GeekBrains

C++: разработка программ с графическим интерфейсом на Qt

Работа с сетью в Qt

Сетевые протоколы. UDP- и TCP-соединения

Высокоуровневые и низкоуровневые сетевые протоколы

Краткий перечень уровней абстракции работы с сетью от низкого к высокому:

- Физический уровень. Уровень драйвера устройства.
- Канальный уровень. Предназначен для передачи данных узлам, находящимся в том же сегменте локальной сети. Также может использоваться для обнаружения и, возможно, исправления ошибок, возникших на физическом уровне.
- Сетевой уровень. Предназначается для определения пути передачи данных. Наиболее часто используемые в повседневной жизни протоколы — IPv4, IPv6.
- Транспортный уровень. На транспортном уровне работает большинство сетевых приложений и протоколов. Например, UDP (DNS-серверы), TCP (веб-сервисы), RTP (потоковая передача мультимедийных данных; протокол основан на UDP).

Мы в нашей работе в основном будем использовать протоколы TCP и UDP.

UDP — протокол, который обеспечивает обмен данными с DNS-сервером, работу STUF-серверов, P2P-протокола BitTorrent, взаимодействие серверов и клиентов сетевых многопользовательских игр и функционирование различных сервисов, например синхронизацию времени в ОС. UDP служит для отправки данных по сети Ethernet без проверки успешности приема данных получателем.

На протоколе TCP основана работа веб-серверов (протоколы HTTP, HTTPS) и протоколы передачи файлов (например, FTP). Его отличительная черта — подтверждение получения пакета данных от получателя. При передаче данных принимающая сторона проверяет пакет и при нахождении ошибки или потери пакета запрашивает повторную отправку

Соединение через сокеты

Сокет — идентификатор приложения, для которого пришли данные по сети. Каждый выделяемый слот для приема должен быть привязан к свободному порту (0...65535). Часть портов занята для системных нужд — обмена данных с BIOS (UEFI), с периферийными устройствами (принтеры, сканеры). Для определенных сервисов есть стандартный список используемых портов (FTP-сервер — 21, HTTP-сервер — 80, HTTPS-сервер — 443).

Стандартные классы для работы с сокетами в Qt:

- QUdpSocket — сокет для установки соединения по UDP-протоколу.
- QTcpSocket — сокет для установки соединения по TCP-протоколу. TCP работает поверх UDP. Устанавливает соединение клиент-сервер.

Транспортный пакет содержит в себе следующие основные элементы:

- флаг, который указывает, какой протокол используется (TCP/IPv4, TCP/IPv6);
- MAC-адреса приемного и передающего устройства (для TCP/IPv4);
- IP-адреса приемного и передающего устройства (в протоколе TCP/IPv6 IP-адреса достаточно для идентификации устройства в сети, поэтому MAC-адрес не используется);
- размер данных. Максимальный размер — 65535 байт для UDP-датаграммы (8 байт на заголовок и 65527 байт на данные);
- контрольная сумма для проверки на наличие ошибок в полученном пакете;
- сам блок данных.

При использовании протокола TCP можно быть уверенным, что отправляемые данные дойдут до приемной стороны, но при этом TCP медленнее, чем UDP. Для UDP проверку получения при необходимости придется реализовывать в коде самостоятельно.

Выбор протокола зависит от того, какие требования предъявляются к пакету данных, передаваемых через сеть. Если важна скорость передачи и потеря части пакетов не страшна для работы ПО, то предпочтительней выбрать UDP. Например, для сетевой игры потеря нескольких пакетов не так страшна, так как следом обязательно придёт новый пакет данных с более актуальной информацией. Для интернет-радио потеря одного пакета тоже не слишком заметна и не скажется на качестве звучания, поэтому UDP также будет более подходящим выбором, чем TCP (в настоящие время чаще используется протокол RTP).

Для примера использования UDP-соединения запросим IP-адрес у DNS-сервера. Создадим новый проект. Добавим слот, обрабатывающий принятую датаграмму:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QLineEdit>
#include <QUdpSocket>

class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow(QWidget *parent = 0);
   ~MainWindow();
private:
   QUdpSocket *dnsServer;
private slots:
   void receive(); // Сокет обработки ответа
};

