ROS-AquaNet-Adapter Design

Step-by-step overview

Version 0.1 – 24/11/2021

Outline

REVISION HISTORY

Version	Description of changes
V0.1	Initial version

1. Introduction

ROS-AquaNet-Adapter presents a software solution for integrating a stack of underwater communication protocols to a Robot Operating System (ROS) framework – a well-known platform for research and development in a field of robotics **[ref to ROS]**.

The stack of underwater communication protocols is presented by AquaNet software suite **[ref to aquanet]**, which implements various underwater-specific protocols on every OSI layer, including L4, L3, L2 as well as providing interfaces to underwater modems, transducers, and underwater channel emulation on L1.

Therefore, the ROS-AquaNet-adapter's main purpose is to seamlessly interconnect both ROS and AquaNet software and thus enable the communication between different underwater robots and their parts using the well-established publisher-subscriber paradigm from the ROS side, and the actual socket interfaces for the underwater communication from the AquaNet side.

Figure 1 presents an underwater communication scenario, where the ros-aquanet-enabled AUVs communicate with each other underwater.

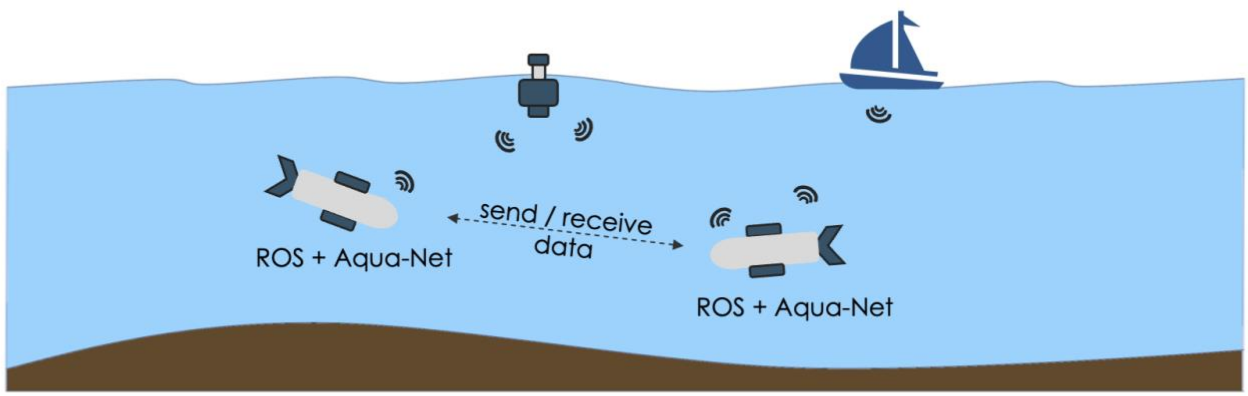

Figure 1 – Possible communication scenario underwater using ROS+Aquanet software stack

2. Communication Structure in ROS

ROS framework implements a native mechanism for communication among different parts inside a robot, based on a well-established publisher-subscriber model. This model presumes a data communication flow between different ROS topics, where a particular robotic part (i.e., a ROS package implementing a specific robotic application, i.e. a motion sensor), needs to send its data to another part of a robot (say, an actuator to perform some task).

To enable such communication scenario, a ROS package publishes its data to the corresponding ROS topic, so that an actuator can retrieve it by subscribing to the same topic name. A generic communication structure in ROS is presented in **Figure 2**.

