vsomeip

Contents

1 Copyright

2 License

3 Version

This documentation was generated for version 2.0.1 of vsomeip.

4 vsomeip Overview

The vsomeip stack implements the Scalable service-Oriented MiddlewarE over IP (SOME/IP) protocol. The stack consists out of:

- a shared library for SOME/IP (`libvsomeip.so`)
- a second shared library for SOME/IP's service discovery (`libvsomeip-sd.so`) which is loaded during runtime if the service discovery is enabled.

5 Build Instructions

5.1 Dependencies

- A C++11 enabled compiler like gcc >= 4.8 is needed.
- vsomeip uses cmake as buildsystem.
- vsomeip uses Boost >= 1.54:
  - Ubuntu 14.04:
    - `sudo apt-get install libboost-system1.54-dev libboost-thread1.54-dev libboost-log1.54-dev`
  - Ubuntu 12.04: a PPA is necessary to use version 1.54 of Boost:
    - URL: https://launchpad.net/~boost-latest/+archive/ubuntu/ppa
    - `sudo add-apt-repository ppa:boost-latest/ppa`
    - `sudo apt-get install libboost-system1.54-dev libboost-thread1.54-dev libboost-log1.54-dev`
- For the tests Google's test framework gtest in version 1.7.0 is needed
  - URL: direct link, version 1.7.0
- To build the documentation asciidoc, source-highlight, doxygen and graphviz is needed:
  - `sudo apt-get install asciidoc source-highlight doxygen graphviz`

5.2 Compilation

For compilation call:

```
mkdir build
cd build
cmake ..
make
```

To specify a installation directory (like `--prefix=` if you're used to autotools) call cmake like:

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=$YOUR_PATH ..
make
make install
```

5.2.1 Compilation of examples

For compilation of the examples call:

```
mkdir build
cd build
cmake ..
make examples
```

5.2.2 Compilation of tests

To compile the tests, first unzip gtest to location of your desire. Some of the tests require a second node on the same network. There are two cmake variables which are used to automatically adapt the json files to the used network setup:

- `TEST_IP_MASTER`: The IP address of the interface which will act as test master.
- `TEST_IP_SLAVE`: The IP address of the interface of the second node which will act as test slave.

If one of this variables isn't specified, only the tests using local communication exclusively will be runnable.

Example, compilation of tests:

```
mkdir build
cd build
export GTEST_ROOT=$PATH_TO_GTEST/gtest-1.7.0/
cmake -DTEST_IP_MASTER=10.0.3.1 -DTEST_IP_SLAVE=10.0.3.125 ..
make check
```

Additional make targets for the tests:

- Call `make build_tests` to only compile the tests
- Call `ctest` in the build directory to execute the tests without a verbose output
- To run single tests call `ctest --verbose --tests-regex $TESTNAME` short form: `ctest -V -R $TESTNAME`
- To list all available tests run `ctest -N`.
- For further information about the tests please have a look at the `readme.txt` in the `test` subdirectory.

For development purposes two cmake variables exist which control if the json files and test scripts are copied (default) or symlinked into the build directory. These settings are ignored on Windows.

- TEST_SYMLINK_CONFIG_FILES: Controls if the json and scripts needed to run the tests are copied or symlinked into the build directory. (Default: OFF, ignored on Windows)
- TEST_SYMLINK_CONFIG_FILES_RELATIVE: Controls if the json and scripts needed to run the tests are symlinked relatively into the build directory. (Default: OFF, ignored on Windows)

Example cmake call:

```
cmake  -DTEST_SYMLINK_CONFIG_FILES=ON -DTEST_SYMLINK_CONFIG_FILES_RELATIVE=ON ..
```

For compilation of only a subset of tests (for a quick functionality check) the cmake variable TESTS_BAT has to be set:

Example cmake call:

