Developer documentation
for the project
Diagnostics over IP for Car Simulator

Contents

Introduction

The *libdoip* is a library for the *car-simulator* (carsimulator.org) which provides that diagnostic messages can be send via Ethernet connection from the test application to the *car-simulator*. The *DoIPServer* is an instance in the *car-simulator*, which takes TCP/UDP diagnostic messages specified in the ISO Norm 13400-2 and reaches the diagnostic messages to the simulated ECUs of the *car-simulator.* For the development of the *DoIPServer* it was also necessary to develop an client application to test the *DoIPServer*. This client application represents the diagnostic test device which sends the diagnostic messages to the *DoIPServer***.**

Development Environment

Set up Linux and installing DoIP-Library

Set up Linux:

1.) Download the Ubuntu-18.04. iso and the free VmWare Player.
2.) Install VmWare Player and run it.
3.) Select “Create a New Virtual Machine”
4.) Select “Installer disc image file” and browse to the ubuntu iso.

Installing DoIP-Library:

1.) To install the library on the system, first get the source files with:
Git clone https://github.com/GerritRiesch94/libdoip
2.) Enter the directory ‘libdoip’ and build the library with:
make
3.) To install the builded library into /usr/lib/libdoip use:
sudo make install

Travis CI

TravisCI is used as a continuous integration tool for this project. After each push of commits to the git repository, the project will be build and the unit tests will be called. Hereby it is possible to detect failures which have been caused by the push of changes. This provides that failures can be located and corrected early after the code has been changed.

GoogleTest

For testing the DoIP library, mainly with unit tests, a test framework was needed. The used test framework must fullfill following criteria:

- Integration difficulty
- xUnit conform reports

After evaluating some different test framework, the decision was made to use *GoogleTest*, which is a unit test library developed by Google and allows testing of C and C++ sources.

To compile the unit tests, the GoogleTest framework is required to be installed on the system.

sudo apt-get install libgtest-dev

The package will be in the directory */usr/src/gtest* and can be built with the following commands:

sudo cmake CmakeLists.txt

sudo make

While there are some options to link the builded framework with the project, it was decided to place the GoogleTest files into the user's library directory.

sudo cp libgtest.a /usr/lib

sudo cp libgtest_main.a /usr/lib

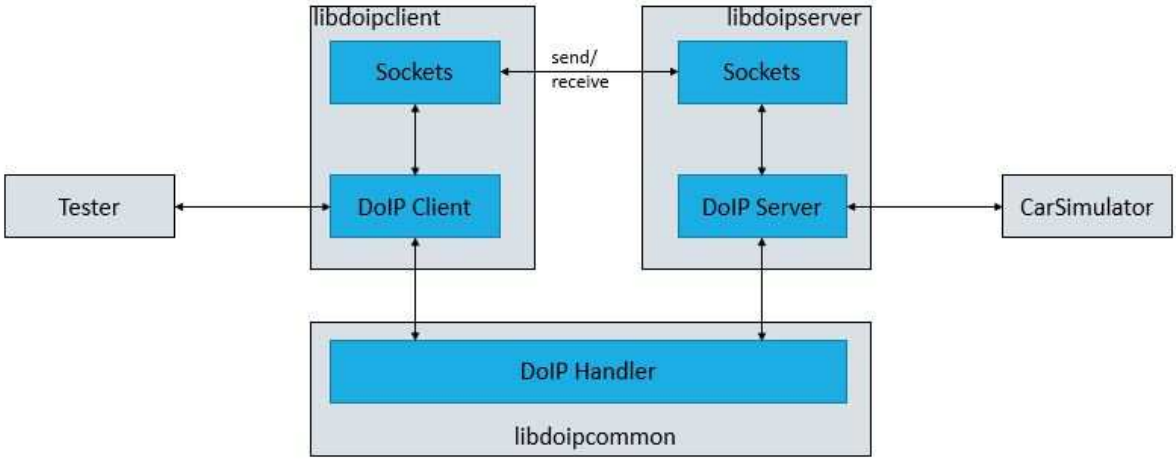
After these steps the project's Makefile can build the unit test. There are two options how to manually execute the tests. To simply run the tests and see the results in the terminal, the test executable must be started. To receive the results in a separately file the following command can be used:

./testExecutable –gtest_output="xml:./testOutput.xml"

The builded test executable is also used by the continuous integration tool TravisCI, as described in its section.

Functions of DoIP Library

Overview of the architecture



DoIP Client

All functionalities of a DoIP client are handled in the DoIPClient component. It is responsible for:

- initialize, start and closing communication sockets
- receiving and sending of DoIP messages

Member variable	**Description**
sockaddr_in _serverAddr	serverAddr is used for TCP and consists of server's IP address and port number (13400)
sockaddr_in _clientAddr	clientAddr is used for UDP and consists of client's IP address and port number (13400)
int _sockFd	Socket descriptor for the TCP socket.
int _SockFd_udp	Socket descriptor for the UDP socket.
int _connected	Client is connected to server

void DoIPClient::startTcpConnection()
Description: The function creates a TCP socket and try to connect to the server.

void DoIPClient::closeTcpConnection()
Description: The function closes the existing TCP connection to the server.

