
# AGL Message Signaling

## Developer Guidelines

Version 1.0

July 2016

# Abstract

This document is a complement to another document "**AGL Message Signaling Architecture**". It gives some development guidelines through the example of a GPS signaling agent.

# Document revisions

| Date | Version | Designation | Author |
|---|---|---|---|
| 8 Jul 2016 | 0.1 | Initial release | S. Desneux [ Iot.bzh ] |
| 19 Jul 2016 | 0.2 | Second release | José Bollo [ Iot.bzh ] |
| 20 Jul 2016 | 1.0 | Review for delivery | S. Desneux [ Iot.bzh ]<br>Y. Gickel [ Iot.bzh ] |
| | | | |

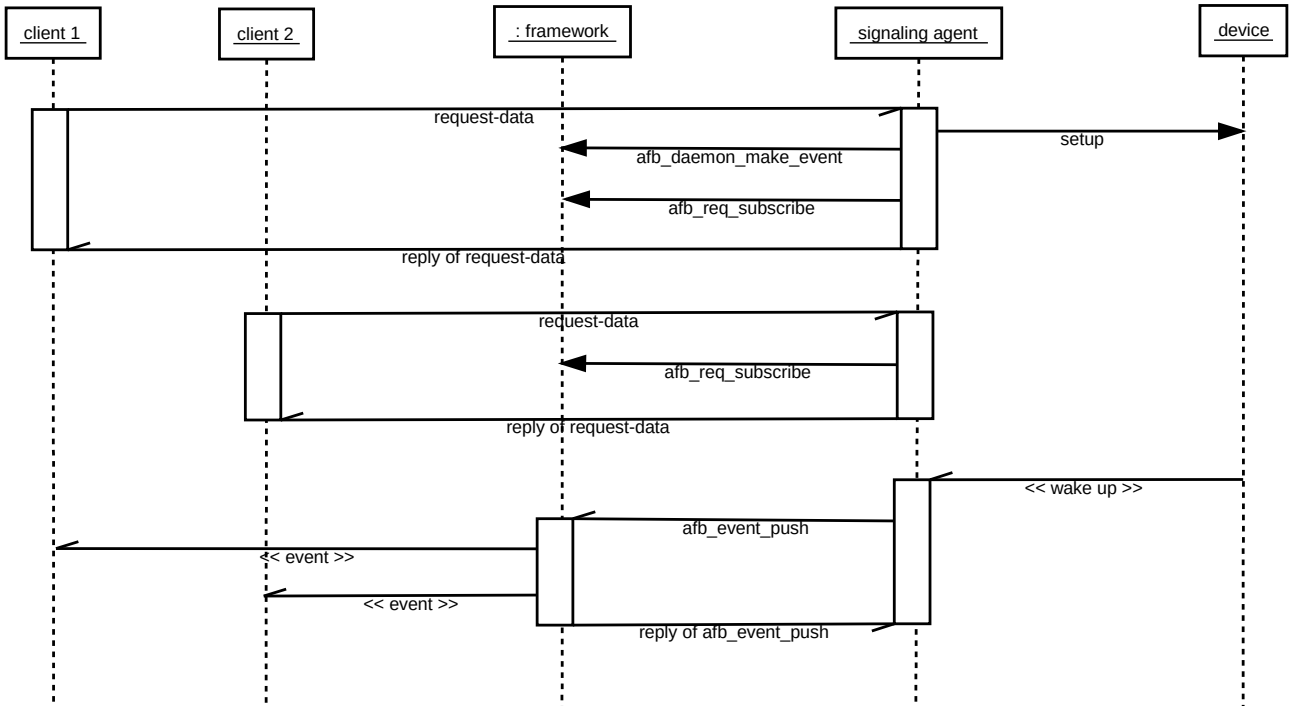
# Table of contents

# 1. Developing signaling agents

Signaling agents are services that send data to the clients that subscribed for receiving these data.

To have a good understanding of how to write a signaling agent, the actions of subscribing, unsubscribing, producing, sending, receiving data must be described and explained.

The basis of a signaling agent is shown on the following figure:




This figure shows the main role of the signaling framework for the propagation of events.

This document describes signaling within the AGL Application Framework as it exists in July 2016. However the content of the document is of more general interest. For people not familiar with the framework, a signaling agent and a “binding” are similar.

## 1.1. Subscribing and unsubscribing

Subscribing and subscription is the action that makes a client able to receive data from a signaling agent. Subscription must create resources for generating the data and for delivering the data to the client. These two aspects are not handled by the same piece of software: generating the data is the responsibility of the developer of the signaling agent while delivering the data is handled by the framework.

