ezCar2X

Release 2.3.0

Fraunhofer IKS

Dec 23, 2021

Contents:

1 Introduction

1.1 General

ezCar2X is a flexible software framework for rapid prototyping and evaluation of cooperative driver assistance systems as well as intelligent transport solutions. It's a collection of reusable software components for protocols and applications around the connected vehicles and infrastructures. It covers the basic ETSI ITS communication stack and various extensions.

The framework is provided as a set of C++14 libraries with limited interdependencies and few requirements on the target system. Platform specific implementation details are kept in separate plugins thus most of the functionality of the framework should be available on a wide range of platforms.

Beside rapid prototyping and evaluation on embedded targets and backend platforms the framework can be used inside simulation environments, reusing existing code for the simulation. Avoiding reimplementation for simulation and target enables fast transition from simulation to field tests. Abstraction principles allow integration of simulation specific code where needed.

Overview of used technologies

- C++14 Shared Libraries
- Main Dependencies: Boost, Google Protocol Buffers, crypto++
- Build: CMake
- Testing: Testing Framework Catch2
- Documentation: Doxygen (API), Sphinx (html, pdf)
- Continuous Integration: Gitlab

This manual is work-in-progress and will be extended on the fly. If you notice some inconsistencies or errors, please let us know. Your contribution is highly appriciated!

However the API documentation should be always correct and is thoroughly maintained.

Feel free to visit https://www.ezcar2x.fraunhofer.de/en.html for additional information.

1.2 Design Principles

Flexibility: The framework is suitable for a variety of different usage scenarios. It was previously deployed as communication and/or application unit on onboard units in vehicles, on road side units, edge platforms and as backend service. Single components or a complete framework can be deployed depending on the already existing environment and conditions.

Extensibility: Integration of new hardware, sensors or communication technologies and additional functionalities is easy through abstraction classes or by adding new plugins and libraries.

Portability: Because of only few external dependencies, the framework can be deployed on target platforms x86 (Linux), ARM and Android.

Abstraction: Interfaces (C++ abstract base classes) define functionality of the components. Therefore, different implementations of the same interface are interchangeable. Factory patterns allow to select specific implementations at run-time, e.g. from a configuration file

Event-driven design: External events, e.g. received packets, messages from sensors or input from the user can trigger defined behaviour. Components can schedule arbitrary events via an *EventScheduler* (interface), e.g. packet timeouts or connection retries. Periodic events are used for routine tasks, e.g. evaluation of measured data or distribution of status information.

1.3 Main Features

Facility Layer Features

- **Cooperative Awareness Basic Service (CAM) (ETSI EN 302 637-2 v1.4.1)**
  - Create CAM according to CAM generation rules
  - Populate CAM with vehicle data
  - Path history in CAMs
  - Sending CAMs on CCH
  - Processing of incoming CAMs
- **Decentralized Environmental Notification Basic Service (DENM) (ETSI EN 302 637-3 v1.3.1)**
  - Sending DENMs on CCH
  - Path history in DENMs
  - Keep-alive forwarding (KAF) (tbd)
  - Processing of incoming DENMs
- **Common Data Dictionary (CDD) (ETSI TS 102 894-2 v1.3.1)**
  - Common Data Dictionary (CDD) with protobuf v3 presentation
  - Common Data Dictionary (CDD) with ASN.1 presentation (non public)

Network & Transport Layer Features

- **GeoNetworking (GN) Media-independent (ETSI EN 302 636-4-1 v1.4.1)**
  - **Geographical addressing and forwarding**
    - Single Hop Broadcasting (SHB)
    - GeoBroadcast (GBC)

        * GeoUnicast (GUC)
        * GeoAnyCast (GAC)
        * TopologyScopedBroadcast (TSB)
    - **GN Forwarding Algorithms**
        * Greedy Forwarding algorithm
        * Contention-based forwarding algorithm
        * Simple area forwarding algorithm
        * Contention-based area forwarding algorithm
    - **GN Basic Features**
        * Beaconing
        * Location service
        * Duplicate packet detection
        * Circular, rectangular and ellipsoidal geographical areas
- Basic Transport Protocol (BTP) (ETSI EN 302 636-5-1 v2.2.0)

Security Layer Features (ETSI TS 103 097 v1.3.1, IEEE 1609.2)

- **Basic Features**
    - Time-stamping of messages
    - Authorization (signing) of messages
    - Validation of authorization of messages (verification of signature, certificate chain, timestamp)
    - Sending and receiving secured messages
    - Using security profiles specified in the standard
    - Signatures based on ECDSA-256 using Brainpool algorithms
- **PKI Handling (based on locally stored certificates)**
    - Provision and update of authorization tickets
    - Provision, update and removal of enrolment credentials
    - Update of local authorization status repository
    - Interfaces to PKI's Authorization and Enrollment Authority (communication is not fully implemented yet)
- **Privacy**
    - Pseudonym change
    - Change of all addresses and identifiers of other layers when pseudonym is changed (Station ID, GN/MAC source address)
    - Clearing path history cache when pseudonym is changed
    - Pseudonym change blocking

1.4 Extensions

The extensions to the basic stack are integrated in form of plugins or additional libraries.

Libraries are facilities or services which provide additional functions to the application or application plugins. Library examples are CAM/DENM facilities or cooperative perception / cooperative maneuver service.

Plugins enhance or redefine the functionality of the framework. A component (also a library) may have alternative implementations which can be loaded and selected in the config file. Examples for alternative implementations are GPSD Position Source (Default is Static Position Source), ASN.1 Data Presentation (Default is Protobuf) or GeoNetworking via UDP (Default is ITS-G5). Plugins can also change the functionality of the framework, e.g. providing infrastructure-based message reflector with the GeoService plugin.

Some mentioned components are included within the framework, some available upon request.

1.5 Simulation Integration

ezCar2X framework can also be integrated into simulation environments and installed on every virtual node (e.g. vehicle, edge, backend) within the simulation. The source code for the integration of ezCar2X with network simulator ns-3 and traffic simulator SUMO can be found in the ns3 repository group below.

```
https://gitlab.cc-asp.fraunhofer.de/ezcar2x/ns3
```

A readme with setup instructions is located in

```
https://gitlab.cc-asp.fraunhofer.de/ezcar2x/ns3/ezc2x
```

A readme with an example of simulation and evaluation of a simple traffic scenario is located in

```
https://gitlab.cc-asp.fraunhofer.de/ezcar2x/ns3/sim-examples/-/tree/master/examples/reference-
↪scenario-2019
```

1.6 Contact & How To Contribute

We would very much appreciate your contribution to further improve and extend ezCar2X framework. Please contact us with your contribution requests (questions, bugfixes or additional features) at ezcar2x@iks.fraunhofer.de and you will receive additional rights to be able to contribute. Feel free to contact us as well for any kind of collaboration, e.g. prototyping or evaluation of your application or protocol for connected automated driving.