int DoIPClient::getSockFd()
Description: The *getSockFd* -Method returns the TCP socket descriptor.

int DoIPClient::getConnected()
Description: The *getConnected*-Method returns a value if the client is connected to the server.

void DoIPClient::startUdpConnection()
Description: The function creates a UDP socket which is listen to the port number 13400.

void DoIPClient::closeUdpConnection()
Description: The function closes the UDP socket which is listen to the port number 13400.

void DoIPClient::receiveMessage()
Description: The function receives DoIP messages over the TCP connection from the server. For example: Routing Activation Response, Diagnostic Message, Diagnostic Message Positive/Negative Ack

void DoIPClient::receiveUdpMessage()
Description: The function receives DoIP messages over UDP from the server. For example: Vehicle Identification Response

DoIP Server

All functionalities of a DoIP server are handled in the DoIPServer component. It is responsible for:

- initialize, setup and closing communication sockets
- receiving and sending of DoIP messages
- configure parameters (see configuration DoIP server sections)

Sockets

Since a socket can only support one protocol at the time, the server is using one socket each for TCP and UDP. For this the *DoIPServer* component must manage some member variables which are essential for the functionality of the sockets.

Member variable	Description
sockaddr_in serverAddress	Struct which defines a combination of the server's IP address and port number.
Sockaddr_in clientAddress	Struct which defines a combination of the client's IP address and port number and is used for UDP.
int server_socket_tcp	Socket descriptor for the TCP socket.
int server_socket_udp	Socket descriptor for the UDP socket.
int client_socket_tcp	Socket descriptor for the established TCP connection.

To setup a TCP socket the *setupTcpSocket* method can be used and for a UDP socket, *setupUdpSocket*.

void DoIPServer::setupTcpSocket()
Description: Creates a TCP socket by using the socket() system call with the associated TCP/IP parameters (AF_INET, SOCK_STREAM). The needed sockaddr_in struct *serverAddress* will be set to the default values from ISO-13400-2. Binds the socket's file descriptor to the address and port specified in the serverAddress.

void DoIPServer::setupUdpSocket()
Description: Creates a UDP socket by using the socket() system call with the associated UDP parameters (AF_INET, SOCK_DGRAM). The needed sockaddr_in struct serverAddress will be set to the default values from ISO-13400-2. Binds the socket's file descriptor to the address and port specified in the serverAddress. Since UDP is a connectionless protocol, it doesn't need to listen and accept an incoming connection.

For TCP, the server socket needs to listen and accept a connection which is implemented in the *listenTcpConnection* method.

void DoIPServer::listenTcpConnection()
Description: This method blocks the current process while listen and accepts a connection using the *listen* and *accept* system calls. The member variable *client_socket_tcp* contains the file descriptor referring to the accepted connection.

After using the described methods, the sockets are ready to receive and send data. To close a TCP or UDP socket the methods *closeSocket* and *closeUdpSocket* can be used.

void DoIPServer::closeSocket()
Description: Closes the sockets which are responsible for the TCP connection.

void DoIPServer::closeUdpSocket()
Description: Closes the socket which is responsible for the UDP connection.

Receive

The server is the passive participant of the communication between a client and a server, should only react to the client's requests and in the best case only be stopped if necessary. To ensure that the application, using the *DoIPServer*, has full control of the server, the TCP and UDP receive methods should be run in separate threads in the application.

int DoIPServer::receiveTcpMessage()
Description: Receives a TCP message from the client and calls the *reactToReceivedTcpMessage* method if the amount of received bytes is greater than 0. This method returns the amount of bytes which were sent back to the client, or -1 if an error occurred.

int DoIPServer::reactToReceivedTcpMessage(int readedBytes)	
int readedBytes	represents amount of received bytes
Description: Determines the payload type, as described in the section *Negative Acknowledge*, and processes the message. For some payload types this method also sends back the appropriate response to the client through TCP. Following payload types are received with TCP and processed with this method: • Routing activation request (see section Routing activation response) • Diagnostic message (see section Diagnostic message) • Alive Check Response (see section Alive Check Response) If a negative acknowledge was determinate, this method initiates the transmission of a negative acknowledge message. This method returns the amount of bytes which were sent back to the client, or -1 if an error occurred.	

int DoIPServer::receiveUdpMessage()
Description: Receives a UDP message from the client and calls the *reactToReceivedUdpMessage* method if the amount of received bytes is greater than 0. This method returns the amount of bytes which were sent back to the client, or -1 if an error occurred.

int reactToReceivedUdpMessage(int readedBytes)	
int readedBytes	represents amount of received bytes
Description: Determines the payload type, as described in the section *Negative Acknowledge*, and processes the message. For some payload types this method also sends back the appropriate response to the client through UDP. Following payload types are received with UDP and processes with this method: • Vehicle identification request (see section *Vehicle identification request*) If a negative acknowledge was determinate, this method initiates the transmission of a negative acknowledge message. This method returns the amount of bytes which were sent back to the client, or -1 if an error occurred.	

If running in a thread, the server will wait for a new message, after receiving and processing a TCP or UDP message, and repeat the procedure with a new received message.