When a client subscribes for data, the agent must:

1. check that the subscription request is correct;
2. establish the computation chain of the required data, if not already done;
3. create a named event for the computed data, if not already done;
4. ask the framework to establish the subscription to the event for the request;
5. optionally give indications about the event in the reply to the client.

The first two steps are not involving the framework. They are linked to the business logic of the binding. The request can be any description of the requested data and the computing stream can be of any nature, this is specific to the binding.

However, the framework uses and integrates "libsystemd" and its event loop. This is the standard API for bindings expecting to setup and handle I/O, timer or signal events.

Steps 3 and 4 are bound to the framework.

The agent must create an object for handling the propagation of produced data to its clients. That object is called "event" in the framework. An event has a name that allows clients to distinguish it from other events.

Events are created using the **afb_daemon_make_event** function that takes the name of the event. Example:

```
event = afb_daemon_make_event(afb_daemon, name);
```

Once created, the event can be used either to push data to its subscribers or to broadcast data to any listener.

The event must be used to establish the subscription for the requesting client. This is done using the **afb_req_subscribe** function that takes the current request object and event and associates them together. Example:

```
rc = afb_req_subscribe(afb_req, event);
```

When successful, this function make the connection between the event and the client that emitted the request. The client becomes a subscriber of the event until it unsubscribes or disconnects. The **afb_req_subscribe** function will fail if the client connection is weak: if the request comes from a HTTP link. To receive signals, the client must be connected. The AGL framework allows connections using WebSocket.

The name of the event is either a well known name or an ad hoc name forged for the usecase.

Let's see a basic example: client A expects to receive the speed in km/h every second while client B expects the speed in mph twice a second. In that case, there are two different events because it is not the same unit and it is not the same frequency. Having two different events allows to associate clients to the correct event. But this doesn't tell any word about the name of these events. The designer of the signaling agent has two options for naming:

1. names can be the same (“speed” for example) with sent data self-describing itself or having a specific tag (requiring from clients awareness about requesting both kinds of speed isn't safe).
2. names of the event include the variations (by example: “speed-km/h-1Hz” and “speed-mph-2Hz”) and, in that case, sent data can self-describe itself or not.

In both cases, the signaling agent might have to send the name of the event and/or an associated tag to its client in the reply of the subscription. This is part of the step 5 above.

The framework only uses the event (not its name) for subscription, unsubscription and pushing.

When the requested data is already generated and the event used for pushing it already exists, the signaling agent must not instantiate a new processing chain and must not create a new event object for pushing data. The signaling agent must reuse the existing chain and event.

Unsubscribing is made by the signaling agent on a request of its client. The `afb_req_unsubscribe` function tells the framework to remove the requesting client from the event's list of subscribers. Example:

```
afb_req_unsubscribe(afb_req, event);
```

Subscription count does not matter to the framework: subscribing the same client several times has the same effect that subscribing only one time. Thus, when unsubscribing is invoked, it becomes immediately effective.

### 1.1.1. More on naming events

Within the AGL framework, a signaling agent is a binding that has an API prefix. This prefix is meant to be unique and to identify the binding API. The names of the events that this signaling agent creates are automatically prefixed by the framework, using the API prefix of the binding.

Thus, if a signaling agent of API prefix `api` creates an event of name `event` and pushes data to that event, the subscribers will receive an event of name `api/event`.

## 1.2. Generating and pushing signals and data

This of the responsibility of the designer of the signaling agent to establish the processing chain for generating events. In many cases, this can be achieved using I/O or timer or signal events inserted in the main loop. For this case, the AGL framework uses "libsystemd" and provide a way to integrates to the main loop of this library using afb_daemon_get_event_loop. Example:

```
sdev = afb_daemon_get_event_loop(af_daemon);
rc = sd_event_add_io(sdev, &source, fd, EPOLLIN, myfunction, NULL);
```

In some other cases, the events are coming from D-Bus. In that case, the framework also uses "libsystemd" internally to access D-Bus. It provides two methods to get the available D-Bus objects, already existing and bound to the main libsystemd event loop. Use either `afb_daemon_get_system_bus` or `afb_daemon_get_user_bus` to get the required instance. Then use functions of "libsystemd" to handle D-Bus.

In some rare cases, the generation of the data requires to start a new thread. The framework doesn't provide any help for that.

