Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

Дисциплина: “Объектно-ориентированное программирование”
Тема: “Разработка узла ROS для управления роботом KUKA youBot”

Студент гр. 3331506/10401 Пельменев Д.К., Барсуков И.А.
Преподаватель Ананьевский М.С.

Санкт-Петербург
2024

Содержание

Введение

KUKA youBot – это мобильный манипулятор, состоящий из всенаправленной мобильной платформы (омни-платформы) и руки манипулятора (с пятью степенями свободы и двухпальцевым захватом) [1]. Сам *KUKA youBot* представлен на рисунке 1.

Рисунок 1 – KUKA youBot

На робот установлен дистрибутив *Linux – Ubuntu* 16.04. В качестве управляющей системы используется робототехническая операционная система *ROS Kinetic*.

Цель данной работы заключается в разработке узла *ROS* для управления роботом и установке необходимого для этого ПО.

1. Необходимое ПО

Для написания управляющей программы необходимо на персональный компьютер установить ОС *Ubuntu* 16.04, а на неё поставить *ROS Kinetic*. Это последние версии данных ОС, поддерживающие драйвера для *youBot*.

1.1. Установка *Ubuntu*

Для пользователей *Windows* данный дистрибутив *Linux* можно установить несколькими способами:

- С помощью утилиты *rufus* или любой другой утилиты создания загрузочных дисков записать на *usb*-диск *ISO*-образ *Ubuntu* 16.04 *AMD64*, которую можно скачать с официального сайта [2], и установить с него новую ОС параллельно старой (на тот же *ssd* или на дополнительный) или установить новую ОС вместо старой;
- Использовать *Windows Subsystem for Linux* (*WSL*);
- Использовать виртуальную машину (например, *Virtual Box*).

Второй способ подходит только для установки последних версий Ubuntu, поэтому его стоит заведомо исключить.

1.2. Установка *ROS Kinetic*

Установка *ROS Kinetic* проводится через терминал *Ubuntu* путём последовательного выполнения необходимых команд, указанных на официальном сайте [3].

Для работы с *youBot* также необходимо загрузить драйвера и дополнительные пакеты. Сделать это можно с помощью следующей команды:

```
$ sudo apt install ros-kinetic-youbot-driver ros-kinetic-pr2-msgs ros-kinetic-brics-actuator ros-kinetic-moveit git ros-kinetic-ros-control ros-kinetic-ros-controllers ros-kinetic-gazebo-ros-control ros-kinetic-joy
```

2. Кратко о *ROS*

Robot Operating System (*ROS*) — это набор программных библиотек и инструментов с открытым исходным кодом, которые помогают создавать приложения для роботов [4]. Платформа *ROS* предоставляет следующие возможности для программирования роботов:

- Интерфейс передачи сообщений между процессами;
- Функции, аналогичные операционной системе;
- Поддержка и инструменты языков программирования высокого уровня;
- Наличие сторонних библиотек;
- Готовые алгоритмы;
- Удобство прототипирования;
- Поддержка экосистемы и сообщества;
- Обширные инструменты и симуляторы [5].

ROS основан на архитектуре графов, где обработка данных происходит в узлах (nodes или нодах), которые могут получать и передавать сообщения между собой. Существует три основных способа коммуникации узлов: топики (*Topic*), сервис (*Service*) и действия (*Action*). Все способы представлены на рисунке 2.

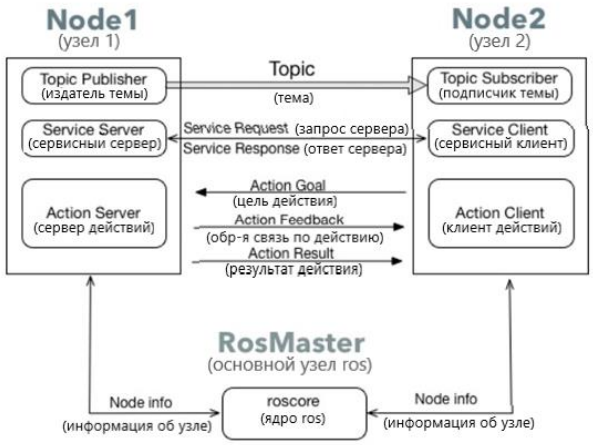

Рисунок 2 – Структура обмена сообщениями ROS

2.1. *Nodes*

"*Node*" (узел) представляет собой исполняемую программу, которая выполняет определенную функциональность в системе. По сути, ноды являются основными строительными блоками, из которых состоит система *ROS*.