2 Installation

2.1 Get Source Code

Get the source files, e.g. with

```
git clone https://gitlab.cc-asp.fraunhofer.de/ezcar2x/ezcar2x.git ~/src/ezcar2x
```

Update 3rd party git submodules within your source directory (e.g. *~/src/ezcar2x*)

```
git submodule update --init
```

Depending on your configuration you might need some plugins as well. You can find all publicly available plugins in the link below. Instructions how to build and use the plugins are within corresponding plugin repository.

```
https://gitlab.cc-asp.fraunhofer.de/ezcar2x/plugins
```

2.2 Building ezCar2X

2.2.1 Using CMake on Ubuntu 20.04 (and 19.xx)

Preparation

0. Install required libraries.

```
sudo apt install build-essential cmake ninja-build libprotobuf-dev protobuf-compiler
sudo apt install libboost-dev libboost-system-dev libboost-filesystem-dev libboost-thread-dev libboost-
↪regex-dev libboost-program-options-dev
```

1. Optional: Install further tools for code analysis and documentation generation

```
sudo apt install doxygen mscgen dia graphviz clang-tidy clang-format cppcheck python3-sphinx python3-
↪sphinx-rtd-theme
```

Build locally

2. Create an empty build folder and change into it:

```
mkdir build && cd build
```

3. Create the build environment for the project, assuming `/path/to/repo` is the location of the repository, defining the installation directory with *-DCMAKE_INSTALL_PREFIX* (e.g. ~/local).

```
cmake -GNinja -DCMAKE_INSTALL_PREFIX=~/local /path/to/repo
```

   In case you skipped installing optional packages for building of docs and static checks, disable those:

```
cmake -GNinja -DBUILD_DOC=OFF -DWITH_STATIC_CHECKS=OFF -DCMAKE_INSTALL_PREFIX=~/local /path/to/repo
```

   If not sure, follow this configuration example:

```
cmake -GNinja -DBUILD_DOC=OFF -DWITH_STATIC_CHECKS=OFF -DCMAKE_INSTALL_PREFIX=~/local ~/src/ezcar2x
```

4. Build and install the libraries:

```
ninja
ninja install
```

2.2.2 Using CMake with older Ubuntu versions (18.xx)

For older Ubuntu versions please replace step 0 of the preparation with the following instructions, depending on your setup.

Preparation on Ubuntu 18.04

0. Install required libraries. Unfortunately, boost, protobuf and CMake shipped with 18.04 LTS are too old, therefore we need to install newer versions separately.

   Install essential libraries and tools

```
sudo apt install build-essential ninja-build python-pip software-properties-common
```

   Register PPA for more recent boost versions and install boost from PPA

```
sudo add-apt-repository ppa:mhier/libboost-latest
sudo apt install libboost1.67-dev
```

   Register PPA for more recent protobuf version (3.6.1) and install protobuf from PPA

```
sudo add-apt-repository ppa:maarten-fonville/protobuf
sudo apt install -y libprotobuf-dev protobuf-compiler
```

   Install CMake via pip (add *sudo* to install system-wide)

```
pip install cmake
```

Preparation on Ubuntu 18.10

0. Install essential libraries and tools

```
sudo apt install build-essential cmake ninja-build software-properties-common
sudo apt install libboost-dev libboost-system-dev libboost-filesystem-dev libboost-thread-dev libboost-
↪regex-dev libboost-program-options-dev
```

   Register PPA for more recent protobuf version (3.6.1) and install protobuf from PPA

```
sudo add-apt-repository ppa:maarten-fonville/protobuf
sudo apt install -y libprotobuf-dev protobuf-compiler
```

2.2.3 Using conan

Preparation

0. Install conan using a virtual environment

```
sudo apt install python3-venv
python3 -m venv ~/.venv/conan
source ~/.venv/conan/bin/activate

pip install -U pip
pip install conan
```

1. Add the bincrafters public conan repository:

```
conan remote add bincrafters https://api.bintray.com/conan/bincrafters/public-conan
```

Build locally

2. Create an empty build folder and change into it:

```
mkdir build && cd build
```

3. Install dependencies via conan, assuming `/path/to/repo` is the path of the repository:

```
conan install /path/to/repo
```

4. Build ezCar2X using conan:

```
conan build /path/to/repo
```

   Alternatively, CMake can be used directly to build the package (adding further options as you like, e.g. using the build tool `ninja` here):

```
cmake -GNinja /path/to/repo
ninja
```

Build conan package

2. Build the package using conan:

```
conan create /path/to/repo esk/testing
```

   Where `/path/to/repo` is the location of the repo and `esk/testing` is the name of the channel the package is generated for.

2.3 Building Manual

The manual is generated with Sphinx. It can generate an HTML version as well as LaTeX/PDF (with additional tools).

2.3.1 Installing Sphinx

Via Ubuntu repository

Sphinx and the used theme can be installed directly from the Ubuntu repositories.

```
sudo apt install python3-sphinx python3-sphinx-rtd-theme
```

Via pip

It is useful to create a virtual environment similar to the description for conan above.

```
sudo apt install python3-venv
python3 -m venv ~/.venv/sphinx
source ~/.venv/sphinx/bin/activate

pip install -U pip
pip install sphinx sphinx-rtd-theme
```

2.3.2 Generate HTML manual

If the CMake project is generated with `-DBUILD_MANUAL=y` (which is the default) there is a build target `manual` (or `manual_html`) for the manual:

```
ninja manual
```

The manual will be generated in the build folder in: `doc/manual/html`

2.3.3 Generate PDF manual

For the PDF output to work, a proper installation of basic LaTeX tools is required, namely `pdflatex` and the `latexmk` build helper. On Ubuntu, some additional packages are required as well (more details):

```
sudo apt install texlive-latex-recommended texlive-fonts-recommended texlive-latex-extra latexmk
```

Since pdf output is not enabled as default, you have to configure the CMake project with `-DBUILD_MANUAL_PDF=y`. If it finds all required components, there is a separate build target `manual_pdf` that should create the PDF version:

```
ninja manual_pdf
```

The PDF manual can be found in the build folder at `doc/manual/tex/ezcar2x.pdf` .

Sometimes, the tex output needs a little tweaking before the generation of the final PDF. For such cases, an additional build target `manual_tex` is provided that only generated the tex files (in `doc/manual/tex`) but skips the generation of the PDF during the build process. Thus, the tex files can be edited and once done, `pdflatex` (or `make` in the folder) has to be invoked manually.

Note: The `manual_tex` target is available, even if `BUILD_MANUAL_PDF` is not enabled.

2.4 Building API Documentation

The API documentation is generated with Doxygen. Doxygen can generate an on-line documentation browser based on HTML.

2.4.1 Installing Doxygen

Doxygen can be installed directly from the Ubuntu repositories.

```
sudo apt install doxygen mscgen dia graphviz
```

2.4.2 Generate API documentation (HTML)

If the CMake project is generated with `-DBUILD_DOC=ON` (which is the default) there is a build target `doc_api` for the API documentation:

```
ninja doc_api
```

The manual will be generated in the build folder in: `doc/api/html`.

3 Overview

The ezCar2X framework is provided as a set of loosely coupled components covering most aspects of the development and evaluation of cooperative applications for driver assistance and intelligent transport systems.

This section provides an overview of some libraries of the ezCar2X framework - some as brief overview, some in more detail in following chapters. For further details please refer to API documentation.

3.1 Libraries

The following libraries are currently available:

core provides functionality used by all other components and libraries of the ezCar2X framework. It covers buffer and packet concepts, logging, events, exceptions, an object aggregation mechanism, a generic object factory and methods to access time related information.

cdd or *common data dictionary* provides definitions of various internal and external data types and structures standardized by ETSI.

access provides access layer functionality including ETSI ITS-G5. IP-based functionalities are mainly covered in separate plugins.

network provides network layer protocols like GeoNetworking and the Basic Transport Protocol standardized by ETSI.

security provides security mechanisms like signing, signature checking, encryption and decryption of messages as well as certificate management for ITS systems.

facility provides ITS facilities as defined by ETSI.

framework provides components to manage multiple libraries as part of a framework. It includes simple mechanisms to configure various system parameters as well as helpers for framework management.

util provides utilities based on or complementing the functionality of the other libraries.

3.1.1 Core

Component

Aggregatable is a base class for all classes that should be aggregatable to an *Bundle*.

Configurable is an interface class for configurable components. Configurable components must implement this interface and will be configured by the *Builder* at runtime.

Runnable is an interface class for components that need to be started. This interface enables components to use a multi-stage run concepts, where all components are expected to have executed operation for a single phase before advancing to the next. Therefore, *run()* is invoked once for each phase and the implementation is free to decide what to do in each phase.