When a data is generated and ready to be pushed, the signaling agent should call the function `afb_event_push`. Example:

```
rc = afb_event_push(event, json);
if (rc == 0) {
      stop_generating(event);
      afb_event_drop(event);
}
```

The function `afb_event_push` pushes json data to all the subscribers. It then returns the count of subscribers. When the count is zero, there is no subscriber listening for the event. The example above shows that in that case, the signaling agent stops to generate data for the event and delete the event using afb_event_drop. This is one possible option. Other valuable options are: do nothing and continue to generate and push the event or just stop to generate and push the data but keep the event existing.

## 1.3. Receiving the signals

Understanding what a client expects when it receives signals, event, data shall be the most important topic of the designer of a signaling agent. The good point here is that because JSON[1] is the exchange format, structured data can be sent in a flexible way.

The good design is to allow as much as possible the client to describe what is needed with the goal to optimize the processing to the requirements only.

## 1.4. The exceptional case of wide broadcast

Some data or events have so much importance that they can be widely broadcasted to alert any listening client. Examples of such an alert are:

- system is entering/leaving "power safe" mode
- system is shutting down
- the car starts/stops moving
- …

An event can be broadcasted using one of the two following methods: `afb_daemon_broadcast_event` or `afb_event_broadcast`.

Example 1:

```
afb_daemon_broadcast_event(afb_daemon, name, json);
```

Example 2:

```
event = afb_daemon_make_event(afb_daemon, name);
…
afb_event_broadcast(event, json);
```

As for other events, the name of events broadcasted using `afb_daemon_broadcast_event` are automatically prefixed by the framework with API prefix of the binding (signaling agent).

1 There are two aspect in using JSON: the first is the flexible data structure that mixes common types (booleans, numbers, strings, arrays, dictionaries, nulls), the second, is the streaming specification. Streaming is often seen as the bottleneck of using JSON (see http://bjson.org). When the agent share the same process, there is no streaming at all.

# 2. Example: NMEA GPS signaling agent

## 2.1. Overview

The NMEA GPS agent listens to a NMEA server that produces GPS data. The GPS data received are made available to clients of the signaling agent.

Clients can query the last known GPS position at any time using the "get" method.

Example of answer to the "gps/get" query:

```
{
      "response": {
            "type":"WGS84",
            "time":76994000,
            "latitude":47.410213545797532,
            "longitude":357.04750632886282,
            "speed":6.7083555549760003
            },
      "jtype":"afb-reply",
      "request":{"status":"success"}
}
```

When the client queries the last known position, it can specify the type of the position it requires. The NMEA GPS signaling agent offers four different types:

| type | longitude & latitude | speed | altitude | track |
|---|---|---|---|---|
| WGS84 | degree | m/s | meter | degree |
| DMS.km/h | deg°min'sec"... | km/h | | |
| DMS.mph | | mph | | |
| DMS.kn | | kn | | |

A client can also subscribe to be periodically notified of the position. The subscription can specify the type of position expected and the period in milliseconds between two notifications.

Note that the agent does not send the data parts of the position that are missing. In the returned example, the data for altitude and track (heading) are missing.

Example of answer to the "gps/subscribe" subscription query:

```
{
      "response":{"name":"GPS","id":1},
      "jtype":"afb-reply",
      "request":{"status":"success"}
}
```

This reply includes the event name (“GPS”) and a numeric id that must be used when unsubscribing.

Notifications of the position are events. Example of received event:

```
{
      "event":"gps/GPS",
      "data": {
            "type":"WGS84",
            "time":78021000,
            "latitude":47.361904282981413,
            "longitude":357.06160208259365,
            "speed":7.119911110496
      },
      "jtype":"afb-event"
}
```

The agent implements basic logic for delivering events: when a new position arrives to the NMEA sockets, the clients whose period is expired receive the notification. So the period of subscription is the minimal period between two notifications.

A client can unsubscribe to an event using the numeric id received during subscription.

## 2.2. Code review

The code implements a simple agent that scans NMEA data and request in the same thread. For agent harder constraints of service, multiple thread have to be started and synchronized.

The code can be found as described in section 3.1. The source file is located in `af-gps-binding/src/af-gps-binding.c`. This source file is made of 5 sections:

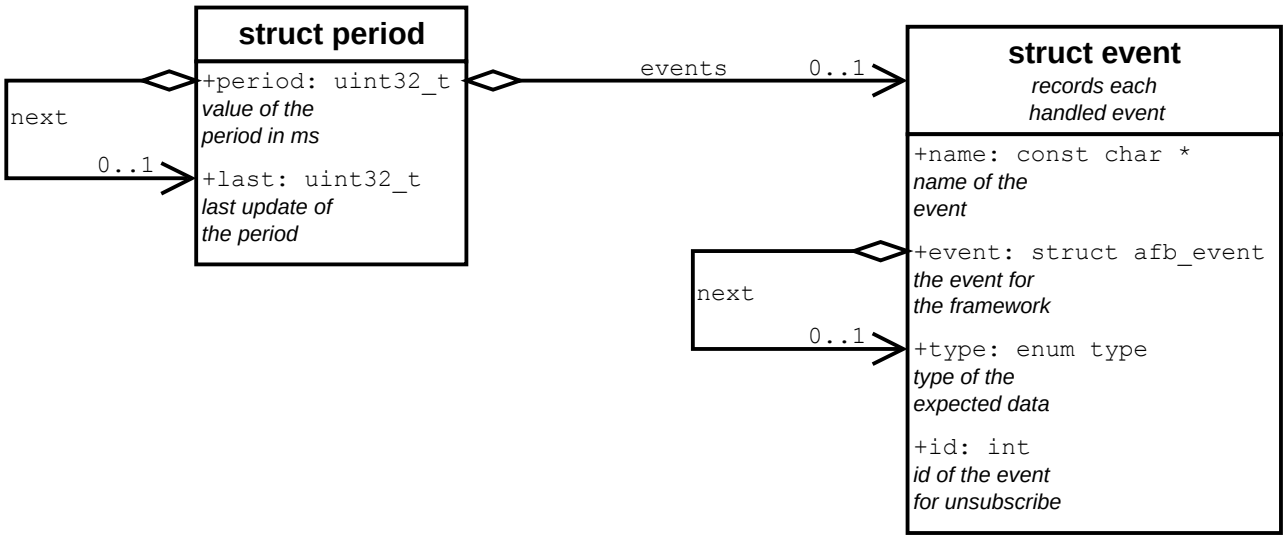
- FORMATTING JSON POSITIONS
- MANAGING EVENTS
- HANDLING NMEA
- HANDLING OF CONNECTION
- BINDING VERBS IMPLEMENTATION

This is a standard binding of the application framework's binder. This document will not explain how to write such a binding. To have this background knowledge, we recommend the reading of chapter 7 “How to write a plugin for afb-daemon” in document **Document-AppFW-Core.pdf** (see the delivery PO4510136005)[2].

2 This chapter is also available in HTML format at this URI: https://gerrit.automotivelinux.org/gerrit/gitweb?p=src/app-framework-binder.git;a=blob_plain;f=doc/afb-bindings-writing.html;hb=refs/heads/master

### 2.2.1. Data structure

The structure is summarized on the figure below.




The agent manages a list of periods. Each period is different. Attached to each period, a list of events records the expected type for the period and the data needed to push the event and to unsubscribe to it: the event of type struct afb_event and its numeric id.

This defines exactly one event for a given couple of period and type.

Using an id for the event is thus not mandatory because the event can be retrieved using its period and its type.

### 2.2.2. Handling NMEA events

At first, the GPS agent connects itself to the NMEA server using a socket.

For this simple example, the default configuration is to connect to the sinagot.net server at port 5001. This server is a fake one, that emits AIS and NMEA frames for boats. Its rate of GPS positions is low but enough for demonstration.

By setting the environment variables **`AFBGPS_HOST`** and **`AFBGPS_SERVICE`**, you can change the server and the port that the agent will use. The agent expects a kind of GPSD server to be used thus it sends a frame to commute the service to NMEA. This can be removed by defining the environment variable **`AFBGPS_ISNMEA`**.

The **`connect_to`** function defined line 844 is the main function for the connection:

```
/*
 * connection to nmea stream for the host and the port
 */
```

```
static int connect_to(const char *host, const char *service, int isgpsd)
{
	sd_event_source *source;
	int rc, fd;

	/* TODO connect to somewhere else */
	fd = open_socket_to(host, service);
	if (fd < 0) {
		ERROR(afbitf, "can't connect to host %s, service %s", host, service);
		return fd;
	}
	if (isgpsd) {
		static const char gpsdsetup[] = "?WATCH={\"enable\":true,\"nmea\":true};\r\n";
		write(fd, gpsdsetup, sizeof gpsdsetup - 1);
	}

	/* adds to the event loop */
	rc = sd_event_add_io(afb_daemon_get_event_loop(afbitf->daemon), &source, fd, EPOLLIN,
on_event, NULL);
	if (rc < 0) {
		close(fd);
		ERROR(afbitf, "can't connect host %s, service %s to the event loop", host, service);
	} else {
		NOTICE(afbitf, "Connected to host %s, service %s", host, service);
	}
	return rc;
}
```

