Setting up ROS development environment

Step-by-step guide

Version 0.4

Outline

1. INSTALLING AND BUILDING ROS SYSTEM FROM SOURCE 3

2. REBUILDING A SPECIFIC ROS PACKAGE 4

3. RUNNING A PACKAGE 5

4. CREATING A NEW EXECUTABLE FILE WITHIN A PACKAGE 6

5. INTEGRATING AQUANET CODE INTO ROS 9

5.1 BUILDING TURTLESIM-CLIENT 10

5.2 BUILDING TURTLESIM-SERVER 11

6. RUNNING AQUANET WITH ROS 12

6.1 RUNNING THE CLIENT SIDE 15

6.2 RUNNING THE SERVER SIDE 17

7. CREATING A ROS PACKAGE 20

APPENDIX A. CODE EXAMPLE FOR TURTLE_TELEOP_KEY_CLIENT 21

APPENDIX B. CODE EXAMPLE FOR TURTLE_TELEOP_KEY_SERVER 25

REVISION HISTORY

Version	**Description of changes**
V0.1	Initial version
V0.2	- Section 5 changed to Section 7 Section 1: - added more dependencies to sources.list - added command to resolve ROS dependencies - change the order of the command - added $ROS_FOLDER variable Section 4: - changed TARGETS definition in CMakeLists - added note about "[rosrun] couldn't find executable" - added note about listing available executables - added Section 5: integrating AquaNet to ROS - added Section 6: running AquaNet and ROS Section 7: - added TODO for creating a separate AquaNet ROS package - added Appendix A: source code for teleop_client - added Appendix B: source code for teleop_server
V0.3	Section 4 and 5: - $ROS_FOLDER and $AQUANET_FOLDER are used now, instead of absolute paths in some commands Section 5: - added "source $ROS_FOLDER/install_isolated/setup.bash" command before recompiling Added Sections 5.1 and 5.2 describing the build process for client and server Added Sections 6.1 and 6.1 describing how to run the client and the server sides Appendix B: - fixed the duplicate name issue: client is renamed to server
V0.4	Section 6: - added turtlesim and turtlesim + aquanet demo explanation - reduced code size in Appendix A and B

1. Installing and building ROS system from source

The complete guide how to install and build the latest ROS code is available here:
http://wiki.ros.org/noetic/Installation/Source

Here are some important steps from the link above:

- add ROS dependencies into repository sources:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) m
ain" > /etc/apt/sources.list.d/ros-latest.list'

sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6
E31E6BADE8868B172B4F42ED6FBAB17C654

sudo apt-get update
```

- install ROS-install packages

```
sudo apt-get install python3-rosdep2
sudo apt install python3-rosinstall-generator
```

- create catkin workspace:

```
mkdir ~/ros_catkin_ws
cd ~/ros_catkin_ws
```

- resolve ROS dependencies

```
mkdir ./src

rosdep install --from-paths ./src --ignore-packages-from-source --rosdistro n
oetic -y
```

- install the required dependencies, for Ubuntu:

```
sudo apt-get install python3-rosdep python3-rosinstall-generator python3-vcsto
ol build-essential

sudo rosdep init
rosdep update
```

- download the source code:

```
rosinstall_generator desktop --rosdistro noetic --deps --tar > noetic-desktop
.rosinstall

vcs import --input noetic-desktop.rosinstall ./src
```

- build the catkin workspace:

```
export ROS_FOLDER=/home/ubuntu/ros_catkin_ws
./src/catkin/bin/catkin_make_isolated --install -DCMAKE_BUILD_TYPE=Release
```

This will take some time, around 10-30 minutes, depending on CPU resources and Internet connection speed.

- initialize the environmental variables (**important**):

```
source $ROS_FOLDER/install_isolated/setup.bash
```

Note: this command **must be executed** every time a new bash-terminal is initialized. Otherwise, the consecutive ros-commands executed next will not be found.

Recommended: alternatively, you can create/modify a `bash_profile` file, which would be initializing the environment automatically for every new terminal session:

```
cd ~
nano .bash_profile
```

- insert the following lines into `bash_profile`:

```
#!/bin/bash
source ~/ros_catkin_ws/install_isolated/setup.bash
export ROS_FOLDER=/home/ubuntu/ros_catkin_ws
export AQUANET_FOLDER=/home/ubuntu/aquanet
```

- save and close the file

Now, whenever a new bash-session is started, all necessary paths and variables will be automatically initialized.

2. Rebuilding a specific ROS package

After the initial build, we can start working with the source code of ROS, meaning to modify the existing or create new files.
In order to recompile only the new code, without rebuilding the entire ROS project, we can explicitly tell the *catkin_make_isolated* program to compile a specific package.

For example, the command below will only **recompile turtlesim package** and reinstall it at the same isolated environment:

