this point, the healthd console should show something like:

```
DEBUG   <create_health_application in plugin_bluez.c:1028> Created health
application: /org/bluez/health_app_18
DEBUG   <create_health_application in plugin_bluez.c:1028> Created health
application: /org/bluez/health_app_19
```

This means that HDP types requested by client have been registered. At this point, you can connect a Bluetooth HDP device with your system. Both *healthd* and *test_healthd.py* should show activity. The test client may print XML data related to measurements.

The HDP device must be paired before it can connect. Antidote does not take care of Bluetooth pairing, you must do it using the tools of your Linux distro. Please mind the items 2.15 and 3.12 regarding Linux kernel compatibility problems with HDP.

4.3.1 Advanced healthd parameters

The health service is not only a good example; it is a full-featured health service with many additional resources:

--autotest **--auto**	Auto-testing. Configure HDP specializations and shows measurement data without any client connected.
--tcp **--tcpserver**	Instead of D-Bus, health service is offered over TCP/IP, port 9005. The client-server protocol is one-way, but it could easily be extended.
--bluez	Use HDP (BlueZ) communication plug-in. This is the default behavior.
--trans	Enable device transcoding support, in addition to HDP (and to USB, if activated). Actual support depends on loading transcoding plug-ins. The standard version loads just an example oximeter (TCP/IP port 10404).
--usb	Use USB communication plug-in in addition of HDP. Supports USB devices that implement PHDC class.

4.3.2 Advanced test_healthd.py parameters

The client test script is a powerful tool. The standard behavior is just to receive measurement data and fetch the device configuration, but it can exercise almost every API of health service, once it is invoked with the proper parameter.

Every XML received by the script is saved on a file named *prefix_dddd_item.xml*, where prefix is "XML" by default, *dddd* are the last four hex digits of device ID, and *item* is a label of XML contents (configuration, measurement data, association data, etc.).

These XML files may be useful for parser testing, learning the format, compare data coming from different devices, and so on.

List of parameters accepted by *test_healthd.py*:

--mds	Fetch MDS object attributes as soon as device is associated. The MDS attribute list is a superset of the attribute list received upon association. A “diff” on XMLs can show that.
--prefix PREFIX	Change the file name prefix where XML data is saved.
--get-segment	Fetch PM-Store segment data from associated devices. Implies --store and --instance.
--clear-segment	Clear PM-Store segment of associated devices. Implies --store and --instance.
--clear-all-segments	Clear all PM-Store segments of associated devices. Implies --store and --instance.
--store HANDLE **--pmstore HANDLE** **--pm-store HANDLE**	Sets PM-Store handle number. Default is 11. Note that script does not interpret configuration in order to get the “right” PM-Store handle. There may be several stores as well. You need to supply the correct handle.
--instance NUMBER **--inst NUMBER** **--segment NUMBER** **--pm-segment NUMBER** **--pmsegment NUMBER**	Sets PM-Segment instance number. Default is 0. Note that script does not interpret PM-Segment info to determinate the “right” instance number. There can be many instances as well. You need to supply the correct instance number.
--interpret	Interpret XML and show data in human-readable format.

4.4 Bluetooth test agent

This applet is useful when you need to make tests with 11073 but does not have any real device in hand. It is very simple to use:

```
$ sample_bt_agent 00:11:22:33:44:55
```

The parameter is the Bluetooth MAC address of the IEEE 11073 manager, to which the agent should connect. This agent simulates a pulse oximeter with standard configuration 0x0190.

4.5 Running on Android

Antidote has been ported to Android, but it needs to interface with Java code to be useful. Consonant to this need, Signove supplies two Android applications that provide a very good starting point: *HealthService* and *HealthServiceTest*.

Antidote itself compiles and works in Android 2.2 and 2.3, but version 4.0 (ICS, Ice Cream Sandwich) added HDP support, which is needed by *HealthService*. So, unfortunately, these projects require ICS devices to run.

Since Antidote library is written in C, the developer must install Android NDK to compile it. Refer to Android documentation for details.

First, the developer needs to download the projects, either in tarballs or cloning the Git repositories (see 3.4). Import each project in Eclipse, or accordingly to your Android SDK environment.

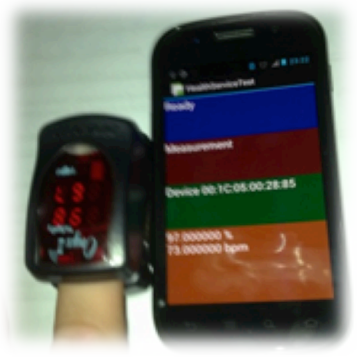
Second, Antidote source must be copied to *HealthService/jni/antidote*, and compiled with NDK. The log should be something like this:

```
HealthService/jni $ ~/android/android-ndk-r7/ndk-build NDK_DEBUG=1

Gdbserver      : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
Compile thumb  : healthd <= manager.c
Compile thumb  : healthd <= agent.c
…
SharedLibrary  : libhealthd.so
Install        : libhealthd.so => libs/armeabi/libhealthd.so
```

Once the native part is ready, Eclipse can be used to make the APK and/or install *HealthService* on device. *HealthService* opens an Activity screen but it does nothing. You need to install and run *HealthServiceTest* application to actually use the service.

Make sure that Bluetooth is active, and pair with the HDP device. You can use the test agent if you don't have one. If everything goes right, you should see something like the picture below while running *HealthServiceTest*.



4.6 DIY manager

In this topic, we analyze the code of a simple but functional manager. It uses the BlueZ communication plug-in, Glib framework and runs on Linux.

The source code is at *doc/examples/mgr*, file *mgr.c*. It is basically a slimmed-down version of *healthd* that compiles outside Antidote source tree. Antidote must be installed in your system.

First, some includes. We include the plug-in header as well, because the application is tasked with choosing and initializing plug-ins.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <glib.h>
#include <gio/gio.h>

#include <ieee11073.h>
#include <communication/plugin/plugin_bluez.h>
#include <communication/service.h>
```

The following structure will be filled some callbacks at *main()*.

```
static PluginBluezListener bluez_listener;
```

Some prototypes.

```
static gboolean display_measurementdata(ContextId, char *);
static gboolean display_disassociated(ContextId);
static gboolean display_associated(ContextId, char *);
```

The following three functions implement the necessary timeout servicing for Antidote. First and third functions are called by Antidote and their pointers will be supplied as callbacks in *main()*.

```
static void timer_reset_timeout(Context *ctx)
{
	if (ctx->timeout_action.id) {
		g_source_remove(ctx->timeout_action.id);
	}
}

static gboolean timer_alarm(gpointer data)
{
	Context *ctx = data;
	void (*f)() = ctx->timeout_action.func;
	if (f)
		f(ctx);
	return FALSE;
}

static int timer_count_timeout(Context *ctx)
{
	ctx->timeout_action.id = g_timeout_add(ctx->timeout_action.timeout
	                                 * 1000, timer_alarm, ctx);
	return ctx->timeout_action.id;
}
```

We used Glib facilities to schedule and cancel timers. Every framework supplies a (different) way to do the same. Note that Glib timeouts are measured in milliseconds while Antidote requests whole-second timeouts.

The next function is called by Antidote when there is a measurement data event report. A *DataList* is passed. It is a simple representation of original IEEE 11073 structure. See sections 5.4, 6.4.2 and 6.4.4 for details.

```
void new_data_received(Context *ctx, DataList *list)
{
	char *data = xml_encode_data_list(list);
```

```
        if (data) {
                display_measurementdata(ctx->id, data);
                free(data);
        }
}
```

We encoded the DataList in XML format and forward the string to another function.

The function *device_associated()* will be called by Antidote when a device associates. The association data is encoded to XML and forwarded.

```
void device_associated(Context *ctx, DataList *list)
{
        char *data = xml_encode_data_list(list);

        if (data) {
                display_associated(ctx->id, data);
                free(data);
        }
}
```

Simetrically, the next function will be called when a device disassociates.

```
void device_disassociated(Context *ctx)
{
        display_disassociated(ctx->id);
}
```

In *hdp_configure()*, the application instructs the BlueZ plug-in to be a sink for a list of data types, which correspond to 11703 specializations.

```
void hdp_configure()
{
        // oximeter, scale, blood pressure
        guint16 hdp_data_types[] = {0x1004, 0x1007, 0x1029, 0x100f, 0x0};
        plugin_bluez_update_data_types(TRUE, hdp_data_types); // TRUE=sink
}
```

HDP does not have a “wildcard” sink; it is necessary to configure the supported types explicitly.

The next functions are just the “consumers” of XML data generated earlier. Normally, a manager will do something useful here, instead of just displaying messages.

```
static gboolean display_connected(ContextId conn_handle, const char *low_addr)
{
	printf("Device connected context %d:%d addr %s\n",
		conn_handle.plugin, (int) conn_handle.connid,
		low_addr);

	return TRUE;
}

static gboolean display_associated(ContextId conn_handle, char *xml)
{
	printf("Device associated context %d:%d\n",
		conn_handle.plugin, (int) conn_handle.connid);
	printf("\n\nAssociation data:\n%s\n\n", xml);

	return TRUE;
}

static gboolean display_measurementdata(ContextId conn_handle, gchar *xml)
{
	printf("Device context %d:%d\n",
		conn_handle.plugin, (int) conn_handle.connid);
	printf("\n\nMeasurement data:\n%s\n\n", xml);

	return TRUE;
}
```

More data consumption functions, nothing special about them.