#endif // MAINWINDOW_H
```

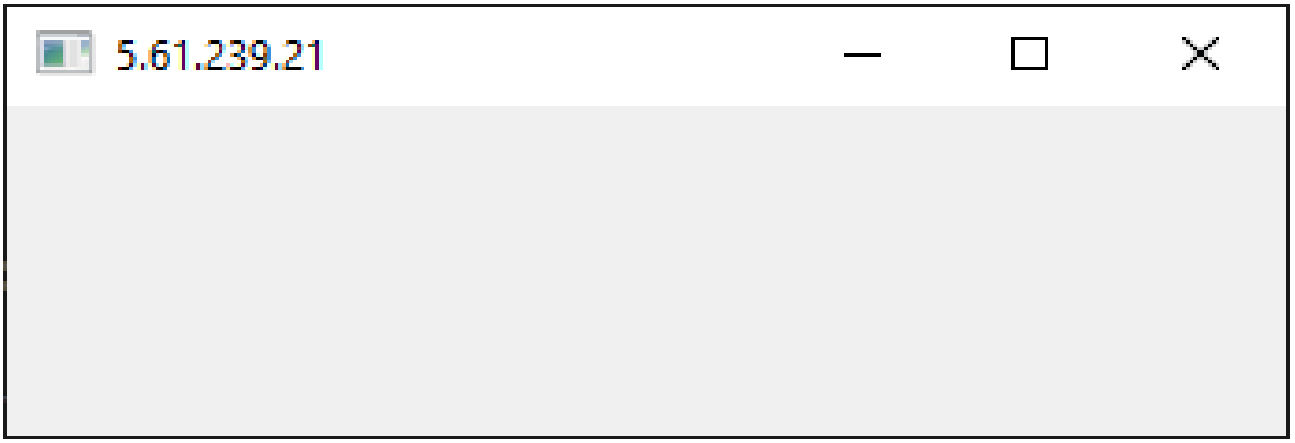
Установим IP адрес, соответствующий домену второго уровня (например, geekbrains.ru), в качестве заголовка окна:

```
#include "mainwindow.h"
#include <QNetworkDatagram>

const unsigned char aap[] = {0xAA, 0xA1, 01, 00, 00, 01, 00, 00, 00, 00, 00,
00}; // Заголовок DNS-запроса. DNS-запрос стандартный, его протокол можно
     // найти с помощью поисковых систем

MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent)
{
   dnsServer = new QUdpSocket(this);
   dnsServer->bind(53);
   // Создаем локальное соединение, указав приемный порт
   connect(dnsServer, &QUdpSocket::readyRead, this, &MainWindow::recieve,
Qt::DirectConnection);


   QByteArray barr;
   barr.append((const char*)aap, sizeof aap);
   // Заполняем структуру запроса
   QString str = "geekbrains";
   unsigned char length = str.length();
   barr.append(length);
   barr.append(str);
   str = "ru";
   length = str.length();
   barr.append(length);
   barr.append(str);
   const unsigned char ch[] = {
       00, 00, 01, 00, 01,
   };
   barr.append((const char *)ch, sizeof ch);
// Воспользуемся DNS-сервером от Google, по умолчанию порт у DNS 53
   auto bb = dnsServer->writeDatagram(barr, QHostAddress("8.8.8.8"),53);
}

MainWindow::~MainWindow()
{

}

void MainWindow::receive()
{
   while (dnsServer->hasPendingDatagrams()) // пока есть данные
   {
       QNetworkDatagram datagram = dnsServer->receiveDatagram();
       QByteArray bArr = datagram.data();
       int  length = bArr.size() - 4;
       char *data = bArr.data();
```