```
./src/catkin/bin/catkin_make_isolated --pkg turtlesim --install -DCMAKE_BUILD_
TYPE=Release
```

3. Running a package

Before running any executable from a ROS package, we need to make sure that the *roscore* program is started.
The *roscore* is a central program in ROS, which receives and forwards ROS-messages via specific port number (**port 11311**, by default).
To run it, simply execute the following command in a new terminal:

```
roscore
```

Note: please make sure that the environmental variables are set up in the new terminal (see the last command in Section 1).

To run an executable from a ROS package, the *rosrun* command is used, with the following input format:

```
rosrun --help
Usage: rosrun [--prefix cmd] [--debug] PACKAGE EXECUTABLE [ARGS]
  rosrun will locate PACKAGE and try to find
  an executable named EXECUTABLE in the PACKAGE tree.
  If it finds it, it will run it with ARGS.
```

where:
PACKAGE – name of the ROS package (e.g., *turtlesim*)
EXECUTABLE – name of the executable within a package (e.g., *turtle_teleop_key*)

For example, to run the program for capturing keyboard input and publishing ROS messages to a “turtle visualization” robot, the following command is executed:

```
rosrun turtlesim turtle_teleop_key
```

4. Creating a new executable file within a package

To create a new executable file within an existing package, the following steps should be made:

- place a new source-code file under `/src/<package_name>/` folder
- modify *CMakeLists.txt* inside the package folder to tell the build system to create and install a new executable

Example:
Let's create a copy of the original *turtle_teleop_key* program, rename it and place it into the same folder:

- check where we are now, make sure that we're in the root folder (**ros_catkin_ws,** by default):

```
cd $ROS_FOLDER

pwd

/home/ubuntu/ros_catkin_ws
```

- copy and rename the existing *turtle_teleop_key.cpp* file:

```
cp src/ros_tutorials/turtlesim/tutorials/teleop_turtle_key.cpp src/ros_tutoria
ls/turtlesim/tutorials/teleop_turtle_key_client.cpp
```

- open *CMakeLists.txt* file

```
vi src/ros_tutorials/turtlesim/CMakeLists.txt
```

- add the following text blocks there (marked in **RED**):

```
set(turtlesim_node_SRCS
  src/turtlesim.cpp
  src/turtle.cpp
  src/turtle_frame.cpp
)
set(turtlesim_node_HDRS
  include/turtlesim/turtle_frame.h
)

qt5_wrap_cpp(turtlesim_node_MOCS ${turtlesim_node_HDRS})


add_executable(turtlesim_node ${turtlesim_node_SRCS} ${turtlesim_node_MOCS})
target_link_libraries(turtlesim_node Qt5::Widgets ${catkin_LIBRARIES} ${Boost_LIBRARIES})
add_dependencies(turtlesim_node turtlesim_gencpp)

add_executable(turtle_teleop_key tutorials/teleop_turtle_key.cpp)
target_link_libraries(turtle_teleop_key ${catkin_LIBRARIES})
add_dependencies(turtle_teleop_key turtlesim_gencpp)


add_executable(turtle_teleop_key_client tutorials/teleop_turtle_key_client.cpp)
target_link_libraries(turtle_teleop_key_client ${catkin_LIBRARIES})
add_dependencies(turtle_teleop_key_client turtlesim_gencpp)

add_executable(draw_square tutorials/draw_square.cpp)
target_link_libraries(draw_square ${catkin_LIBRARIES} ${Boost_LIBRARIES})
add_dependencies(draw_square turtlesim_gencpp)

add_executable(mimic tutorials/mimic.cpp)
target_link_libraries(mimic ${catkin_LIBRARIES})
add_dependencies(mimic turtlesim_gencpp)

install(TARGETS turtlesim_node turtle_teleop_key turtle_teleop_key_client draw_square mimic
  RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})

install(DIRECTORY images
  DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
  FILES_MATCHING PATTERN "*.png" PATTERN "*.svg")

```

Block (1) tells the compiler to build a new source-file, named *teleop_turtle_key_client.cpp*, and specifies which ROS-dependencies it needs.

Block (2) tells the compiler to install the file into default catkin binary destination (CATKIN_PACKAGE_BIN_DESTINATION).

After saving the changes in *CMakeLists.txt* file, rebuild the entire *turtlesim* package using the following command:

```
./src/catkin/bin/catkin_make_isolated --pkg turtlesim --install -DCMAKE_BUILD_TYPE=Release
```

Note:
If the following error occurs:

```
[rosrun] Couldn't find executable named turtle_teleop_key_client below /opt/ros/noetic/share/turtlesim
```

Please make sure that there are no ROS packages installed from the packet manager:

```
sudo apt-get remove ros*
```

This is necessary, since we're building everything from source, and we need to make sure that no alternative ROS executables exist outside the `$ROS_FOLDER` path.

After a successful build, we can now execute the newly built file using the standard command:

```
rosrun turtlesim turtle_teleop_key_client
```

Note:
A list of all available executables within a ROS package can be listed by double-tapping after the package name, like this:

```
rosrun turtlesim <tab><tab>

draw_square                turtlesim_node             turtle_teleop_key_client

mimic                      turtle_teleop_key
```

Now we're ready to create/modify/build new executables in a ROS package.

Section 5 will provide a description on how to integrate AquaNet source code into ROS.

Section 6 will describe how to run AquaNet alongside ROS.

Section 7 will briefly describe how to create your own package in ROS, if we would need one at some point later.

5. Integrating AquaNet code into ROS

To use AquaNet socket interface inside ROS, the following steps should be made:

- get and compile the latest aquanet code:

install `subversion`:

```
sudo apt-get install subversion
```

- get the code:

```
svn co svn+ssh://dmitrii@hudson.ccny.cuny.edu/var/svn/repos/aquanet aquanet
export AQUANET_FOLDER=/home/ubuntu/aquanet
```

- install dependencies:

```
sudo apt-get install libglib2.0-dev
sudo apt-get install libgsl-dev
```

- build AquaNet:

```
cd $AQUANET_FOLDER/trunk
make
```

Now, when AquaNet is built in a separate folder, the following headers and configuration files should be linked to a ROS package (`turtlesim` is used as example):

Header files:

```
ln -s $AQUANET_FOLDER/trunk/aquanet_log.h $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/aquanet_log.h

ln -s $AQUANET_FOLDER/trunk/aquanet_netif.h $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/aquanet_netif.h

ln -s $AQUANET_FOLDER/trunk/aquanet_pdu.h $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/aquanet_pdu.h

ln -s $AQUANET_FOLDER/trunk/aquanet_socket.h $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/aquanet_socket.h

ln -s $AQUANET_FOLDER/trunk/aquanet_time.h $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/aquanet_time.h
```

Configuration files:

```
ln -s $AQUANET_FOLDER/trunk/test_example/mesh/node2/config_add.cfg $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/config_add.cfg

ln -s $AQUANET_FOLDER/trunk/test_example/mesh/node2/config_arp.cfg $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/config_arp.cfg

ln -s $AQUANET_FOLDER/trunk/test_example/mesh/node2/config_conn.cfg $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/config_conn.cfg

ln -s $AQUANET_FOLDER/trunk/test_example/mesh/node2/config_net.cfg $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials/config_net.cfg
```

5.1 Building turtlesim-client

Now, when all the necessary files are linked to the `turtlesim` package, let's modify the `turtle_teleop_key_client.cpp` example to enable the communication via `aqua_sockets`.

You may use the corresponding source codes for the client and the server parts in **Appendix A** and **B**, correspondingly.

- copy and paste the code from **Appendix A** into `turtle_teleop_key_client.cpp`
- make sure that `CMakeLists.txt` file has a new executable `turtle_teleop_key_client` (see **Section 4**)
- rebuild ROS package with the modified source code, containing aqua-sockets:

```
cd $ROS_FOLDER

source $ROS_FOLDER/install_isolated/setup.bash

./src/catkin/bin/catkin_make_isolated --pkg turtlesim --install -DCMAKE_BUILD_TYPE=Release
```

Now, the `turtle_teleop_key_client` executable will be using `aqua_sockets` for communication with the rest of the AquaNet stack.

5.2 Building turtlesim-server

On the server-side, another executable should be compiled to receive incoming messages from the AQUA-sockets. Use the following steps to build the server-part:

- copy and paste the code from **Appendix B** into `turtle_teleop_key_server.cpp`
- make sure that `CMakeLists.txt` file has a new executable `turtle_teleop_key_server` (see **Section 4**). Add the following lines there:

```
add_executable(turtle_teleop_key_server tutorials/teleop_turtle_key_server.cpp)

target_link_libraries(turtle_teleop_key_server ${catkin_LIBRARIES})

add_dependencies(turtle_teleop_key_server turtlesim_gencpp)
```

```
install(TARGETS turtlesim_node turtle_teleop_key turtle_teleop_key_server draw_square mimic

  RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

- rebuild ROS package with the modified source code, containing aqua-sockets:

```
cd $ROS_FOLDER

source $ROS_FOLDER/install_isolated/setup.bash

./src/catkin/bin/catkin_make_isolated --pkg turtlesim --install -DCMAKE_BUILD_TYPE=Release
```

Now you should be able to run both client and server parts via AquaNet system.

6. Running AquaNet with ROS

The section will talk about how to run the modified `turtlesim` demo with Aqua-Net.

The original `turtlesim` demo (before introducing AquaNet and changing the source code) has the following two main components:

- `turtle_teleop_key` program:
Intercepts the keyboard input, wraps it into `ros::twist` messages and publishes them into "*turtle1/cmd_vel*" topic.

- `turtlesim_node` program:
Subscribes to "*turtle1/cmd_vel*" topic and gets the messages from it. Visualizes a movement of a "turtle" on a screen.

The corresponding message flow between those components is shown on Figure 1.