When the socket is created, its file descriptor is added to the main event loop. The line doing that addition is shown in bold above. As it was said before, the event loop provided by the binder is provided by "libsystemd". The function `afb_daemon_get_event_loop` retrieves that event loop and the function `sd_event_add_io` of libsystemd adds an I/O event handling readable status of the socket handled by the function `on_event` defined line 788.

```
/*
 * called on an event on the NMEA stream
 */
static int on_event(sd_event_source *s, int fd, uint32_t revents, void *userdata)
{
	/* read available data */
	if ((revents & EPOLLIN) != 0) {
		nmea_read(fd);
		event_send();
	}

	/* check if error or hangup */
	if ((revents & (EPOLLERR|EPOLLRDHUP|EPOLLHUP)) != 0) {
		sd_event_source_unref(s);
		close(fd);
		connection(fd);
	}

	return 0;
}
```

When data comes on the NMEA socket, the function `on_event` is called. It will call the function `nmea_read` followed by a call to the function `event_send`.

The function `nmea_read` reads NMEA stream. Note that the function that opened the socket (`open_socket_to`) configured that socket to be non-blocking. This read is non-

blocking. When a full frame is read, only NMEA sentences GGA and RMC[3] are treated and processed to record the position using the function `nmea_set`.

### 2.2.3. Sending events

The function event_send is defined line 421.

```
/*
 * Sends the events if needed
 */
static void event_send()
{
	struct period *p, **pp;
	struct event *e, **pe;
	struct timeval tv;
	uint32_t now;

	/* skip if nothing is new */
	if (!newframes)
		return;

	/* computes now */
	gettimeofday(&tv, NULL);
	now = (uint32_t)(tv.tv_sec * 1000) + (uint32_t)(tv.tv_usec / 1000);

	/* iterates over the periods */
	pp = &list_of_periods;
	p = *pp;
	while (p != NULL) {
		if (p->events == NULL) {
			/* no event for the period, frees it */
			*pp = p->next;
			free(p);
		} else {
			if (p->period <= now - p->last) {
				/* its time to refresh */
				p->last = now;
				pe = &p->events;
				e = *pe;
				while (e != NULL) {
					/* sends the event */
					if (afb_event_push(e->event, position(e->type)) != 0)
						pe = &e->next;
					else {
						/* no more listeners, free the event */
						*pe = e->next;
						afb_event_drop(e->event);
						free(e);
					}
					e = *pe;
				}
			}
			pp = &p->next;
		}
		p = *pp;
	}
}
```

This function does nothing if no new position was set (test on newframes).

When a new position is available (newframes), it iterates over the list periods. When a period expired, the events recorded for that period are generated using the function `afb_event_push` and the function `position` for the data to be sent. The function

3 For definitions of NMEA sentences, see http://www.gpsinformation.org/dale/nmea.htm or https://en.wikipedia.org/wiki/NMEA_0183

`position` is optimized to compute on needed only the required data for the given type.

The function `afb_event_push` is defined as below:

```
/*
 * Pushes the 'event' with the data 'object' to its observers.
 * 'object' can be NULL.
 *
 * For convenience, the function calls 'json_object_put' for 'object'.
 * Thus, in the case where 'object' should remain available after
 * the function returns, the function 'json_object_get' shall be used.
 *
 * Returns the count of clients that received the event.
 */
int afb_event_push(struct afb_event event, struct json_object *object);
```

As the function `afb_event_push` returns 0 when there is no more subscriber, the loop removes such unexpected event using the function `afb_event_drop`. Dropping the events may lead to period without event. These empty periods are removed the next time that a function is entered.

The function `afb_event_drop` is defined as below:

```
/*
 * Drops the data associated to the event
 * After calling this function, the event
 * MUST NOT BE USED ANYMORE.
 */
void afb_event_drop(struct afb_event event);
```

## 2.2.4. Subscribing and unsubscribing

The binding verb "subscribe" is implemented by the function `subscribe` defined line 965:

```
/*
 * subscribe to notification of position
 *
 * parameters of the subscription are:
 *
 *    type:   string:  the type of position expected (defaults to WCS84 if not present)
 *                     see the list above (get)
 *    period: integer: the expected period in milliseconds (defaults to 2000 if not present)
 *
 * returns an object with 2 fields:
 *
 *    name:   string:  the name of the event without its prefix
 *    id:     integer: a numeric identifier of the event to be used for unsubscribing
 */
static void subscribe(struct afb_req req)
{
      enum type type;
      const char *period;
      struct event *event;
```

```
	struct json_object *json;

	if (get_type_for_req(req, &type)) {
		period = afb_req_value(req, "period");
		event = event_get(type, period == NULL ? DEFAULT_PERIOD : atoi(period));
		if (event == NULL)
			afb_req_fail(req, "out-of-memory", NULL);
		else if (afb_req_subscribe(req, event->event) != 0)
			afb_req_fail_f(req, "failed", "afb_req_subscribe returned an error: %m");
		else {
			json = json_object_new_object();
			json_object_object_add(json, "name", json_object_new_string(event->name));
			json_object_object_add(json, "id", json_object_new_int(event->id));
			afb_req_success(req, json, NULL);
		}
	}
}
```