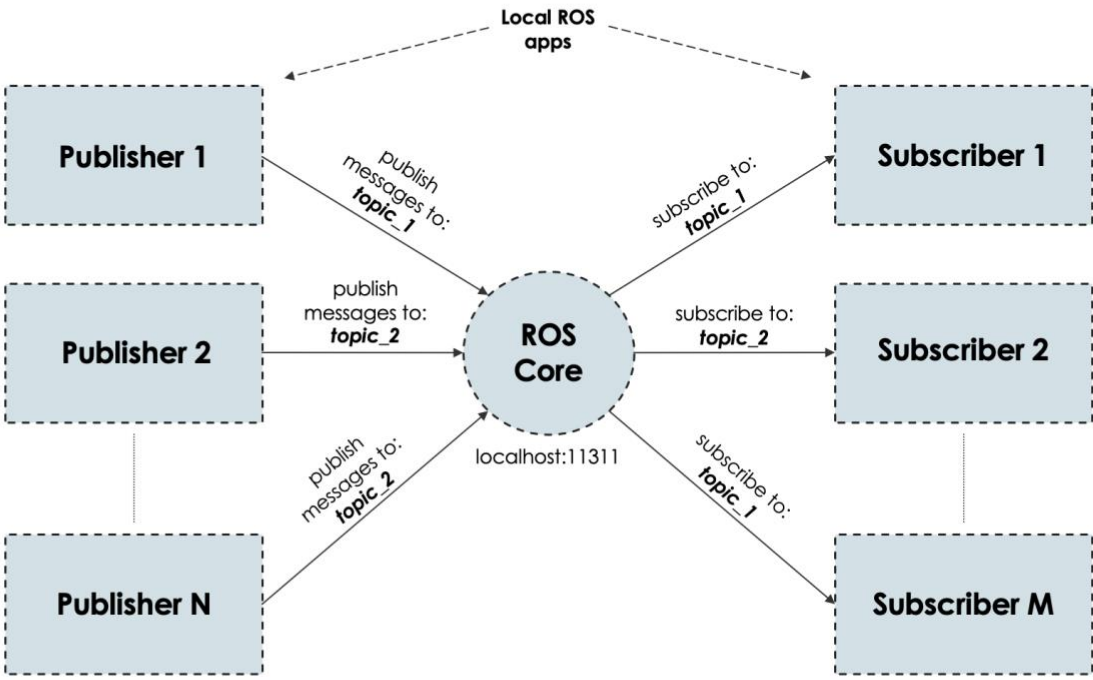

Figure 2 – conventional ROS publisher-subscriber model for the local message exchange

As **Figure 2** shows, there is no direct correspondence between specific senders/publishers and receivers/subscribers. That is, a single sender may broadcast its data to multiple subscribers and, vice-versa, a multiple publishers can advertise their data to a single subscriber or some subset of subscribers.

3. AquaNet Software Architecture

AquaNet presents an independent suite of protocols for underwater communication. It contains protocols implementations for transport (L4), network (L3) and medium access control layers (L2). Besides that, AquaNet implements communication interfaces/drivers to various underwater acoustic modems and transducers, as well as provides a convenient TCP/IP-based acoustic channel emulation layer for quick protocol development and debugging. The generic AquaNet architecture is presented on **Figure 3**.

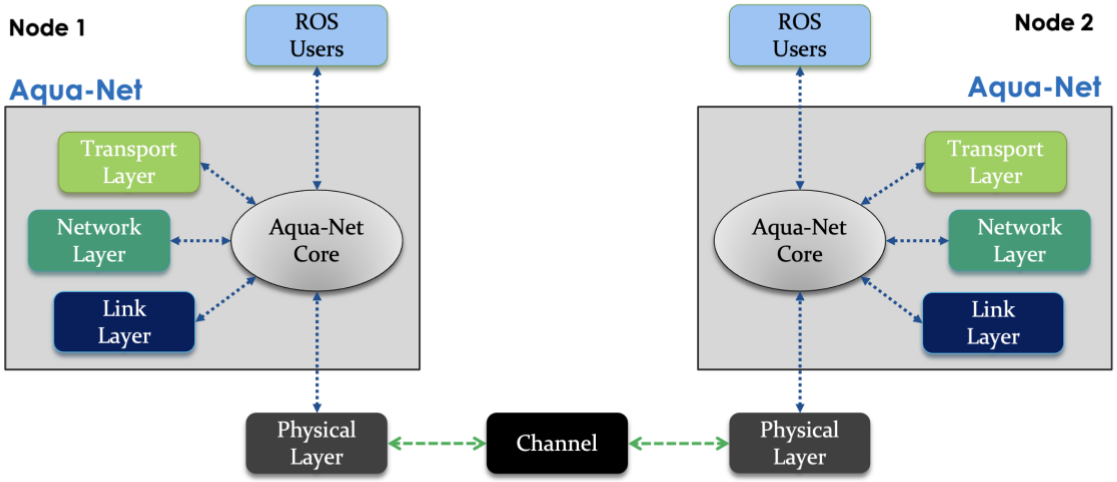

Figure 3 – AquaNet software architecture

As shown on **Figure 3**, the AquaNet protocol stack can be reached over the AquaNet-core layer, available over a standard unix domain socket interface. Besides that, AquaNet provides a convenient UDP-based aquanet-socket interface for the actual communication over the configured protocol stack.