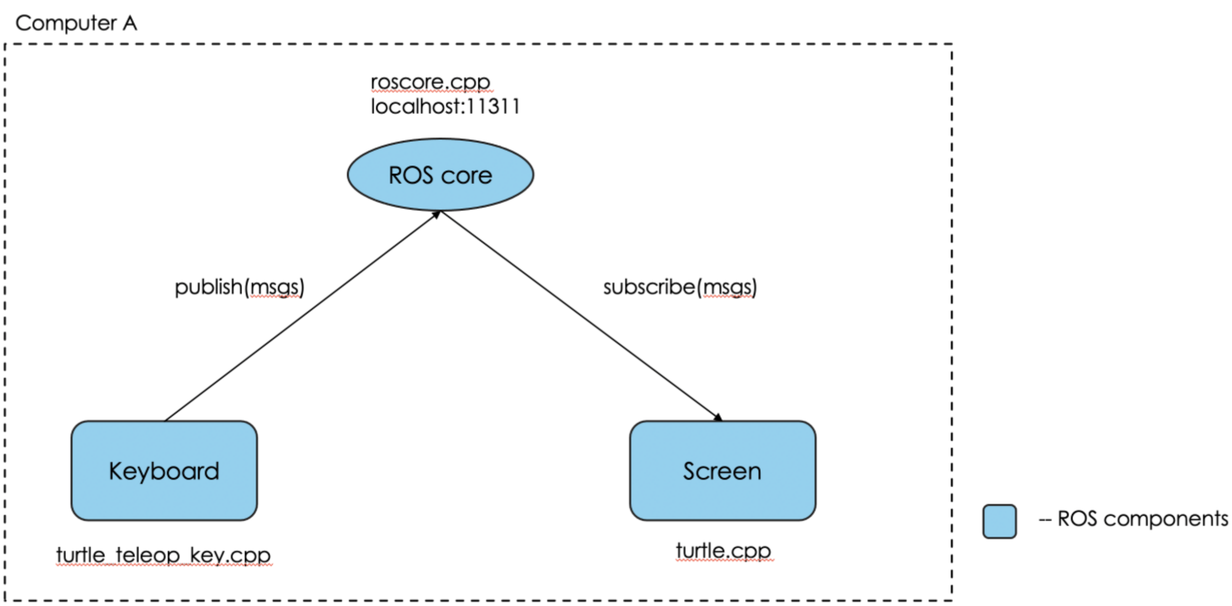

Figure 1. Original message flow inside the turtlesim package

The operation example of the original `turtlesim` package is shown on Figure 2. By pressing the arrow buttons on the keyboard where `turtle_teleop_key` program is executed, a user can control the movements of a turtle, visualized in the `turtlesim_node` program.

Figure 2. Turtlesim operation example

Now, let's take a look at the modified version of the `turtlesim` package, with the introduced aquanet stack. There are 3 main components now:

- `turtle_teleop_key_client` program:
Intercepts the keyboard input and sends it to the aquanet socket to the other machine.

- `turtle_teleop_key_server` program:
Receives the incoming messages from the aquanet socket, wraps them into `ros::twist` messages and publishes them to "*turtle1/cmd_vel*" topic on the receiver side.

- `turtlesim_node` program:
This is the same program as in the original turtlesim package. It is running on the receiver side and subscribes to the messages, published by `turtle_teleop_key_server` program.

The corresponding message flow in the modified turtlesim + AquaNet structure is shown on Figure 3.