Основные характеристики нод:

1. Автономность. Каждый нод работает независимо от других, выполняя свою специфическую задачу.

2. Взаимодействие. Ноды взаимодействуют друг с другом через *ROS*-коммуникационные механизмы, такие как топики, службы и параметры.

3. Гибкость. Ноды могут быть легко добавлены, заменены или удалены из системы без нарушения работы других компонентов.

Каждый нод выполняет определенную задачу и взаимодействует с другими нодами через стандартизованные механизмы *ROS*, такие как публикация/подписка на топики, вызов сервисов и доступ к общим параметрам.

2.2. *Topics*

Тема (*topic*) — это канал, через который узлы (ноды) обмениваются сообщениями (*messages*). Другими словами, топик представляет собой именованный канал связи, по которому данные передаются от одного узла к другому. Топики играют ключевую роль в организации децентрализованного, асинхронного и масштабируемого обмена данными между различными компонентами робототехнической системы. Вот основное назначение топиков в *ROS*:

1. Асинхронный обмен данными:

- Узлы могут публиковать сообщения в тему, не зная, какие другие узлы подписаны на эту тему.

- Узлы могут подписываться на темы, получая сообщения, опубликованные другими узлами.

Это позволяет организовать асинхронный обмен данными между узлами системы.

2. Слабая связанность:

Использование тем помогает достичь слабой связанности между узлами, так как узлы не зависят друг от друга напрямую. Узлы могут быть добавлены, удалены или заменены без необходимости изменять другие узлы, если они подписываются/публикуют в те же темы.

3. Масштабируемость:

Темы позволяют легко масштабировать систему, добавляя новые узлы, публикующие или подписывающиеся на темы.

2.3. *Service/action server* и *service/action client*

1. *Service client* и *Service server*:

- *Service server* - узел, который предоставляет определенную услугу (*service*) другим узлам.
- *Service client* - узел, который запрашивает эту услугу у *service server*.

Взаимодействие происходит по схеме "запрос-ответ": *client* отправляет запрос, *server* обрабатывает его и возвращает ответ.

2. *Action client* и *Action server*:

- *Action server* - узел, который предоставляет возможность выполнения длительной, возможно асинхронной, операции.
- *Action client* - узел, который инициирует выполнение этой операции и получает обратную связь о ходе ее выполнения.

Взаимодействие происходит через передачу сообщений "*goal*" (цель), "*feedback*" (обратная связь) и "*result*" (результат).

Основные различия между *service* и *action*:

- *Service* - синхронный запрос-ответ, *action* - асинхронная операция с обратной связью.

- *Service* - один запрос - один ответ, *action* - один запрос - множество сообщений обратной связи.
- *Service* обычно используется для быстрых, атомарных операций, *action* - для длительных, возможно распределенных во времени операций.

2.4. *ROS Master*

ROS Master – это основной узел (*node*), который играет роль центрального координатора в *ROS*-системе. Он выполняет следующие ключевые функции:

1. Регистрация узлов. *ROS Master* отвечает за регистрацию всех узлов, которые запускаются в *ROS*-системе. Когда узел запускается, он регистрируется в *ROS Master*, чтобы быть доступным для других узлов.

2. Управление темами и сервисами. *ROS Master* осуществляет регистрацию и отслеживание всех тем (*topics*) и сервисов (*services*), используемых в системе. Он предоставляет информацию об активных темах и сервисах другим узлам.

3. Обеспечение связи. *ROS Master* отвечает за установление и поддержание соединений между различными узлами в *ROS*-системе.

3. Разработка узла

Узел разрабатывался с использованием готовых драйверов для *KUKA youBot* [6].