The AquaNet protocol stack configuration is defined in a separate configuration file, where a user may select a specific set of protocols for a given communication scenario, as presented in **Figure 4**.

```
# start the protocol stack
../aquanet_bin/aquanet-stack &
sleep 2
# start the VMDM
#/home/ubuntu/aquanet/trunk/bin/aquanet-vmdc 10.13.13.101 2021 1 10 10 10 &
#../aquanet_bin/aquanet-vmdc 130.160.143.6 2102 1 10 10 10 &
../aquanet_bin/aquanet-vmdc 10.13.13.102 2021 1 10 10 10 &
sleep 4
# start the MAC
../aquanet_bin/aquanet-uwaloha &
sleep 2
# start the routing protocol
../aquanet_bin/aquanet-sroute &
sleep 2
# start the transport layer
../aquanet_bin/aquanet-tra &
```
Figure 4 – AquaNet start-up configuration file example

The supported list of protocols for AquaNet is presented on **Table 1**:

Table 1. AquaNet supported protocol list

Layer 4	Layer 3	Layer 2	Layer 1
Dummy Transport	Static Routing	Broadcast MAC	TCP/IP-based channel emulation (aquanet-hub)
	VBF	Underwater ALOHA	Drivers to acoustic modems, i.e. Benthos

4. ROS-AquaNet Adapter Architecture

ROS-AquaNet-Adapter software is implemented as an independent module/package for the ROS framework. Thus, it can be plugged into any existing ROS installation on a particular robot/hardware to automatically start the AquaNet stack of protocols and to enable a two-way communication among the underwater robots.

ROS-AquaNet-Adapter presents a convenient way to send/receive data over underwater network by providing a ROS developer a unified set of topics for both outbound and inbound communication: aquanet_outbound and aquanet_inbound ROS topics, correspondingly. That is, if a user wants to send data from a particular ROS application to a remote application on the side of a network, a user publishes the corresponding data to the aquanet_outbound topic, stating the destination node and the data to be sent. And, in contrary, for accepting any incoming messages, a user simply subscribes to the aquanet_inbound topic to process the incoming data. The generic ros-aquanet-adapter architecture is presented on **Figure 5**.

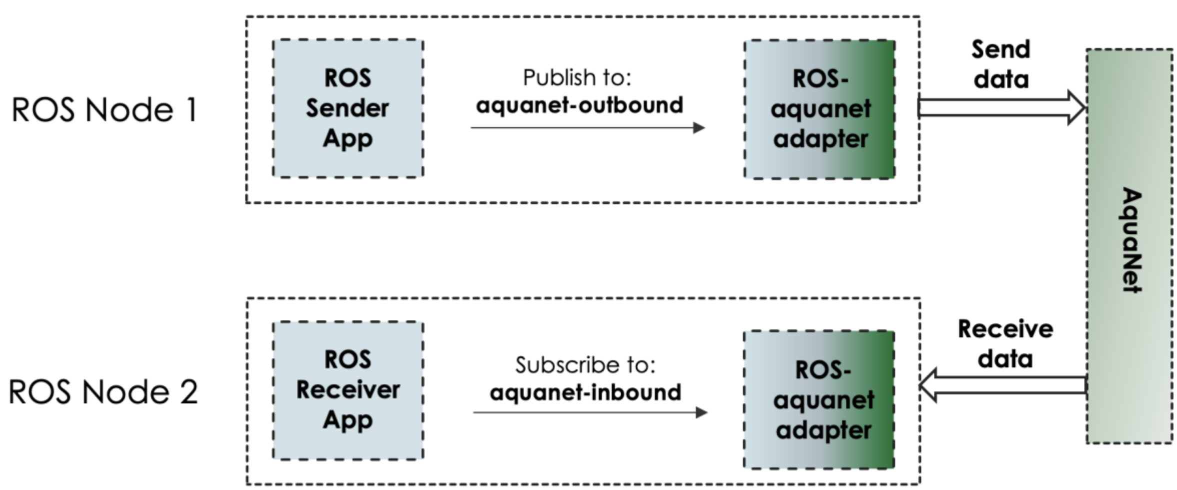

Figure 5 – ROS-AquaNet-Adapter architecture

Figure 5 presents a simple one-way communication scenario, where the ROS application on Node 1 sends data over AquaNet to another ROS application on Node 2. The corresponding Ros-aquanet-adapter instances on both sides are responsible for: retrieving original messages from the local ROS applications; converting the messages into a serialized transport unit/frame for the aquanet stack; sending the frame; receiving the frame and conducting the backward operation with data deserialization and uploading the original messages at the ROS application on the destination side (Node 2).

4.1 DCCL/GBP Message Support

The ROS-AquaNet-Adapter package provides an optional support for the DCCLv3/GPB message marshalling library **[ref to DCCL]**. The library presents an extension to Google Protocol Buffers (GPB) serialization/deserialization tool and it is aimed to preserving the amount of bits required to transmit a particular piece of information, described in the DCCL message fields. An information field can be described as a combination of a data structure (i.e., an integer, double, enumeration, etc.) and the range of values that data structure can take, such as minimum value, maximum value and its precision.

The range of a value and its precision affects the total number of bits required to send it. The higher the range or the precision are, the higher the number of bits required. The eventual size of a message, given the range and the precision, is calculated by DCCL on a compilation/translation stage, using `protoc` compiler.