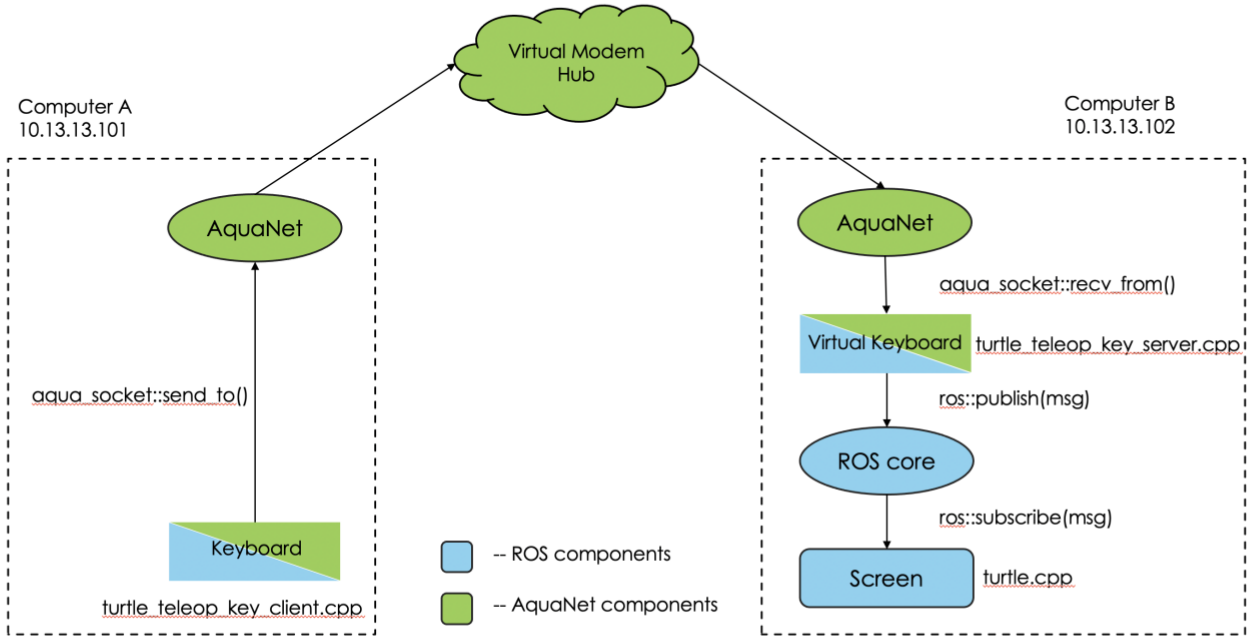

Figure 3. Modified turtlesim + AquaNet message flow

The operation example of the modified turtlesim + AquaNet program is shown on Figure 4. The workflow is very similar to the original turtlesim behavior, however, the only difference is that the client-machine forwards the keyboard input to the server-machine via AquaNet socket. Upon receiving the keyboard messages, the server machine wraps them back up into ros-messages, publishes them into the same "*turtle1/cmd_vel*" topic to be visualized by `turtlesim_node` locally.

Figure 4. Modified turtlesim + AquaNet operation example

6.1 Running the client side

After we enabled `aqua_socket` communication inside the ROS executable, we can now initialize the rest of the AquaNet stack (L1-L4) and use the `turtle_teleop_key_client.cpp` on L5 – the application layer of AquaNet.

To initialize the AquaNet stack from L1 to L4, do the following:

- run AquaNet Virtual Modem Server (VMDS) to emulate L1 over TCP/IP:

```
cd $AQUANET_FOLDER/trunk
./bin/aquanet-vmds <VMDS_PORT> &
```

where:
<VMDS_PORT> - port number to VMDS server, e.g. 2021

- create a bash-script inside the ROS package to start the layers for L2 to L4:

```
cd $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials
touch vm_aloha_sroute_tra_ros.sh
```

- insert the following code into vm_aloha_sroute_tra_ros.sh file:

```
#!/bin/sh
# Initialize L2-L4 modules on localhost

# start the protocol stack
$AQUANET_FOLDER/trunk/bin/aquanet-stack &
sleep 2
# start the VMDM
# modify IP and Port number, if necessary
# $AQUANET_FOLDER/trunk/bin/aquanet-vmdc 127.0.0.1 2021 2 20 20 20 &
$AQUANET_FOLDER/trunk/bin/aquanet-vmdc <VMDS_IP> <VMDS_PORT> 1 20 20 20 &
sleep 4
# start the MAC
$AQUANET_FOLDER/trunk/bin/aquanet-uwaloha &
sleep 2
# start the routing protocol
$AQUANET_FOLDER/trunk/bin/aquanet-sroute &
sleep 2
# start the transport layer
$AQUANET_FOLDER/trunk/bin/aquanet-tra &
```

- please change **<VMDS_IP>** and **<VMDS_PORT>** with the actual IP address and port number of the VMDS emulator:
e.g., if the VMDS server is running on the same machine where the client is running, then the IP address is local: 127.0.0.1

- run the script:

```
chmod +x vm_aloha_sroute_tra_ros.sh
./vm_aloha_sroute_tra_ros.sh
```

At this point, we have started the VMDM server and clients on L1, and the rest of the stack up to L4.

Now, to start the application layer, execute the modified `turtle_teleop_key_client` in the same folder:

```
source $ROS_FOLDER/install_isolated/setup.bash
```

```
roscore
```

- in a new terminal:

```
cd $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials
rosrun turtlesim turtle_teleop_key_client
```

If everything is initialized correctly, you should be able to see **AQUA_*** socket files inside the same directory:

```
ls $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials
AQUA_APP  AQUA_TRA        aquanet_socket.h  config_conn.cfg  teleop_turtle_key.cpp
AQUA_MAC  aquanet_log.h   aquanet_time.h    config_net.cfg   teleop_turtle_key_client.cpp
AQUA_NET  aquanet_netif.h config_add.cfg    draw_square.cpp  vm_aloha_sroute_tra_ros.sh
AQUA_PHY  aquanet_pdu.h   config_arp.cfg    mimic.cpp
```

6.2 Running the server side

To connect another machine to the same stack, use the IP address and port of the machine where VMDM-server is started.