Сначала требуется подготовить директорию для работы с пакетом.

```
$ mkdir -p ~/catkin_ws/src
```

Затем клонировать в неё git репозиторий и собрать проект.

```
$ cd ~/catkin_ws/src
$ git clone -b kinetic --recursive https://github.com/ut-ims-robotics/youbot
$ cd ~/catkin_ws
$ catkin_make
$ sourse ~/catkin_ws/devel/setup.bash
```

Передача сообщений осуществляется через топик для отправки управляющих команд, поскольку необходима двунаправленная синхронная связь.

Необходимо создать и собрать новый пакет.

```
$ cd ~/catkin_ws/src
$ roscreate-pkg youbot_rc geometry_msgs roscpp
$ rosmake youbot_rc
```

Далее создаются папки для нод и заголовочных файлов, а также сами файлы.

```
$ cd ~/youbot_rc
$ mkdir src
$ touch src/youbot_rc_node.cpp
$ mkdir include
$ mkdir include/youbot_rc
$ touch include/youbot_rc/youbot_rc_node.hpp
$ touch include/youbot_rc/net_protocol.hpp
```

В файлах прописываются следующие коды.

`youbot_rc_node.cpp`

Код представляет собой узел *ROS* для управления мобильным роботом и его манипулятором *Youbot*. Примерная структура этого узла:

1. В конструкторе класса *YoubotServer* происходит инициализация узлов *ROS*, создание паблишеров для управления движением робота и позицией манипулятора, а также инициализация *TCP*-сервера для взаимодействия с внешним управляющим устройством.

2. Метод *pub_cmdvel* используется для публикации команды движения робота в виде сообщения типа *geometry_msgs::Twist*.

3. Метод *pub_arm_joint* используется для публикации команды управления позицией манипулятора робота в виде сообщения с указанием позиции каждого сустава.

4. Метод *pub_gripper_joint* используется для публикации команды управления захватом манипулятора для сжатия или открытия захвата.

5. Метод *connect* используется для установления соединения с внешним управляющим устройством.

6. Метод *receive_command* используется для приема команд от внешнего управляющего устройства, обновления состояния робота и отправки обратного сообщения о состоянии.

7. Метод *spin* содержит основной цикл работы ноды, в котором осуществляется обработка входящих команд и отправка управляющих команд роботу.

8. Функция *main* инициализирует узел *ROS* и запускает цикл работы класса *YoubotServer*.