The message structure is described by Google's `proto2` language. The aquanet message structure can have the following format, described as:

```
import "dccl/option_extensions.proto";

message AquanetMessage {
  option (dccl.msg) = { codec_version: 3
                        id: 1
                        max_bytes: 32 };
  required uint32 ros_msg_id = 1 [(dccl.field) = { min: 0 max: 10 precision: 1 }];
  required double x = 2 [(dccl.field) = { min: -10 max: 10 precision: 1 }];
  required double y = 3 [(dccl.field) = { min: -10 max: 10 precision: 1 }];
  required double z = 4 [(dccl.field) = { min: -10 max: 10 precision: 1 }];
  enum VehicleClass { AUV = 1; USV = 2; SHIP = 3; }
  optional VehicleClass veh_class = 5;
  optional bool battery_ok = 6;
}
```

Figure 6 – DCCL/GPB ROS-AquaNet-Adapter message structure example

5. Operation Example

As a baseline example, showcasing the operational behavior of the ros-aquanet-adapter, a turtlesim package from the standard ROS repository has been selected. The ros-turtlesim application consists of the two main components: the turtle_teleopkey program for capturing the keystrokes from a keyboard; and the turtlesim program which receives the ros-twist messages from the turtle_teleopkey program and visualizes a movement of a turtle on a screen. Thus, a simple communication flow in a traditional local ROS architecture is presented: one side sends messages over a topic to the other side which receives the messages and processes them further. **Figure 7** demonstrates that.

Figure 7 – original ROS turtlesim program communication flow

If, say, we want to separate those two instances (turtle_teleopkey and turtlesim programs) by introducing the underwater network in-between and enabling the message flow over the network, then the ROS-aquanet-adapter comes into play. Using the same communication flow, the aquanet-adapter provides the ros_outbound topic for the turtlesim_teleopkey program and forwards all the messages over the AquaNet stack to the other side, which is running a turtlesim application independently. The turtlesim application on the other side is listening over the aquanet_inbound topic to receive the incoming commands for the turtle movement. The corresponding scheme is presented on **Figure 8**.

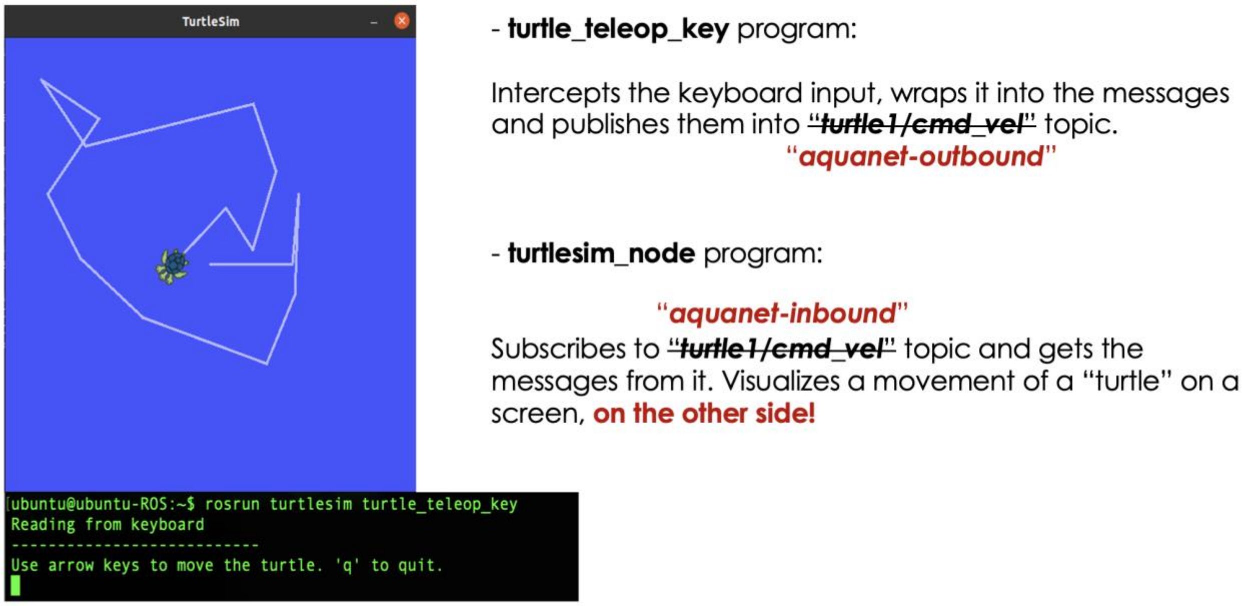

Figure 8 – Modified ROS turtlesim communication over ROS-AquaNet-Adapter

Thus, we successfully enabled the communication over underwater channel among a sender (turtle_teleopkey) and receiver (turtlesim). The main source code of the ROS-AquaNet-adapter program is presented in **Appendix A**.

