CommonAPI C++ SOME/IP in 10 minutes

CommonAPI C++ and bindings have been moved to github, see overview at https://github.com/orgs/GENIVI/teams/someip/repositories and see also the CommonAPI C++ homepage at http://genivi.github.io/capicxx-core-tools/. Please make sure that you use code generators and runtime libraries with the same version by getting the code generator binaries from capicxx-core-tools/releases repository, capicxx-dbus-tools/releases repository and capicxx-someip-tools/releases repository. Otherwise you could get same strange compilation errors. Please note that a newer version of this wiki page can be accessed via capicxx-core-tools/wiki repository.

Valid for CommonAPI 3.1.3 and vsomeip 1.3.0

Step 1 & 2: Preparation / Prerequisites

The following description is based on CommonAPI 3.1.3 and assumes that you use a standard Linux distribution (I tested it with Xubuntu 14.04) and that you have installed git and (CMake >=2.8). If you didn't pass the instructions of the page "CommonAPI C++ D-Bus in 10 minutes (from scratch)" then do it now for **step 2** in order to get and build the CommonAPI runtime library.

Step 3: Build the CommonAPI SOME/IP Runtime Library

Start again with cloning the source code of CommonAPI-SomeIP:

Clone CommonAPI SOME/IP

```
x@ubuntu:~/work$ git clone https://github.com/GENIVI/capicxx-someip-runtime.git

Cloning into 'capicxx-someip-runtime'...
remote: Counting objects: 95, done.
remote: Compressing objects: 100% (88/88), done.
remote: Total 95 (delta 17), reused 0 (delta 0)
Unpacking objects: 100% (95/95), done.
Checking connectivity... done.
x@ubuntu:~/work$ ls
capicxx-core-runtime  capicxx-someip-runtime
```

Just as CommonAPI C++ D-Bus requires the D-Bus library libdbus, the SOME/IP binding builds on the SOME/IP core library `vsomeip`. Fortunately. the source code of this library can be easily downloaded by cloning the vsomeip repository on GitHub.

Get vsomeip

```
x@ubuntu:~/work$ git clone http://github.com/GENIVI/vSomeIP.git
x@ubuntu:~/work$ ls
capicxx-core-runtime  capicxx-someip-runtime  vSomeIP
```

- At GENIVI there is another SOME/IP implementation in the repository someip.git. This implementation cannot be used together with `capicxx-someip-runtime`. Make sure that you cloned the right repository.
- CommonAPI 3.1.3 depends on `vsomeip` 1.3.0.

`vsomeip` uses the standard cross-platform C++ library `Boost.Asio` for the implementation of some core network functions. Before you can compile the `vsomeip` library make sure that Boost is installed on your system. You can follow the installation instructions at http://www.boost.org/doc/libs/1_59_0/more/getting_started/unix-variants.html, but if you have an Ubuntu platform it is easier to install it with the package manager APT (see also http://docs.projects.genivi.org/vSomeIP/1.3.0/html/README.html):

- 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`

When Boost has been successfuly installed, you can build `vsomeip` without further difficulty as usual:

Build vsomeip

```
x@ubuntu:~/work$ cd vSomeIP
x@ubuntu:~/work/vSomeIP$ mkdir build
x@ubuntu:~/work/vSomeIP$ cd build
x@ubuntu:~/work/vSomeIP/build$ cmake ..
x@ubuntu:~/work/vSomeIP/build$ make
```

On my system CMake prints out the following lines concerning Boost:

Boost

```
-- Boost version: 1.54.0
-- Found the following Boost libraries:
--   system
--   thread
--   log
```

Now it should be no problem to build the CommonAPI C++ SOME/IP runtime library:

Build CommonAPI C++ SOME/IP

```
x@ubuntu:~/work$ cd capicxx-someip-runtime
x@ubuntu:~/work/capicxx-someip-runtime$ mkdir build
x@ubuntu:~/work/capicxx-someip-runtime$ cd build
x@ubuntu:~/work/capicxx-someip-runtime/build$ cmake -DUSE_INSTALLED_COMMONAPI=OFF ..
x@ubuntu:~/work/capicxx-someip-runtime/build$ make
```

As with D-Bus we do not install the CommonAPI C++ SOME/IP library, the library `libCommonAPI-SomeIP.so` is (should be!) just in the actual build directory.

Step 4: Write the Franca file and generate code

Now follow again the instructions of the page "CommonAPI C++ D-Bus in 10 minutes", Step 4:

- Create a sub-directory project in your work directory and then the sub-directory `fidl` in the project directory. Change to this sub-directory.
- Now create HelloWorld.fidl as described in the D-Bus tutorial. It's CommonAPI: concerning the interface definition in the fidl there is no difference.