Общая структура кода является типичной для узла управления роботом в *ROS*, включая инициализацию узла, создание паблишеров и принятие команд от внешнего управляющего устройства через *TCP*-соединение.

```
#include "youbot_rc/youbot_rc_node.hpp"
```

```
        #include <boost/units/systems/si/length.hpp>
        #include <boost/units/systems/si/plane_angle.hpp>
        #include <boost/units/io.hpp>
        #include <algorithm>

        YoubotServer::YoubotServer()
        {
            cmdVelPub = nh.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
            armPosPub           =           nh.advertise<brics_actuator::JointPositions           >
("arm_1/arm_controller/position_command", 1);

gripperPosPub              =              nh.advertise<brics_actuator::JointPositions              >
("arm_1/gripper_controller/position_command", 1);

            int port;
            nh.getParam("youbot_rc_node/port", port);
            tcpServer.set_socket("", port);
            tcpServer.set_keepalive(1, 1, 1);
            if(tcpServer.socket_bind() < 0) {
                ROS_ERROR("Bind error. Try changing the port, wait a few minutes or reboot.");
                ros::shutdown();
            }

            ROS_INFO("Init success!");
        }

        YoubotServer::~YoubotServer()
        {
            pub_cmdvel(0, 0, 0);
        }

        void YoubotServer::pub_cmdvel(double x, double y, double angle)
        {
            geometry_msgs::Twist twist_msg;

            twist_msg.linear.x = x;
            twist_msg.linear.y = y;
            twist_msg.linear.z = 0;
            twist_msg.angular.x = 0;
            twist_msg.angular.y = 0;
            twist_msg.angular.z = angle;

            cmdVelPub.publish(twist_msg);
        }

        void YoubotServer::pub_arm_joint()
        {
            brics_actuator::JointPositions command;

            std::vector <brics_actuator::JointValue> arm_joint_positions;
            arm_joint_positions.resize(5);
```

```
            for (int idx = 0; idx < 5; ++idx)
            {
                arm_joint_positions[idx].joint_uri = std::string("arm_joint_")+ std::to_string(idx +
1);
                arm_joint_positions[idx].value = rxMsg.axis[idx] / 100.0;
                arm_joint_positions[idx].unit = boost::units::to_string(boost::units::si::radians);
            };

            command.positions = arm_joint_positions;
            armPosPub.publish(command);
        }

        void YoubotServer::pub_gripper_joint()
        {
            switch (rxMsg.grip_cmd)
            {
            case GripControl::WAIT:
                return;
                break;
            case GripControl::COMPRESS:
                gripperPosition -= gripperSpeed;
                break;
            case GripControl::OPEN:
                gripperPosition += gripperSpeed;
                break;
            default:
                break;
            }

            gripperPosition = std::max(gripperPosition, 0.0);
            gripperPosition = std::min(gripperPosition, 0.0115);

            brics_actuator::JointPositions command;
            std::vector<brics_actuator::JointValue> gripper_joint_positions;
            gripper_joint_positions.resize(2);

            gripper_joint_positions[0].joint_uri = "gripper_finger_joint_l";
            gripper_joint_positions[0].value = gripperPosition;
            gripper_joint_positions[0].unit = boost::units::to_string(boost::units::si::meter);
            gripper_joint_positions[1].joint_uri = "gripper_finger_joint_r";
            gripper_joint_positions[1].value = gripperPosition;
            gripper_joint_positions[1].unit = boost::units::to_string(boost::units::si::meter);

            command.positions = gripper_joint_positions;
            gripperPosPub.publish(command);
        }

        void YoubotServer::connect()
        {
            ROS_INFO("Waint connection");
```

```
            int res = tcpServer.socket_listen();

            if(res < 0) {
                ROS_WARN("Client connection error! Code: %d", res);
                return;
            }

            isConnected = true;

            ROS_INFO("Connected");
        }

        int YoubotServer::receive_command()
        {
            if (tcpServer.receive(reinterpret_cast<char*>(&rxMsg), YOUBOT_MSG_SIZE) < 0) {
                ROS_WARN("Receive error!");
                return -1;
            }

            update_state();

            if (tcpServer.send_mes(reinterpret_cast<char*>(&youbotState), YOUBOT_MSG_SIZE) < 0) {
                ROS_WARN("Send error!");
                return -1;
            }

            return 0;
        }

        void YoubotServer::update_state()
        {
            bool state_changed = false;
            if (rxMsg.ang_speed != youbotState.ang_speed || rxMsg.x_vel != youbotState.x_vel ||
rxMsg.y_vel != youbotState.y_vel)
            {
                pub_cmdvel(rxMsg.x_vel/100.0, rxMsg.y_vel/100.0, rxMsg.ang_speed/100.0);
                state_changed = true;
            }

            if ((rxMsg.axis[0] != youbotState.axis[0]) || (rxMsg.axis[1] != youbotState.axis[1])
                || (rxMsg.axis[2] != youbotState.axis[2]) || (rxMsg.axis[3] != youbotState.axis[3])
                || (rxMsg.axis[4] != youbotState.axis[4]))
            {
                pub_arm_joint();
                state_changed = true;
            }

            pub_gripper_joint();
            if (state_changed) youbotState = rxMsg;
        }
```

```
void YoubotServer::spin()
{
    while (nh.ok())
    {
        if (!isConnected) {
            connect();
            continue;
        }

        if (receive_command() < 0) {
            isConnected = false;
            pub_cmdvel(0, 0, 0);
        }
        ros::spinOnce();
        rate.sleep();
    }
}

int main(int argc, char * argv[])
{
    ros::init(argc, argv, "youbot_rc_node");
    YoubotServer youbot_server;
    youbot_server.spin();
    return 0;
}
```

net_protocol.hpp

В данном коде определена структура *youBotMsg*, представляющая информацию о сообщении для управления роботом. Структура содержит четыре поля типа *int16_t* для скорости по осям *x* и *y*, угловой скорости и массив *axis* размером 5 элементов. Также присутствует перечисление *GripControl* для команд управления захватом.