The rest of the steps are the same, except for an application – you may want to change it to `turtle_teleop_key_server` to be able to receive data from aqua-socket:

- create a bash-script inside the ROS package to start the layers for L2 to L4:

```
cd $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials
touch vm_aloha_sroute_tra_ros_server.sh
```

- insert the following code into `vm_aloha_sroute_tra_ros_server.sh` file:

```
#!/bin/sh
# Initialize L2-L4 modules on localhost

# start the protocol stack
$AQUANET_FOLDER/trunk/bin/aquanet-stack &
sleep 2
# start the VMDM
# modify IP and Port number, if necessary
# $AQUANET_FOLDER /trunk/bin/aquanet-vmdc 127.0.0.1 2021 2 20 20 20 &
$AQUANET_FOLDER/trunk/bin/aquanet-vmdc <VMDS_IP> <VMDS_PORT> 2 20 20 20 &
sleep 4
# start the MAC
$AQUANET_FOLDER/trunk/bin/aquanet-uwaloha &
sleep 2
# start the routing protocol
$AQUANET_FOLDER/trunk/bin/aquanet-sroute &
sleep 2
# start the transport layer
$AQUANET_FOLDER/trunk/bin/aquanet-tra &
```

- please change **<VMDS_IP>** and **<VMDS_PORT>** with the actual IP address and port number of the VMDS emulator:
e.g., if the VMDS server is still running on the machine where the client-program is running, then the IP address should be a public address of the client machine, e.g.: 10.13.13.101.

- run the script:

```
chmod +x vm_aloha_sroute_tra_ros_server.sh
./vm_aloha_sroute_tra_ros_server.sh
```

At this point, we have started the server side and the rest of the stack up to L4.

```
source $ROS_FOLDER/install_isolated/setup.bash
```

```
roscore
```

Now, in a new terminal, start the server-application layer by executing the modified `turtle_teleop_key_server` in the same folder:

```
cd $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials
rosrun turtlesim turtle_teleop_key_server
```

Again, if everything is initialized correctly, you should be able to see **AQUA_*** socket files inside the same directory:

```
ls $ROS_FOLDER/src/ros_tutorials/turtlesim/tutorials
AQUA_APP  AQUA_TRA        aquanet_socket.h  config_conn.cfg  teleop_turtle_key.cpp
AQUA_MAC  aquanet_log.h   aquanet_time.h    config_net.cfg   teleop_turtle_key_client.cpp
AQUA_NET  aquanet_netif.h config_add.cfg    draw_square.cpp  vm_aloha_sroute_tra_ros.sh
AQUA_PHY  aquanet_pdu.h   config_arp.cfg    mimic.cpp
```

Note:
The AquaNet Virtual Modem Server (VMDS) for L1 emulation can be run on a separate machine, independently from both client and server machines. If that is the case, please make sure that the IP address and the port number of the VMDS server are publicly accessible for both the client and the server parts. The corresponding .sh files must also be modified to reflect the public IP address of the VMDS emulator.

7. Creating a ROS package

TODO: Create a separate AquaNet ROS-package, which is subscribing to a given topic to receive messages and send them to the other side via aqua-sockets.

A detailed guide on how to create and build custom ROS packages is provided in the official tutorial over here:
http://wiki.ros.org/ROS/Tutorials/CreatingPackage
http://wiki.ros.org/ROS/Tutorials/BuildingPackages

In short, the following steps should be made:

- create meta-information about a new package:

Create an *.xml* file containing the name, description, license and dependencies of a new package (see the format of *.xml* file in the link above).