But the SOME/IP specification additionally requires the definition of service and method identifiers. Franca IDL provides the possibility to provide this kind of IPC framework specific parameters by means of so-called Franca deployment files. These deployment files have the ending .fdepl and have a Franca-like syntax (deployment parameter file). The exact content (which deployment parameters must be provided) must be specified also in a fdepl-file (deployment specification file). The deployment specification file which is implemented by the CommonAPI C++ SOME/IP specification can be found in the SOME/IP code generator project. If you don't want to check out the complete tools project just have a look at the Common API C++ SOME/IP Tools repository. You will find the file `CommonAPI-SOMEIP_deployment_spec.fdepl` in the directory org.genivi.commonapi.someip/deployment.

Based on this deployment specification it is possible to write the deployment parameter file; usually we call it like the fidl file, just with the ending fdepl.

HelloWorld.fdepl

```
import "platform:/plugin/org.genivi.commonapi.someip/deployment/CommonAPI-SOMEIP_deployment_spec.fdepl"
import "HelloWorld.fidl"
define org.genivi.commonapi.someip.deployment for interface commonapi.HelloWorld {
    SomeIpServiceID = 4660

    method sayHello {
        SomeIpMethodID = 33000
    }
}

define org.genivi.commonapi.someip.deployment for provider MyService {
    instance commonapi.HelloWorld {
        InstanceId = "test"
        SomeIpInstanceID = 22136
    }
}
```

The SOME/IP deployment specification has some mandatory parameters; it imports the binding independent CommonAPI deployment specification. Therefore we find the parameter `InstanceId` in the instance deployment.

Now you need the SOME/IP code generator:

Get SOME/IP Code Generator

```
x@ubuntu:~/work/project/cgen$ wget http://docs.projects.genivi.org/yamaica-update-site/CommonAPI/generator/3.1
/3.1.3/commonapi_someip_generator.zip
x@ubuntu:~/work/project/cgen$ unzip commonapi_someip_generator.zip -d commonapi_someip_generator
x@ubuntu:~/work/project/cgen$ chmod +x ./commonapi_someip_generator/commonapi-someip-generator-linux-x86
```

And generate code:

Generate SOME/IP Code

```
x@ubuntu:~/work/project/cgen$ cd ..
x@ubuntu:~/work/project$ ./cgen/commonapi_someip_generator/commonapi-someip-generator-linux-x86 -ll verbose .
/fidl/HelloWorld.fdepl
```


Please note that there are some SOME/IP parameter mandatory. Therefore the input file for the code generator is the deployment parameter file; the fidl file itself cannot be used.

Go to the directory with the generated sources and have a look:

Generatet SOME/IP Code

```
x@ubuntu:~/work/project/cgen$ cd ../v1_0/commonapi
x@ubuntu:~/work/project/v1_0/commonapi$ ls
HelloWorld.hpp  HelloWorldSomeIPDeployment.cpp  HelloWorldSomeIPProxy.hpp  HelloWorldStubDefault.cpp
HelloWorldProxyBase.hpp  HelloWorldSomeIPDeployment.hpp  HelloWorldSomeIPStubAdapter.cpp  HelloWorldStubDefault.
hpp  HelloWorldProxy.hpp  HelloWorldSomeIPProxy.cpp  HelloWorldSomeIPStubAdapter.hpp  HelloWorldStub.hpp
```

Step 5: Write the client and the service application

It's CommonAPI C++! Just follow step 5 of "CommonAPI C++ D-Bus in 10 minutes (from scratch)". There is no difference.

Step 6: Build and run

We start with creating the file `CMakeLists.txt`. It is very similar to the `CMakeLists.txt` that we use with D-Bus. Note that we link to the `vsomeip` library instead of libdbus:

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread -std=c++0x")

include_directories(
     src-gen
     ../capicxx-core-runtime/include
     ../capicxx-someip-runtime/include
     ../vSomeIP/interface
)

link_directories(
     /home/x/work/capicxx-core-runtime/build
     /home/x/work/capicxx-someip-runtime/build
     /home/x/work/vSomeIP/build
)

add_executable(HelloWorldClient
    src/HelloWorldClient.cpp
    src-gen/v1_0/commonapi/HelloWorldSomeIPProxy.cpp
    src-gen/v1_0/commonapi/HelloWorldSomeIPDeployment.cpp
)
target_link_libraries(HelloWorldClient CommonAPI CommonAPI-SomeIP vsomeip)

add_executable(HelloWorldService
    src/HelloWorldService.cpp
    src/HelloWorldStubImpl.cpp
    src-gen/v1_0/commonapi/HelloWorldSomeIPStubAdapter.cpp
    src-gen/v1_0/commonapi/HelloWorldStubDefault.cpp
    src-gen/v1_0/commonapi/HelloWorldSomeIPDeployment.cpp
)
target_link_libraries(HelloWorldService CommonAPI CommonAPI-SomeIP vsomeip)
```

Now you can call `CMake` and build everything as usual:

Build everything