```
cmake  -DTESTS_BAT=ON ..
```

5.2.3 Generating the documentation

To generate the documentation call cmake as described in [Compilation] and then call make doc. This will generate:

- The README file in html: $BUILDDIR/documentation/README.html
- A doxygen documentation in $BUILDDIR/documentation/html/index.html

6 Starting vsomeip Applications / Used environment variables

On startup the following environment variables are read out:

- VSOMEIP_APPLICATION_NAME: This environment variable is used to specify the name of the application. This name is later used to map a client id to the application in the configuration file. It is independent from the application's binary name.
- VSOMEIP_CONFIGURATION: vsomeip uses the default configuration file /etc/vsomeip.json and/or the default configuration folder /etc/vsomeip. This can be overridden by a local configuration file ./vsomeip.json and/or a local configuration folder ./vsomeip. If VSOMEIP_CONFIGURATION is set to a valid file or directory path, this is used instead of the standard configuration (thus neither default nor local file/folder will be parsed).

Note
If the file/folder that is configured by VSOMEIP_CONFIGURATION does *not* exist, the default configuration locations will be used.

Note
vsomeip will parse and use the configuration from all files in a configuration folder but will *not* consider directories within the configuration folder.

In the following example the application my_vsomeip_application is started. The settings are read from the file my_settings.json in the current working directory. The client id for the application can be found under the name my_vsomeip_client in the configuration file.

```
#!/bin/bash
export VSOMEIP_APPLICATION_NAME=my_vsomeip_client
export VSOMEIP_CONFIGURATION=my_settings.json
./my_vsomeip_application
```

7 Configuration File Structure

The configuration files for vsomeip are JSON-Files and are composed out of multiple key value pairs and arrays.

> - An object is an unordered set of name/value pairs. An object begins with `{ (left brace)` and ends with `} (right brace)`. Each name is followed by `:(colon)` and the name/value pairs are separated by `, (comma)`.
> - An array is an ordered collection of values. An array begins with `[ (left bracket)` and ends with `] (right bracket)`. Values are separated by `, (comma)`.
> - A value can be a *string* in double quotes, or a *number*, or `true` or `false` or `null`, or an *object* or an *array*. These structures can be nested.
>
> — *json.org*

Configuration file element explanation:

- *unicast*
  The IP address of the host system.
- *netmask*
  The netmask to specify the subnet of the host system.
- *logging*
  - *level*
    Specifies the log level (valid values: *trace*, *debug*, *info*, *warning*, *error*, *fatal*).
  - *console*
    Specifies whether logging via console is enabled (valid values: *true, false*).
  - *file*
    - *enable*
      Specifies whether a log file should be created (valid values: *true, false*).
    - *path*
      The absolute path of the log file.
  - *dlt*
    Specifies whether Diagnostic Log and Trace (DLT) is enabled (valid values: *true, false*).
- *applications (array)*
  Contains the applications of the host system that use this config file.
  - *name*
    The name of the application.
  - *id*
    The id of the application.
  - *num_dispatchers*
    The number of threads that shall be used to execute the callbacks to the application. If *num_dispatchers* is set to *0*, the callbacks will be executed within the application thread. If an application wants/must do time consuming work directly within event, availability or message callbacks, *num_dispatchers* should be set to *2* or higher.
  - `services` (array)
    Contains the services of the service provider.
    - `service`
      The id of the service.
    - `instance`
      The id of the service instance.

  * `protocol` (optional)
    The protocol that is used to implement the service instance. The default setting is *someip*. If a different setting is provided, vsomeip does not open the specified port (server side) or does not connect to the specified port (client side). Thus, this option can be used to let the service discovery announce a service that is externally implemented.
  * `unicast` (optional)
    The unicast that hosts the service instance.

    Note
    The unicast address is needed if external service instances shall be used, but service discovery is disabled. In this case, the provided unicast address is used to access the service instance.

  * `reliable`
    Specifies that the communication with the service is reliable respectively the TCP protocol is used for communication.
    · `port`
      The port of the TCP endpoint.
    · `enable-magic-cookies`
      Specifies whether magic cookies are enabled (valid values: *true*, *false*).