The function subscribe calls the function **event_get** to retrieve or create the structure recording event metadata for the requested period and type. When the structure is obtained, it is used to register the subscription of the requesting client to that event using the function **afb_req_subscribe**.

The function **afb_req_subscribe** is defined as below:

```
/*
 * Establishes for the client link identified by 'req' a subscription
 * to the 'event'.
 * Returns 0 in case of successful subscription or -1 in case of error.
 */
int afb_req_subscribe(struct afb_req req, struct afb_event event);
```

The function **event_get** is defined line 352. Its role is to search for an existing structure recording the event defined for the period and the type requested. If a such event is not existing, it creates it. Here is the extract that creates that structure (variable e) when needed:

```
	/* creates the type if needed */
	if (e == NULL) {
		e = calloc(1, sizeof *e);
		if (e == NULL)
			return NULL;

		e->name = "GPS"; /* TODO */
		e->event = afb_daemon_make_event(afbitf->daemon, e->name);
		if (e->event.itf == NULL) {
			free(e);
			return NULL;
		}

		e->next = p->events;
		e->type = type;
		do {
			id++;
			if (id < 0)
				id = 1;
		} while(event_of_id(id) != NULL);
		e->id = id;
		p->events = e;
	}
```

The event is created using the function **afb_daemon_make_event** that is defined as below:

```
/*
 * Creates an event of 'name' and returns it.
 * 'daemon' MUST be the daemon given in interface when activating the binding.
 */
struct afb_event afb_daemon_make_event(struct afb_daemon daemon, const char
*name);
```

The binding verb "unsubscribe" is implemented by the function **unsubscribe** defined line 995:

```
/*
 * unsubscribe a previous subscription
 *
 * parameters of the unsubscription are:
 *
 *    id:   integer: the numeric identifier of the event as returned when subscribing
 */
static void unsubscribe(struct afb_req req)
{
       const char *id;
       struct event *event;

       id = afb_req_value(req, "id");
       if (id == NULL)
              afb_req_fail(req, "missing-id", NULL);
       else {
              event = event_of_id(atoi(id));
              if (event == NULL)
                     afb_req_fail(req, "bad-id", NULL);
              else {
                     afb_req_unsubscribe(req, event->event);
                     afb_req_success(req, NULL, NULL);
              }
       }
}
```

The structure recording the event is retrieved using its id by the function **event_of_id**. Then the function afb_req_unsubscribe is called to remove the subscription to the event for the client of the request.

The function **afb_req_unsubscribe** is defined as below:

```
/*
 * Revokes the subscription established to the 'event' for the client
 * link identified by 'req'.
 * Returns 0 in case of successful subscription or -1 in case of error.
 */
int afb_req_unsubscribe(struct afb_req req, struct afb_event event);
```

The unsubscription removes one client of the event. When the count of client decreases to 0, it makes the function **afb_event_push** returning zero. At the end, this will drop the event as seen when looking to function **event_send**.

# 3. Building, installing and running the GPS agent

For this section, we present building, installing and running the GPS agent in the context described by the documents "AGL Devkit – Image and SDK for Porter" and "AGL Devkit – Build your 1st AGL Application".

The workspace in this context designates the directory shared between the host and the docker instance of AGL Devkit, as described in "3.5. Set up a persistent workspace" of the document "AGL Devkit – Image and SDK for Porter".

In the following examples of commands, the typed text is written in bold.