Директива *#pragma pack*(*push*, 1) используется для указания компилятору упаковать поля структуры в 1-байтовые блоки, что может быть полезно при работе с сетевым протоколом.

Код защищен директивами *#ifndef* и *#define*, что предотвращает повторное включение файла net_protocol.hpp в программу.

Константа *YOUBOT_MSG_SIZE* определяет размер структуры *YoubotMsg* в байтах.

В целом, структура кода указывает на наличие некоторого сетевого протокола для взаимодействия с роботом и управлением захватом.

```
#ifndef NET_PROTOCOL_H
#define NET_PROTOCOL_H

#include <cstdint>
#include <cstddef>

enum class GripControl : uint8_t {WAIT, COMPRESS, OPEN};

#pragma pack(push, 1)
struct YoubotMsg
{
    int16_t x_vel = 0;
    int16_t y_vel = 0;
    int16_t ang_speed = 0;
    int16_t axis[5];
    GripControl grip_cmd = GripControl::WAIT;
};
#pragma pack(pop)
const size_t YOUBOT_MSG_SIZE = sizeof(YoubotMsg);

#endif
```

youbot_rc_node.hpp

Этот код представляет класс *YoubotServer*, который содержит методы для управления роботом *youBot* через сетевое соединение. Он включает заголовочные файлы и библиотеки для работы с *ROS*, *TCP*-сервером, геометрическими сообщениями и сообщениями для управления суставами *Youbot*.

В классе определены переменные для управления публикацией сообщений движения, положения руки и захвата *youBot*. Также есть методы для установки соединения с роботом, получения команд от клиента и обновления состояния робота.

Этот код является основным управляющим узлом для управления роботом *Youbot* через сеть, публикуя команды движения и положения суставов.

```
#ifndef YOUBOT_RC_NODE_HPP
#define YOUBOT_RC_NODE_HPP

#include <ros/ros.h>
#include <ros/console.h>
#include <cstddef>
#include <vector>
#include <iostream>

#include <geometry_msgs/Twist.h>
#include <brics_actuator/JointPositions.h>

#include "simple_socket/simple_socket.hpp"
#include "net_protocol.hpp"

class YoubotServer
{
public:
    YoubotServer();
    ~YoubotServer();

    void spin();
    void pub_cmdvel(double x, double y, double angle);
    void pub_arm_joint();
    void pub_gripper_joint();

private:
    void connect();
    int receive_command();
    void update_state();

private:
    ros::NodeHandle nh;
    ros::Publisher cmdVelPub;
    ros::Publisher armPosPub;
    ros::Publisher gripperPosPub;
    ros::Rate rate = ros::Rate(30);

    sockets::TCPServer tcpServer;
    bool isConnected = false;
    YoubotMsg rxMsg;
    YoubotMsg youbotState;

    double gripperPosition = 0;
    double gripperSpeed = 0.002;
};

#endif
```

Далее необходимо сконфигурировать файл формата *.launch.*

ROS Launch — это инструмент для запуска и управления набором узлов (*nodes*) в *ROS*-системе. Файлы запуска (*.launch*) позволяют описывать, какие узлы должны быть запущены, с какими параметрами, а также настраивать сетевые соединения между ними.

Ниже представлен программный код файла */catkin_ws/src/youBot_rc/start_server.launch.*

```
<launch>
    <arg name="port" default="10001" />
    <node name="youbot_rc_node" pkg="youbot_rc" type="youbot_rc_node" output="screen">
        <param name="port" type="int" value="$(arg port)" />
    </node>

</launch>
```

Обмен сообщениями между сервером и клиентом осуществляется через Интернет с помощью технологии сокетов.

Сокеты (*sockets*) используются для организации сетевых соединений между различными компонентами системы. Сокеты позволяют узлам (*nodes*) обмениваться данными по сети, независимо от их физического местоположения. Основные типы сокетов:

- *TCP*-сокеты для точечного обмена данными между узлами (используется в нашем случае);

- *UDP*-сокеты для рассылки данных по сети.