  * `unreliable`
    Specifies that the communication with the service is unreliable respectively the UDP protocol is used for communication (valid values: the *port* of the UDP endpoint).
  * `multicast`
    A service can be offered to a specific group of clients via multicast.
    · `address`
      The specific multicast address.
    · `port`
      The specific port.
  * `events` (array)
    Contains the events of the service.
    · `event`
      The id of the event. [Error: itemizedlist too deeply nested]
  * `eventgroups` (array)
    Events can be grouped together into on event group. For a client it is thus possible to subscribe for an event group and to receive the appropriate events within the group.
    · `eventgroup`
      The id of the event group.
    · `events` (array)
      Contains the ids of the appropriate events.
    · `is_multicast`
      Specifies whether the events should be sent via multicast (valid values: *true*,*false*).
    · `multicast`
      The multicast address which the events are sent to.

- `payload-sizes` (array)

  Array to specify the maximum allowed payload sizes per IP and port. If not specified, or a smaller value than the default values is specified, the default values are used. The settings in this array only affect communication over TCP and local communication over UNIX domain sockets.

  – `unicast`
    On client side: the IP of the remote service to which the oversized messages should be sent. On service side: the IP of the offered service which should receive the oversized messages and is allowed to respond with oversized messages. If client and service only communicate locally, any IP can be entered here as for local communication only the maximum specified payload size is relevant.
  – `ports` (array)
    Array which holds pairs of port and payload-size statements.

    * port
      On client side: the port of the remote service to which the oversized messages should be sent. On service side: the port of the offered service which should receive the oversized messages and is allowed to respond with oversized messages. If client and service only communicate locally, any port number can be entered.
    * max-payload-size
      On client side: the maximum payload size in bytes of a message sent to the remote service hosted on beforehand specified IP and port. On service side: the maximum payload size in bytes of messages received by the service offered on previously specified IP and port. If multiple services are hosted on the same port all of them are allowed to receive oversized messages and send oversized responses.

- routing

  The name of the application that is responsible for the routing.

- service-discovery

  Contains settings related to the Service Discovery of the host application.

  – enable
    Specifies whether the Service Discovery is enabled (valid values: *true*, *false*). The default value is *true*.
  – multicast
    The multicast address which the messages of the Service Discovery will be sent to. The default value is "224.0.0.1".
  – port
    The port of the Service Discovery. The default setting is 30490.
  – protocol
    The protocol that is used for sending the Service Discovery messages (valid values: *tcp*, *udp*). The default setting is *udp*.
  – initial_delay_min
    Minimum delay before first offer message.
  – initial_delay_max
    Maximum delay before first offer message.
  – repetitions_base_delay
    Base delay sending offer messages within the repetition phase.
  – repetitions_max
    Maximum number of repetitions for provided services within the repetition phase.
  – ttl
    Lifetime of entries for provided services as well as consumed services and eventgroups.
  – cyclic_offer_delay
    Cycle of the OfferService messages in the main phase.
  – request_response_delay
    Minimum delay of a unicast message to a multicast message for provided services and eventgroups.

8 Autoconfiguration

vsomeip supports the automatic configuration of client identifiers and the routing. The first application that starts using vsomeip will automatically become the routing manager if it is *not* explicitly configured. The client identifiers are generated from the diagnosis address that can be specified by defining DIAGNOSIS_ADDRESS when compiling vsomeip. vsomeip will use the diagnosis address as the high byte and enumerate the connecting applications within the low byte of the client identifier.

9 vsomeipd

The vsomeipd is a minimal vsomeip application intended to offer routing manager functionality on a node where one system wide configuration file is present.

The vsomeipd uses the application name `vsomeipd` by default. This name can be overridden by specifying `-DROUTING=$DESIRED_NAME` during the cmake call.

Example: Starting the daemon on a system where the system wide configuration is stored under `/etc/vsomeip.json`:

```
VSOMEIP_CONFIGURATION=/etc/vsomeip.json ./vsomeipd
```

When using the daemon it should be ensured that:

- In the system wide configuration file the vsomeipd is defined as routing manager, meaning it contains the line `"routing" : "vsomeipd"`. If the default name is overridden the entry has to be adapted accordingly. The system wide configuration file should contain the information about all other offered services on the system as well.
- There's no other vsomeip configuration file used on the system which contains a `"routing"` entry. As there can only be one routing manager per system.

10 vsomeip Hello World

In this paragraph a Hello World program consisting out of a client and a service is developed. The client sends a message containing a string to the service. The service appends the received string to the string `Hello` and sends it back to the client. Upon receiving a response from the service the client prints the payload of the response ("Hello World"). This example is intended to be run on the same host.

All files listed here are contained in the `examples\hello_world` subdirectory.

10.1 Build instructions

The example can build with its own CMakeFile, please compile the vsomeip stack before hand as described in [Compilation]. Then compile the example starting from the repository root directory as followed:

```
cd examples/hello_world
mkdir build
cd build
cmake ..
make
```

10.2 Starting and expected output

10.2.1 Starting and expected output of service

```
$ VSOMEIP_CONFIGURATION=../helloworld-local.json \
  VSOMEIP_APPLICATION_NAME=hello_world_service \
  ./hello_world_service
2015-04-01 11:31:13.248437 [info] Using configuration file: ../helloworld-local.json
2015-04-01 11:31:13.248766 [debug] Routing endpoint at /tmp/vsomeip-0
2015-04-01 11:31:13.248913 [info] Service Discovery disabled. Using static routing ↩
    information.
2015-04-01 11:31:13.248979 [debug] Application(hello_world_service, 4444) is initialized.
2015-04-01 11:31:22.705010 [debug] Application/Client 5555 got registered!
```

10.2.2 Starting and expected output of client

```
$ VSOMEIP_CONFIGURATION=../helloworld-local.json \
  VSOMEIP_APPLICATION_NAME=hello_world_client \
  ./hello_world_client
2015-04-01 11:31:22.704166 [info] Using configuration file: ../helloworld-local.json
2015-04-01 11:31:22.704417 [debug] Connecting to [0] at /tmp/vsomeip-0
2015-04-01 11:31:22.704630 [debug] Listening at /tmp/vsomeip-5555
2015-04-01 11:31:22.704680 [debug] Application(hello_world_client, 5555) is initialized.
Sending: World
Received: Hello World
```

10.3 CMakeFile

```
# Copyright (C) 2015 Bayerische Motoren Werke Aktiengesellschaft (BMW AG)
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.

cmake_minimum_required (VERSION 2.8.7)
project (vSomeIPHelloWorld)

# This will get us acces to
#   VSOMEIP_INCLUDE_DIRS - include directories for vSomeIP
#   VSOMEIP_LIBRARIES    - libraries to link against
find_package(vsomeip)
if (NOT vsomeip_FOUND)
    message("vsomeip was not found. Please specify vsomeip_DIR")
endif()

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")

include_directories(${VSOMEIP_INCLUDE_DIRS})

add_executable (hello_world_service hello_world_service.cpp)
target_link_libraries(hello_world_service ${VSOMEIP_LIBRARIES})

add_executable (hello_world_client hello_world_client.cpp)
target_link_libraries(hello_world_client ${VSOMEIP_LIBRARIES})
```

10.4 Configuration File For Client and Service

```
{
   "unicast" : "134.86.56.94",
   "logging" :
   {
      "level" : "debug",
      "console" : "true"
   },

   "applications" :
   [
      {
         "name" : "hello_world_service",
         "id" : "0x4444"
      },

      {
```

```
         "name" : "hello_world_client",
         "id" : "0x5555"
      }
   ],

   "servicegroups" :
   [
      {
         "name" : "default",
         "unicast" : "local",
         "services" :
         [
            {
               "service" : "0x1111",
               "instance" : "0x2222",
               "unreliable" : "30509"
            }
         ]
      }
   ],

   "routing" : "hello_world_service",
   "service-discovery" :
   {
      "enable" : "false"
   }
}
```

10.5 Service

```
// Copyright (C) 2015 Bayerische Motoren Werke Aktiengesellschaft (BMW AG)
// This Source Code Form is subject to the terms of the Mozilla Public
// License, v. 2.0. If a copy of the MPL was not distributed with this
// file, You can obtain one at http://mozilla.org/MPL/2.0/.

#include <vsomeip/vsomeip.hpp>
#include <chrono>
#include <thread>
#include <condition_variable>
#include <mutex>

static vsomeip::service_t service_id = 0x1111;
static vsomeip::instance_t service_instance_id = 0x2222;
static vsomeip::method_t service_method_id = 0x3333;

class hello_world_service {
public:
    // Get the vSomeIP runtime and
    // create a application via the runtime, we could pass the application name
    // here otherwise the name supplied via the VSOMEIP_APPLICATION_NAME
    // environment variable is used
    hello_world_service() :
                    rtm_(vsomeip::runtime::get()),
                    app_(rtm_->create_application()),
                    stop_(false),
                    stop_thread_(std::bind(&hello_world_service::stop, this))
    {
    }

    ~hello_world_service()
```

```
{
    stop_thread_.join();
}

void init()
{
    // init the application
    app_->init();

    // register a message handler callback for messages sent to our service
    app_->register_message_handler(service_id, service_instance_id,
            service_method_id,
            std::bind(&hello_world_service::on_message_cbk, this,
                    std::placeholders::_1));