References

TBD.

Appendix A. ROS-AquaNet-Adapter start.cpp program

```
// This program initializes the aquanet stack of protocols and
// creates outbound/inbound topics for passing the data over aquanet
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/String.h>
#include <iomanip> // for std::setprecision and std::fixed

// dccl
#include "dccl.h"
#include "dccl_messages/aquanet.message.pb.h"

// Socket communication
#include<stdlib.h> //exit(0);
#include<arpa/inet.h>
#include<sys/socket.h>
#include <iostream>
#include <sstream>
#include <limits>
// thread-related stuff
#include <thread>
// Aqua-sockets
#include <sys/un.h>
#include "aquanet_include/aquanet_log.h"
#include "aquanet_include/aquanet_socket.h"

// // udp socket stuff
// #include <sys/types.h>
// #include <sys/socket.h>
// #include <arpa/inet.h>
// #include <netinet/in.h>

char log_file[BUFSIZE];
char* log_file_name = log_file;

// Define inbound and outbound topics for communication over aquanet-enabled nodes
std::string inbound_topic = "aquanet_inbound";
std::string outbound_topic = "aquanet_outbound";

// Initialize publishers
ros::Publisher aquanet_inbound_publisher_twist;
ros::Publisher aquanet_inbound_publisher_string;

// Aquanet socket
int m_socket = -1;
struct sockaddr_aquanet m_to_addr;
dccl::Codec m_codec;

// // udp
// int send_sockfd;
// struct sockaddr_in servaddr;

struct aqua_header {
    unsigned short int frame_len;
    unsigned short int msg_id;
    char pkt_data[450];
};

// serialize/deserialize messages for sending/receiving to/from the sockets
std::string serializeStringMsg(const std_msgs::String::ConstPtr& msg)
{
    // Construct aqua-message with string-message inside
    std::string sent_msg;
    dccl::Codec send_codec;
    // m_codec.load<AquanetMessage>();
    send_codec.load<AquanetMessage>();
    {
        AquanetMessage r_out;
```

```
        r_out.set_ros_msg_id(2);                // 2 - ros string-message;
        r_out.set_body_message(msg->data.c_str());
        r_out.set_veh_class(AquanetMessage::AUV);
        r_out.set_battery_ok(true);
        // std::cout << r_out.ByteSize() << "\n";

        send_codec.encode(&sent_msg, r_out);
        // m_codec.encode(&sent_msg, r_out);
    }

    // Append length of the dccl serialized message

    unsigned char len_byte;

    len_byte = (sent_msg.size());

    // std::cout << sizeof(len_byte) << "\n";
    std::string s((const char*)&(len_byte), sizeof(len_byte));
    // std::cout << s << std::endl;
    // std::cout << s + sent_msg << "\n";

    // return sent_msg;
    return s + sent_msg;
}

// Callback functions for different ros data structures
// twist velocity message
void twistMessageReceived(const geometry_msgs::Twist::ConstPtr& vel)
{
    ROS_INFO_STREAM(std::setprecision(2) << std::fixed << "position=(" << vel->linear.x << "," <<
vel->linear.y << ")" << " angle=" << vel->angular.z);

    // Construct aqua-message with twist-message inside
    std::string sent_msg;
    m_codec.load<AquanetMessage>();
    {
        AquanetMessage r_out;
        r_out.set_ros_msg_id(1);                // 1 - ros twist-message;
        r_out.set_x(vel->angular.z);            // set angular velocity to x
        r_out.set_y(vel->linear.x);            // set linear velocity to y
        r_out.set_z(0);
        r_out.set_veh_class(AquanetMessage::AUV);
        r_out.set_battery_ok(true);

        m_codec.encode(&sent_msg, r_out);
    }

    // Send aqua-message over aqua-socket
    if (aqua_sendto(m_socket, sent_msg.data(), sent_msg.size(), 0, (struct sockaddr *) & m_to_add
r, sizeof (m_to_addr)) < 0) {
        printf("failed to send to the socket");
        perror("m_socket closed");
        exit(1);
    }

    // if (sendto(send_sockfd, sent_msg.data(), sent_msg.size(), 0, (const struct sockaddr *) &se
rvaddr, sizeof(servaddr)))
    // {
    //     printf("failed to send to the socket");
    //     perror("m_socket closed");
    //     exit(1);
    // }

}

// string message
void stringMessageReceived(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("String message: [%s]", msg->data.c_str());

    // std::string send_str;
```

```
    // send_str = "hello";

    // // Construct aqua-message with string-message inside
    // std::string sent_msg;
    // dccl::Codec send_codec;
    // // m_codec.load<AquanetMessage>();