```
static gboolean display_disassociated(ContextId conn_handle)
{
	printf("Device disassociated context %d:%d\n",
		conn_handle.plugin, (int) conn_handle.connid);
	return TRUE;
}

static gboolean display_disconnected(ContextId conn_handle, const char *low_addr)
{
	printf("Device disconnected context %d:%d addr %s\n",
		conn_handle.plugin, (int) conn_handle.connid,
		low_addr);

	return TRUE;
}
```

Next coding does some housekeeping.

```
static GMainLoop *mainloop = NULL;

static void app_finalize(int sig)
{
	g_main_loop_quit(mainloop);
}

static void app_setup_signals()
{
	signal(SIGINT, app_finalize);
	signal(SIGTERM, app_finalize);
}
```

Then we have *main()*. Since we use Glib and the BlueZ plug-in uses DBus-Glib, we need to init the *GObject* type system. This is a Glib-specific need.

```
int main(int argc, char *argv[])
{
	g_type_init();

	CommunicationPlugin plugin;
	app_setup_signals();
```

In the next part, we allocate a communication plug-in structure and offer it to the BlueZ plug-in.

```
	plugin = communication_plugin();

	plugin_bluez_setup(&plugin);
	bluez_listener.peer_connected = display_connected;
	bluez_listener.peer_disconnected = display_disconnected;
	plugin_bluez_set_listener(&bluez_listener);
```

This plug-in is capable of notifying the application when a device connects (Antidote itself just notifies us about associations). The interested party must pass a listener structure. The application would work without that, anyway.

Next lines configure the plug-in to use the timeout servicing functions we have defined earlier.

```
	plugin.timer_count_timeout = timer_count_timeout;
	plugin.timer_reset_timeout = timer_reset_timeout;
```

Antidote looks for timeout servicing at communication plug-in, but BlueZ plug-in does not implement it, so we supply the missing functionality at this point.

If an application uses many communication plug-ins, it makes more sense to provide timeout servicing at just one place (like we did), and configure every plug-in to use it.

Next, we initiate the IEEE 11073 manager, providing a list of communication plug-ins.

```
	CommunicationPlugin *plugins[] = {&plugin, 0};

	manager_init(plugins);
```

In this case, we just supply one plug-in for HDP (made for Linux and BlueZ).

The following code configures and adds a “manager listener”, providing some functions that we defined earlier as callbacks. This is how Antidote gets knowledge of how to notify the application about IEEE 11073 events.

```
	ManagerListener listener = MANAGER_LISTENER_EMPTY;
	listener.measurement_data_updated = &new_data_received;
	listener.segment_data_received = 0;
	listener.device_available = &device_associated;
	listener.device_unavailable = &device_disassociated;

	manager_add_listener(listener);
	manager_start();
```

After the listener is in place, the manager is started. The manager will start the communication plug-ins.

At this point, the BlueZ plug-in is activated, so we can change the list of supported data types.

```
	hdp_configure();
```

This example is very simple but it operates in asynchronous mode, using a Glib event loop:

```
	mainloop = g_main_loop_new(NULL, FALSE);
	g_main_loop_ref(mainloop);
	g_main_loop_run(mainloop);
```

The application “sleeps” in main loop until it is cancelled by a signal, which in turn calls *app_finalize()*.

When the main loop exits, execution continues until the end of *main()*:

```
        manager_finalize();

        return 0;
}
```

Finishing the manager before exiting is not strictly necessary but it is good practice.

4.7 DIY agent

In this topic, we analyze the code of a simple but functional agent. It also uses the BlueZ plug-in, and depends on Linux and Glib as well.

The source code is at *doc/examples/mgr*, file *agt.c*. It is basically a slimmed-down version of *sample_bt_agent* that compiles outside Antidote source tree. Antidote must be installed in your system.

First, the bureaucracy of include files.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <unistd.h>
#include <time.h>
#include <glib.h>
#include <gio/gio.h>

#include <ieee11073.h>
#include <communication/plugin/plugin_bluez.h>
#include <specializations/pulse_oximeter.h>
#include <agent.h>
```

The following “constants” define which kind of 11073 agent we are.

```
static const intu8 AGENT_SYSTEM_ID_VALUE[] = { 0x11, 0x33, 0x55, 0x77, 0x99,
             0xbb, 0xdd, 0xff};

static int oximeter_specialization = 0x0190;
guint16 hdp_data_types[] = {0x1004, 0x00};
```

Then are some application variables, including the communication plug-in. We will use them later.

```
static CommunicationPlugin comm_plugin = COMMUNICATION_PLUGIN_NULL;
static PluginBluezListener bluez_listener;
static GMainLoop *mainloop = NULL;

static int alarms = 0;
static guint alrm_handle = 0;

static ContextId cid;
```

The clean-up function is executed upon exit and makes Valgrind happy.

```
static void app_clean_up()
{
	g_main_loop_unref(mainloop);
}
```

Next, there are the timeout servicing functions. They are exactly the same as in the manager example.

```
static void timer_reset_timeout(Context *ctx)
{
	if (ctx->timeout_action.id) {
		g_source_remove(ctx->timeout_action.id);
	}
}

static gboolean timer_alarm(gpointer data)
{
	Context *ctx = data;
	void (*f)() = ctx->timeout_action.func;
	f(ctx);
	return FALSE;
}

static int timer_count_timeout(Context *ctx)
{
	ctx->timeout_action.id = g_timeout_add(ctx->timeout_action.timeout
	                                  * 1000, timer_alarm, ctx);
	return ctx->timeout_action.id;
}
```

The following section implements agent-like behavior, again using Glib timers: send some measurements and finally take the initiative of disconnection.

```

static gboolean sigalrm(gpointer data);
static void schedule_alarm(ContextId id, int to)
{
	if (alrm_handle) {
		g_source_remove(alrm_handle);
	}
	cid = id;
	alrm_handle = g_timeout_add(to * 1000, sigalrm, 0);
}

static gboolean sigalrm(gpointer data)
{
	fprintf(stderr, "==== alarm %d ====\n", alarms);

	alrm_handle = 0;

	if (alarms > 2) {
		agent_send_data(cid);
		schedule_alarm(cid, 3);
	} else if (alarms == 2) {
		agent_request_association_release(cid);
		schedule_alarm(cid, 2);
	} else if (alarms == 1) {
		agent_disconnect(cid);
		schedule_alarm(cid, 2);
	} else {
		g_main_loop_quit(mainloop);
	}

	--alarms;

	return FALSE;
}
```

Normally, IEEE 11073 agents take the initiative in everything, while IEEE 11073 managers wait and react. The section shown above reflected this agent-typical behavior. Timeouts are used to trigger events; in a real device, sensors drive the events.

The following function is called by Antidote when device associates with the manager. At this point, we start the timer that sends periodic measurements.

```
void device_associated(Context *ctx)
{
	fprintf(stderr, "Associated\n");
```

```
        alarms = 5;
        schedule_alarm(ctx->id, 3);
}
```

Next function is called when device disassociates from manager, either by our initiative or by manager's.

```
void device_unavailable(Context *ctx)
{
        fprintf(stderr, "Disasociated\n");
        if (alarms > 2) {
                // involuntary; go straight to disconnection
                alarms = 1;
        }
}
```

The *device_connected()* function is called back by Antidote when device is connected.

```
void device_connected(Context *ctx)
{
        fprintf(stderr, "Connected\n");
        agent_associate(ctx->id);
}
```

The following function is called back by BlueZ plug-in when device is connected.

```
static gboolean bt_connected(ContextId context, const char *btaddr)
{
        fprintf(stderr, "bt_connected %s\n", btaddr);
        return TRUE;
}
```

Note that there are two connection-related callbacks: one at plug-in level and another at IEEE 11073 level.

Next function is called back by communication plug-in when device is disconnected.

```
static gboolean bt_disconnected(ContextId context, const char *btaddr)
{
        fprintf(stderr, "bt_disconnected %s\n", btaddr);
        return TRUE;
}
```

The next function produces measurement data. It is called back by Antidote when a measurement event report is being composed and this data is needed.

```
static void *event_report_cb()
{
	time_t now;
	struct tm nowtm;
	struct oximeter_event_report_data* data =
		malloc(sizeof(struct oximeter_event_report_data));

	time(&now);
	localtime_r(&now, &nowtm);

	data->beats = 60.5 + random() % 20;
	data->oximetry = 90.5 + random() % 10;
	data->century = nowtm.tm_year / 100 + 19;
	data->year = nowtm.tm_year % 100;
	data->month = nowtm.tm_mon + 1;
	data->day = nowtm.tm_mday;
	data->hour = nowtm.tm_hour;
	data->minute = nowtm.tm_min;
	data->second = nowtm.tm_sec;
	data->sec_fractions = 50;

	return data;
}
```

The data structure supplied by this function is defined by the specialization plug-in (in this case, the oximeter). The specialization must implement agent-related features to be employed in agent applications.

When the stack needs agent’s MDS attributes, the *mds_data_cb()* function is called:

```
static struct mds_system_data *mds_data_cb()
{
	struct mds_system_data* data = malloc(sizeof(struct mds_system_data));
	memcpy(&data->system_id, AGENT_SYSTEM_ID_VALUE, 8);
	return data;
}
```

Currently, only the system id attribute is required to be filled in. The *mds_system_data* structure may be extended in the future.