    // register a state handler to get called back after registration at the
    // runtime was successful
    app_->register_state_handler(
            std::bind(&hello_world_service::on_state_cbk, this,
                    std::placeholders::_1));
}

void start()
{
    // start the application and wait for the on_event callback to be called
    // this method only returns when app_->stop() is called
    app_->start();
}

void stop()
{
    std::unique_lock<std::mutex> its_lock(mutex_);
    while(!stop_) {
        condition_.wait(its_lock);
    }
    std::this_thread::sleep_for(std::chrono::seconds(5));
    // Stop offering the service
    app_->stop_offer_service(service_id, service_instance_id);
    // unregister the state handler
    app_->unregister_state_handler();
    // unregister the message handler
    app_->unregister_message_handler(service_id, service_instance_id,
            service_method_id);
    // shutdown the application
    app_->stop();
}

void on_state_cbk(vsomeip::state_type_e _state)
{
    if(_state == vsomeip::state_type_e::ST_REGISTERED)
    {
        // we are registered at the runtime and can offer our service
        app_->offer_service(service_id, service_instance_id);
    }
}

void on_message_cbk(const std::shared_ptr<vsomeip::message> &_request)
{
    // Create a response based upon the request
    std::shared_ptr<vsomeip::message> resp = rtm_->create_response(_request);

    // Construct string to send back
```

```
        std::string str("Hello ");
        str.append(
                reinterpret_cast<const char*>(_request->get_payload()->get_data()),
                0, _request->get_payload()->get_length());

        // Create a payload which will be sent back to the client
        std::shared_ptr<vsomeip::payload> resp_pl = rtm_->create_payload();
        std::vector<vsomeip::byte_t> pl_data(str.begin(), str.end());
        resp_pl->set_data(pl_data);
        resp->set_payload(resp_pl);

        // Send the response back
        app_->send(resp, true);
        // we're finished stop now
        std::lock_guard<std::mutex> its_lock(mutex_);
        stop_ = true;
        condition_.notify_one();
    }

private:
    std::shared_ptr<vsomeip::runtime> rtm_;
    std::shared_ptr<vsomeip::application> app_;
    bool stop_;
    std::mutex mutex_;
    std::condition_variable condition_;
    std::thread stop_thread_;
};

int main(int argc, char **argv)
{
    hello_world_service hw_srv;
    hw_srv.init();
    hw_srv.start();
    return 0;
}
```

The service example results in the following program execution:

Main

1. *main()*

   First the application is initialized. After the initialization is finished the application is started.

Initialization

1. *init()*

   The initialization contains the registration of a message handler and an event handler.

   The message handler declares a callback (*on_message_cbk*) for messages that are sent to the specific service (specifying the service id, the service instance id and the service method id).

   The event handler declares a callback (*on_event_cbk*) for events that occur. One event can be the successful registration of the application at the runtime.

Start

1. *start()*

   The application will be started. This function only returns when the application will be stopped.

Callbacks

1. *on_state_cbk()*

   This function is called by the application when an state change occurred. If the application was successfully registered at the runtime then the specific service is offered.

2. *on_message_cbk()*

   This function is called when a message/request from a client for the specified service was received.

   First a response based upon the request is created. Afterwards the string *Hello* will be concatenated with the payload of the client's request. After that the payload of the response is created. The payload data is set with the previously concatenated string. Finally the response is sent back to the client and the application is stopped.

Stop

1. *stop()*

   This function stops offering the service, unregister the message and the event handler and shuts down the application.

10.6 Client

```
// Copyright (C) 2015 Bayerische Motoren Werke Aktiengesellschaft (BMW AG)
// This Source Code Form is subject to the terms of the Mozilla Public
// License, v. 2.0. If a copy of the MPL was not distributed with this
// file, You can obtain one at http://mozilla.org/MPL/2.0/.

#include <vsomeip/vsomeip.hpp>
#include <iostream>

static vsomeip::service_t service_id = 0x1111;
static vsomeip::instance_t service_instance_id = 0x2222;
static vsomeip::method_t service_method_id = 0x3333;

class hello_world_client {
public:
    // Get the vSomeIP runtime and
    // create a application via the runtime, we could pass the application name
    // here otherwise the name supplied via the VSOMEIP_APPLICATION_NAME
    // environment variable is used
    hello_world_client() :
                    rtm_(vsomeip::runtime::get()),
                    app_(rtm_->create_application())
    {
    }

    void init(){
        // init the application
        app_->init();

        // register a state handler to get called back after registration at the
        // runtime was successful
        app_->register_state_handler(
                std::bind(&hello_world_client::on_state_cbk, this,
                        std::placeholders::_1));