- generate *CMakeLists.txt* file with the instructions to the build system, using the following command:

```
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

where:

<package_name>	- name of new package, specified in the metainformation
[depend1…N]	- dependencies for a new package (such as roscpp, std_msgs, etc.)

Appendix A. Code example for turtle_teleop_key_client

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <signal.h>
#include <stdio.h>
#ifndef _WIN32
# include <termios.h>
# include <unistd.h>
#else
# include <windows.h>
#endif
// Socket communication
#include<stdlib.h> //exit(0);
#include<arpa/inet.h>
#include<sys/socket.h>
#include <iostream>
#include <sstream>
#include <limits>
// Aqua-sockets
#include <sys/un.h>
#include "aquanet_log.h"
#include "aquanet_socket.h"
char log_file[BUFSIZE];
char* log_file_name = log_file;
#define SERVER "10.13.13.102"
#define BUFLEN 512 //Max length of buffer
#define PORT 8888  //The port on which to send data
const std::size_t maxPrecision = std::numeric_limits<double>::digits;
#define KEYCODE_RIGHT 0x43
#define KEYCODE_LEFT 0x44
#define KEYCODE_UP 0x41
#define KEYCODE_DOWN 0x42
#define KEYCODE_B 0x62
#define KEYCODE_C 0x63
#define KEYCODE_D 0x64
#define KEYCODE_E 0x65
#define KEYCODE_F 0x66
#define KEYCODE_G 0x67
#define KEYCODE_Q 0x71
#define KEYCODE_R 0x72
#define KEYCODE_T 0x74
#define KEYCODE_V 0x76
class KeyboardReader
{
public:
  KeyboardReader()
#ifndef _WIN32
    : kfd(0)
#endif
  {
#ifndef _WIN32
    // get the console in raw mode
    tcgetattr(kfd, &cooked);
    struct termios raw;
    memcpy(&raw, &cooked, sizeof(struct termios));
    raw.c_lflag &=~ (ICANON | ECHO);
    // Setting a new line, then end of file
    raw.c_cc[VEOL] = 1;
    raw.c_cc[VEOF] = 2;
    tcsetattr(kfd, TCSANOW, &raw);
#endif
  }
  void readOne(char * c)
  {
#ifndef _WIN32
    int rc = read(kfd, c, 1);
    if (rc < 0)
    {
      throw std::runtime_error("read failed");
```

```
    }
#else
    for(;;)
    {
      HANDLE handle = GetStdHandle(STD_INPUT_HANDLE);
      INPUT_RECORD buffer;
      DWORD events;
      PeekConsoleInput(handle, &buffer, 1, &events);
      if(events > 0)
      {
        ReadConsoleInput(handle, &buffer, 1, &events);
        if (buffer.Event.KeyEvent.wVirtualKeyCode == VK_LEFT)
        {
          *c = KEYCODE_LEFT;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == VK_UP)
        {
          *c = KEYCODE_UP;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == VK_RIGHT)
        {
          *c = KEYCODE_RIGHT;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == VK_DOWN)
        {
          *c = KEYCODE_DOWN;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x42)
        {
          *c = KEYCODE_B;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x43)
        {
          *c = KEYCODE_C;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x44)
        {
          *c = KEYCODE_D;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x45)
        {
          *c = KEYCODE_E;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x46)
        {
          *c = KEYCODE_F;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x47)
        {
          *c = KEYCODE_G;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x51)
        {
          *c = KEYCODE_Q;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x52)
        {
          *c = KEYCODE_R;
          return;
        }
```

```
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x54)
        {
          *c = KEYCODE_T;
          return;
        }
        else if (buffer.Event.KeyEvent.wVirtualKeyCode == 0x56)
        {
          *c = KEYCODE_V;
          return;
        }
      }
    }
#endif
  }
  void shutdown()
  {
#ifndef _WIN32
    tcsetattr(kfd, TCSANOW, &cooked);
#endif
  }
private:
#ifndef _WIN32
  int kfd;
  struct termios cooked;
#endif
};
KeyboardReader input;
class TeleopTurtle
{
public:
  TeleopTurtle();
  void keyLoop();
  void close_socket();
private:
  int m_socket = -1;
  ros::NodeHandle nh_;
  double linear_, angular_, l_scale_, a_scale_;
};
TeleopTurtle::TeleopTurtle():
  linear_(0),
  angular_(0),
  l_scale_(2.0),
  a_scale_(2.0)
{
  nh_.param("scale_angular", a_scale_, a_scale_);
  nh_.param("scale_linear", l_scale_, l_scale_);
}
void quit(int sig)
{
  (void)sig;
  input.shutdown();
  ros::shutdown();
  exit(0);
}
// store linear and angular values from ROS and send them over aqua-socket
struct aqua_message
{
  double angular_value;
  double linear_value;
};
int main(int argc, char** argv)
{
  ros::init(argc, argv, "teleop_turtle_client");
  TeleopTurtle teleop_turtle;
  signal(SIGINT,quit);
  teleop_turtle.keyLoop();
  quit(0);
  return(0);
}
// Helper-functions for UDP socket
void TeleopTurtle::close_socket()
```

```
{
  perror("m_socket closed");
  exit(1);
}
void TeleopTurtle::keyLoop()
{
  char c;
  bool dirty=false;
  puts("Reading from keyboard");
  puts("---------------------------");
  puts("Use arrow keys to move the turtle. 'q' to quit.");
  // Init Socket //
  ///////////////
  /* Create the socket.  */
  if ((m_socket = aqua_socket(AF_AQUANET, SOCK_AQUANET, 0)) < 0) {
      printf("socket creation failed\n");
      close_socket();
  }
  /* The addresses of Aqua-Net */
  // sending from node 2 to node 1
  unsigned short int local_addr = 2;
  unsigned short int remote_addr = 1;
  struct sockaddr_aquanet to_addr;
  to_addr.sin_family = AF_AQUANET;
  to_addr.sin_addr.s_addr = local_addr;
  to_addr.sin_addr.d_addr = remote_addr;
  ///////////////
  for(;;)
  {
    // get the next event from the keyboard
    try
    {
      input.readOne(&c);
    }
    catch (const std::runtime_error &)
    {
      perror("read():");
      return;
    }
    linear_=angular_=0;
    ROS_DEBUG("value: 0x%02X\n", c);
      switch(c)
    {
      case KEYCODE_LEFT:
        ROS_DEBUG("LEFT");
        angular_ = 1.0;
        dirty = true;
        break;
      case KEYCODE_RIGHT:
        ROS_DEBUG("RIGHT");
        angular_ = -1.0;
        dirty = true;
        break;
      case KEYCODE_UP:
        ROS_DEBUG("UP");
        linear_ = 1.0;
        dirty = true;
        break;
      case KEYCODE_DOWN:
        ROS_DEBUG("DOWN");
        linear_ = -1.0;
        dirty = true;
        break;
      case KEYCODE_Q:
        ROS_DEBUG("quit");
        return;
    }
    // Changes for AquaNet:
    // Send the captured data from keyboard and send it to UDP socket
    if(dirty ==true)
    {
```

```
      dirty=false;
      aqua_message values;
      values.angular_value = angular_;
      values.linear_value = linear_;
      // Send message via socket
      if (aqua_sendto(m_socket, &values, sizeof(values), 0, (struct sockaddr *) & to_addr, sizeof
(to_addr)) < 0) {
          printf("failed to send to the socket");
          close_socket();
      }
    }
  }
  return;
}
```