Then we reach the *main()* function, beginning with agent initialization.

```
int main(int argc, char **argv)
{
	g_type_init();
	comm_plugin = communication_plugin();

	if (argc != 2) {
		fprintf(stderr, "Usage: sample_bt_agent <bdaddr>\n");
		exit(1);
	}

	fprintf(stderr, "\nIEEE 11073 sample agent\n");
```

We configure the plug-in with our timeout servicing kit.

```
	comm_plugin.timer_count_timeout = timer_count_timeout;
	comm_plugin.timer_reset_timeout = timer_reset_timeout;
```

Next, the BlueZ plug-in is configured, using the structure that we allocated.

```
	CommunicationPlugin *plugins[] = {&comm_plugin, 0};

	plugin_bluez_setup(&comm_plugin);
```

The timeout callbacks that we have configured before are not overwritten because BlueZ plug-in does not implement timeout servicing.

Then we configure a listener for connection and disconnection events.

```
	bluez_listener.peer_connected = bt_connected;
	bluez_listener.peer_disconnected = bt_disconnected;
	plugin_bluez_set_listener(&bluez_listener);
```

These events are emitted by the communication plug-in.

The following call initializes agent with the chosen communication plug-ins, desired service specialization and suppliers of data for event reports and MDS.

```
	agent_init(plugins, oximeter_specialization,
			event_report_cb, mds_data_cb);
```

Analogous to manager, the agent callback functions need to be passed to Antidote through a listener structure, below:

```
	AgentListener listener = AGENT_LISTENER_EMPTY;
	listener.device_connected = &device_connected;
	listener.device_associated = &device_associated;
	listener.device_unavailable = &device_unavailable;

	agent_add_listener(listener);
```

Then, agent is started and BlueZ plug-in is updated with desired data type, matching agent specialization. This time, HDP channel has the source role.

```
	agent_start();

	plugin_bluez_update_data_types(FALSE, hdp_data_types); // FALSE=source
```

The next call asks the plug-in to establish a transport connection. This call is blocking, but could be made asynchronous with little effort.

Then we ask the plug-in to establish a transport connection. This call is blocking, but could be made asynchronous with little effort.

```
	if (!plugin_bluez_connect(argv[1], hdp_data_types[0],
	                                            HDP_CHANNEL_RELIABLE)) {
		exit(1);
	}
```

That call bypassed the IEEE 11073 stack. Antidote is notified only when the connection is brought up, by the plug-in. When this happens, the agent state machine automatically tries to establish an association, without our intervention.

That activity happens asynchronously, so the top-level application must be running an event loop:

```
	alarms = 0;

	fprintf(stderr, "Main loop started\n");
	mainloop = g_main_loop_new(NULL, FALSE);
	g_main_loop_ref(mainloop);
	g_main_loop_run(mainloop);
	fprintf(stderr, "Main loop stopped\n");

	agent_finalize();
	app_clean_up();

	return 0;
}
```

The call to *g_main_loop_run()* returns olny when the event loop is interrupted by calling *sigalrm()*.

5 Antidote C API

This section describes selected parts of Antidote's C API. Note that this document does not describe every API function. Doxygen-generated documentation fills this role. This section is a guide to show the developer where to start looking.

The developer is encouraged to read first the relevant topics in Section 4.6, which shows parts of the API under a practical perspective.

The section format is: a relevant piece of include file, followed by the corresponding API documentation.

5.1 Manager

This API offers high-level functionality for IEEE 11073 manager applications. The include file is *manager.h*.

```
typedef struct ManagerListener {
	void (*measurement_data_updated)(Context *ctx, DataList *list);
	void (*segment_data_received)(Context *ctx, int handle, int instnumber,
				      DataList *list);
	void (*device_available)(Context *ctx, DataList *list);
	void (*device_unavailable)(Context *ctx);
	void (*timeout)(Context *ctx);
} ManagerListener;
```

This structure is filled by the application with callback functions and passed to manager API. It is the major channel for a manager app to receive IEEE 11073 events.

If the app is not interested in some particular event (e.g. timeout), just fill it with a null pointer.

The *segment_data_received* callback has a semantic difference from the others: it passes the ownership of *DataList* to the caller. This is because PM-Store data lists may be very large, and interpretation/encoding may take a long time. Passing the ownership allows the application to delegate the list handling to another thread, or postpone it.

```
#define MANAGER_LISTENER_EMPTY;
```

This is an empty listener structure with all callbacks filled with NULL.

```
void manager_init(CommunicationPlugin **plugins);
```

Initializes (but does not start) the manager with a list of communication plug-ins. In, turn Antidote initializes the plug-ins.

```
void manager_start();
```

Starts the IEEE 11073 manager. Associations are accepted from this moment on.

```
void manager_stop();
```

Stops the IEEE 11073 manager. It can be restarted if necessary.

```
void manager_finalize();
```

Finalizes the IEEE 11073 manager, freeing internal structures. Normally used just before application exit. Manager cannot be restarted after this call.

```
void manager_connection_loop(ContextId context_id);
```

Used by applications that lack an event loop. This function provides a simple main loop that returns when the passed context is disconnected.

Passing a context ID implies that it can handle only one connection at a time; it is quite limited and for simple tests only.

```
int manager_add_listener(ManagerListener listener);
```

Used by application to register the listener callbacks with manager. Several listeners can be added, but normally only one is necessary.

```
DataList *manager_get_mds_attributes(ContextId id);
```

Returns a data list with known MDS attributes for the context. Internal format of data list is described in Section 6.4.5.

```
Request *manager_request_measurement_data_transmission(ContextId id,
                                        service_request_callback callback);
```

Request transmission of measurement data. The device must support this feature. Returns NULL if the request cannot be made.

```
Request *manager_request_get_all_mds_attributes(ContextId id,
                                    service_request_callback callback);
```

Request all MDS attributes from remote device. Returns a request structure or NULL if request cannot be made.

Once the request callback is called, the application can get them using *manager_get_mds_attributes()*.

```
Request *manager_request_get_pmstore(ContextId id, int handle,
                                   service_request_callback callback);
```

Request PM-Store attributes from device. Returns a request structure or NULL if handle is invalid or if the request cannot be made by some other reason.

Upon callback, data can be actually be obtained by calling *manager_get_pmstore_data()*.

```
DataList *manager_get_pmstore_data(ContextId id, int handle);
```

Gets a data list with known PM-Store attributes. This is a “fast” call (i.e. does not have to carry a remote request to the agent), but it only returns current agent data if the application had called *manager_request_get_pmstore()* and waited for the callback. Returns NULL if context or handle are invalid.

The internal format of data list is described in Section 6.4.6.

```
Request *manager_request_get_segment_info(ContextId id, int handle,
                                       service_request_callback callback);
```

Makes a request to get segment data. Returns NULL if request cannot be made or parameters are invalid. When callback is called, data can be fetched using *manager_get_segment_info_data()*.

```
DataList *manager_get_segment_info_data(ContextId id, int handle);
```

Returns segment info data, or NULL if context or handle are invalid. Internal format of data list is described in Section 6.4.7.

This is a “fast” call but it only returns actual segment info data if the application had called *manager_request_get_segment_info()* and waited for the callback.

```
Request *manager_set_operational_state_of_the_scanner(ContextId id,
               HANDLE handle, OperationalState state,
               service_request_callback callback);
```

Set operational state of scanner object.

```
Request *manager_request_get_segment_data(ContextId id, int handle,
                   int instnumber, service_request_callback callback);
```

Requests PM-Store segment data. Returns NULL if request cannot be made (perhaps because some parameter is invalid). The request response at *request->return_data* contains only the response status (successful or not).

It can fail temporarily even if handle and instance number are correct - for example, if the segment is busy writing new entries. If successful, the actual data is sent via listener’s *segment_data_received* callback.

```
Request *manager_request_clear_segment(ContextId id, int handle, int instnumber,
                               service_request_callback callback);
```

Requests to clear a PM-Store segment.

Returns NULL if some parameter is wrong or request cannot be made for some other reason.

```
Request *manager_request_clear_segments(ContextId id, int handle,
                                     service_request_callback callback);
```

Request to clear all segments of a PM-Store.

Returns NULL if some parameter is wrong or request cannot be made by any reason.

```
DataList *manager_get_configuration(ContextId id);
```

Returns a DataList with device configuration.

This is a “fast”, local call, because configuration is known as soon as device associates. Returns NULL if context is invalid. The internal format of data list is described in Section 6.4.3.

```
void manager_request_association_release(ContextId id);
```

Requests the manager to disassociate from a given device.

```
void manager_request_association_abort(ContextId id);
```

Requests the manager to abort association with a given device.

Normally, the application should never need to abort an association. Abortion signals an error condition, and Antidote handles most error conditions automatically.

5.2 Agent

This API offers high-level functionality for IEEE 11073 agent applications. Include file is *agent.h*.

```
typedef struct AgentListener {
	void (*device_connected)(Context *ctx);
	void (*device_associated)(Context *ctx);
	void (*device_unavailable)(Context *ctx);
	void (*timeout)(Context *ctx);
} AgentListener;
```

Akin to manager API, an agent application communicates with the stack mainly via this listener structure, which contains a set of callbacks.

Different from manager, there is a hook for connection events. This is necessary because agent normally takes the initiative of association when connection is established.