ROS использует сокеты для реализации механизмов публикации/подписки на темы (*topics*) и вызова удаленных сервисов (*services*).

Сокет находится в папке */catkin_ws/src/lib/simple_socket/*. Код конфигурации сокета находится в папке */src* и представлен ниже.

```
#include "simple_socket/simple_socket.hpp"
```

```
namespace sockets
{
Socket::Socket(const SocketType socket_type) : socket_type_{socket_type}
{
#ifdef _WIN32
    if (!socket_count) {
        WSADATA wsa;
        if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) {
            throw  SimpleSocketException(__FILE__,  __LINE__,  "Error  initializing  Winsock  "  +
WSAGetLastError());
        }
        ++socket_count;
    }
#endif

    sockfd_ = socket(AF_INET, static_cast<int>(socket_type_), 0);
    if (sockfd_ < 0) {
        throw SimpleSocketException(__FILE__, __LINE__, "Could not create socket");
    }
    address_.sin_family = AF_INET;
}

int Socket::set_socket(const std::string& ip_address, uint16_t port)
{
    set_port(port);
    return set_address(ip_address);
}

void Socket::set_port(uint16_t port)
{
    address_.sin_port = htons(port);
}

int Socket::set_address(const std::string& ip_address)
{
    if (ip_address.empty()) {
        address_.sin_addr.s_addr = htonl(INADDR_ANY);
        return 0;
    }
    if(!inet_pton(AF_INET, ip_address.c_str(), &address_.sin_addr)) {
        return static_cast<int>(SocketErrors::INCORRECT_ADDRESS);
    }
    return 0;
}

void Socket::socket_update()
{
    socket_close();
    sockfd_ = socket(AF_INET, static_cast<int>(socket_type_), 0);
    if (sockfd_ < 0) {
        throw SimpleSocketException(__FILE__, __LINE__, "Could not create socket");
```

```
    }
}
void Socket::socket_close()
{
#ifdef _WIN32
    ::closesocket(sockfd_);
    --socket_count;
#else
    ::close(sockfd_);
#endif
}

Socket::~Socket()
{
    socket_close();

#ifdef _WIN32
    if (!socket_count){
        WSACleanup();
    }
#endif
}

UDPClient::UDPClient(const std::string& ip_address, uint16_t port) : Socket(SocketType::TYPE_DGRAM)
{
    set_address(ip_address);
    set_port(port);
}

int UDPClient::send_mes(const char* mes, const int mes_size)
{
    if(sendto(sockfd_, mes, mes_size, 0, reinterpret_cast<sockaddr*>(&address_), sizeof(address_))
< 0)
        return static_cast<int>(SocketErrors::SEND_ERROR);
    return 0;
}

UDPServer::UDPServer(const std::string& ip_address, uint16_t port)
: Socket(SocketType::TYPE_DGRAM)
{
    set_port(port);
    set_address(ip_address);
}

int UDPServer::socket_bind()
{
    if (bind(sockfd_, reinterpret_cast<sockaddr*>(&address_), sizeof(address_)) < 0)
        return static_cast<int>(SocketErrors::BIND_ERROR);
    return 0;
}
```

```
int UDPServer::receive(char* recv_buf, const int recv_buf_size)
{
    return  recvfrom(sockfd_,  recv_buf,  recv_buf_size,  0,  reinterpret_cast<sockaddr*>(&client_),
&client_size_);
}

TCPSocket::TCPSocket()