However, some basic rules should be applied:

- Init: Is intended for internal setup of the component or setup of connections with the outside world. Avoid interaction with other internal components in this stage to be on the safe side and allow them to be properly setup first.
- Normal: Setup all the interactions between internal components, e.g. register callbacks for signals, ...
- Final: Finalize operation, e.g. start sending beacons, schedule regular cleanup events, ...

Datahub component provides unified access to vehicle status information and configuration data with following high-level features:

- Extensible structure via path like access to individual data elements
- Simple synchronization scheme to ensure consistent data for read/write accesses
- Subscription to updates of certain values (or even entire sub-trees)
- Type safe read/write operations
- Support for complex data types, e.g. custom structs
- Flexible input components (data providers) that could be implemented by 3rd parties through simple APIs

Event class provides the handle to access and cancel pending events. EventScheduler class allows to schedule events some time in the future. Events may be executed for a single time or recur with a fixed interval.

Geographic components provide methods for coordinate transformations and geographic region description, e.g.

- Basic class for capsulating WGS84 positions
- Transformation between WGS84 and a local vehicle coordinate system
- Distance calculation between two points
- Extrapolation functions for geographic positions
- Geographic region definitions (ellipse, circle, polygon, ...)

IO component provides aggregatable context wrapper for the boost::asio::io_context. This is used to share an asio io_context between components of the framework. Its processing is either run by a dedicated thread or manually, e.g. in the running environment's main thread.

IP provides an interface to access UDP functionality through a proper socket implementation. The default UDP abstraction implementation is based on boost::asio.

Logging provides mainly a logger for multiple messages with a common context. Logger class is mainly intended to be used as a member variable in high-level components that want to generate logging records. It provides a couple of convenience functions to simplify logging from a user's perspective.

It can be used (in cpp file) as follows:

```
Logger log_("MyContext");
log_.info() << "This might be interesting";
log_.error() << "This is an error";
```

Following severity classes are predefined:

- Trace : Detailed diagnostic information related to a specific piece of code (use sparingly)
- Debug : Diagnostic information helpful to more than just developers
- Info : Useful information to log, e.g. start/setup of a component
- Warning : Condition that may influence service behavior but can usually be recovered from automatically
- Error : Error fatal to a specific operation but not the entire application/service
- Fatal : Fatal error usually forcing shutdown or termination of the application/service

Plugin provides mainly a manager for ezC2X plugins. The *PluginManager* is responsible for loading plugins. It loads the shared object file, searches for the exported symbol and tries to cast this symbol to a Plugin implementation. After that it calls the populateTypeSet method of the loaded plugin in order to give the plugin a chance to export it's types and register it's creators. After a plugin was loaded, it is stored internally and released when the PluginManager is destroyed.

Position provides mainly *PositionVector* data structure and *EgoPositionProvider* class.

> **PositionVector** data structure provides information about the current position, speed, heading, altitude and position confidence ellipse. Data structure is modeled after the standardized ReferencePosition data frame and augmented with other information that logically belongs together. This information can be provided e.g. by a GPS sensor.
>
> **EgoPositionProvider** component provides the ego position data in a concise form based on the DataHub. Position provider reads the position data from the DataHub under a configurable root node which is by default "position". Furthermore, each data element that resides under the root e.g. speed, heading has configurable paths with sensible default values given below:
>
> - Root: "position"
> - Latitude: "latitude"
> - Longitude: "longitude"
> - Speed: "speed"
> - SpeedConfidence: "speed_confidence"
> - Heading: "heading"
> - HeadingConfidence: "heading_confidence"
> - Altitude: "altitude"
> - AltitudeConfidence: "altitude_confidence"
> - PositionConfidenceSemiMajorConfidence: "confidence_ellipse/semi_major_confidence"
> - PositionConfidenceSemiMinorConfidence: "confidence_ellipse/semi_minor_confidence"
> - PositionConfidenceSemiMajorOrientation: "confidence_ellipse/semi_major_orientation"
>
> Each data source populating the DataHub for providing position data e.g. gpsd, should write its fix quality under a configurable path which by default is "fix_type". The component can be configured to ignore updates with a fix type that is less then a minimum set.

Random provides an interface for a generator of uniformly distributed random bits. The default implementation is a thread-safe implementation of a random bit generator based on the STL Mersenne Twister.

Serialization provides convenience functions for network encoded serialization/deserialization.

Time provides an interface for a time source. The default time provider *SystemTimeProvider* is based on the current system time. The component also include *ItsClock* definition based on the ITS epoch 2004-01-01 midnight and *ItsTimestamp* creating an ITS time stamp from the provided system time.

3.1.2 Framework

FrameworkBuilder class is a builder for the creation of an entire framework instance at once. Initially, the framework builder does not provide any component builders, they have to be added first.

> **setup** method creates framework components and aggregates them to the provided bundle. The component builders will be executed in the order of addition. After all framework builders were processed, the presence of a *PluginProvider* is checked. If available and a *BuilderList* is found, additional builders loaded from plugins will be executed as well. Note: Only framework components that have at least their node present in the properties will be installed.
>
> **setupAndRun** method also creates framework components, aggregates them to the provided bundle and additionally run *Runnable* objects. All stages will be executed for all objects in the bundle that are runnable (i.e. inherit from *Runnable*).

ApplicationBuilder class is a builder for applications loaded from plugins.

> **addApplicationFactoriesFromPlugins** method adds application factories defined in plugins.
>
> **createApplications** method creates applications using the populated application factory. You should call *addApplicationFactoriesFromPlugins* first if you want to create applications defined in plugins. The properties should consist of a list of nodes where each node represents a single application. The name of the node is used to identify the application and to query the factory. Therefore the node must have the same name as the application class. All children of the node are forwarded to the factory method creating the specific application.
>
> Example:

```
<YourApplication1>
    <TriggerStart>10s</TriggerStart>
</YourApplication1>
```

3.1.3 Facility

Currently, publicly available facilities are

- CAM
- DENM
- Data Presentation
- Vehicle Control (Interface only)
- Object Sensor (Interface only)

3.2 Basic Concepts and Patterns

3.2.1 Abstraction

Abstraction is a fundamental concept to separate concepts from their implementations and used extensively throughout the framework. It is essential to create decoupled components and allow replacement of certain aspects of a complex system.

3.2.2 Interfaces

Pure abstract base classes[1], i.e. classes consisting of method signatures without any implementation, are used to describe concepts. They are often referred to as *interface classes* or *interfaces* (similar to the Java world).

Implementation details should always depend on interfaces rather than specific implementations to avoid tight coupling of several components. Furthermore, several distinct interfaces with a limited functional scope can be used to provide different *views* on a component improving reusability and decoupling at the same time.

3.2.3 Object Aggregation

Communication stacks usually have a defined structure with specified interfaces. Having its origins in research projects with varying requirements and innovative ideas, ezCar2X does not force a specific architecture for component composition. Instead ezCar2X adopts the idea of *object aggregation* from the network simulator *ns-3*[5].

Different objects, e.g. protocol instances, interface managers, etc., can be aggregated to a flexible stack or framework object: the `Bundle`. A `Bundle` is essentially a container of objects implementing `Aggregatable` base class. Each aggregated object can access all other aggregated objects. This way even cross-layer schemes can easily be implemented and tested but decoupling and separation of components remains.

3.2.4 Object Lifecycle

Following our goal of flexibility, there is no default object management since there is also no common base class for all objects. Instead, we provide different capabilities to the objects that need them, e.g. *Object Aggregation* through the `Aggregatable` base class.