Agents normally take the initiative of connecting as well, but the agent API does not take care of this. Each communication plug-in is in charge of providing some API to start a connection.

```
#define AGENT_LISTENER_EMPTY;
```

Pre-defined listener with all callbacks set to NULL. Null callbacks are ignored by the stack.

```
void agent_init(CommunicationPlugin **plugins, int specialization,
             void *(*event_report_cb)(),
             struct mds_system_data *(*mds_data_cb)());
```

Initialize the agent (but does not start it) with given communication plug-ins, a IEEE 11073 specialization code (e.g. 4100 or 0x1004 for oximeter), and two data provider callbacks.

The stack will call these callbacks when it needs the System-Id for MDS and when it emits a measurement report.

```
void agent_start();
```

Starts the agent.

```
void agent_stop();
```

Stops the agent. It still can be restarted if necessary.

```
void agent_finalize();
```

Finalize the agent and free internal structures. Normally called just before application quits. Agent cannot be restarted after this call.

```
void agent_connection_loop(ContextId context_id);
```

This is a simple, blocking event loop implementation. It can be used by applications that don't have their own event loop. Handles only one connection at a time.

```
int agent_add_listener(AgentListener listener);
```

Used by application to register a listener structure. Many listeners can be registered, but normally only one is used.

```
void agent_associate(ContextId id);
```

Assuming that context represents a connection, requests the stack to initiate an association.

Note that the agent stack does not take the initiative of association. The application must call this function for that. On the other hand, if manager takes the initiative to associate, the stack will accept it.

Agent calls listener's *agent_connected*; when connected with (but still not associated to) a manager. Normally an agent application calls *agent_associate()* upon this moment.

```
void agent_request_association_release(ContextId id);
```

Request that association with given device is released.

```
void agent_disconnect(ContextId id);
```

Request a disconnection from given device.

```
void agent_send_data(ContextId id);
```

Request that agent stack sends a measurement event report to the given device.

5.3 Service

This API is of interest to manager developers mainly because of the *Request* structure. Include file is *communication/service.h*.

This structure is returned upon a request, and the same structure is passed to request callback. This allows an application to keep some control of its pending requests.

Normally, the application may “forget” the first Request, because the callback will be called even in case of error, allowing for graceful, asynchronous error handling.

```
typedef struct Request {
       // ‘private’ members omitted
       void *context;
       struct RequestRet *return_data;
} Request;
```

The request’s *context* member (not to be confused with Context type used throughout Antidote) is a hook where the application can hang some request-related data that it wishes to receive when request is fulfilled.

It is automatically released when the request is destroyed, so the application does not need to do this (if the application does release it, then it must set *context* to NULL to avoid a double free.)

The *return_data* member contains some “subclass” of the *RequestRet* structure. The actual structure depends on the nature of original request. It is employed by PM-Store requests that return data. See *dim/pmstore_req.h* to know these structures in detail.

```
typedef void (*service_request_callback)(Context *ctx, struct Request *r,
                                       DATA_apdu *response_apdu);
```

This is the typedef for all request callbacks. Note that it includes the “raw” APDU sent as response by the remote device. Normally the application won’t have to look into the APDU, but it is there in case of need.

```
typedef void (*request_ret_data_free)(void *);


struct RequestRet {
       request_ret_data_free del_function;
};
```

This is the “abstract class” for request’s *return_data*. Regardless of how elaborate the actual *return_data* object is, it always “knows” how to release itself, thanks to *del*_function.

```
InvokeIDType service_get_current_invoke_id(Context *ctx);
Request *service_get_request(Context *ctx, InvokeIDType invoke_id);
```

If the application prefers to store invoke IDs instead of *Request*'s, fearing dangling pointers, it can get the invoke ID, and get back the *Request* afterwards, using the above functions.

5.4 Data encoder API (DataList)

Some manager and agent APIs supply data to application using the *DataList* format. The developer needs to know the *DataList* format (defined in this API) in order to interpret the data. Include file is *api/api_definitions.h*.

Also, there are the encoder functions for XML *(api/xml_encoder.h)* and JSON *(api/json_encoder.h)*. They are employed in *healthd*, Android and other sample applications.

```
char *xml_encode_data_list(DataList *list);
char *json_encode_data_list(DataList *list);
```

These functions get a *DataList* and return it in encoded format. The caller is responsible by freeing the struct.

```
typedef struct DataList {
	int size;
	DataEntry *values;
} DataList;
```

DataList itself is a simple structure. It is just a list of entries.

```
typedef struct DataEntry {
	MetaData meta_data;
	DataEntry_choice choice;
	union {
		SimpleDataEntry simple;
		CompoundDataEntry compound;
	} u;
} DataEntry;
```

Each data entry is represented by a *DataEntry* instance. Data entries may be simple or compound. Basically, a compound entry contains other entries. Regardless of type, each entry has a name, and may have a list of "metadata" attributes attached.

```
typedef struct MetaAtt {
	char *name;
	char *value;
} MetaAtt;


typedef struct MetaData {
	int size;
	MetaAtt *values;
} MetaData;
```

These are the metadata structures; simple name-value string pairs.

```
typedef enum {
	SIMPLE_DATA_ENTRY
	COMPOUND_DATA_ENTRY
} DataEntry_choice;

typedef struct SimpleDataEntry {
	char *name;
	APIDEF_type type;
	char *value;
} SimpleDataEntry;


typedef struct CompoundDataEntry {
	char *name;
	int entries_count;
	struct DataEntry *entries;
} CompoundDataEntry;
```

These structures are part of the *DataEntry* union. Note that both simple and compound entries have a name.

Every data list tree must eventually end in simple data entries. A simple entry is a name-type-value triplet. All three parameters are stored as strings.

```
typedef char *APIDEF_type;
#define APIDEF_TYPE_STRING "string"
#define APIDEF_TYPE_INT32 "int32"
#define APIDEF_TYPE_INTU32 "intu32"
#define APIDEF_TYPE_INT16 "int16"
#define APIDEF_TYPE_INTU16 "intu16"
#define APIDEF_TYPE_INT8 "int8"
#define APIDEF_TYPE_INTU8 "intu8"
#define APIDEF_TYPE_FLOAT "float"
#define APIDEF_TYPE_HEX "hex"
```

Those macros define the possible types of a simple entry. Any other type must be constructed as a compound of the basic types.

```
void data_list_del(DataList *pointer);
```

This API frees a data list and all associated entries. Normally, the stack releases data lists itself upon callback returning, and the application does not need to call this function.

The only exception is the segment data callback (see manager API). Each IEEE 11703 message that is sent to the application (configuration, measurement, attributes) is converted to a data list in predetermined formats. Refer to the topics of Section 6.4 to learn about those formats.

5.5 Communication plug-in

This API is for communication plug-in writers. Include file is *communication/plugin/plugin.h*. A communication plug-in is, in essence, a collection of functions tied together by a structure named *CommunicationPlugin*.

```
typedef struct CommunicationPlugin {
	network_init_ptr network_init;
	network_finalize_ptr network_finalize;
	network_get_apdu_stream_ptr network_get_apdu_stream;
	network_wait_for_data_ptr network_wait_for_data;
	network_send_apdu_stream_ptr network_send_apdu_stream;
	network_disconnect_ptr network_disconnect;
	thread_lock_ptr thread_lock;
	thread_unlock_ptr thread_unlock;
	thread_init_ptr thread_init;
	thread_finalize_ptr thread_finalize;
	int (*timer_count_timeout)(PluginContext *ctx);
	timer_reset_timeout_ptr timer_reset_timeout;
	int type;
} CommunicationPlugin;
```

The *CommunicationPlugin* structure has many members, but not every plug-in is obligated to fill every item.

- *type* is a bitmap filled by the stack, **not** by the plug-in. But the plug-in may query the contents. The plug-in may behave differently if it is used in manager or agent contexts. See *MANAGER_CONTEXT*, *AGENT_CONTEXT* and *TRANS_CONTEXT* at *communication/context.h*.
- *timer_count_timeout* and *timer_reset_timeout* are called by the stack to service timeouts. The plug-in is, in theory, responsible by filling these items. In practice, they are often filled in by the top-level application, because this makes more sens in a multiple-plugin scenario.
- *network_init* is a plug-in function called by the stack when manager or agent is started. The plug-in must go to operating state. The *plugin_label* parameter is the plug-in ID from stack point of view; it must be saved in e.g. static variable because this value is needed to generate new *ContextId*’s.
- *network_finalize* is called by the stack when the agent or manager is stopped. The plug-in must stop and release resources at this point.
- *network_get_apdu_stream* is called by stack to fetch a received APDU. Normally this call should not block; the stack “knows” that there is an available APDU because the plug-in called *communication_read_input_stream()* before.
- *network_wait_for_data* is a blocking function called by stack to wait for an APDU to arrive. It is only used by applications without an event loop (and “normal” applications always have one). An asynchronous plug-in does not have to implement this.
- *network_send_apdu_stream* is called by stack when an APDU must be sent.
- *network_disconnect* is called by stack when a connection must be closed.
- *thread_lock*, *thread_unlock*, *thread_init* and *thread_finalize* are used to synchronize access to a context. Only necessary in multithreaded scenarios. Each *Context* contains a *void *multithread* hook to contain some synchronization-related object, like a mutex. The plug-in may use it at will.

```
typedef struct Context PluginContext;
```

PluginContext is a local typedef for the all-important *Context* type, used to bypass the need of forward declaration.

```
typedef void (*thread_lock_ptr)(PluginContext *ctx);
typedef void (*thread_unlock_ptr)(PluginContext *ctx);
typedef void (*thread_init_ptr)(PluginContext *ctx);
typedef void (*thread_finalize_ptr)(PluginContext *ctx);
typedef int (*network_init_ptr)(unsigned int plugin_label);
typedef int (*network_finalize_ptr)();
typedef ByteStreamReader* (*network_get_apdu_stream_ptr)(PluginContext *ctx);
typedef int (*network_wait_for_data_ptr)(PluginContext *ctx);
typedef int (*network_send_apdu_stream_ptr)(PluginContext *ctx,
                                            ByteStreamWriter *stream);
typedef int (*network_disconnect_ptr)(PluginContext *ctx);
typedef void (*timer_wait_for_timeout_ptr)(PluginContext *ctx);
typedef void (*timer_reset_timeout_ptr)(PluginContext *ctx);
typedef int (*timer_count_timeout_ptr)(PluginContext *ctx);

#define NETWORK_ERROR
#define NETWORK_ERROR_NONE
```

These are the typedefs for the callbacks in communication plug-in structure. Note that some of them are expected to return an integer status code; they must return either of the macro-defined values.