Appendix B. Code example for turtle_teleop_key_server

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <signal.h>
#include <stdio.h>
#ifndef _WIN32
# include <termios.h>
# include <unistd.h>
#else
# include <windows.h>
#endif
// Socket communication
#include<stdlib.h> //exit(0);
#include<arpa/inet.h>
#include<sys/socket.h>
#include <iostream>
#include <sstream>
#include <limits>
#define BUFLEN 512 //Max length of buffer
#define PORT 8888  //The port on which to send data
// Aqua-sockets
#include <sys/un.h>
#include "aquanet_log.h"
#include "aquanet_socket.h"
char log_file[BUFSIZE];
char* log_file_name = log_file;
#define KEYCODE_RIGHT 0x43
#define KEYCODE_LEFT 0x44
#define KEYCODE_UP 0x41
#define KEYCODE_DOWN 0x42
#define KEYCODE_B 0x62
#define KEYCODE_C 0x63
#define KEYCODE_D 0x64
#define KEYCODE_E 0x65
#define KEYCODE_F 0x66
#define KEYCODE_G 0x67
#define KEYCODE_Q 0x71
#define KEYCODE_R 0x72
#define KEYCODE_T 0x74
#define KEYCODE_V 0x76
class TeleopTurtle
{
public:
  TeleopTurtle();
  void keyLoop();
  void close_socket();
private:
  int m_socket = -1;
  ros::NodeHandle nh_;
  double linear_, angular_, l_scale_, a_scale_;
  ros::Publisher twist_pub_;
};
```

```
TeleopTurtle::TeleopTurtle():
  linear_(0),
  angular_(0),
  l_scale_(2.0),
  a_scale_(2.0)
{
  nh_.param("scale_angular", a_scale_, a_scale_);
  nh_.param("scale_linear", l_scale_, l_scale_);

  twist_pub_ = nh_.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 1);
}
void quit(int sig)
{
  (void)sig;
  ros::shutdown();
  exit(0);
}
// store linear and angular values from ROS and send them over aqua-socket
struct aqua_message
{
  double angular_value;
  double linear_value;
};
int main(int argc, char** argv)
{
  ros::init(argc, argv, "teleop_turtle_server");
  TeleopTurtle teleop_turtle;
  signal(SIGINT,quit);
  teleop_turtle.keyLoop();
  quit(0);
  return(0);
}
// Helper-functions for UDP socket
void TeleopTurtle::close_socket()
{
  perror("m_socket closed");
  exit(1);
}
void TeleopTurtle::keyLoop()
{
  char c;
  puts("Reading from keyboard");
  puts("---------------------------");
  puts("Use arrow keys to move the turtle. 'q' to quit.");
  // Init Socket //
  ///////////////
  struct sockaddr_in si_me, si_other;
  int i;
  socklen_t slen = sizeof(si_other) , recv_len;
  // create AQUA socket
  if ((m_socket = aqua_socket(AF_AQUANET, SOCK_AQUANET, 0)) < 0) {
      printf("failed to create a socket");
      exit(-1);
  }
  for(;;)
  {
    // printf("Waiting for data...");
    fflush(stdout);
    linear_=angular_=0;
    ROS_DEBUG("value: 0x%02X\n", c);
    struct sockaddr_aquanet remote_addr;
    int addr_size = sizeof (remote_addr);
    aqua_message received_values;
    if (aqua_recvfrom(m_socket, &received_values, sizeof(received_values), 0, (struct sockaddr *)
& remote_addr, &addr_size) < 0) {
        printf(log_file_name, "failed to read from aqua_socket");
        break;
    }
         printf("Angular: %f\n" , received_values.angular_value);
         printf("Linear: %f\n" , received_values.linear_value);
         printf("--------------------\n");
```

```
    // Place the received values into the ROS datastructure
    geometry_msgs::Twist twist;
    twist.angular.z = a_scale_*received_values.angular_value;
    twist.linear.x = l_scale_*received_values.linear_value;
    twist_pub_.publish(twist);
  }
  return;
}
```