However, more complex scenarios require the creation of the required object, setting up external connections as well as internal dependencies, scheduling periodic events or simply start of the work. To structure these tasks, classes can implement the `Runnable` interface. Note, that this is independent of an object being aggregatable, i.e. each implementation may choose to implement none of the base classes, either one or even both.

An `Runnable` implementation provides a single `run` method that will be invoked in multiple stages to accommodate different phases of the setup. The setup routine of your application has to ensure that each stage was called on all runnable objects before the next stage is invoked. The order of execution within one phase is implementation specific and therefore not specified. The helper methods within ezCar2X all provide this behavior, thus it is expected by all implementation classes.

Currently, there are 3 run stages defined:

- **INIT**: Used to setup external connections (e.g. sockets to other applications/processes). It is recommended to avoid interacting with other internal components in this stage. However, it can be used to check if all *Dependencies* are available to fail early.
- **NORMAL**: This stage is used for the normal setup of internal component interaction, e.g. subscribing for signals.
- **FINAL**: The last stage is intended to start normal operation of a component. Furthermore, it should be used to start regular activity of a component, e.g. sending beacons, listening for packets, analyzing sensor data.

3.2.5 Dependencies

Components may use other components to provide their service, e.g. a position-based routing protocol needs to access the current station position as well as time information. Throughout this manual, we refer to this kind of relationship

[1] http://en.wikibooks.org/wiki/C++_Programming/Classes/Abstract_Classes/Pure_Abstract_Classes
[5] http://www.nsnam.org

as a dependency, i.e. the implementation of the routing protocol depends on the time information or more specifically `ezC2X::EtsiGeoNetworking` depends on `TimeProvider`. Specific implementations should always depend on interfaces rather than specific types (`TimeProvider` instead of `SystemTimeProvider`) to enable reuse with other components (e.g. within a simulation) providing the required interface.

ezCar2X itself does not provide or force a specific way to handle this kind of relationship. However, most of the included component implementations follow the same pattern. Dependencies can be set with `using Dependencies = component::Handle<TimeProvider>`.

Once set up, the link to the dependency is stored in the dependent (implementation) class using an `Handle`.

`Handle` class is used for easier dependency management. This class allows to reference one or more dependencies by their type. Usually, this will reference the interfaces required for a specific component to do its job. Most of the access is provided through templated functions that expect the type, i.e. the element of the dependency list, they are working on. However, several convenience functions are provided to work on the entire set at once:

- isSet() checks if all dependencies are not null
- reset() resets all dependencies to null
- setFromAggregationIfNotSet() tries to update all internal reference from a bundle or aggregate

One example given is the definition of the implementation class `EtsiGeoNetworking` using the interface class `GeoNetworking` and defining dependencies.

```
class EtsiGeoNetworking : public GeoNetworking,
                          public component::Aggregatable,
                          public component::Configurable,
                          public component::Runnable,
...
{
public:
    //! External dependencies of this component
    using Dependencies = component::Handle<EventScheduler, TimeProvider, RandomBitGenerator,
            ItsG5Access, DataHub, component::Optional<PseudonymManager>, component::Optional
↪<SecurityEngine>>;
...
private:
    //! Handle to the dependencies of this component
    Dependencies deps_;
```

Dependencies are checked when an object is *run*, e.g.:

```
void
EtsiGeoNetworking::run(Stage stage)
{
    switch (stage)
    {
        case Stage::Init:
            deps_.setFromAggregationIfNotSet(*this);
            component::throwOnMissingDependencies(deps_, "EtsiGeoNetworking");
...
```

With the `getOrThrow` method you can get the requested dependency and throw if it is not available, e.g.:

```
// make sure dependencies are there before we change something
auto now = deps_.getOrThrow<TimeProvider, MissingDependency>("TimeProvider is missing")->now();
auto es = deps_.getOrThrow<EventScheduler, MissingDependency>("EventScheduler is missing");
```

3.2.6 Callbacks and Signal-Slot-Mechanisms

Interfaces are one way to decouple modules or components. However, there are often cases in which one component provides information about certain events to other components interested in those events. Of course, this could be implemented using direct invocation of certain methods but this would require explicit knowledge of the components involved. *Callbacks*[3] provide an indirection - usually through function pointers or function objects - allowing one

[3] https://en.wikipedia.org/wiki/Callback_(computer_programming)

piece of code to call a function without any dependency between the caller and the executed code.

Signal-Slot-Mechanisms[4] extend this concepts in case of multiple event sinks and reduce the amount of boiler plate code required to manage registration of sinks and connection handling.

ezCar2X uses both concepts. Callbacks are implemented using `boost::function` in conjunction with `boost::bind` whenever a simple callback suffices. Signals and slots in ezCar2X use *boost::signals2*.

One example using signals and slots for variable tracing can be found in section *Variable Tracing*.

3.2.7 Plugins

With a strong focus on abstraction and flexible component setup, plugins provide a powerful tool to introduce new or updated implementations of interfaces without rebuilding the ezCar2X libraries itself. Furthermore, it allows for 3rd party extensions and helps keeping dependencies of the core framework low by moving code with external dependencies to (optional) plugins.

Plugins have to be provided as shared objects (`*.so` on Linux, `*.dll` on Windows) exposing only a minimal interface to the `PluginManager`.

3.3 External Dependencies

ezCar2X has few mandatory external dependencies to enable a wide range of target platforms. However, to avoid reinventing the wheel *boost* is required for all ezCar2X libraries to build and run. Furthermore, *Google Protocol Buffers* are required for message serialization (e.g. inter-process communication) and internal data structures. Cryptographic algorithms are based on *crypto++* which is therefore mandatory for the security library to work.

3.3.1 boost

Boost provides free peer-reviewed portable C++ libraries. They cover a broad spectrum of applications and use cases while maintaining a high code standard. Many of the libraries are provided as *header-only* thus they are only required at compile time.

For further information visit the project website: http://www.boost.org

3.3.2 Google Protocol Buffers

Google Protocol Buffers provides simple means to define *messages* in a plain text format, so called *proto files* (usually with the extension `.proto`). Proto files can then be used to generate code for the message data structures themselves as well as their serialization and deserialization for various programming languages and platforms.

ezCar2X uses *Google Protocol Buffers* for socket based communication and to model certain data structures that need to be exchanged with other components or might change frequently.

For further information visit the project website: https://code.google.com/p/protobuf/

3.3.3 crypto++

Crypto++ is a C++ library providing cryptographic primitives and algorithms. It is used for all security related matters in the security library.

For further information visit the project website: http://www.cryptopp.com

[4] https://en.wikipedia.org/wiki/Signals_and_slots

3.3.4 CMake

CMake is a cross-platform build tool that can generate native makefiles as well as project files for a wide range of compilers and development environments. It is required to build the ezCar2X libraries.