Send

Since the server is the passive participant of the communication, as described in the previous section, it will never send a message by itself (except of Vehicle Announcment messages). The server uses the *sendMessage* and *sendUdpMessage* methods only to send back a response to a received DoIP message.

int DoIPServer::sendMessage(unsigned char* message, int messageLength)	
unsigned char* message	Pointer to the message which should be send back to the client with TCP.
int messageLength	Length of the message to send.
Description: Sends the given message to the client using the TCP socket. This method returns the amount of bytes sent or -1 if an error occurred.	

int DoIPServer::sendUdpMessage(unsigned char* message, int messageLength)	
unsigned char* message	Pointer to the message which should be send with UDP
int messageLength	Length of the message to send
Description: Sends the given message using the UDP socket. This method returns the amount of bytes sent or -1 if an error occurred.	

If the application needs to send a message, for example after processing a diagnostic message, the server has certain methods to send an answer. These methods are described in detail in their sections.

Generic Header

Every DoIP message which should be send needs to start with the generic header. The content of such a header is defined in the ISO-13400-2 and contains:

- Protocol version
- Inverse protocol version
- Payload type
- Payload length

Therefore, the implemented *DoIPGenericHeaderHandler* can create and return a valid generic header with the specified Payload type and Payload length depending on the message type which should be send.

unsigned char* createGenericHeader(PayloadType type, uint32_t length)	
PayloadType type	The required payload type which should be included into the generic header
uint32_t length	Length of the payload type specific message content
Description: Creates a generic header containing the indicated informations, so the client or server can identify a DoIP message and read out the required information to further process the received message Annotation: *PayloadType* is an enumeration defined in the *DoIPGenericHeaderHandler* header file. It contains every payload type which is currently supported by the library.	

The *DoIPGenericHeaderHandler* is located in the *libdoipcommon* component to allow both the DoIPClient and DoIPServer to create generic headers with the same method, to reduce potential errors.

Negative Acknowledge

When the DoIP server receives a message it always checks the generic header if all required information, to further process the message, are valid. There are multiple reasons when a message isn't considered as valid which needs to be checked before processing the message:

- Protocol version or inverse protocol version does not match
- Payload type is not supported by the server
- Payload length does not match the expected length for the payload type

The message will be parsed by the *DoIPGenericHeaderHandler* and returns a *GenericHeaderAction.* A *GenericHeaderAction* is the result of parsing the message, stores the determined payload type and a byte, and is used for further message processing (see section DoIP Server, Receive).

GenericHeaderAction parseGenericHeader(unsigned char* data, int dataLength)	
unsigned char* data	Pointer to the received message which will be parsed
int dataLength	Length of the received message
Description: Checks if the generic header of the given message is valid. It returns a *GenericHeaderAction* with the payload type if the header is valid. Otherwise the payload type of the *GenericHeaderAction* will be set to negative acknowledge and the byte will be set to a negative acknowledge code depending on which error occurred. Annotation:	

struct GenericHeaderAction	
PayloadType type	Payload type of the message or negative acknowledge if a error occurred. *DoIPServer* can further process the message depending on this type.
unsigned char value	Holds the negative acknowledge code if an error occurred.

If the *DoIPServer* receives a message that is considered as not valid after parsing it, a negative acknowledge will be send back to the client. The received message will be discarded and the server can receive a new message.

int DoIPServer::sendNegativeAck(unsigned char ackCode)	
unsigned char ackCode	The negative acknowledge code which should be send to the client.
Description: Creates a negative acknowledge message, by calling *createGenericHeader* to construct the generic header with negative acknowledge payload type. The given *ackCode* will then be inserted as content into the message and at last send to the client through the server's *sendMessage* function.	

If no error has been found while parsing, the received message can be further processed, which is descript in the payload type specific sections.

Routing Activation Request

void DoIPClient::sendRoutingActivationRequest()
Description: The Routing Activation Request is used by the Client to activate routing on a TCP socket. When the routing is activated the client can send diagnostic messages over the TCP socket to the *DoIP-Server*. The *DoIP-Server* is an instance in the car-simulator which receives the messages and transfer to the appropriate *Electronic Control Unit* for further processing.

const std::pair<int,unsigned char*>* DoIPClient::buildRoutingActivationRequest()
Description: The function builds the header data and the payload for the *Routing Activation Request* which is specified in the ISO standard and returns the message and the length of the message in bytes.

Routing Activation Response

The RoutingActivationResponse message will only be send whenever the DoIP server receives a *RoutingActivationRequest.* Upon receiving a *RoutingActivationRequest*, the payload specific content will be parsed by the *RoutingActivationHandler* to determine if the server should allow further messages of the client.

unsigned char parseRoutingActivation(unsigned char* data)	
unsigned char* data	Pointer to received message which has the routing activation request in its payload specific content
Description: Checks if the routing activation request is valid which determine if the server should receive further messages from this client. Since the client's logical address needs to be in a specific range, this method first checks if the address is located in this range. Hereafter the routing activation type will be checked whether the server supports this type. This method returns a routing activation response code depending of the outcome of these two checks.	