In the following part, we have some functions that are defined at communication level, that every plug-in must call upon communication events.

```
communication_transport_connect_indication(ContextId);
```

Called by plug-in to notify the stack that a new connection is up. The *ContextId* must be generated by the communication plug-in. It is a two-value tuple: the stack-attributed plug-in ID (participated to the plug-in in *network_init* callback) and a connection ID whose namespace is the plug-in.

The plug-in is responsible to generate unique connection IDs for every device in its own domain, and reuse the same ID when the same device reconnects.

The first connection of a device is the exact moment when the Antidote stack "learns" the *ContextId*. This ID is used in all subsequent operations with that device.

```
communication_transport_disconnect_indication(ContextId);
```

Called by plug-in to notify the stack that a connection is down.

```
communication_read_input_stream(Context*);
```

Called by plug-in to notify the stack that an APDU has been received from a device.

The stack will call plug-ins *network_get_apdu_stream* callback right after, to get APDU data.

```
communication_process_input_data(ContextId, stream);
```

This is an alternative way to notify the stack about new data.

The APDU stream is sent along, so the stack does not have to call *network_get_apdu_stream* to get it.

```
CommunicationPlugin communication_plugin();
```

This function fills a communication plugin structure with stub callbacks.

It is important to use this function instead of a zeroed structure for plug-in setup, because a plug-in may not fill some callbacks, the stack might call them without checking for NULL.

5.6 Transcoding plug-in API

This API is for transcoding plug-in writers. Include file is *trans/trans.h*. Note that, in order to write a transcoding plug-in, the developer needs to have reasonable knowledge about IEEE 11073, access to 11073 documents, and needs to use Antidote's PHD APIs, documented in Doxygen. This document shows the high-level API only.

Wring a transcoding plug-in has some things in common with writing an agent, because the transcoding layer actually represents a (fake) agent to satisfy the IEEE 11073 manager stack.

Very few manager stack functions discriminate between native and transcoded devices; this ensures that there is little code duplication.

```
typedef struct TransPlugin
{
	int (*init)();
	int (*finalize)();
	void (*force_disconnect)(const char *);
	agent_connected_cb conn_cb;
	agent_disconnected_cb disconn_cb;
} TransPlugin;
```

This structure must be filled by the transcoding plug-in, when it is set up. The application is responsible by calling plug-in setup function.

Structure allocation may be done by application or by the plug-in itself. The only requisite is being unique: one pointer for each plug-in; the same plug-in must not be set up twice.

Note that initialization is different from setup. The plug-in is initialized when the stack calls the *init* callback.

The *force_disconnect* function is called by stack when connection with device should be ended preemptively.

The *agent_connected_cb* and *agent_disconnected_cb* are called by the plug-in when a device connects and disconnects. This mechanism is used to notify the application about those events. The pointers are supplied by the application (the stack is not interested in these particular events).

```
typedef int (*agent_connected_cb)(ContextId id, const char *lladdr);
typedef int (*agent_disconnected_cb)(ContextId id, const char *lladdr);
```

Typedefs for the connection callbacks. Note that a *lladdr* string is passed. This is an arbitrary string, defined by the transcoding plug-in, that represents a device.

The *lladdr* string may contain anything; the only requirement is to be unique across all transcoding plug-ins.

```
void trans_register_plugin(TransPlugin *plugin);
```

This function registers a transcoding plug-in with the stack. The plug-in must have been set up by the application beforehand (using some API provided by the plug-in itself).

```
int trans_connected(TransPlugin *plugin,
                    char *lladdr,
                    AttributeList spec,
                    PhdAssociationInformation assoc_info,
                    ConfigReport config);
```

This function is called by the transcoding plug-in when a device connects and it is ready to associate.

- *plugin* is a pointer to the transcoding plug-in structure. Since the plug-in must pass it back to the stack, it means that plug-in must have saved this pointer upon set-up.
- *lladdr* is the low-level address of device. The string can be chosen arbitrarily by the plug-in. It can be anything; the only requisite is to be unique across all transcoded devices. A good choice is device's MAC address, when available, because it is guaranteed to be unique.
- *spec* is an *AttributeList* containing device's MDS object attributes. At the very least, it should contain device specialization.
- *assoc_info* is association information in IEEE 11073 format. This is used to simulate an agent in front of IEEE 11073 manager. Important information like *System-Id* goes into this structure.
- *config* is the IEEE 11073 configuration report for the device. It may be a standard or extended configuration.

The last three parameters are IEEE 11073 structures that must be composed using Antidote's PHD APIs. You can use *trans/plugin/example_oximeter.c* as a starting point.

The stack takes ownership of all IEEE 11073 structures; the plug-in must not delete any of them.

```
int trans_event_report_fixed(TransPlugin *plugin,
                   char *lladdr,
                   ScanReportInfoFixed report);
```

This function is called by the transcoding plug-in when a device has some measurement data.

The *ScanReportInfoFixed* structure must be generated using Antidote PHD API. The stack takes the ownership; the plug-in must not delete it.

```
int trans_event_report_var(TransPlugin *plugin,
                   char *lladdr,
                   ScanReportInfoVar report);
```

This function is called by the transcoding plug-in when a device has some measurement data, and variable-format report is to be used.

The *ScanReportInfoVar* structure must be generated using Antidote PHD API. The stack takes the ownership; the plug-in must not delete it.

```
int trans_disconnected(TransPlugin *plugin, char *lladdr);
```

This function is called by the transcoding plug-in when a device disconnects.

5.7 Linking with Antidote

This section describes how to link a given application against the IEEE static library in a GNU/Linux environment.

First, Antidote must be installed, either by downloading and installing the development package, or by installing it from source *(configure && make && make install)*.

This library uses *pkg-tools*. An application that employs *autoconf* can be linked to Antidote very easily. In *configure.in* file, the following line is added:

PKG_CHECK_MODULES(ANTIDOTE, antidote)

In *Makefile.am*, the compilation and link flags are added, like in this example:

```
someapp_CFLAGS = @LIB1_CFLAGS@ @LIB2_CFLAGS@ @ANTIDOTE_CFLAGS@
```

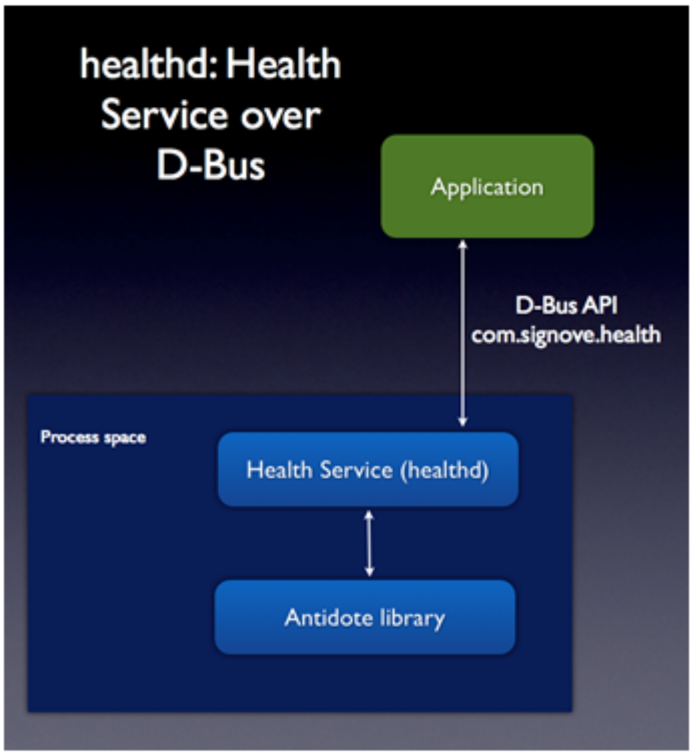
```
someapp_LDADD = \
             @LIB1_LIBS@ \
             @LIB2_LIBS@ \
             @ANTIDOTE_LIBS@
```

"Lib1" and "Lib2" are other hypothetical dependencies of the application. Check the example in *doc/examples/mgr* for more details.

If the application does not use *autoconf*, the compiler/linker flags and include paths must be configured manually. Each environment may present different configurations, so your best bet is learn the configurations by querying the *pkgconfig* file, at */usr/lib/pkgconfig/antidote.pc* or */usr/local/lib/pkgconfig/antidote.pc*.