    // send_codec.load<AquanetMessage>();
    // {
    //     AquanetMessage r_out;
    //     r_out.set_ros_msg_id(2);                // 2 - ros string-message;
    //     r_out.set_body_message(msg->data.c_str());
    //     r_out.set_veh_class(AquanetMessage::AUV);
    //     r_out.set_battery_ok(true);
    //     std::cout << r_out.ByteSize() << "\n";

    //     send_codec.encode(&sent_msg, r_out);
    //     // m_codec.encode(&sent_msg, r_out);
    // }

    // // Send aqua-message over aqua-socket
    // if (aqua_sendto(m_socket, sent_msg.data(), sent_msg.size(), 0, (struct sockaddr *) & m_to_
addr, sizeof (m_to_addr)) < 0) {
    //     printf("failed to send to the socket");
    //     perror("m_socket closed");
    //     exit(1);
    // }

    // // connect to server
    // if(connect(send_sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
    // {
    //     printf("\n Error : Connect Failed \n");
    //     exit(0);
    // }

    // std::string sent_msg = serializeStringMsg(msg);

    aqua_header aqua_frame;
    aqua_frame.frame_len = msg->data.size();
    aqua_frame.msg_id = 2;      // 2 - ros string-message;
    memcpy(aqua_frame.pkt_data, msg->data.c_str(), msg->data.size());

    // std::cout << aqua_frame.frame_len << "\n";

    // if (sendto(send_sockfd, &aqua_frame, sizeof(aqua_frame), 0, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0)
    // // if (sendto(send_sockfd, sent_msg.c_str(), sent_msg.size(), 0, (struct sockaddr *) &serv
addr, sizeof(servaddr)) < 0)
    // // if (sendto(send_sockfd, msg->data.c_str(), msg->data.size(), 0, (struct sockaddr *) &se
rvaddr, sizeof(servaddr)) < 0)
    // {
    //     printf("failed to send to the socket");
    //     perror("m_socket closed");
    //     exit(1);
    // }

    // Send aqua-message over aqua-socket
    if (aqua_sendto(m_socket, &aqua_frame, sizeof(aqua_frame), 0, (struct sockaddr *) & m_to_addr
, sizeof (m_to_addr)) < 0) {
        printf("failed to send to the socket");
        perror("m_socket closed");
        exit(1);
    }

}

// string message
void stringMessageReceivedDccl(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("String message: [%s]", msg->data.c_str());

    // Construct aqua-message with string-message inside
```

```
    std::string sent_msg;
    dccl::Codec send_codec;
    // m_codec.load<AquanetMessage>();
    send_codec.load<AquanetMessage>();
    {
        AquanetMessage r_out;
        r_out.set_ros_msg_id(2);                // 2 - ros string-message;
        r_out.set_body_message(msg->data.c_str());
        r_out.set_veh_class(AquanetMessage::AUV);
        r_out.set_battery_ok(true);
        std::cout << r_out.ByteSize() << "\n";

        send_codec.encode(&sent_msg, r_out);
        // m_codec.encode(&sent_msg, r_out);
    }

    // Send aqua-message over aqua-socket
    if (aqua_sendto(m_socket, sent_msg.data(), sent_msg.size(), 0, (struct sockaddr *) & m_to_add
r, sizeof (m_to_addr)) < 0) {
        printf("failed to send to the socket");
        perror("m_socket closed");
        exit(1);
    }

    // // connect to server
    // if(connect(send_sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
    // {
    //     printf("\n Error : Connect Failed \n");
    //     exit(0);
    // }

    // std::string sent_msg = serializeStringMsg(msg);

    // aqua_header aqua_frame;
    // aqua_frame.frame_len = msg->data.size();
    // aqua_frame.msg_id = 2;      // 2 - ros string-message;
    // memcpy(aqua_frame.pkt_data, msg->data.c_str(), msg->data.size());

    // std::cout << aqua_frame.frame_len << "\n";

    // if (sendto(send_sockfd, &aqua_frame, sizeof(aqua_frame), 0, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0)
    // // if (sendto(send_sockfd, sent_msg.c_str(), sent_msg.size(), 0, (struct sockaddr *) &serv
addr, sizeof(servaddr)) < 0)
    // // if (sendto(send_sockfd, msg->data.c_str(), msg->data.size(), 0, (struct sockaddr *) &se
rvaddr, sizeof(servaddr)) < 0)
    // {
    //     printf("failed to send to the socket");
    //     perror("m_socket closed");
    //     exit(1);
    // }

    // // Send aqua-message over aqua-socket
    // if (aqua_sendto(m_socket, &aqua_frame, sizeof(aqua_frame), 0, (struct sockaddr *) & m_to_a
ddr, sizeof (m_to_addr)) < 0) {
    //     printf("failed to send to the socket");