Regardless of the result, the server will send back a *RoutingActivationResponse* with the response code to the client. Therefore the *RoutingActivationHandler* has the capability to construct a routing activation response with all needed information.

unsigned char* createRoutingActivationResponse(unsigned char clientAddress[2], unsigned char responseCode)	
unsigned char clientAddress[2]	Logical address of the client that requested the routing activation.
unsigned char responseCode	The routing activation response code determined by parsing the *RoutingActivationRequest*.
Description: Creates a routing activation response, by calling *createGenericHeader* to construct a generic header with routing activation response as payload type and the required payload specific length. The logical address of the server, the given logical address of the client and the response code will then be inserted as content into the message. This method returns the pointer to the constructed routing activation response message. Annotation: All available response codes can be seen in ISO-13400-2, Table 25.	

The constructed routing activation response is then sent with the server's *sendMessage* method to the client. Afterwards the response code required action will be executed which is to either activate and register the client's source address on the socket or to not activate routing and close the socket.

Vehicle Identification Request

void DoIPClient::sendVehicleIdentificationRequest(const char* address)	
const char* address	request will be sent to this address Structure: xxx.xxx.xxx.xxx Example: 255.255.255.255
Description: The *Vehicle Identification Request* is used by the Client to identify a DoIP entity in the network. Therefore the request can be sent as broadcast, multicast or unicast, based on the address which is given to the function as parameter.	

const std::pair<int,unsigned char*>* DoIPClient::buildVehicleIdentificationRequest()
Description: The function fills the message header for the request, specified in the ISO Norm and returns the length of the header and the header data.

void DoIPClient::parseVIResponseInformation(unsigned char* data)	
unsigned char* data	received data
Description: If the client application receives a *Vehicle Identification Response* message the client saves the values of the *Vehicle Identification Number (VIN), Logical Address of the DoIP entity (LogicalAddress), EID, GID and „further action required" (FurtherActionReq)* as member variables. These parameters are explained in a further chapter *„Configuration of a DoIPServer instance"*.	

void DoIPClient::displayVIResponseInformation()
Description: Displays the *Vehicle Identification Data* (VIN, LogicalAddress, EID, GID, FurtherActionReq) as console output.

Vehicle Identification Response

Description:

The *Vehicle Identification Response* will be sent to a client application when the server application receives a *Vehicle Identification Request* message. The message contains the values of *VIN, LogicalAddress, EID, GID, FurtherActionReq* as payload which are member variables of the server instance.

VehicleIdentificationHandler.cpp (libdoipserver)

This module contains a function which is used for building the header data and the payload for the *Vehicle Identification Response* and *Vehicle Announcement* message. This module is used by the *DoIPServer.cpp* module.

unsigned char* createVehicleIdentificationResponse(std::string VIN, unsigned char* LogicalAddress, unsigned char* EID, unsigned char* GID, unsigned char FurtherActionReq)	
std::string VIN	value of vehicle identification number
unsigned char* LogicalAddress	logical address of an DoIP entity
unsigned char* EID	MAC address of an DoIP entity network interface
unsigned char* GID	identifier of an DoIP entity group
unsigned char FurtherActionReq	further action code
Description: The function creates the header data and the payload for the *Vehicle Identification Response* and *Vehicle Announcement* message. The parameters are the member variables *VIN, LogicalAddress, EID, GID, FurtherActionReq* of the server instance. These parameters are explained in the chapter *"Configuration of a DoIPServer instance"*.	

Vehicle Announcement Message

The *Vehicle Announcement Message* will be sent to a client application after the server instance and the sockets have been successfully created. This message has the same data as the *Vehicle Identification Response* message.

int DoIPServer::sendVehicleAnnouncement()
Description: This function sends a Vehicle Announcement Message. There are two variables that are used in this function. The member variable *A_DoIP_Announce_Num* specifies the number of messages that are sent and the member variable *A_DoIP_Announce_Interval* specifies the interval in which the messages are sent. These two variables are set in the *"carsimconfig.lua"* LUA Script file which will be explained in the chapter *"Configuration of an DoIPServer instance"*.

Setter Functions for DoIPServer

The following setter functions are used to set the values of member variables which can be configured by an LUA Script file called *„carsimconfig.lua"*

Setter function	Description
void DoIPServer::setVIN(std::string VINString)	Sets the value of member variable *VIN* to the value of *VINString*
void DoIPServer::setLogicalAddress(const unsigned int inputLogAdd)	Sets the value of the member variable *LogicalAddress* to the value of *inputLogAdd*
void DoIPServer::setEID(const uint64_t inputEID)	Sets the value of the member variable *EID* to the value of *inputEID*
void DoIPServer::setGID(const uint64_t inputGID)	Sets the value of the member variable *GID* to the value of *inputGID*
void DoIPServer::setFAR(const unsigned int inputFAR)	Sets the value of the member variable *FurtherActionReq* to the value of *inputFAR*
void DoIPServer::setA_DoIP_Announce_Num(int Num)	Sets the value of the member variable *A_DoIP_Announce_Num* to the value of *Num*
void DoIPServer::setA_DoIP_Announce_Interval(int Interval)	Sets the value of the member variable *A_DoIP_Announce_Interval* to the value of *Interval*
void DoIPServer::setGeneralInactivityTime (uint16_t seconds)	Sets the value of the alive check timers timeout *maxSeconds* to the value of *seconds*.

void DoIPServer::setEIDdefault()
Description: Sets the value of the *EID* member variable to the MAC Address of the first found ethernet controller. (not loopback)

There is an example for the usage of LUA Script Files in the chapter *"Example for LUA Script files"*.