6 Healthd D-Bus API

As explained before, Healthd is a manager application bundled with Antidote, and based on it, which exports a D-Bus API for client UI applications.



It provides a way to write health applications based in any language that supports D-Bus, without even linking with Antidote. D-Bus is, in practice, a Linux-only IPC mechanism, but the basic idea could be easily translated to other platforms and IPCs (as we did for Android).

This section is a formal description of the D-Bus API served by Healthd. It follows loosely the format of D-Bus API description documents (which are normally written in raw text). The script *src/test_healthd.py* is a good example of how to use this API.

6.1 Manager interface *(com.signove.health.manager)*

Service: *com.signove.health*

Path: */com/signove/health*

Manager methods:

void ConfigurePassive(object agent, int data_types[])

Puts the health service in listening mode (accepting connections from agents).

The agent is a client-provided D-Bus object that receives events, by having its methods called back. (Don't confuse this agent with the IEEE 11073 agent.)

data_types is an array of integers that configure which HDP data types should be accepted. The values can be found in HDP specification. For example, 0x1004 hex is the oximeter, and this is the value that should be passed in this API, if HDP oximeters are to be supported.

Note that *data_types* does not currently limit the specializations handled by non-HDP transports.

The configuration survives until client or server process dies. The *healthd* service detects when a client goes away.

The client application can detect an eventual *healthd* death/restart event, and redo configuration when this happens. The standard D-Bus way to detect this condition is by listening the *NameOwnerChanged* signal. (You can look into *src/test_healthd.py* to learn how.)

void Configure()

Makes the health service to initiate a connection with an Agent. Not implemented.

6.2 Agent interface *(com.signove.health.agent)*

The agent supplied by client in *ConfigurePassive()* may have any path, but it must implement a number of methods under the *com.signove.health.agent* interface.

Don't confuse this agent object with the IEEE 11073 agent. In D-Bus patterns, an object is an object that is registered with a service in order to receive "callbacks" from the service.

These callbacks are the mechanism through which the client "discovers" the IEEE 11073 devices associated with *healthd* manager. Once a device connects and associates, the application may simply wait for measurement data or make active requests (configuration, PM-Store data, etc.).

Service: *(any)*

Path: *(any)*

Agent methods:

void Connected(object device, string address)

Called back when a device connects. *Device* is a D-Bus device object. *Address* is the transport layer address (in case of Bluetooth HDP, it is the MAC address).

The received device object implements the interface *com.signove.health.device*, which is described later in this chapter. The path itself is opaque.

The name of first parameter *(device)* is a bit misleading because, if a device establishes two separate data channels with the manager, that will translate to 2 paths. (In practice, most devices use a single channel.)

It is **not** guaranteed that a new device calls *Connected()* before *Associated()*. This method is called only when a) transport layer emits a connection event, and b) there is a transport address. (Transports like FIFOs or pipes don't have a meaningful address that uniquely identifies a device.)

Also, a device may stay associated in IEEE 11073 level while disconnected, and connection/disconnection events bear no relationship with association status! Clients are advised to use this callback just for informational purposes (e.g. user feedback).

void Associated(object device, string xmldata)

Called back when a device goes into associated state.

The first parameter is the opaque device path. The application may relate it to the actual MAC address by keeping a map and update it upon *Connected()* and *Disconnected()* agent calls.

The second parameter contains a XML representation of data sent by agent in association APDU. Format is described at section 6.4.5.

Note that it is not guaranteed that *Connected()* is ever called before *Associated()*. If the application needs a unique ID for the device, it must get the *System-Id* attribute from XML data.

void MeasurementData(object device, string xmldata)

Called back upon a measurement event report. The second parameter is the XML representation of MDS object(s) whose value has changed.

XML format is described at 6.4.4.

void DeviceAttributes(object device, string xmldata)

Called back when MDS attributes have been fetched from device. This is the asynchronous response to an earlier call to *device.RequestDeviceAttributes()*, described in next topic.

XML format is described in section 6.4.5.

void Disassociated(object device)

Device entered into disassociated state.

Normally, *Disconnected()* is right after disassociation, but it might happen that the connection is recycled in a new association.

The client may "forget" the device at this point. (It does not have to; since device object are not "recycled"; a given D-Bus device object is always related to the same device, unless *healthd* is restarted).

void Disconnected(object device)

Device (channel) disconnected.

Normally, a disconnection implies previous disassociation, but IEEE 11073 allows association persistence across connections. This is **not** the right moment to "forget" a device; the *Disassociated()* callback is.

Clients are advised to use this just for informational purposes (e.g. user feedback).

void PMStoreData(object device, int handle, string xmldata)

Asynchronous response for a previous *device.GetPMStore(handle)* request.

handle is the PM-Store handle in MDS. The client can discover handles via *device.GetConfiguration()*.

xmldata is the PM-Store attribute list encoded in XML format, described in section 6.4.6.

void SegmentInfo(object device, int handle, string xmldata)

Asynchronous response for a previous *device.GetSegmentInfo(handle)* request.

handle is the PM-Store handle in MDS. The client can discover handles via *device.GetConfiguration()*.

xmldata is the segment info encoded in XML format, including available segments in the store. Format is described at section 6.4.7.

void SegmentDataResponse(object device, int handle, int segment, int response)

Asynchronous response for a previous *device.GetSegmentData(handle, segment)*.

handle is the PM-Store handle in MDS. “segment” is the segment instance number, that was discovered via *GetSegmentInfo()/SegmentInfo()*.

response is the IEEE 11073’s *TrigSegmXferRsp* response code: 0=success, 1=no such segment, 2=busy (being cleared now), 3=segment empty, other=unspecified error.

If response is “success” (zero) the next method will be called back in short notice.

void SegmentData(object device, int handle, int segment, string xmldata)

Second asynchronous response for *device.GetSegmentData(handle, segment)*. It is received after *SegmentDataResponse()*, and only if the request was reported successful.

XML format is described at section 6.4.8.

The whole segment data is delivered at once. In IEEE 11073 level, segment data transfer is carried out in many chunks due to APDU size restrictions.

void SegmentCleared(object device, int handle, int segment, int retstatus)

Asynchronous response to *device.ClearSegment(handle, segment)*.

“retstatus” is 0=ok, other=error.

6.3 Device interface *(com.device.health.device)*

The agent receives a D-Bus device object in almost every method. This topic describes the methods implemented by device object.

Each object relates to a single device, and *healthd* reuses the same object for a given device when it associates again. The application may safely “cache” object paths and relate them with System-Ids and/or transport addresses.

The objects are invalidated only when the *healthd* process exits/restarts. The standard D-Bus way to detect this condition is by listening the *NameOwnerChanged* signal. (You can look into *src/test_healthd.py* to learn how.)

All methods return immediately. When the response depends on communication with the IEEE 11073 device, the response is sent asynchronously to D-Bus agent (seen in last topic).

This mode of operation was chosen because it is normally easier to implement that in client applications than handling asynchronous D-Bus calls.

Service: *com.signove.health*

Path: *(any)*

Device methods:

void RequestDeviceAttributes()

Request MDS device attributes.

Data is received via *agent.DeviceAttributes()*.

string GetConfiguration()

Return device configuration in XML, immediately. XML format is described at 6.4.3.

This is useful to discover the MDS object layout, and the handles of individual objects e.g. PM-Stores.

void ReleaseAssociation()

Request disassociation. The RLRQ (release request) is used. This is the preferred method to disassociate.

void AbortAssociation()

Request disassociation. The ABRT (abort association request) is used. May be called even between *Connected()* and *Associated()*.

Normally, this method should not be used. Abortion is a response to unexpected behavior, and Antidote automatically aborts the association when this happens.

int GetPMStore(int handle)

Request attributes of PM-Store object with given *handle*.

Passing a handle of -1 is a wildcard for “any store”, useful when device has only one store, and/or for testing purposes.

Returns 0=ok, other=error.

If ok, data is received via *agent.PMStoreData()*.

int GetSegmentInfo(int handle)

Request segment info. Target is the PM-Store object with given *handle*.

A handle of -1 is a wildcard for “any store”, useful when device has only one store, and/or for testing purposes.

Returns 0=ok, other=error.

If ok, data is received via agent.SegmentInfo().

int GetSegmentData(int handle, int number)

Request data of PM-Segment with the given instance *number*, in PM-Store object with given *handle*.

Important: you must have called *GetSegmentInfo(handle)* before during the association, so the manager had an opportunity to discover the available segments.

A handle of -1 is a wildcard for "any store", and passing a instance number of -1 is a wildcard for "any instance".

Returns 0 when request is apparently ok; the definite response is received via *agent.SegmentDataResponse()* and may be an error.

If final response is successful, store data is received via *agent.SegmentData()*.

int ClearSegment(int handle, int instance number)

Request that the given store segment is cleared.

Important: you must have called *GetSegmentInfo(handle)* before during the association, so the manager had an opportunity to discover the available segments.

A handle of -1 is a wildcard for "any store", and passing a instance number of -1 is a wildcard for "any instance number".

Returns 0 when request is apparently ok; the definite response is received via *agent.SegmentCleared()* and may be an error.

int ClearAllSegments(int handle)

The same as *ClearSegment()*, but clears all segments of the given store.

void RequestActivationScanner(int handle)

Request activation scanner identified by *handle*.

void RequestDeactivationScanner(int handle)

Request deactivation of the scanner identified by *handle*.

void RequestMeasurementDataTransmission()

Request transmission of measurement data. (Rarely used.)

void Connect()

Request (re)connection. (Currently a stub.)

void Disconnect()

Request disconnection. (Currently a stub.)