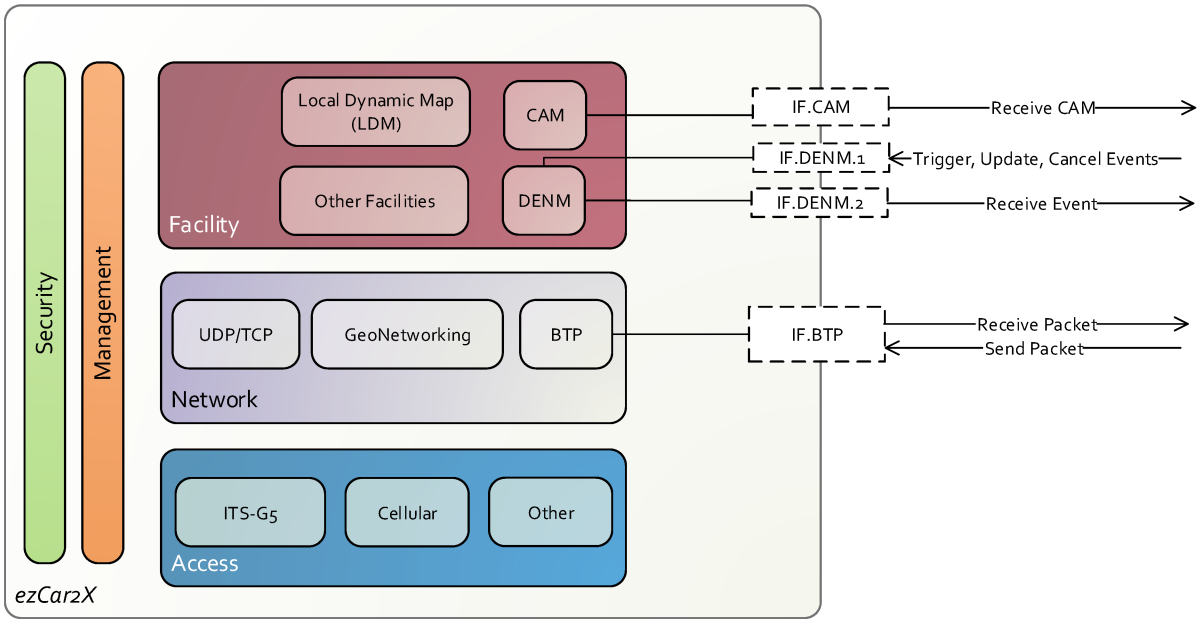
For further information visit the project website: http://www.cmake.org

4 External Interfaces

This section provides a quick overview of the external interfaces of the ezCar2X framework.

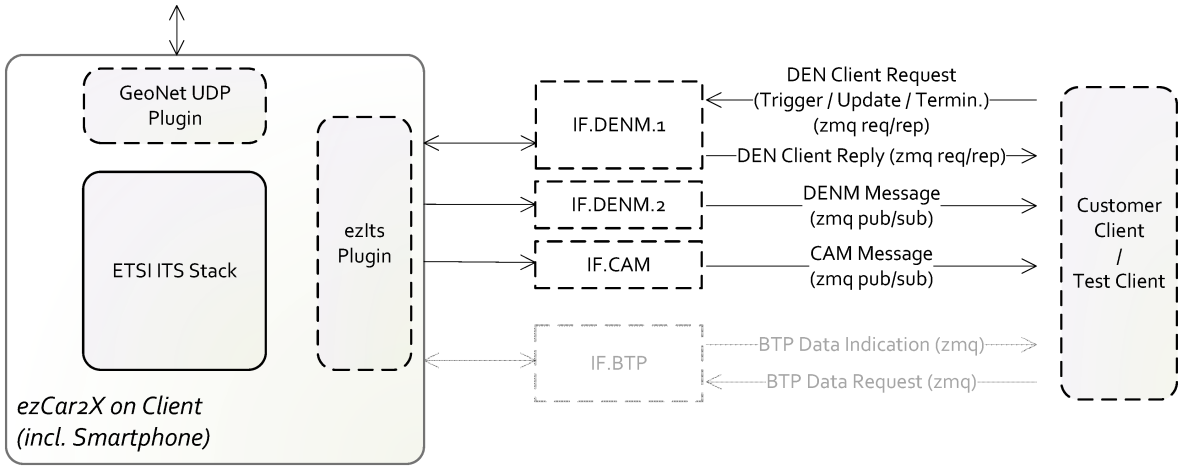
4.1 Architecture

This document describes the interfaces between ezCar2X stack on a client (or smartphone) and an application on this client (or smartphone). The interfaces consist of several logical interfaces that expose interfaces of the ETSI ITS stack. An overview of the logical interfaces is shown in the figure below. Detailed description of the logical interfaces is given in the following section.

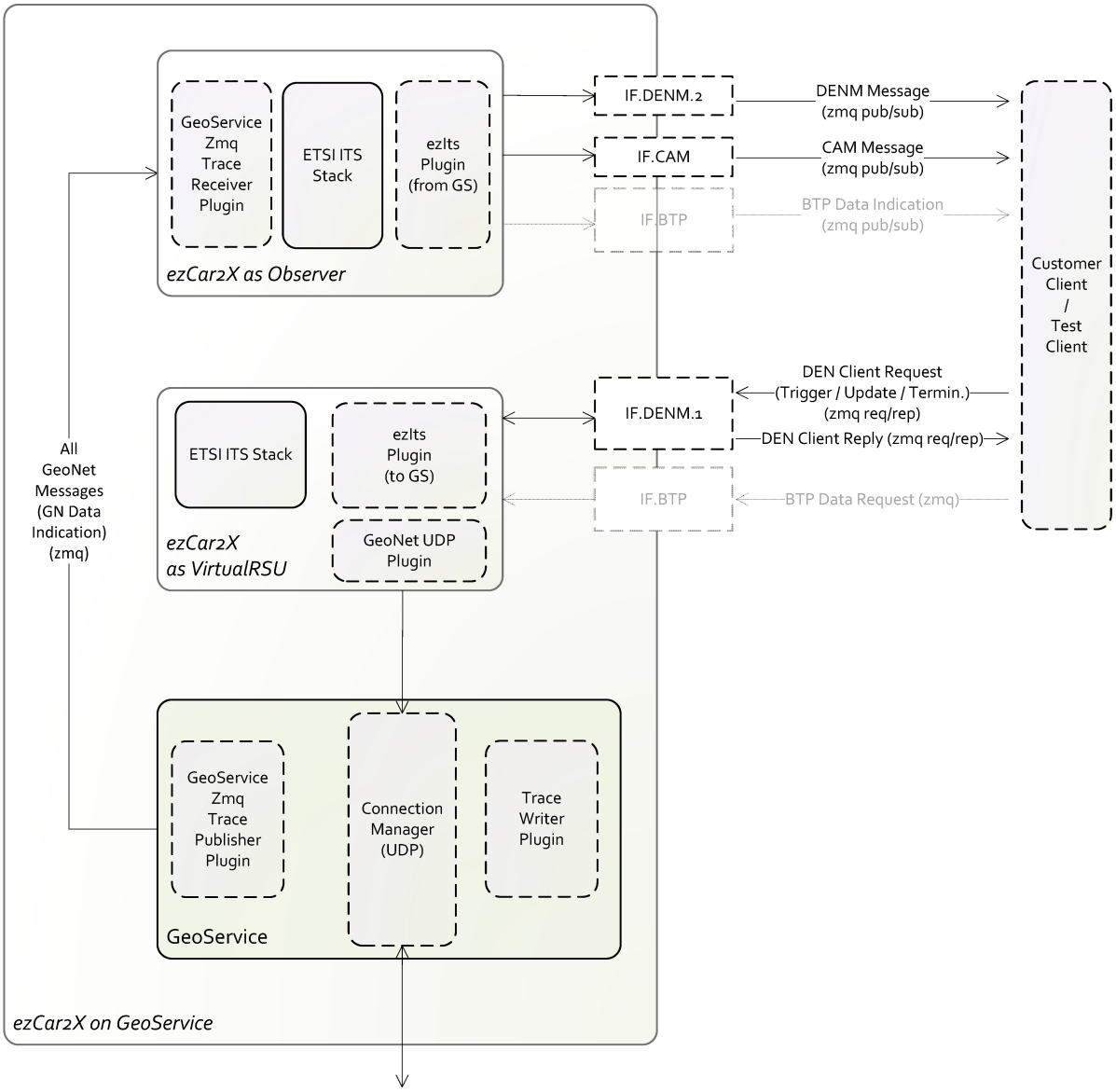


All logical interfaces are provided as ZeroMQ (e.g. via TCP) running on the same IP host in the subnet. The subnet address, host IP and port numbers of the interfaces are configurable.

The figure below depicts the detailed interfaces between the ETSI ITS stack and an application on a client / smartphone. Please note that this configuration requires additional plugins.