Configuration of a DoIPServer instance

The following member variables of a DoIPServer instance can be configured with an LUA Script file called „*carsimconfig.lua*" or have default values which are specified in the ISO Norm 13400-2 after creating an server instance (*DoIPServer.cpp*).

Therefore the setter functions are used to set the values of the member variables given by the „*carsimconfig.lua*" file. All variables have default values which will be created with the server instance, except of the default value for *EID*. If there are no specified values for the variables in the „*carsimconfig.lua*" file the default values are used.

Member variable	Default value	Naming in carsimconfig.lua
std::string VIN	"00000000000000000"	VIN
unsigned char LogicalAddress [2]	{0x00, 0x00}	LOGICAL_ADDRESS
unsigned char EID [6]	Setted by the *setEIDdefault()* function	EID
unsigned char GID [6]	{0x00, 0x00, 0x00, 0x00, 0x00, 0x00}	GID
unsigned char FurtherActionReq	0x00	FURTHER_ACTION
int A_DoIP_Announce_Num	3	ANNOUNCE_NUM
int A_DoIP_Announce_Interval	500	ANNOUNCE_INTERVAL
std::uint16_t generalInactivity (Member variable of *AliveCheckTimer)*	300	T_TCP_General_Inactivity

Member variable	Description
std::string VIN	represents the vehicle identification number of an DoIP entity
unsigned char LogicalAddress [2]	represents the logical address of an DoIP entity
unsigned char EID [6]	represents the MAC address of an DoIP entities network interface
unsigned char GID [6]	represents the unique identifier of an DoIP entity group
unsigned char FurtherActionReq	further information about used security approach for building a connection to the DoIP entity
int A_DoIP_Announce_Num	specifies the number of Vehicle Announcement Messages which will be sent when the function is *sendVehicleAnnouncement* is called
int A_DoIP_Announce_Interval	specifies the interval in which the Vehicle Announcement Messages will be sent when the function is *sendVehicleAnnouncement* is called
std::uint16_t generalInactivity (Member variable of *AliveCheckTimer)*	specifies the timeout for the call of the function *aliveCheckTimeout*

Diagnostic Message

The Diagnostic Message is used by the client to send diagnostic requests to the server. This message type is also used to transmit the diagnostic response back to the client. Every diagnostic message must contain following information:

- Source address
- Target address
- User data

The *DiagnosticMessageHandler* is located in the *libdoipcommon* component to allow both the DoIPClient and DoIPServer to create and parse diagnostic messages with the same methods, to reduce potential errors. To construct a diagnostic message, both the client and server uses the *createDiagnosticMessage* method from the *DiagnosticMessageHandler.*

unsigned char* createDiagnosticMessage(unsigned char sourceAddress[2], unsigned char targetAddress[2], unsigned char* userData, int userDataLength)	
unsigned char sourceAddress[2]	Logical address of the sender of a diagnostic message. Limited to two bytes.
unsigned char targetAddress[2]	Logical address of the receiver of a diagnostic message. Limited to two bytes.
unsigned char* userData	Pointer to the userData which should be transmitted.
int userDataLength	Length of the userData to send.
Description: Creates a diagnostic message, by calling *createGenericHeader* to first construct a generic header with diagnostic message as payload type and the required payload specific length. The payload specific length is being calculated from the length of the sourceAddress, targetAddress and the userDataLength. The sourceAddress, targetAddress and the userData will be inserted as content into the message. This method returns the pointer to the constructed diagnostic message which is ready to be send.	

The constructed diagnostic message can then be sent with the client's or server's send method, depending on who is the sender of the diagnostic message.

Only the application, which uses this library, can interpret the *userData* from the diagnostic message. Therefore, we use callbacks since the server doesn't know anything about the application which uses the library. There are two callbacks defined in the *DiagnostcMessageHandler:*

DiagnosticCallback std::function<void(unsigned char*, unsigned char*, int)>	Used to pass over the targetAddress, userData and the length of the user data to the application which on the other hand can process the received userData. This callback is only executed if the parsing of the received message was successful.
DiagnosticMessageNotification std::function<bool(unsigned char*)>	Used to notify the application whenever a diagnostic message is received regardless of whether the diagnostic message is valid or not. The application can simulate a desired situation which either ends up with sending a positive or negative diagnostic message acknowledge. The callback method must return a boolean to indicate if the server should further process the message. This callback passes only the targetAddress to the application.

When a diagnostic message was passed to the application with the *DiagnosticCallback*, sending a diagnostic message with the response can be done using the *receiveDiagnosticPayload* method.

void DoIPServer::receiveDiagnosticPayload(unsigned char* address, unsigned char* data, int length)	
unsigned char* address	The logical address of an electronic control unit which has generated the response.
unsigned char* data	The generated response data of an electronic control unit.
int length	The length of data.
Description: This method receives the address, data and length of data from the application to create a diagnostic message by using *createDiagnosticMessage,* to send the response of an electronic control unit back to the client.	