: Socket(SocketType::TYPE_STREAM)
{}

int TCPSocket::set_keepalive(const int& keepidle, const int& keepcnt, const int& keepintvl)
{
    int optval = 1;
    if (setsockopt(sockfd_, SOL_SOCKET, SO_KEEPALIVE, (char *) &optval, sizeof(optval)) < 0) {
        return -1;
    }

    //set the keepalive options
    if (setsockopt(sockfd_, IPPROTO_TCP, TCP_KEEPCNT, (char *) &keepcnt, sizeof(keepcnt)) < 0) {
        return -1;
    }

    if (setsockopt(sockfd_, IPPROTO_TCP, TCP_KEEPIDLE, (char *) &keepidle, sizeof(keepidle)) < 0) {
        return -1;
    }

    if (setsockopt(sockfd_, IPPROTO_TCP, TCP_KEEPINTVL, (char *) &keepintvl, sizeof(keepintvl)) < 0)
{
        return -1;
    }

    return 0;
}

TCPClient::TCPClient(const std::string& ip_address, uint16_t port)
{
    set_address(ip_address);
    set_port(port);
}

int TCPClient::make_connection()
{
    if (is_connected) {
        socket_update();
        is_connected = false;
    }

    if (connect(sockfd_, reinterpret_cast<sockaddr*>(&address_), sizeof(address_)) < 0)
        return static_cast<int>(SocketErrors::CONNECT_ERROR);
    is_connected = true;
```

```
    return 0;
}
int TCPClient::receive(char* recv_buf, const int recv_buf_size)
{
    return recv(sockfd_, recv_buf, recv_buf_size, 0);
}

int TCPClient::send_mes(const char* mes, const int mes_size)
{
#ifdef _WIN32
    int flags = 0;
#else
    int flags = MSG_NOSIGNAL;
#endif

    if(send(sockfd_, mes, mes_size, flags) < 0)
        return static_cast<int>(SocketErrors::SEND_ERROR);
    return 0;
}

TCPServer::TCPServer(const std::string& ip_address, uint16_t port)
{
    set_port(port);
    set_address(ip_address);
};

TCPServer::~TCPServer()
{
    close_connection();
}

int TCPServer::socket_bind()
{
    if (bind(sockfd_, reinterpret_cast<sockaddr*>(&address_), sizeof(address_)) < 0)
        return static_cast<int>(SocketErrors::BIND_ERROR);
    return 0;
}

int TCPServer::socket_listen()
{
    if (listen(sockfd_, 5) < 0)
        return static_cast<int>(SocketErrors::LISTEN_ERROR);

    close_connection();

    client_sock_ = accept(sockfd_, reinterpret_cast<sockaddr*>(&client_), &client_size_);

    if (client_sock_ < 0)
        return static_cast<int>(SocketErrors::ACCEPT_ERROR);

    is_open = true;
```

```
    return 0;
}
void TCPServer::close_connection()
{
    if(is_open) {
#ifdef _WIN32
        ::closesocket(client_sock_);
#else
        ::close(client_sock_);
#endif
        is_open = false;
    }
}

int TCPServer::receive(char* recv_buf, const int recv_buf_size)
{
    return recv(client_sock_, recv_buf, recv_buf_size, 0);
}

int TCPServer::send_mes(const char* mes, const int mes_size)
{
#ifdef _WIN32
    int flags = 0;
#else
    int flags = MSG_NOSIGNAL;
#endif

    if(send(client_sock_, mes, mes_size, flags) < 0)
        return static_cast<int>(SocketErrors::SEND_ERROR);

    return 0;
}

} // namespace sockets
```