        // register a callback for responses from the service
        app_->register_message_handler(vsomeip::ANY_SERVICE,
                service_instance_id, vsomeip::ANY_METHOD,
                std::bind(&hello_world_client::on_message_cbk, this,
                        std::placeholders::_1));
```

```
    // register a callback which is called as soon as the service is available
    app_->register_availability_handler(service_id, service_instance_id,
            std::bind(&hello_world_client::on_availability_cbk, this,
                    std::placeholders::_1, std::placeholders::_2,
                    std::placeholders::_3));
}

void start()
{
    // start the application and wait for the on_event callback to be called
    // this method only returns when app_->stop() is called
    app_->start();
}

void on_state_cbk(vsomeip::state_type_e _state)
{
    if(_state == vsomeip::state_type_e::ST_REGISTERED)
    {
        // we are registered at the runtime now we can request the service
        // and wait for the on_availability callback to be called
        app_->request_service(service_id, service_instance_id);
    }
}

void on_availability_cbk(vsomeip::service_t _service,
        vsomeip::instance_t _instance, bool _is_available)
{
    // Check if the available service is the the hello world service
    if(service_id == _service && service_instance_id == _instance
            && _is_available)
    {
        // The service is available then we send the request
        // Create a new request
        std::shared_ptr<vsomeip::message> rq = rtm_->create_request();
        // Set the hello world service as target of the request
        rq->set_service(service_id);
        rq->set_instance(service_instance_id);
        rq->set_method(service_method_id);

        // Create a payload which will be sent to the service
        std::shared_ptr<vsomeip::payload> pl = rtm_->create_payload();
        std::string str("World");
        std::vector<vsomeip::byte_t> pl_data(std::begin(str), std::end(str));

        pl->set_data(pl_data);
        rq->set_payload(pl);
        // Send the request to the service. Response will be delivered to the
        // registered message handler
        std::cout << "Sending: " << str << std::endl;
        app_->send(rq, true);
    }
}

void on_message_cbk(const std::shared_ptr<vsomeip::message> &_response)
{
    if(service_id == _response->get_service()
            && service_instance_id == _response->get_instance()
            && vsomeip::message_type_e::MT_RESPONSE
                    == _response->get_message_type()
            && vsomeip::return_code_e::E_OK == _response->get_return_code())
    {
```

```
            // Get the payload and print it
            std::shared_ptr<vsomeip::payload> pl = _response->get_payload();
            std::string resp = std::string(
                    reinterpret_cast<const char*>(pl->get_data()), 0,
                    pl->get_length());
            std::cout << "Received: " << resp << std::endl;
            stop();
        }
    }

    void stop()
    {
        // unregister the state handler
        app_->unregister_state_handler();
        // unregister the message handler
        app_->unregister_message_handler(vsomeip::ANY_SERVICE,
                service_instance_id, vsomeip::ANY_METHOD);
        // shutdown the application
        app_->stop();
    }

private:
    std::shared_ptr<vsomeip::runtime> rtm_;
    std::shared_ptr<vsomeip::application> app_;
};

int main(int argc, char **argv)
{
    hello_world_client hw_cl;
    hw_cl.init();
    hw_cl.start();
    return 0;
}
```

The client example results in the following program execution:

Main

1. *main()*

   First the application is initialized. After the initialization is finished the application is started.

Initialization

1. *init()*

   The initialization contains the registration of a message handler, an event handler and an availability handler.

   The event handler declares again a callback (*on_state_cbk*) for state changes that occur.

   The message handler declares a callback (*on_message_cbk*) for messages that are received from any service, any service instance and any method.

   The availability handler declares a callback (*on_availability_cbk*) which is called when the specific service is available (specifying the service id and the service instance id).

Start

1. *start()*

   The application will be started. This function only returns when the application will be stopped.

Callbacks

1. *on_state_cbk()*

   This function is called by the application when an state change occurred. If the application was successfully registered at the runtime then the specific service is requested.

2. *on_availability_cbk()*

   This function is called when the requested service is available or no longer available.

   First there is a check if the change of the availability is related to the *hello world service* and the availability changed to true. If the check is successful a service request is created and the appropriate service information are set (service id, service instance id, service method id). After that the payload of the request is created. The data of the payload is *World* and will be set afterwards. Finally the request is sent to the service.

3. *on_message_cbk()*

   This function is called when a message/response was received. If the response is from the requested service, of type *RESPONSE* and the return code is *OK* then the payload of the response is printed. Finally the application is stopped.

Stop

1. *stop()*

   This function unregister the event and the message handler and shuts down the application.