On the server side, the diagnostic message will be checked, if the received *sourceAddress* is registered on the socket (see Routing Activation Request). Therefore, the *DiagnosticMessageHandler's parseDiagnosticMessage* method will be called, which checks the *sourceAddress*. If the check was successful the *DiagnosticCallback* is executed, so the application can use the received *userData*.

unsigned char parseDiagnosticMessage(DiagnosticCallback callback, unsigned char sourceAddress[2], unsigned char* data, int diagMessageLength)	
DiagnosticCallback callback	Callback to pass over the received data to the application.
unsigned char sourceAddress[2]	Current registered source address on the socket.
unsigned char* data	Received message which has the diagnostic message as payload specific content.
int diagMessageLength	Length of received message. Parameter is needed because the userData doesn't have a fixed length.
Description: This method checks if the sourceAddress of the received message matches the registered source address on the socket. If the check is successful the given callback will be executed. Other checks, as defined in the ISO-13400, were dropped due to no need in the simulation. This method returns either a positive or negative acknowledge code.	

The returned acknowledge code from the *parseDiagnosticMessage* method, can be used to either send a diagnostic message positive or negative acknowledge, which is described in the next sections.

Diagnostic Message positive Acknowledge

The diagnostic message positive acknowledge is sent by the server to inform the client, that his diagnostic message was successful checked, and the containing user data were passed to the application. This message type only contains the source address, target address and the positive acknowledge code, which is for example produced by the *parseDiagnosticMessage* method.

To create a diagnostic message positive acknowledge the *DiagnosticMessageHandlers createDiagnosticACK* method is used with the diagnostic message positive acknowledge payload type.

unsigned char* createDiagnosticACK(bool ackType, unsigned char sourceAddress[2], unsigned char targetAddress[2], unsigned char responseCode)	
bool ackType	Defines if a positive or negative acknowledge should be created.
unsigned char sourceAddress[2]	Logical address of the receiver of the previous diagnostic message. Limited to two bytes.
unsigned char targetAddress[2]	Logical address of the sender of the previous diagnostic message. Limited to two bytes.
unsigned char responseCode	The diagnostic message positive or negative acknowledge code.
Description: Creates a diagnostic message positive or negative acknowledge, by calling *createGenericHeader* to first construct the generic header with the given type and the required payload specific length. The sourceAddress, targetAddress and response code will then be inserted as payload specific content into the message. This method returns the pointer to the constructed diagnostic message acknowledge.	

The constructed diagnostic message positive acknowledge can then be send through the servers *send* method.

Due to current requirements, the method executed by the *DiagnosticMessageNotification* callback calls the *sendDiagnosticAck* method of the *DoIPServer*, which both constructs and sends a diagnostic message acknowledge.

void DoIPServer::sendDiagnosticAck(bool ackType, unsigned char ackCode)	
bool ackType	Defines if a positive or negative acknowledge should be created.
unsigned char ackCode	The diagnostic message positive or negative acknowledge code.
Description: Creates a diagnostic message acknowledge by calling *createDiagnosticACK* with the given acknowledge type and acknowledge code. Sends the constructed diagnostic message acknowledge using the DoIPServer *sendMessage* method.	

Diagnostic Message negative Acknowledge

Each DoIP entity send the *Diagnostic Message negative Acknowledgement* and close the TCP socket when the received diagnostic message:

- contains a source address which is not activated on the TCP socket
- contains an unknown target address
- exceeds the maximum supported length of the transport protocol of the target network

See section *Diagnostic Message positive Acknowledge* which describes sending a Diagnostic Message positive/negative Acknowledge.

Disconnect TCP Connection

This function can be called through an LUA Script file. This usage will be explained in the chapter "*Examples for LUA Script Files*".

void DoIPServer::triggerDisconnection()
Description: Closes the TCP socket of a server instance and waits for a client applications TCP connection request.

Alive Check Response

The *Alive Check Response* will be sent from the client to the server as a response to an *alive check request*, to keep the connection established. This message type only contains the client's source address as payload type specific content. The DoIPClient can send a alive check response with the *sendAliveCheckResponse* method.

void DoIPClient::sendAliveCheckResponse()
Description: Creates an *alive check response*, by calling *createGenericHeader* to first construct the generic header with the alive check response payload type. The client's source address will then be inserted as payload specific content into the message. Using the system call *write*, the constructed alive check response will be sent to the server.

On server side, the alive check needs to be configured in order to close the established connection when a timeout occurred (see section *Configuration of a DoIPServer instance*). For managing the alive check timeout, the *DoIPServer* uses the *AliveCheckTimer* component.