6.4 XML data format

In many cases, this D-Bus service returns XML-encoded data to the client application. XML is used when original data is of hierarchical nature, as many IEEE 11073 messages are.

XML was selected due to widespread support in programming languages. In the future, as JSON is equally well supported, the API may provide additional D-Bus interfaces for JSON support.

D-Bus arrays (plain and associative) could have been used, but they are very inconvenient to work with, at least in static languages, putting a heavy burden on client development. (The D-Bus API for arrays is arid and poorly documented.)

Moreover, non-Linux platforms (including Android) don’t use D-Bus as IPC, so it would be a wasted effort. In the other hand, XML strings are easily adaptable to any IPC.

6.4.1 General

Antidote translates some IEEE 11073 messages to a *DataList* tree-like structure (described in section 5.4), which is uniform and easier to interpret. Each XML (or JSON) sent by the API is simply an encoded *DataList*.

There is a 1:1 relationship between *DataList* elements and XML elements. Each *DataList* is translated to a XML document with a single *data-list* element in it:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<data-list>
…
</data-list>
```

A *DataList* may have any number of *DataEntry* children. Each *DataEntry* is an element inside <*data-list*>:

```
<data-list>
<entry>…</entry>
<entry>…</entry>
…
</data-list>
```

Each *DataEntry* may have some meta-data that helps to describe it. Besides, each entry may be simple or compound. A simple entry has exactly three attributes: name, type and value.

```
<entry>
    <meta-data>
        <meta name=”unit-code”>544</meta>
        <meta name=”unit”>%</meta>
    </meta-data>
    <simple>
         <name>Basic-Nu-Observed-Value</name>
         <type>float</type>
         <value>29.00000</value>
    </simple>
</entry>
```

A compound entry contains other entries (that can be themselves simple or compound).

```
<entry>
    <meta-data>
        <meta name=”unit-code”>3872</meta>
        <meta name=”unit”>mmHg</meta>
    </meta-data>
    <compound>
         <name>Compound-Basic-Nu-Observed-Value</name>
         <entries>
                     … three simple entries meaning systolic,
                       diastolic and MAP blood pressures …
         </entries>
    </compound>
</entry>
```

This is the complete (if informal) description of XML DTD.

The script *test_healthd.py* saves all XML data it receives to files named `XML_(id)_(contents).xml`. It is a handy source of XML samples that you can use to test a parser and/or understand the internal structure.

6.4.2 Describing data lists

The next topics will describe how data lists are used to represent IEEE 11073 data, like configurations, attributes and measurements.

The final XML result will not be described directly because it gets too long very fast. XML is meant to be interpreted mechanically, following the "DTD" described in 6.4.1.

The data list format (that is encoded to XML, or JSON) is described instead, using a table format like the example below:

Level	Name and description	Type	Meta-data
1	Foo	Compound	Spam: "eggs"
2	Bar	intu16	
2	Baz	intu16	

The "Level" value establishes parent/child/sibling relationships between elements. In the example, "Bar" has level 2, meaning that it is boxed inside a compound, level-1 "Foo" and is a sibling to "Baz". Name indentation provides an additional visual cue to this relationship.

The table is actually a template, not a final data list, because we did not specify the values for "Bar" and "Baz".

If "Bar" and "Baz" had values 555 and 666 respectively, the data list would translate to the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<data-list>
<entry>
    <meta-data>
        <meta name="Spam">eggs</meta>
    </meta-data>
    <compound>
         <name>Foo</name>
         <entries>
              <entry>
                   <meta-data>
                   </meta-data>
                   <simple>
                        <name>Bar</name>
                        <type>intu16</type>
                        <value>555</value>
                   </simple>
              </entry>
             <entry>
                  <meta-data>
                  </meta-data>
                 <simple>
                      <name>Baz</name>
                      <type>intu16</type>
                      <value>666</value>
                 </simple>
             </entry>
         </entries>
    </compound>
</entry>
</data-list>
```

The table and this XML describe the same data list hierarchy, but we believe that the table format is much easier to read and comprehend.

6.4.3 Configuration

The configuration structure reveals the layout of MDS objects. By analyzing the configuration, an application knows beforehand which kinds of measurement data it may receive.

It is not strictly necessary to parse the device configuration – unless the application is going to access the PM-Store. In this case, the configuration is the only way to discover the PM-Store handle number.

Level	Name and description	Type	Meta-data
1	Configuration object Name: (object class)	Compound	HANDLE: handle #
2	Type	Compound	
3	Code	intu16	
3	Partition	intu16	
2	Metric-Spec-Small	intu16	
2	Unit-Code	intu16	
2	Attribute-Value-Map	Compound	

For example, an oximeter that follows standard configuration 0x0190 would have the following list structure with two objects:

Level	Name	Value	Meta-data
1	Numeric		HANDLE: 1
2	Type		
3	Code	19384 MDC_PULS_OXIM_SAT_O2 Pulse oximetry	
3	Partition	2 MDC_PART_SCADA	
2	Metric-Spec-Small	16448 0x4040 bitmap avail-stored-data, acc-agent-init, measured	
2	Unit-Code	544 MDC_DIM_PERCENT %	
2	Attribute-Value-Map	(not shown here for brevity)	
1	Numeric		HANDLE: 10
2	Type		
3	Code	18458 MDC_PULS_OXIM_PULSE_RATE Pulse rate	
3	Partition	2 MDC_PART_SCADA	
2	Metric-Spec-Small	16448 0x4040 bitmap avail-stored-data, acc-agent-init,	

		measured
2	Unit-Code	2720 MDC_DIM_BEAT_PER_MIN beats per minute
2	Attribute-Value-Map	(not shown here for brevity)

Interpreting the configuration structure requires some knowledge about IEEE 11073. At least, discovering the number of objects and their units is more or less straightforward.

Attribute-Value-Map is a compound type that describes the format of measurement events. Since Antidote handles it automatically, the client application should never need to interpret this item. It is included for completeness.

6.4.4 Measurement data

In practical terms, measurement data is the most important structure, because it is the one that carries actual sensor data, which is the whole point of developing an IEEE 11073 manager.

Level	Name and description	Type	Meta-data
1	Object Name: (object class)	Compound	HANDLE: handle #
2	Data content Name: (Data content PHD type)	(Data content primitive type)	partition metric-id unit-code unit

The naming of each element looks a bit confusing. An example (a measurement from a standard oximeter) will help:

Level	Name and description	Type Value	Meta-data
1	Numeric		HANDLE: 1

2	Basic-Nu-Observed-Value	float 97.0000	partition: 2 metric-id: 19384 unit-code: 544 unit: “%”
1	Numeric		HANDLE: 10
2	Basic-Nu-Observed-Value	float 65.000	partition: 2 metric-id: 18458 unit-code: 2720 unit: “bpm”

It can be seen that many things refer to configuration example: handle IDs, unit codes etc. Antidote adds a human-readable version of *unit*; a simple client app can show the measurements without any knowledge of IEEE 11073 unit codes.

Now we can clarify the confusion about class names, type names, and so on:

- *Numeric* is the class of each MDS object;
- Both *Numeric* objects contain a *Basic-Nu-Observed-Value* information;
- A *Basic-Nu-Observed-Value* is worth a float;
- In this example, the *Numeric* object contains only the float, but it could contain more attributes, like an *Absolute-Time-Stamp*.

6.4.5 Association data and MDS attributes

When a device associates, some MDS attributes are sent immediately. If the manager desires to know all attributes, it needs to call *RequestDeviceAttributes()*.

Note that we are talking about attributes of the root MDS object. The children objects are not sent along; those are discovered via configuration and/or measurement data.

Level	**Name and description**	**Type**	**Meta-data**
1	MDS	Compound	
2	MDS attribute Name: (variable)	(attribute type)	attribute-id

This structure is relatively simple, because it is just a list of attributes. There is only one object at level 1: the MDS itself.

Of course, an attribute may be itself a compound. In that case, structure may look more complex. Here we have a partial example of a MDS attribute list, where all attributes except one are compound:

Level	Name	Type	Meta-data
1	MDS		
2	System-Type		attribute-id: 2438
3	Code	intu16	
3	Partition	intu16	
2	System-Model		attribute-id: 2344
3	manufacturer	string e.g “Acme corp.”	
3	model-number	string e.g. “Super BP 543”	
2	System-Id	hex e.g. “001f12233445c”	attribute-id: 2436
2	System-Type-Spec-List		attribute-id: 2650
3	0		
4	version	Intu16 e.g. 1	
4	type	Intu16 e.g. 4103 – blood pressure specialization	

The very important *System-Type-Spec-List* contains a list of specializations, each specialization being itself a compound of “version” and “type” values.

6.4.6 PM-Store attributes

The structure of PM-Store attribute list looks like the MDS attribute list; both are lists of an object's attributes.

The PM-Store list is more stable, meaning that length and quality of attributes is always the same for every store. Most attributes are necessary for a client to manipulate the store correctly, so they always need to be there.

While the XML is simple, the meaning of each attribute depends on IEEE 11073 knowledge. The following table gives a very brief explanation for each attribute. Refer to [4] section "PM-Store class" for more details (Antidote source code can provide some information as well).