```
x@ubuntu:~/work/project$ cd build
x@ubuntu:~/work/project/build$ cmake ..
x@ubuntu:~/work/project/build$ make
x@ubuntu:~/work/project/build$
```

Now we are on the verge to start the service and the client application. But `vsomeip` needs little more preparation (see http://docs.projects.genivi.org/vSomeIP/1.3.0/html/README.html). Forst we need to create a configuration file in the json format (see http://www.json.org/). Let's create an `ASCII` file `vsomeip.json` in the sub-directory `fidl` of the project directory:

vsomeip.json

```
{
    "unicast" : "local",
    "logging" :
    {
        "level" : "debug",
        "console" : "true",
        "file" : { "enable" : "false" },
        "dlt" : "false"
    },
    "applications" :
    [
        {
            "name" : "client-sample",
            "id" : "0x1343"
        },
        {
            "name" : "service-sample",
            "id" : "0x1277"
        }
    ],
    "servicegroups" :
    [
        {
            "name" : "default",
            "unicast" : "local",
            "services" :
            [
                {
                    "service" : "0x1234",
                    "instance" : "0x5678",
                    "unreliable" : "31000"
                }
            ]
        }
    ],
    "routing" : "service-sample",
    "service-discovery" :
    {
        "enable" : "false",
        "multicast" : "224.244.224.245",
        "port" : "30490",
        "protocol" : "udp"
    }
}
```

For a detailed description of all these entries in the configuration file please read the `vsomeip` user guide. At this point I only list some notes:

- We communicate here only locally on the same machine. Therefore we do not have to configure a unicast address.
- We log only to the console. Other possibilities are not considered.
- For both applications (client and service) we have to define names and identifiers. There is no relationship to the deployment file of the SOME/IP binding.
- Services belong to service groups. Here we have to enter the `SomeIpServiceID` and the `SomeIpInstanceID` of the deployment file.
- After the key word "routing" we have to enter the application which contains the vsomeip routing manager. The routing manager must be instantiated by exactly one application on your machine.
- We do not enable the service discovery.

Two more hints before we finally start client and service. We must set one environment variable in order to define the path of our json file:

Environment

```
x@ubuntu:~/work/project$ cd build
x@ubuntu:~/work/project/build$ export VSOMEIP_CONFIGURATION_FILE=../fidl/vsomeip.json
x@ubuntu:~/work/project/build$
```

And we must set a second environment variable `VSOMEIP_APPLICATION_NAME` which is set to the name of the application. As we intend to start both applications in the same console window, we assign this variable together with the start of the application:

Call Service

```
x@ubuntu:~/work/project/build$ VSOMEIP_APPLICATION_NAME=service-sample ./HelloWorldService &
[1] 5049
x@ubuntu:~/work/project/build$ 2015-09-18 01:54:25.924662 [info] Using configuration file: ../fidl/vsomeip.json
2015-09-18 01:54:25.928805 [debug] Routing endpoint at /tmp/vsomeip-0
2015-09-18 01:54:25.929315 [info] Service Discovery disabled. Using static routing information.
2015-09-18 01:54:25.929443 [debug] Application(service-sample, 1277) is initialized (uses 0 dispatcher threads).
Successfully Registered Service! Waiting for calls... (Abort with CTRL+C)
```

And we call the client:

Call Client

```
x@ubuntu:~/work/project/build$ VSOMEIP_APPLICATION_NAME=client-sample ./HelloWorldClient
2015-09-18 01:58:42.382432 [info] Using configuration file: ../fidl/vsomeip.json
2015-09-18 01:58:42.382750 [debug] Connecting to [0] at /tmp/vsomeip-0
2015-09-18 01:58:42.382969 [debug] Listening at /tmp/vsomeip-1343
2015-09-18 01:58:42.383024 [debug] Application(client-sample, 1343) is initialized (uses 0 dispatcher threads).
Checking availability!
2015-09-18 01:58:42.383918 [debug] Application/Client 1343 got registered!
Available...
sayHello('Bob'): 'Hello Bob!'
Got message: 'Hello Bob!'
2015-09-18 01:58:42.385127 [debug] Application/Client 1343 got deregistered!
2015-09-18 01:58:42.385230 [error] Local endpoint received message (Operation canceled)
```

The last error message appears at the moment because the client terminates after having called the `sayHello` function (there is no while loop in the main function).