The *AliveCheckTimer* instance in the DoIPServer will be initiated, after a *RoutingActivationRequest* is received. That's because a timeout should only be possible when a client is routed to the server. The *AliveCheckTimer* contains a *CloseConnectionCallback* which is set to the *aliveCheckTimeout* method and is executed when a timeout occurs.

void DoIPServer::aliveCheckTimeout()
Description: Closes the sockets of this *DoIPServer* instance by using *closeSocket* and *closeUdpSocket*. Calls the saved *CloseConnectionCallback close_connection* from the *DoIPServer* instance, which was set from the application. The callback method in *close_connection* should stop receiving messages and closing the responsible threads in the application.

When the server receives any message from the client, the alive check timer needs to be reseted.

void AliveCheckTimer::resetTimer()
Description: Sets the startTime to the current time and sets the alive check timer to active.

The Alive Check Request, which is send from the server to the client, is currently not implemented.

Integration into Car Simulator

DoIP Simulator

The *DoIPSimulator* is the main component responsible for integrating the DoIP library into the Car-Simulator. When the Car-Simulator is executed, the *start* method of the *DoIPSimulator* will be called.

void DoIPSimulator::start()
Description: Initialize a server instance. Therefore the needed callbacks will be created, the DoIP server will be configured by reading the DoIP configuration file (see section Configuration of the DoIPConfiguration File). Also the threads for the TCP and UDP receiver methods will be created to continuously receive messages from a client. • TCP thread is executing the *DoIPSimulator::listenTcp()* method • UDP thread is executing the *DoIPSimulator::listenUdp()* method This method also initialize the sending of Vehicle Announcment Messages (see section Vehicle Announment Messages)

void DoIPSimulator::listenTcp()
Description: Setup the TCP socket and listen for a connection by using *setupTcpSocket* and *listenTcpConnection* (see section DoIP Server). When a connection is established it checks permantly if a TCP message was received, as long as the status variable *serverActive* is true.

void DoIPSimulator::listenUdp()
Description: Setup the UDP Socket by using *setupUdpSocket* (see section DoIP Server). This method checks permantly if a UDP message was received, as long as the status variable *serverActive* is true.

Both threads, which runs the two described methods are stored in *std::vector<std::thread> doipReceiver.*

The termination of the Car-Simulator is based on if any threads are still running. The two doip receiver threads needs to be added into the applications thread pool, otherwise when only running DoIP in the Car-Simulator, the program will terminate as soon as the *main* method finish.

As mentioned, the *start* method of the DoIPSimulator is also creating the needed callbacks as described in the Diagnostic Message section. The *DiagnosticCallback* will be set to the *receiveFromLibrary* method and *DiagnosticMessageNotification* to *diagnosticMessageReceived*.

bool DoIPSimulator::diagnosticMessageReceived(unsigned char* targetAddress)	
unsigned char* targetAddress	The logical address of a ECU which was targeted by the diagnostic message
Description: This method will be called from the DoIP library to notify the application that a diagnostic message was received. The targetAddress is used to search in the Car-Simulator if a ECU with the given logical address is currently running. Since this method is implemented to simulate specific situations, it currently initializes the sending of a diagnostic message positive acknowledge and returns true, if the search was successfully. Otherwise a diagnostic message negative acknowledge will be send and this method returns false.	

void DoIPSimulator::receiveFromLibrary(unsigned char* address, unsigned char* data, int length)	
unsigned char* address	The logical address of the ECU which was targeted by the diagnostic message
unsigned char* data	Transmitted user data of the received diagnostic message
int length	Length of the user data
Description: This method will be called from the DoIP library with the callback, to pass the logical address, user data and the length of the user data to the application. With the help of the logical address the targeted ECU will be searched with the *findECU* method. If a *ElectronicControlUnit* with the targeted logical address was found, the user data will be passed into the simulated ECU, returning the response to the user data as defined in the ECU's lua script. The response and the logical address of the found ECU will be passed to the *sendDiagnosticResponse* method to initialize sending a diagnostic message back to the client.	

void DoIPSimulator::sendDiagnosticResponse(const std::vector<unsigned char> data, unsigned char* logicalAddress)	
const std::vector<unsigned char> data	The response of an ECU to send back to the client.
unsigned char* logicalAddress	The logical address of the ECU from which the response originates.
Description: This method passes the received response from a ECU back to the DoIP library by calling *receiveDiagnosticPayload* method from the DoIP server instance (see section Diagnostic Message).	

As mentioned in the description of the *receiveFromLibrary* method, the electronic control unit to which a diagnostic message was targeted, needs to be found with the *findECU* method. Therefore, the DoIPSimulator needs to manage all running electronic control units.

std::vector<ElectronicControlUnit*> ecus	Holds a pointer to every currently running electronic control unit.

int DoIPSimulator::findECU(unsigned char* address)	
Unsigned char* address	The electronic control unit with this given logical address which should be found.
Description: Searches if any running electronic control unit has the given logical address. Returns the index of the position in the electronic control unit pool.	

Start arguments

To be able to start the Car-Simulator either with only DoIP, only CAN or both simulations we needed to pass command line arguments when starting the Car-Simulator. The required usage was the following:

./car-simulator doip vcan0	For DoIP and CAN simulation
./car-simulator	For DoIP and CAN simulation
./car-simulator doip	Only for DoIP simulation
./car-simulator vcan0	Only for CAN simulation

To determine which simulations should be started the passed arguments, when starting the Car Simulator, must be evaluated which is handled in the *start_arguments* file. Even though there are some external libraries which makes analysing command line arguments easier, it was decided to not use these for not creating unnecessary overhead.