В папке */include/simple_socket/* расположены заголовочные файлы классов и исключений сокета. Они представлены ниже.

simple_socket.hpp

```
#ifndef SIMPLE_SOCKET_HPP_
#define SIMPLE_SOCKET_HPP_
#include "simple_socket_exception.hpp"
#include <cstdint>
#include <string>
#include <memory>

#ifdef _WIN32
    #include <winsock2.h>
    #include <Ws2tcpip.h>
```

```
#else
    #include <sys/socket.h>
    #include <arpa/inet.h>
    #include <netinet/tcp.h>
    #include <unistd.h>
#endif

namespace sockets
{

enum class SocketType {
    TYPE_STREAM = SOCK_STREAM,
    TYPE_DGRAM = SOCK_DGRAM
};

enum class SocketErrors {
    RECEIVE_ERROR = -1,
    SEND_ERROR = -2,
    BIND_ERROR = -3,
    ACCEPT_ERROR = -4,
    CONNECT_ERROR = -5,
    LISTEN_ERROR = -6,
    INCORRECT_ADDRESS = -7
};

#ifdef _WIN32
static int socket_count = 0;
typedef int SockaddrSize;
#else
typedef unsigned int SockaddrSize;
#endif

class Socket
{
public:
    explicit Socket(SocketType socket_type);
    ~Socket();
    int set_socket(const std::string& ip_address, uint16_t port);
protected:
    void set_port(uint16_t port);
    int set_address(const std::string& ip_address);
    void socket_update();
    void socket_close();
protected:
    int sockfd_;
    sockaddr_in address_;
    SocketType socket_type_;
};

class UDPClient : public Socket
{
```

```
public:
    UDPClient(const std::string& ip_address = "127.0.0.1", uint16_t port = 8000);
    int send_mes(const char* mes, const int mes_size);
};

class UDPServer : public Socket
{
public:
    UDPServer(const std::string& ip_address = "127.0.0.1", uint16_t port = 8000);
    int socket_bind();
    int receive(char* recv_buf, const int recv_buf_size);
private:
    sockaddr_in client_;
    SockaddrSize client_size_ = sizeof(sockaddr_in);
};

class TCPSocket : public Socket
{
public:
    TCPSocket();
    int set_keepalive(const int& keepidle, const int& keepcnt, const int& keepintvl);
};

class TCPClient : public TCPSocket
{
public:
    TCPClient(const std::string& ip_address = "127.0.0.1", uint16_t port = 8000);
    int make_connection();
    int receive(char* recv_buf, const int recv_buf_size);
    int send_mes(const char* mes, const int mes_size);
private:
    bool is_connected = false;
};

class TCPServer : public TCPSocket
{
public:
    TCPServer(const std::string& ip_address = "127.0.0.1", uint16_t port = 8000);
    ~TCPServer();
    int socket_bind();
    int socket_listen();
    void close_connection();
    int receive(char* recv_buf, const int recv_buf_size);
    int send_mes(const char* mes, const int mes_size);
private:
    bool is_open = false;
    int client_sock_;
    sockaddr_in client_;
    SockaddrSize client_size_ = sizeof(sockaddr_in);
};
```

```
}  // namespace sockets
#endif  // SIMPLE_SOCKET_HPP_
```

simple_socket_exception.hpp

```
#ifndef SIMPLE_SOCKET_EXCEPTION_HPP_
#define SIMPLE_SOCKET_EXCEPTION_HPP_
#include <stdexcept>
#include <string>

namespace sockets
{

class SimpleSocketException : public std::runtime_error
{
public:
    SimpleSocketException(const char *file, int line, const std::string &arg)
    : std::runtime_error(arg)
    {
    msg_ = std::string(file) + ":" + std::to_string(line) + ": " + arg;
    }
    const char *what() const noexcept override
    {
    return msg_.c_str();
    }

private:
    std::string msg_;
};

}  // namespace sockets

#endif  // SIMPLE_SOCKET_EXCEPTION_HPP_
```

Заключение

В ходе выполнения работы были произведены установка и настройка ПО, необходимого для написания управляющей программы для робота *KUKA youBot*, и написаны узел *ROS* и сокет для обмена сообщениями между роботом (сервером) и ПК (клиентом).

Список использованной литературы

1. KUKA youBot User Manual. — 2012. — URL: https://usermanual.wiki/Document/KUKAyouBotUserManual.1109421140 (Дата обр. 27.05.2024).
2. Ubuntu 16.04.7 LTS (Xenial Xerus). — URL: http://releases.ubuntu.com/16.04/ (Дата обр. 27.05.2024).
3. Ubuntu install of ROS Kinetic. — URL: https://wiki.ros.org/kinetic/Installation/Ubuntu (Дата обр. 27.05.2024).
4. ROS – Robot Operating System. — URL: https://www.ros.org/ (Дата обр. 27.05.2024).
5. Lentin Joseph. Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy. — Apress, 2018. — ISBN 9781484234051.
6. YouBot. — URL: https://github.com/ut-ims-robotics/youbot (Дата обр. 27.05.2024).