## 3.1. Getting the GPS agent

From the delivery PO-4510369035-AGL-Phase2-AppFW-SDK, upload the archive file of name **09-AGL-Message-Signaling-GPS-Agent.tgz** in your workspace and untar it using the commands:

```
$ cd /xdt/workspace
$ tar zxf 09-AGL-Message-Signaling-GPS-Agent.tgz
```

It should have the following content:

```
$ find af-gps-binding
af-gps-binding
af-gps-binding/.gitignore
af-gps-binding/CMakeLists.txt
af-gps-binding/README.md
af-gps-binding/src
af-gps-binding/src/CMakeLists.txt
af-gps-binding/src/af-gps-binding.c
af-gps-binding/src/export.map
```

## 3.2. Build the GPS agent

First the build directory is created:

```
$ mkdir /xdt/workspace/af-gps-binding/build
```

Then the build environment is prepared using the following commands:

```
$ cd /xdt/workspace/af-gps-binding/build
$ source /xdt/sdk/environment-setup-cortexa15hf-vfp-neon-poky-linux-gnueabi
```

```
$ cmake ..
-- The C compiler identification is GNU 5.2.0
-- Check for working C compiler: /xdt/sdk/sysroots/x86_64-pokysdk-
linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc
-- Check for working C compiler: /xdt/sdk/sysroots/x86_64-pokysdk-
linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Found PkgConfig: /xdt/sdk/sysroots/x86_64-pokysdk-linux/usr/bin/pkg-config
(found version "0.28")
-- checking for modules 'json-c;libsystemd;afb-daemon'
--   found json-c, version 0.12
--   found libsystemd, version 225
--   found afb-daemon, version 1.0
-- Creation af-gps-binding for AFB-DAEMON
-- Configuring done
-- Generating done
-- Build files have been written to: /xdt/workspace/af-gps-binding/build
```

Now we can build the plug-in:

```
$ make
Scanning dependencies of target af-gps-binding
[ 50%] Building C object src/CMakeFiles/af-gps-binding.dir/af-gps-binding.c.o
[100%] Linking C shared module af-gps-binding.so
[100%] Built target af-gps-binding
```

## 3.3. Installing the GPS agent

Assuming your target board is up and running and the environment variable BOARDIP has been set to its network IP address, we can now deploy the GPS agent using the following commands:

```
$ export BOARDIP=1.2.3.4
$ scp /xdt/workspace/af-gps-binding/build/src/af-gps-binding.so \
root@$BOARDIP:~/
```

For the test, we will also require a WebSocket command line client, which can be deployed using:

```
$ scp /xdt/build/tmp/work/cortexa15hf-vfp-neon-poky-linux-gnueabi/af-
binder/1.0-r0/build/src/afb-client-demo root@$BOARDIP:/usr/bin
```

## 3.4. Running the GPS agent

To test the GPS agent, we can connect to the target board:

```
$ ssh root@$BOARDIP
```

Note: The GPS agent example plug-in is able to connect to a specific NMEA server by specifying the following environment variables (we will not use those variables in the following, but this can be useful if required to point on a dedicated set of GPS data).

```
# export AFBGPS_HOST=hostname-or-ip
# export AFBGPS_SERVICE=port-or-service
# export AFBGPS_ISNMEA
```

Now, the next step proposed is to launch a stand-alone binder which will provide the service:

```
# cd $HOME
# afb-daemon --binding ./af-gps-binding.so --port 12345 --token hello \
--ldpaths /tmp &
[1] 658
# NOTICE: binding [./af-gps-binding.so] calling registering function
afbBindingV1Register
NOTICE: binding ./af-gps-binding.so loaded with API prefix gps
NOTICE: Waiting port=12345 rootdir=/home/root/.AFB
NOTICE: Browser URL= http:/*localhost:12345
NOTICE: Connected to host sinagot.net, service 5001 {binding gps}
#
```

As the service binder is launched, we can test the agent by using the client test application which will allow to perform requests on the WebSocket from the console:

```
$ afb-client-demo localhost:12345/api?token=hello
```

Note: This test client can be exited by pressing CTRL+C.

At this point, we can perform a single position query, and observe the corresponding answer from the agent:

```
gps get {"type":"DMS.km/h"}
ON-REPLY 1:gps/get: {"response":{"type":"DMS.km/h","time":80172000,
     "latitude":"47°18'5.141\"N","longitude":"2°50'53.044\"W",
     "speed":26.446559997715202},"jtype":"afb-reply","request":{
     "status":"success","uuid":"7f8e4122-3bf0-43a4-a7fe-7ac6adec3575"}}
```

We can also subscribe to the signaling agent in order to get positions data periodically, using:

```
gps subscribe {"type":"DMS.km/h","period":2000}
ON-REPLY 2:gps/subscribe: {"response":{"name":"GPS","id":2},
     "jtype":"afb-reply","request":{"status":"success"}}

ON-EVENT gps/GPS({"event":"gps/GPS","data":{"type":"DMS.km/h",
     "time":8018600,"latitude":"47°18'0.371\"N","longitude":"2°50'46.756\"W",
     "speed":17.871799998456002},"jtype":"afb-event"})

ON-EVENT gps/GPS({"event":"gps/GPS","data":{"type":"DMS.km/h",
     "time":8019700,"latitude":"47°17'59.061\"N","longitude":"2°50'44.246\"W",
     "speed":25.8168799977696},"jtype":"afb-event"})
[snip]
```