The figure below depicts the detailed interfaces between the GeoService on backend server / MEC and an application on backend server / MEC. This configuration allows to observe all GeoNetworking traffic as well as inject messages into the ITS system. Please note that this configuration requires additional plugins.



4.2 External Interfaces

4.2.1 IF.CAM

ETSI Cooperative Awareness Messages (CAM) are periodical messages exchanged between ITS stations to maintain awareness of each other. Therefore, the messages convey information such as position, speed and heading. CAMs are generated periodically by a CAM facility, which is an entity of the ETSI ITS protocol stack, located between application and network layer. The CAM facility asynchronously initiates CAM generation and transmission based on the dynamics of the vehicle. ETSI ITS stack exposes the IF.CAM interface to external clients via publish/subscribe on ZeroMQ sockets. Client can subscribe to a ZeroMQ topic on a configurable endpoint (e.g. TCP port) and receive all incoming CAM messages. The message encoding follows the Google Protocol Buffer Version 3 message format. The data fields of the message correspond to the specification in the ETSI TC ITS CAM standard.

The protobuf v3 definition of the *CAM Message* is described in *ezC2X/facility/cam/* folder.

4.2.2 IF.DEN

ETSI Decentralized Environmental Notification Messages (DENM) are messages used to distribute information about adverse road conditions to other road users. In the ETSI ITS stack a DENM facility is responsible to generate DENM messages from information received from the application layer and disseminate them via the IF.BTP interface in the relevant area. The facility further provides received DENM information to receiving application layer services. IF.DEN.1 is responsible for DENM generation from event information provided by the user of the interface (typically an application). Generated messages will be disseminated to warn surrounding vehicles immediately. Received DENM information are provided via the IF.DEN.2 interface to connected clients. The IF.DEN.1 and IF.DEN.2 interfaces of the ETSI ITS stack are exposed to external clients via ZeroMQ sockets. IF.DEN.1 can receive messages which describe a newly detected event, updates to an already disseminated event or an event termination. Upon reception of a DENM message, the IF.DEN.2 interface provides a connected client with the content of the DENM Message encoded as Google Protocol Buffer Version 3 message.

IF.DEN.1

The IF.DEN.1 interface accepts three message types for triggering, updating and terminating DEN events. The messages are embedded as optional elements in the DEN Client Request Message. Client requests are answered with DEN Client Reply Message, which contains an optional ActionId in case the request was successful. Both messages are encoded following Google Protocol Buffer Version 3 format. A client can trigger the creation of a new DENM message by sending a DEN Trigger message to the IF.DEN.1 interface. In case of successful DENM message creation, the interface replies the id of the created event as an ActionId embedded in a DEN Client Reply Message. The id can later on be used to identify the DENM event for termination or update. The DEN Update and DEN Termination message can be sent to the IF.DEN.1 interface to modify or terminate a previously triggered DEN event. Update and Termination require the ActionId that was returned by the IF.DEN.1 interface when triggering the event. Rules for update and termination of DEN events can be found in Section 6.1.2 of the ETSI TC ITS DENM standard.

The protobuf v3 definition of the messages

- DEN Client Request Message
- DEN Client Reply Message
- DEN Trigger
- DEN Update
- DEN Termination

are described in *ezC2X/facility/denm/ipc/* folder.

IF.DEN.2

DENM Messages received by the communication unit are forwarded to clients connected on the IF.DEN.2 interface. Client can subscribe to a ZeroMQ topic on a configurable endpoint (e.g. TCP port) and receive all incoming DENM messages. The forwarded information is encoded as a DENM Google Protocol Buffer Version 3 message. The data fields of the message correspond to the specification in the ETSI TC ITS DENM standard.

The protobuf v3 definition of the *DENM Message* is described in *ezC2X/facility/denm/ipc/* folder.

4.2.3 IF.BTP

The IF.BTP interface is a bidirectional interface, which encodes and transmits messages according to the ETSI Basic Transport Protocol (BTP). The ETSI BTP is a connectionless network layer protocol that operates on top of the ETSI GeoNetworking layer. It is comparable to UDP, allowing to multiplex messages of different application data streams by BTP ports and BTP node addresses. It further provides additional GeoAddressing capabilities, such that

messages can be addressed to a single or multiple receivers using geographical coordinates. In the receive direction, it delivers the content of BTP messages addressed to the ITS station's BTP address to a connected and subscribed client application. The primary use of the IF.BTP interface is to convey non ETSI standardized V2X messages to vehicles in a geographical region. The IF.BTP interface exposes the BTP data service primitive of the ETSI ITS protocol stack. Clients can connect to the ZeroMQ sockets to subscribe for a specific topic (= destination port) for reception of BTP messages. Received BTP messages will then be delivered as Google Protocol Buffer Version 3 encoded messages (BTP Data Indication) over the same connection. BTP Data Request messages are sent to the IF.BTP interface in order to request a payload to be sent via BTP. In addition to the payload, source and destination port information has to be specified as well as GeoMessaging parameters, which can be used to enable GeoBroadcasting. Different addressing methods (e.g. GeoUnicast / GeoBroadcast) for messages transmitted via the IF.BTP interface can be configured by setting the GeoMessaging parameters in the BTP Data Request.

The protobuf v3 definition of the messages

- BTP Data Request
- BTP Data Indication

are described in *ezcar2x/ezC2X/network/btp/ipc* folder.

5 Getting Started as User

This section contains information about stack configuration for framework users.

5.1 Starting ezCar2X instance

One ezCar2X instance can be started with *ezRun* command, located in the installation directory. Several instances can be executed in parallel on single machine assuming different pseudonym ids. The following options are available:

```
Configuration:
  -c [ --config ] arg         Config file to load
  -h [ --help ]               Show help
  -f [ --ext-file ] arg       File with configuration extensions
  -e [ --ext-value ] arg      Extensions in format <VAR>:<VALUE>

  Information:
  --builders                  Show list of registered builders plus their
                              config paths and quit.
  -v [ --version ]            Show version and quit.

  Logging:
  -l [ --log-level ] arg      Minimum log level to log. Valid values are
                              [Trace, Debug, Info, Warning, Error, Fatal].
                              Fatal is default log level
  --quiet                     Disable all log output (also Fatal)

  Advanced:
  -r [ --reuse-io-context ]   Force ezRun to reuse the aggregated io context.
                              If none is present, ezRun will fail to start.
```

To start one instance of ezCar2X client the command might look like:

```
~/local/bin/ezRun -l Trace -c your_config.xml -e EZC2X_ROOT_DIR:/home/your_user_name/local
```

5.2 Configuration File

The role and fuctions of the ezCar2X instance is defined by its configuration file. It can vary from OBU, RSU to a backend service on embedded target or within a simulation environment, all depending on the configuration file.