Level	Name and description	Type	Meta-data
1	Attributes	Compound	HANDLE: store handle
2	Handle	intu16	attribute-id
2	Capabilities Bitmap of store capabilities e.g. 0x8000: variable number of segments 0x0040: segments cleared are removed 0x0008: multi-person support	intu16	attribute-id
2	Store-Sample-Algorithm Bitmap that describes how samples are derived and averaged e.g. 0x0000: no algorithm 0x0001: moving average	intu16	attribute-id
2	Store-Capacity-Count Maximum number of entries that can be stored in this store (counting all segments)	intu32	attribute-id
2	Store-Usage-Count Number of entries currently stored (counting all segments).	intu32	attribute-id

2	Operational-State 0: disabled 1: enabled 2: not available	intu16 OperationalState	attribute-id
2	PM-Store-Label Human-readable description of this store	string	attribute-id
2	Sample-Period In case of periodic measurements, specifies the frequency at which entries are added. If zero, Sample-Period is specified at individual PM-Segments (if both are zero, measurements are aperiodic).	intu32 RelativeTime	attribute-id
2	Number-Of-Segments Current number of segments that belong to this store. May be variable (see Capabilities attribute).	intu16	attribute-id
2	Clear-Timeout Maximum time in seconds that a manager waits for clear response.	intu32 RelativeTime	attribute-id

6.4.7 PM-Store segment info

The most important information in this structure is the Instance Number for each segment in the store, because the client needs it to request segment data.

The other items may be of importance or not, depending on the case. Some IEEE 11073 knowledge is necessary to interpret them; we give a very brief explanation. Refer to [4], section “PM-Segment class” for all details.

Many attributes use compound types, like *AbsoluteTime*, that is encoded as a compound of values (century, year, month, and so on.)

Level	Name and description	Type	Meta-data

1	Segments	Compound	
2	Segment	Compound	Instance-Number
3	Instance-Number Identifies a segment within a store (as a handle identifies the store in MDS)	intu16	attribute-id
3	Operational-State 0: disabled 1: enabled 2: not available	intu16	attribute-id
3	Sample-period In case of periodic measurements, specifies the frequency at which entries are added. If zero, Sample-Period is specified at individual PM-Segments (if both are zero, measurements are aperiodic).	intu32	attribute-id
3	PM-Segment-Label Human-readable description of this segment	string	attribute-id
3	Person-ID Identifies the person in case of devices that support multiple users	intu16	attribute-id
3	Start-Time Date and time when entries began to be added to this segment	Compound (AbsoluteTime)	attribute-id
3	End-Time Date and time when entries ceased to be added to this segment	Compound (AbsoluteTime)	attribute-id
3	Segment-Statistics Minimum, average and maximum for each	Compound (SegmStatistics)	attribute-id

	element.		
3	Usage-Count Current number of entries in this segment	intu32	attribute-id
3	PM-Segment-Entry-Map Describes the entries' binary format. (Antidote decodes it automatically, the application is not forced to interpret this in order to get segment data.)	Compound (PM-Segment-Entry-Map)	attribute-id
3	Date-And-Time-Adjustment Date and time adjustment made by the user or caused by daylight savings.	Compound (AbsoluteTimeAdjust)	attribute-id

6.4.8 PM-Store segment data

In highest level, the segment data hierarchy was made to look like a measurement event as much as possible. Even the unit is bundled for easy data interpretation.

Since a segment can have any number of entries, and each entry has a header with timestamp information, there are of course additional boxing levels.

Level	Name and description	Type	Meta-data
1	PM-Segment	Compound	
2	Pm-Segment-Entry-Map This name is misleading! This is a compound entry that holds a single store entry. (If there are e.g. 50 entries, there will be 50 of this element in XML.)	Compound	
3	Segm-Entry-Header Some meta information of the entry. Will have one of the three timestamp values described below.	Compound	

4	Segment-Absolute-Time Absolute time of the entry	Compound (AbsoluteTime)	attribute-id
4	Segment-Relative-Time Relative time offset of the entry.	intu32 (RelativeTime)	attribute-id
4	Segment-Hires-Relative-Time Relative time offset of the entry.	Compound (2 x intu32)	attribute-id
3	Segm-Entry-Elem-List Contains the actual measurements of this entry. Each measurement is related to some MDS object, and the structure of tis compound is similar to a measurement event report.	Compound	
4	Name: (object class) Measured object. Classes may be Numeric, Enumeration or RT-SA. The handle allows to relate it to the MDS object configuration. Metric-id identifies the object meaning.	Compound	HANDLE metric-id partition-SCADA-code
5	Name: (PHD type) Data content of the measured object. Has the same information of a measurement event report for the same object, including unit code, etc.	(Data content primitive type)	partition metric-id unit-code unit

7 Android APIs

Even though Android is Linux-based, Android APIs are different from Linux due to a number of reasons:

- Interfacing between native code and Java code is made via JNI, which is far from trivial to use;
- It is difficult (if not impossible) to develop a 100% native application;
- Native applications don't have access to most Android APIs;
- Android does not D-Bus as IPC. Intents fill that role;
- Android does not allow direct access to BlueZ in any case.

Antidote for Android (called AFA from now on) is compiled as a library. It depends on a Java "shell" to run as an application. AFA contains the core library, as well as a slimmed down version of *healthd* manager.

The APIs borrow heavily from *healthd* API, including the use of XML strings to send structured data. Once the developer knows one API, he will feel at home with the other.

7.1 HealthService

HealthService is an Android *Service* that exposes its API to other applications. Any application (even more than one at the same time) may use this service. The only extra step is to copy the AIDL files from *HealthService* folder to the client application.

HealthServiceTest uses this API (and nothing else). This API is analogous, and extremely similar, to the D-Bus API that *healthd* serves to Linux clients. Actually, both share some C code.

There is no need to repeat the text of Section 6; we just point out the differences.

```
interface HealthServiceAPI {
	void ConfigurePassive(HealthAgentAPI agt, in int[] specs);
	String GetConfiguration(String dev);
	void RequestDeviceAttributes(String dev);
	void Unconfigure(HealthAgentAPI agt);
}
```

The main difference from D-Bus API is the absence of a separate Device object. A single service object is the target of all methods. PM-Store methods are still not implemented.

As the D-Bus API, the client must call ConfigurePassive() passing an agent object, that will receive manager callbacks.

```
interface HealthAgentAPI {
	void Connected(String dev, String addr);
	void Associated(String dev, String xmldata);
	void MeasurementData(String dev, String xmldata);
	void DeviceAttributes(String dev, String xmldata);
	void Disassociated(String dev);
	void Disconnected(String dev);
}
```

This is the API of the agent object. (Do not confuse this kind of agent with the IEEE 11073 agent devices!) Again, it is an almost perfect subset of the D-Bus agent.

Refer to Android documentation, and of course to *HealthServiceTest* source code, for more information of how to use those AIDL-defined service APIs.

In Antidote's native code side, these APIs are supported mainly by *src/healthd_android.c*. The name is suggestive: it is basically a ripoff from *src/healthd_service.c*, adapted to Android and JNI.

7.2 JniBridge

This is the low-level Java interface to Antidote for Android (AFA). Every native JNI call, and every callback from C code to Java, passes through *JniBridge*.

It is a very thin layer above AFA. It does not do much useful work, save by synchronizing multithreading access to AFA. (Parts of AFA are not multithread-safe, and it is easier to do synchronization at Java level.)

JniBridge talks with *src/healthd_service.c* and with Android-specifc communication plug-in. Refer to *HealthService/src/com/signove/health/service/JniBridge.java* for more details.

7.3 HealthService and HDP support

The communication plugin used in AFA is a simple proxy that exchanges APDUs via *JniBridge*. Actual HDP communication is implemented at Java level: see *BluetoothHDPService.java* in *HealthService* project. Even the connection IDs (the major ingredient of *ContextId's*) are generated in Java.

As mentioned, native support to Android features is very limited. This means that, if there is a need to support other transport protocols in the future, this must be implemented using Java. In the other hand, the native AFA communication plug-in would not need any changes in this scenario, since APDUs coming from either transport look exactly the same.

7.4 Further developments

It should be stressed that it **is not mandatory** to use the bundled Android support “as is”. Antidote could be used as an IEEE 11073 stack in Android in completely different ways, even though the bundled solution is quite convenient. The same happens in Linux: nobody is forced to use *healthd*, even though it is quite convenient.

These are the main components of Android support:

- *src/healthd_android.c: healthd*-like manager service
- *src/communication/plugin/android:* communication plug-in. Very generic APDU conduit, actual transport is implemented elsewhere.
- *JniBridge.java:* Links with native code functions.
- *HealthService:* Service application that uses Antidote as IEEE 11073 stack and implements HDP support.
- *HealthService API*: interface convention to the *HealthService* service.
- *HealthServiceTest:* UI application that is client of *HealthService* via its API.

The developer is free to improve, alter or replace any of them.

References

[1]Continua Health Alliance. http://www.continuaalliance.org

[2]Google Health. http://health.google.com

[3]Microsoft Health Vault. http://www.microsoft.com/en-us/healthvault/

[4]ISO/IEEE 11073-20601. Health informatics — Personal health device communication —Part 20601: Application profile — Optimized exchange protocol

[5]ISO/IEEE 11073-10101. Health informatics — Point-of-care medical device communication —Part 10101: Nomenclature

[6]Bluetooth SIG: Personal Health Devices transcoding whitepaper.