To unsubscribe the client from the signaling agent, we can type:

```
gps unsubscribe {"id":2}

ON-REPLY 4:gps/unsubscribe: {"jtype":"afb-reply",
     "request":{"status":"success"}}
```

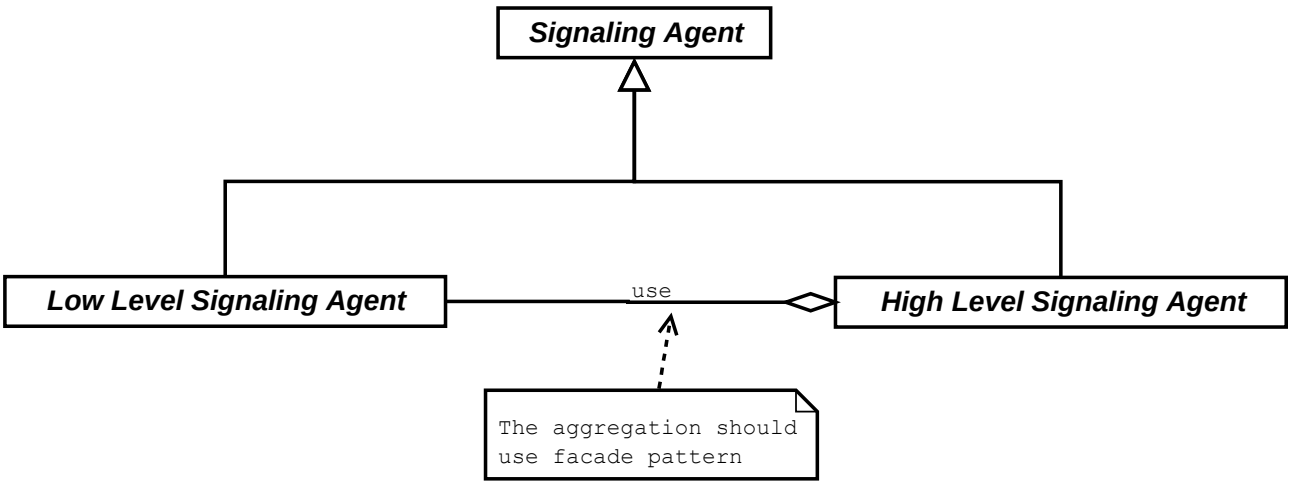
# 4. Architectural digressions

Based on their dependencies to hardware, signaling agents can be split into 2 categories: low-level signaling agents and high-level signaling agents.

Low-level signaling agents are bound to the hardware and focused on interfacing and driving.

High-level signaling agent are independent of the hardware and ocused on providing service.

This separation (that may in the corner look artificial) aim to help in the systems design. The main idea here is that high-level signaling agents are providing "business logic", also known as "application logic", that is proper to the car industry and that can be reused and that can evolve as a foundation for the future of the industry.




The implementation of this decomposition may follow 2 paths: strict separation or soft composition.

## 4.1. Strict separation

The strict separation implements the modularity composition of signaling agent through the framework. The high-level signaling agent subscribes to the low level signaling agent using the standard client API.

Advantages:

- Modularity
- Separation of responsibilities

- Possible aggregation of multiple sources
- Soft binding of agent good for maintenance

Drawbacks:

- Cost of propagation of data (might serialize)
- Difficulties to abstract low-level signaling agent or to find a trade-of between abstracting and specializing

The key is modularity versus cost of propagation. It can be partly solved when logical group of signaling agent are launched together in the same binder process. In that particular case, the cost of propagation of data between agents is reduced[4] because there is no serialization.

This reduction of the propagation cost (and of the resources used) precludes implementation of strong security between the agents because they share the same memory.

## 4.2. Soft composition

The soft composition implements the business logic of high-level signaling agents as libraries that can then be used directly by the low level signaling agents.

Advantages:

- No propagation: same memory, sharing of native structures

Drawbacks:

- Cannot be used for aggregation of several sources
- Difficulties to abstract low-level signaling agent or to find a trade-of between abstracting and specializing
- Source code binding not good for maintenance

4 Within the same process, there is not serialization, the propagation has the cost of wrapping a json data and calling callbacks with the benefit of having a powerful callback manager: the event mechanism of the framework.