    //     perror("m_socket closed");
    //     exit(1);
    // }

}

// Receive thread
void receiveAqua(int recv_socket)
{
    struct sockaddr_aquanet remote_addr;
    int addr_size = sizeof (remote_addr);

    // struct sockaddr_in remote_addr;
    // memset(&remote_addr, 0, sizeof(remote_addr));
    // socklen_t addr_size = sizeof (remote_addr);
```

```
    // aqua_message received_values;
    // TODO: make it a parameter:
    // char recv_buf[512]; // 512 bytes is the maximum aquanet message size, provided by DCCL-pro
tobuf
    // char len_buf[1];
    char recv_buf[450];
    std::string recv_msg;

    // socklen_t len;
    // // struct sockaddr_in servaddr, cliaddr;
    // // struct sockaddr_in cliaddr;
    // // bzero(&servaddr, sizeof(servaddr));

    // // // Create a UDP Socket
    // struct sockaddr_in servaddr1;
    // int listenfd;
    // listenfd = socket(AF_INET, SOCK_DGRAM, 0);
    // servaddr1.sin_addr.s_addr = htonl(INADDR_ANY);
    // servaddr1.sin_port = htons(57777);
    // servaddr1.sin_family = AF_INET;
    // bind(listenfd, (struct sockaddr*)&servaddr1, sizeof(servaddr1));

    int out_value = 0;
    while (true)
    {
        // out_value = aqua_recvfrom(m_socket, &recv_buf, sizeof(recv_buf), 0, (struct sockaddr *
) & remote_addr, &addr_size);
        // out_value = recvfrom(send_sockfd, &recv_buf, sizeof(recv_buf), 0, (struct sockaddr *)
& remote_addr, &addr_size);
        // out_value = recvfrom(send_sockfd, &recv_buf, sizeof(recv_buf), 0, (struct sockaddr*)NU
LL, NULL);
        // out_value = recvfrom(listenfd, &recv_buf, sizeof(recv_buf), 0, (struct sockaddr*)NULL,
NULL);
        struct aqua_header aqua_frame;
        // memset(&aqua_frame, 0, sizeof (aqua_frame));
        memset(recv_buf, 0, sizeof (recv_buf));
        // out_value = recvfrom(listenfd, &recv_buf, sizeof(recv_buf), 0, (struct sockaddr*)NULL,
NULL);
        out_value = aqua_recvfrom(m_socket, &recv_buf, sizeof(recv_buf), 0, (struct sockaddr *) &
remote_addr, &addr_size);

        if (out_value < 0) {
            // printf(log_file_name, "failed to read from aqua_socket");
            break;
        }
        else if (out_value == 0)
        {
            // nothing was received during the socket timeout
            printf("socket timeout\n");
            continue;
        }

        // memcpy(&aqua_frame, recv_buf, 2);      // read first to bytes to get the frame length
        // memcpy(&aqua_frame, recv_buf, 2+aqua_frame.frame_len);
        memcpy(&aqua_frame, recv_buf, sizeof(recv_buf));
        // std::cout << "frame_len: " << aqua_frame.frame_len << "\n";
        uint16_t frame_size = aqua_frame.frame_len;
        memset(&aqua_frame, 0, sizeof (aqua_frame));
        memcpy(&aqua_frame, recv_buf, 4+frame_size);

        if (aqua_frame.msg_id == 2)         // handle string message
        {
            std_msgs::String msg;
            msg.data = aqua_frame.pkt_data;
            aquanet_inbound_publisher_string.publish(msg);
        }
    }
}

// Receive thread for DCCL
```

```
void receiveAquaDccl(int recv_socket)
{
    struct sockaddr_aquanet remote_addr;
    int addr_size = sizeof (remote_addr);

    char recv_buf[256];
    std::string recv_msg;

    int out_value = 0;
    while (true)
    {
        out_value = aqua_recvfrom(m_socket, &recv_buf, sizeof(recv_buf), 0, (struct sockaddr *) &
remote_addr, &addr_size);
        if (out_value < 0) {
            // printf(log_file_name, "failed to read from aqua_socket");
            break;
        }
        else if (out_value == 0)
        {
            // nothing was received during the socket timeout
            printf("socket timeout\n");
            continue;
        }

        // reconstruct aqua-message
        // std::cout << "SIZEOF BUF: " << sizeof(recv_buf) << "\n";
        std::string ret(recv_buf, sizeof(recv_buf));
        recv_msg = ret;

        // std::string sub_str = recv_msg.substr(1, (uint8_t)recv_msg[0]);
        // std::cout << "len buf: " << +(uint8_t)recv_msg[0] << "\n";
        // std::cout << "msg: " << sub_str << "\n";

        dccl::Codec recv_codec;
        m_codec.load<AquanetMessage>();
        // recv_codec.load<AquanetMessage>();
        if(m_codec.id(recv_msg) == m_codec.id<AquanetMessage>())
        // if(recv_codec.id(sub_str) == recv_codec.id<AquanetMessage>())