Te following example config file shows a configuration of one RasPi with cellular interface and gps, configured as passenger car and loading a camdemo application.

```
<ezC2X>
    <Framework>
        <!-- Core components -->
        <core>
            <PluginManager>
                <Plugin>##EZC2X_ROOT_DIR##/share/ezC2X/plugins/GeonetUdp.so</Plugin>
                <Plugin>##EZC2X_ROOT_DIR##/share/ezC2X/plugins/GpsdPositionSource.so</Plugin>
                <Plugin>##EZC2X_ROOT_DIR##/share/ezC2X/plugins/camdemo.so</Plugin>
            </PluginManager>
            <EventScheduler />
            <TimeProvider  />
            <RandomBitGenerator />
            <DataHub />
            <DataHubSources>
                <Source Type="GpsdPosition" />
            </DataHubSources>
            <IoContext>
                <LaunchThread>true</LaunchThread>
            </IoContext>
            <Udp Type="asio" />
        </core>

        <!-- Facility Layer -->
        <facility>
            <CaBasicService>
                <StationType>PASSENGER_CAR</StationType>
                <ProtocolParameters>
                    <SendingEnabled>true</SendingEnabled>
                    <ManualCamGenerationInterval>1s</ManualCamGenerationInterval>
                </ProtocolParameters>
            </CaBasicService>
            <DenBasicService>
                <StationType>PASSENGER_CAR</StationType>
            </DenBasicService>
            <DataPresentation />
        </facility>

        <!-- Network Layer -->
        <network>
            <GeoNetworking Type="GeonetUdp">
                <ProtocolParameters>
                    <itsGnSecurity>false</itsGnSecurity>
                    <itsGnLocalAddrConfMethod>Anonymous</itsGnLocalAddrConfMethod>
                    <itsGnMinUpdateFrequencyEpv>60s</itsGnMinUpdateFrequencyEpv>
                </ProtocolParameters>
                <GeonetUdpParameters>
                    <ServerUrl>10.123.123.123</ServerUrl>
                    <ServerPort>12345</ServerPort>
                </GeonetUdpParameters>
            </GeoNetworking>
            <Btp />
        </network>

        <!-- Security Layer -->
        <security>
            <PseudonymManager Type="list">
                <Pseudonyms>
                    <Id>1</Id>
                </Pseudonyms>
            </PseudonymManager>
        </security>
    </Framework>
    <Applications>
        <camdemoapp />
    </Applications>
</ezC2X>
```

5.2.1 Structure

The config file is a xml file with the following structure:

```
<ezC2X>
    <Framework>
        <LIBRARY_1>
            <COMPONENT_1>
```

(continues on next page)

(continued from previous page)

```
                <PARAMETER_1>VALUE</PARAMETER_1>
                <PARAMETER_x>VALUE</PARAMETER_x>
            </COMPONENT_1>
            <COMPONENT_n>
                <PARAMETER_1>VALUE</PARAMETER_1>
                <PARAMETER_z>VALUE</PARAMETER_z>
            </COMPONENT_n>
        </LIBRARY_1>
        <LIBRARY_y>
        </LIBRARY_y>
    </Framework>
    <Applications>
        <CUSTOM_APPLICATION_1/>
        <CUSTOM_APPLICATION_m/>
    </Applications>
</ezC2X>
```

Within the *Framework* tag, the main framework libraries (core, facility, network, security) are loaded and configured. Within the *Applications* tag, custom applications are loaded and their configurations (incl. component and parameter settings) are defined.

Note, that some components like EventScheduler, TimeProvider, RandomBitGenerator and DataHub must be loaded within the config file because they are dependencies for other components.

5.2.2 Type-based Components

For several components there is more than one implementation which defines its behaviour or function. The specific implementation can be chosen defined by type, e.g.

```
<DataHubSources>
    <Source Type="GpsdPosition" />
</DataHubSources>
```

Is this example, there are several source types available: *StaticPositionSource* (default), *ManualPositionSource* and *GpsdPosition* which is loaded via plugin in the given example. Not providing a specific implementation loads the default one.

5.2.3 Parameter Extensions

Defining extentions on command line starting ezRun with

```
~/local/bin/ezRun -l Trace -c your_config.xml -e EZC2X_ROOT_DIR:/home/your_user_name/local
```

will substitute ##EZC2X_ROOT_DIR## with your local ezCar2X directory.

5.2.4 Component Walkthrough

5.3 Parameter Definition

Tons of parameters and options can be set in the config file. To find settable parameters you need to have a look in the source code of the component and search for *property::Mapper* which is usually located the the beginning of the cpp file or within the Configurable API.

One example given are the protocol parameters of the CAM facility:

```
property::Mapper
mapProperties(EtsiProtocolParameters& pp, property::MapperFlags flags)
{
    property::Mapper pm(flags);

    pm.addProperty("DestinationPort", &pp.destinationPort, false);
    pm.addProperty("DestinationPortInfo", &pp.destinationPortInfo, false);
    pm.addProperty("GnPacketTransportType", &pp.gnPacketTransportType, false);
    pm.addProperty("GnCommunicationProfile", &pp.gnCommunicationProfile, false);
    pm.addProperty("GnSecurityProfile", &pp.gnSecurityProfile, false);
    pm.addProperty("GnTrafficClass", &pp.gnTrafficClass, false);
    pm.addProperty("GnMaximumPacketLifetime", &pp.gnMaximumPacketLifetime, false);

    pm.addProperty("SendingEnabled", &pp.sendingEnabled, false);
    pm.addProperty("ManualCamGenerationInterval", &pp.manualCamGenerationInterval, false);
    pm.addProperty("RsuCamGenerationInterval", &pp.rsuCamGenerationInterval, false);
    pm.addProperty("TGenCamMin", &pp.tGenCamMin, false);
    pm.addProperty("TGenCamMax", &pp.tGenCamMax, false);
    pm.addProperty("TCheckCamGen", &pp.tCheckCamGen, false);
    pm.addProperty("NGenCam", &pp.nGenCam, false);
    pm.addProperty("MaxCamGenerationTime", &pp.maxCamGenerationTime, false);
    pm.addProperty("DistanceThreshold", &pp.distanceThreshold, false);
    pm.addProperty("SpeedThreshold", &pp.speedThreshold, false);
    pm.addProperty("HeadingThreshold", &pp.headingThreshold, false);
    pm.addProperty("LowFrequencyContainerGenerationThreshold", &pp.
↪lowFrequencyContainerGenerationThreshold, false);

    return pm;
}
```

The default values are provided in the corresponding header file.

6 Getting Started as Developer

This section contains hints how to create your own application, plugin or library based on ezCar2X as well as some other useful information for developers.

6.1 Create your own application, plugin or library

The easiest way to create a new application, plugin or library is using the dedicated project wizard *ezProjectWizard* located in your installation directory. All required arguments can be passed directly with command line options, or the wizard can be call with -i in an interactive mode (recommended). In interactive mode, all required parameters will be asked for and can then be entered.

```
ezProjectWizard
usage: ezProjectWizard [-h] [-i] [-v] [-n NAME] [-d DESCRIPTION]
                       [-t {app,app_plugin,library,plugin}] [-o OUTPUT] [-e]
                       [--target TARGET] [--plugin PLUGIN] [--app APP]
                       [--library LIBRARY] [-y] [--data DATA]

optional arguments:
    -h, --help            show this help message and exit
    -i, --interactive     Interactive mode
    -v, --verbose         Enable verbose logging
    -n NAME, --name NAME  Name of the project
    -d DESCRIPTION, --description DESCRIPTION
                          Project description
    -t {app,app_plugin,library,plugin}, --type {app,app_plugin,library,plugin}
                          Type of the project
    -o OUTPUT, --output OUTPUT
                          Output path for the generated project
    -e, --exports         Generate export targets
    --target TARGET       Main target built by the project (app)
    --plugin PLUGIN       Name of the plugin to build (plugin, app_plugin)
    --app APP             Name of the application class (app_plugin)
    --library LIBRARY     Name of the library (library
    -y, --yes             Assume yes to all questions
    --data DATA           Path to the wizard input data folder
```

There are four types of projects which can be created:

1. app - An application based on ezCar2X which runs standalone.
2. app_plugin - An application based on ezCar2X which can be started by the configuration of the ezCar2X stack.
3. library - A library which is part of ezCar2X and can be used within the ezCar2X stack.

4. plugin - An enhancement of the ezCar2X stack which adds new functionality directly within the ezCar2X stack. A plugin must be registered within the config file under the plugin section, and can then directly be used within the config file.