At the start of the Car-Simulator the *parse_Arguments* method will be called with the main's argument count and argument value parameters.

void startargs::parse_arguments(int argc, char argv)**	
int argc	Count of command line arguments
char** argv	Pointer to the char pointer to the command line argument values
Description: Parses the given argument values to determine which simulations the user requires.	

Depending on which simulations are required, the global flags will be set and can be used in the Car Simulator to selective change the programs flow control. There are three variables used for this:

bool startargs::doip_flag	Used to activate DoIP simulation when required
bool startargs::can_flag	Used to activate CAN simulation when required
std::string startargs::can_device	Name of the CAN device only used when CAN activation is required. Needed to set up the CAN correctly.

Separate start of DoIP and CAN

The user has the opportunity to decide how he wants to start the car-simulator. He can start it with DoIP and CAN, only DoIP or only CAN. See section *start arguments* which describes it in detail. Added constructors to start the car-simulator only with DoIP.

BroadcastReceiver::BroadcastReceiver() {}
Description: The empty constructor of *broadcastReceiver* is called when the server only accepts messages via DoIP.

IsoTpReceiver::IsoTpReceiver() {}
Description: The empty constructor of *IsoTpReceiver* is called when the server only accepts messages via DoIP.

IsoTpSender::IsoTpSender() {}
Description: The empty constructor of *IsoTpSender* is called when the server only accepts messages via DoIP.

ElectronicControlUnit::ElectronicControlUnit(std::unique_ptr<EcuLuaScript> pEcuScript)	
std::unique_ptr<EcuLuaScript> pEcuScript	Represents a corresponding lua script
Description: This constructor of *ElectronicControlUnit* is called when the server only accepts messages via DoIP.	

Enhancement of “ecu_lua_script.cpp” and “electronic_control_unit.cpp”

In case the DoIPServer instance receives a diagnostic message which should be reached through to a simulated ECU, the DoIPServer has to know the Logical Address of the ECU. Therefore there was added a member variable *logicalAddress* for the instances of *ecu_lua_script.cpp* and *electronic_control_unit.cpp* and also a getter function.

Getter functions

uint16_t EcuLuaScript::getLogicalAddress() const	Returns the value of the member variable *logicalAddress_* which is specified in the LUA Script file and used by the ECU constructor as parameter to create an instance
uint16_t ElectronicControlUnit::getLogicalAddress() const	Returns the value of the member variable *logicalAddress_* which is being used by the DoIPServer instance to identify the ECU

Configuration of the DoIPConfiguration File

To configure the DoIPServer instance in the car-simulator there had to be created a module that interprets the LUA files.

This module also has the member variables of the server instance (*Chapter: Configuration of a DoIPServer instance*) as member variables.

DoipConfigurationFile::DoipConfigurationFile()
Description: The default constructor is used when there is no configuration file for the DoIPServer instance. The member variables have the default values as shown in the chapter "*Configuration of a DoIPServer instance*".

DoipConfigurationFile::DoipConfigurationFile(const std::string& luaScript)	
const std::string& luaScript	path and name of an LUA Script file
Description: This function takes the path and name of a LUA Script file and interprets the content and executes instructions which depend on the content. At the moment there is only one instruction which sets the values of the member variables.	

Getter functions

The getter functions are used to save the values of the member variables of the loaded LUA Script instance into the member variables of the *DoIPServer* instance.

Getter function	**Description**
std::string DoipConfigurationFile::getVin() const	Returns the value of the member variable *vin*
std::uint16_t DoipConfigurationFile::getLogicalAddress() const	Returns the value of the member variable *logicalAddress*
unsigned long DoipConfigurationFile::getEid() const	Returns the value of the member variable *eid*
unsigned long DoipConfigurationFile::getGid() const	Returns the value of the member variable *gid*
std::uint8_t DoipConfigurationFile::getFurtherAction() const	Returns the value of the member variable *furtherAction*
int DoipConfigurationFile::getAnnounceNumber() const	Returns the value of the member variable *A_DoIP_Announce_Num*
int DoipConfigurationFile::getAnnounceInterval() const	Returns the value of the member variable *A_DoIP_Announce_Interval*
bool DoipConfigurationFile::getEIDflag() const	Returns the value of the member variable *EIDflag*. This variable specifies if the value of *EID* is specified in the LUA Script file.
std::uint16_t DoipConfigurationFile::getGeneralInactivity() const	Returns the value of the member variable *generalInactivity*

Example for LUA Script files

In this chapter there is an example for the content of a LUA Script file to configure a DoIPServer instance (*carsimconfig.lua*)

```
Main = {

        ANNOUNCE_NUM = 3,
        ANNOUNCE_INTERVAL = 500,


        VIN = “SAJGA51D72XC03718”,
        LOGICAL_ADDRESS = 0x15BF,
        EID = 0x156984A5,
        GID = 0x56F2F2AC,
        FURTHER_ACTION = 0x00,

        T_TCP_General_Inactivity = 600

}
```