        {
            AquanetMessage r_in;
            m_codec.decode(recv_msg, &r_in);
            // recv_codec.decode(sub_str, &r_in);
            std::cout << r_in.ShortDebugString() << std::endl;

            // Publish the messsages to the inbound topic
            // identify ros-message type and re-generate the correpsonding ros-message structure
to be published
            if (r_in.ros_msg_id() == 1)              // handle twist message
            {
                geometry_msgs::Twist twist;
                twist.angular.z = 1.0*r_in.x();
                twist.linear.x = 1.0*r_in.y();
                aquanet_inbound_publisher_twist.publish(twist);
            }
            else if (r_in.ros_msg_id() == 2)        // handle string message
            {
                std_msgs::String msg;
                msg.data = r_in.body_message();
                aquanet_inbound_publisher_string.publish(msg);
            }
            else
            {
                ROS_INFO("Error! Unsupported message type: [%s]", r_in.ros_msg_id());
            }
        }
    }
}

int main(int argc, char **argv)
{
    if (argc < 3)
```

```
    {
        std::cout << "Error! No local and/or destination addresses specified!\n";
        return -1;
    }

    // check the additional dccl flag
    bool dccl_enabled = false;
    if (argc == 4)
    {
        if (strcmp(argv[3], "dccl") == 0)
        {
            // enabling dccl
            dccl_enabled = true;
            std::cout << "WARNING! DCCL serialization is experimental! Use small message rates!\n
";
        }
        else
        {
        std::cout << "Error! Unknown DCCL flag is specified!!\n";
        return -1;
        }
    }

    // Start aquanet stack
    system("cd /home/ubuntu/ros_catkin_ws/src/aquanet_adapter/aquanet_scripts && ./run_aquanet.sh
");

    // Initialize the ROS system and become a node.
    ros::init(argc, argv, "aquanet_node");
    ros::NodeHandle nh;

    // // Creating socket file descriptor
    // if ( (send_sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
    //     perror("socket creation failed");
    //     exit(EXIT_FAILURE);
    // }

    // // struct sockaddr_in servaddr;
    // // clear servaddr
    // bzero(&servaddr, sizeof(servaddr));

    // if (std::stoi(argv[1]) == 1)
    // {
    //     servaddr.sin_addr.s_addr = inet_addr("10.13.13.101");
    // }
    // else
    // {
    //     servaddr.sin_addr.s_addr = inet_addr("10.13.13.102");
    // }
    // servaddr.sin_port = htons(57777);
    // servaddr.sin_family = AF_INET;

    // Create socket
    if ((m_socket = aqua_socket(AF_AQUANET, SOCK_AQUANET, 0)) < 0) {
        printf("socket creation failed\n");
        perror("m_socket closed");
        exit(1);
    }

    m_to_addr.sin_family = AF_AQUANET;
    // Set local and dest aquanet addresses from CLI
    m_to_addr.sin_addr.s_addr = std::stoi(argv[1]);
    m_to_addr.sin_addr.d_addr = std::stoi(argv[2]);

    // Create subscriber objects for different message types
    if (dccl_enabled)
    {
        // twist
        ros::Subscriber sub_twist = nh.subscribe("aquanet_outbound_twist/", 1, twistMessageReceiv
ed);  // TODO: change name to Dccl
```

```
        aquanet_inbound_publisher_twist = nh.advertise<geometry_msgs::Twist>("aquanet_inbound_twi
st", 1);
        // string
        ros::Subscriber sub_string = nh.subscribe("aquanet_outbound_string/", 1, stringMessageRec
eivedDccl);
        aquanet_inbound_publisher_string = nh.advertise<std_msgs::String>("aquanet_inbound_string
", 1);

        // Start the receive thread
        std::thread t1(receiveAquaDccl, m_socket);
        // Let ROS take over.
        ros::spin();
    }
    else
    {
        // twist
        ros::Subscriber sub_twist = nh.subscribe("aquanet_outbound_twist/", 1, twistMessageReceiv
ed);
        aquanet_inbound_publisher_twist = nh.advertise<geometry_msgs::Twist>("aquanet_inbound_twi
st", 1);
        // string
        ros::Subscriber sub_string = nh.subscribe("aquanet_outbound_string/", 1, stringMessageRec
eived);
        aquanet_inbound_publisher_string = nh.advertise<std_msgs::String>("aquanet_inbound_string
", 1);

        // Start the receive thread
        std::thread t1(receiveAqua, m_socket);
        // Let ROS take over.
        ros::spin();
    }
}
```