The following parameters can be set in the interactive mode:

- Export targets are required, when the generated project, or parts of it, should be used and linked within an other project.
- Plugin name is the name of the plugin, or application plugin.
- Application class is the name of the main class of the application. This is also the name, under which the application can be registered within the config file.
- Output path is the path for the created source files. Indicate an empty directory.
- Main target is the name of the build application (used with CMake).
- Library name is the name of the build library. It also defines the namespace for the library functions.

The two most common used additions are application plugin and library which will be explained in more detail based on examples.

6.1.1 Application Plugin

Mandatory methods

The project wizard creates a basic code structure for the application plugin. Within your application class there are few methods that will be briefly described in the following section.

configure

Within the *configure* method all settable parameters will be configured. Default values are defined in the corespronding header file. These parameters can be set in the xml config file or with a parameters substition even from command line to enable script-based evaluation of parameter sets. If there are no settable parameters there is no need for this method.

```
void
YourApplication::configure(boost::property_tree::ptree const& properties)
{
    property::Mapper pm;

    pm.addProperty("TriggerStart", &triggerStart_, false);
    pm.addProperty("TriggerInterval", &triggerInterval_, false);

    pm.apply(properties);
    log_.info() << pm;
}
```

Within the xml configutation file the parameters will be set as follows:

```
<Applications>
    <YourApplication>
        <TriggerStart>10s</TriggerStart>
        <TriggerInterval>1s/TriggerInterval>
    </YourApplication>
</Applications>
```

start

Within the *start* method the initial events are scheduled - either as one time event, periodic events or triggered by internal, external or remote event. In the following example some ideas are presented to get the idea for complexity.

```
void
YourApplication::start(component::Bundle const& framework)
{
    log_.info() << "Application started";

    // Get EventScheduler to be able to schedule events, then enable scheduled event triggering:
    // Call 'triggerEvent' after triggerStart_ ms and then repeat after triggerInterval_ ms
    auto es = deps_.getOrThrow<EventScheduler, component::MissingDependency>("EventScheduler",
↪"YourApplication");
    triggerEvent_ = es->schedule([this]() { triggerEvent(0); },␣
↪std::chrono::milliseconds(triggerStart_),
                                 std::chrono::milliseconds(triggerInterval_));

    // Enable trigger based on incoming event: call 'handleReceivedCam' upon CAM reception
    auto pm = deps_.getOrThrow<PseudonymManager, component::MissingDependency>("PseudonymManager",
↪"YourApplication");
    auto cam = framework.get<CaBasicService>();
    camReceptionConnection_ = cam->subscribeOnCam([this](Cam const& cam) { handleReceivedCam(cam);␣
↪});
}
```

stop

Within this method you should cancel any scheduled events, disconnect your signals and than set the state to *NotRunning*.

```
void
ManeuverApplication::stop() noexcept
{
    triggerEvent_.cancel();
    camReceptionConnection_.disconnect();
    state_ = State::NotRunning;
}
```

Examples

One very simple application plugin is camdemo. It subscribes to CAM reception at the startup of the plugin and prints CAM content on command line. This can be done with just a few lines of code:

```
void camdemoapp::start(component::Bundle const& framework)
{
    auto cam = framework.get<CaBasicService>();
    cam->subscribeOnCam([this](Cam const& cam) { handleReceivedCam(cam); });
}

void camdemoapp::handleReceivedCam(Cam const& cam)
{
    std::string sds = cam.ShortDebugString();
    log_.debug() << "Received CAM: " << sds;
}
```

A bit more complex **YourApplication::handleReceivedCam** example shows how to retrieve own position and speed data from Vehicle Control Facility, extract position and speed data from received CAM of the remote vehicle and calculate the distance betwen two vehicles.

```
void YourApplication::handleReceivedCam(Cam const& cam)
{
    log_.debug() << "Received CAM: " << cam.ShortDebugString();

    std::uint32_t sendingStation = cam.header().station_id();

    // Get pointer to your vehicle control implementation, throw exception if missing dependency
    auto vehicleControl = deps_.getOrThrow<VehicleControlInterface, component::MissingDependency>(
            "VehicleControlInterface", "YourApplication");

    // Get current states of own vehicle from vehicleControl
    double ownSpeed = vehicleControl->getSpeed();
    Wgs84Position ownPosition = vehicleControl->getFrontBumperPosition();

    // Debug output of own state
    auto pm = deps_.getOrThrow<PseudonymManager, component::MissingDependency>("PseudonymManager",
↪"YourApplication");
```

(continues on next page)

(continued from previous page)

```
    log_.info() << "Station id: " << pm->getCurrentPseudonymId().value() << " > own state: speed="
↪<< ownSpeed << ", lat=" << ownPosition.getLatitude()
            << ", long=" << ownPosition.getLongitude();

    // Get position of remote vehicle
    auto remotePosition = Wgs84Position((ezC2X::Latitude)cam.payload().containers().basic_
↪container().reference_position().latitude().value(),
             (ezC2X::Longitude)cam.payload().containers().basic_container().reference_position().
↪longitude().value());

    // Compute distance from own vehicle to remote vehicle
    double geoDistance = ezC2X::distance(ownPosition, remotePosition);
}
```

6.1.2 Library

Depending on the intended functionality the setup can be very different. The basic structure will be already created with the project wizard. Some examples of libraries are CAM, DENM, Data Presentation and Vehicle Control facilities.

6.2 Naming Conventions

Developers should follow some basic rules while contributing to ezCar2X development or creating their own plugins.

Namespaces

All classes and functions are a member of the namespace `ezC2X`. Implementation details or classes/functions with a limited scope are located in sub namespaces of `ezC2X`.

Classes, Methods and Variables

Class names are camel case beginning with a capital letter. Header files follow the same convention with the extension `.hpp`, source files with extension `.cpp`.

Methods are camel case beginning with a small letter, e.g. `doSomething()`.

Variable names are camel case beginning with a small letter. Member variables have an underscore appended, e.g. `isRunning_`.

6.3 Variable Tracing

If you want to trace your variables within the stack (e.g. for evaluation), you have to add hook functions which can call registered callbacks. Additionally, the function to register the callback must be added as well as a required signal. Therefore you have two possiblities: Either implement it direct in the according class file (.cpp and .hpp), or extend an aleady existing tracing header file (interface definition), derive the class from this interface definition and implement the required functionality. An example of such an interface definition can be found in the file ezC2X/network/geonet/common/tracing/SourceInternal.hpp. An example for a direct implementation can be found in ezC2X/security/profile/CamProfileHandler.cpp and ezC2X/security/profile/CamProfileHandler.hpp.

What exactly must be done for direct implementation?

For the header file:

- Add a signal definition with the type and arguments of the called callback function (one argument is the variable type which should be traced, here SecurityEventType and void as return value)

```
using SignalOnEncapsulateSign = boost::signals2::signal<void(SecurityEventType)>;
```

- Add a function to register the callback

```
boost::signals2::connection
subscribeOnEncapsulateSign(SignalOnEncapsulateSign::slot_type const& handler);
```

- Add a signal from the previously defined signal definition

```
SignalOnEncapsulateSign onEncapsulateSign_;
```

For the implementation file:

- Add the hook function wherever you want to call the registered callback function

```
onEncapsulateSign_(SecurityEventType::Pre);
```

- Implement the function to register a callback

```
boost::signals2::connection
CamProfileHandler::subscribeOnEncapsulateSign(SignalOnEncapsulateSign::slot_type const&␣
↪handler)
{
    return onEncapsulateSign_.connect(handler);
}
```

For an additional app based on the ezCar2X stack:

- Implement a callback function

```
void handleEncapsulateSign(ezC2X::security::CamProfileHandler::SecurityEventType type)
{
    // do something with type (the traced variable)
}
```

- Register the callback function

```
camProfileHandler.subscribeOnEncapsulateSign([&](auto type) {
    handleEncapsulateSign(type);
});
```