```
        QString answer =
                QString::number((int)data[length + 0] & 0xFF) + "." +
                QString::number((int)data[length + 1] & 0xFF) + "." +
                QString::number((int)data[length + 2] & 0xFF) + "." +
                QString::number((int)data[length + 3] & 0xFF);
        setWindowTitle(answer);
    }
}
```

В файл **.pro** прописываем модуль **QT += network**.

Запустим программу:



IP-адрес сервера — 5.61.239.21.

Модель «клиент — сервер»

Эта модель является технологией обмена информацией в сети. Вычислительная модель «клиент — сервер» изначально связана с парадигмой открытых систем, которая появилась в 90-х годах и быстро эволюционировала. Сами термины «клиент» и «сервер» изначально применялись к архитектуре программного обеспечения, в которой процесс выполнения задачи представлял собой взаимодействие двух процессов: клиентский запрашивал определенную услугу, а серверный обеспечивал ее выполнение. При этом предполагалось, что один серверный процесс может обслужить множество клиентских.

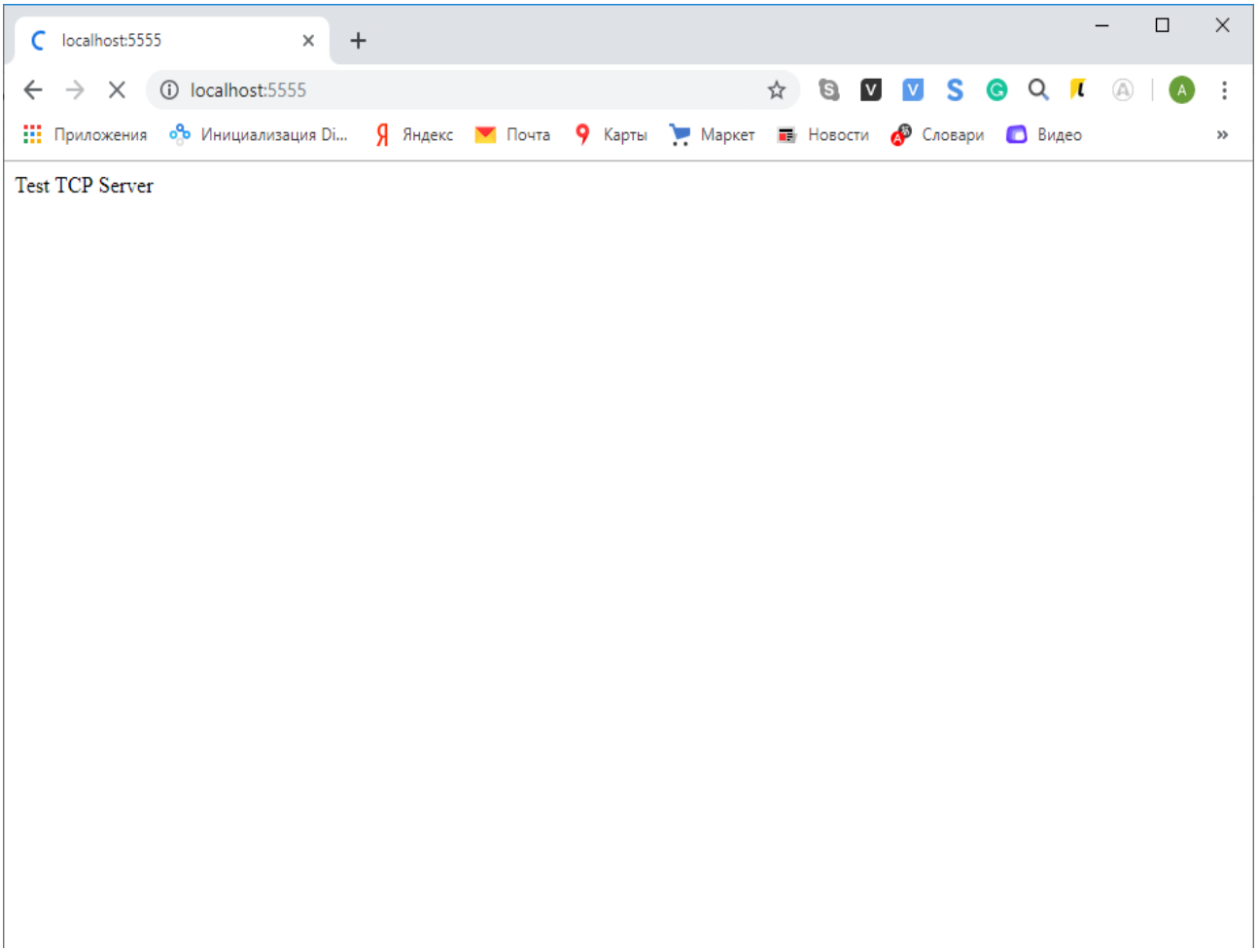
Вот пример кода простого HTTP-сервера (здесь не учитывается полный цикл запроса, в действительности протокол гораздо сложнее):

```
//mainwindow.h
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTcpServer>
```

```
class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow(QWidget *parent = 0);
   ~MainWindow();
private:
   QTcpServer *server;
   QTcpSocket *socket;
private slots:
   void newConnect();
   void readData();
   void disconnectConnect();
};


#endif // MAINWINDOW_H


// mainwindow.cpp
#include "mainwindow.h"
#include <QTcpSocket>

MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent), socket(nullptr)
{
   server = new QTcpServer(this);
   server->listen(QHostAddress::LocalHost, 5555);
   connect(server, &QTcpServer::newConnection, this, &MainWindow::newConnect);
}

MainWindow::~MainWindow()
{
}


void MainWindow::newConnect()
{
   if (socket) return;
   socket = server->nextPendingConnection();

   connect(socket, &QTcpSocket::readyRead, this, &MainWindow::readData,
Qt::QueuedConnection);
   connect(socket, &QTcpSocket::disconnected, this,
&MainWindow::disconnectConnect, Qt::QueuedConnection);
}
void MainWindow::readData()
{
   for (;socket->bytesAvailable() > 0;)
   {
```

```
        QByteArray ba = socket->readAll();
        QString f(ba);
        if (f.indexOf("GET") == 0)
        {
            char msg[] = "HTTP/1.1\n\n <html><body>Test TCP
Server</body></html>";
            socket->write(msg, sizeof msg);
        }
    }
}
void MainWindow::disconnectConnect()
{
    socket->close();
    socket = nullptr;
}
```

Запустим приложение. Затем запустим браузер и введем адрес http://localhost:5555:



Стоит особенно отметить следующий участок кода:

```
if (socket) return;
    socket = server->nextPendingConnection();
```

```
   connect(socket, &QTcpSocket::readyRead, this, &MainWindow::readData,
Qt::QueuedConnection);
   connect(socket, &QTcpSocket::disconnected, this,
&MainWindow::disconnectConnect, Qt::QueuedConnection);
}
void MainWindow::readData()
{
   for (;socket->bytesAvailable() > 0;)
   {
       QByteArray ba = socket->readAll();
       QString f(ba);
       if (f.indexOf("GET") == 0)
       {
           char msg[] = "HTTP/1.1\n\n <html><body>Test TCP
Server</body></html>";
           socket->write(msg, sizeof msg);
       }
   }
```

Объект **socket**, полученный в пакете, содержит IP-адрес и порт отправителя пакета запроса. При разработке клиентского приложения (подключающегося к серверу) должен использоваться один и тот же слот для приема и передачи, а для сервера порт нужно считывать из пришедшей датаграммы для протокола TCP/IPv4. Это связано с тем, что большинство пользователей выходят в интернет через NAT-сервер либо с динамическим IP (достаточно редким в настоящие время).

NAT-сервер — это механизм в сетях TCP/IP, позволяющий преобразовывать IP-адреса транзитных пакетов. Пользователь делает запрос к серверу, где расположен интересующий его сайт (например geekbrains.ru). Браузер создает сокет для приема и формирует структуру с IP-адресом сервера и портом 443 (для HTTPS). В пакет записываются IP и порт сервера и IP (либо другой идентификатор устройства в сети) и порт настройки устройства пользователя. Сервер NAT меняет IP отправителя и присваивает новый порт отправки (он может совпадать с портом отправителя), которой привязывается к порту и IP-адресу устройства отправителя запроса. HTTPS-сервер формирует ответ и отправляет ответ с IP-адреса NAT-сервера и портом (выделенным NAT-сервером).

Создадим UDP-сервер, который будет передавать текущие время и дату. Если есть последовательность символов Time, то возвращается текущее время, при наличии символов Date — текущая дата.

```
//-------------*.h----------------
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QUdpSocket>
```

```
class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow(QWidget *parent = 0);
   ~MainWindow();
private:
   QUdpSocket *udpserver;
private slots:
   void workWithData();
};

#endif // MAINWINDOW_H

//-------------*.cpp----------------
#include "mainwindow.h"
#include <QNetworkDatagram>
#include <QDebug>
#include <QTime>
#include <QDate>

MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent)
{
   udpserver = new QUdpSocket(this);
   udpserver->setReadBufferSize(1024);                        // Устанавливаем
                                                              // максимальный
                                                              // размер
                                                              // приемного
                                                              // буфера
   // Для проверки раскомментировать код ниже:
   /*
   auto m = udpserver->bind(QHostAddress::LocalHost, 8085); // Открываем порт
                                                              // 8085 на
                                                              // прослушивание
                                                              // сетевых пакетов
   connect(udpserver, &QUdpSocket::readyRead, this, &MainWindow::workWithData);
   auto s = new QUdpSocket(this);
   char *y = "DateTime";
   s->writeDatagram((char*)y, sizeof y, QHostAddress("127.0.0.1"), 8085);
   */
}

MainWindow::~MainWindow()
{
}

void MainWindow::workWithData()
{
    QHostAddress clientAddr;//IP-адрес клиента
    unsigned short clientport;//порт, с которого пришел запрос
```

```
    while(udpserver->hasPendingDatagrams())
    {
        char msg[512] = {0};
        udpserver->readDatagram(msg, sizeof msg, &clientAddr, &clientport);
        qDebug() << QString(msg);
        qDebug() << "IP: " << clientAddr.toString() << " port: " <<
QString::number(clientport);
        QString s = QString(msg);
        QString ret = "";
        if (s.indexOf("Date") != -1)
        {
            QDate date = QDate::currentDate();
            ret += QString::number(date.day()) + "." +
QString::number(date.month()) + "." + QString::number(date.year()) + " ";
        }
        if (s.indexOf("Time") != -1)
        {
            QTime time = QTime::currentTime();
            ret += QString::number(time.hour()) + ":" +
QString::number(time.minute()) + ":" + QString::number(time.second());
        }
        udpserver->writeDatagram(ret.toUtf8(), clientAddr, clientport);
        qDebug() << "Ret: " << ret;
    }
}
```

Создадим клиент, который будет запрашивать время и/или дату. По приеме выводим дату и время на заголовке окна.

```
//**********MainWindow.h******************
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QUdpSocket>
#include <QPushButton>
#include <QCheckBox>
#include <QLineEdit>
#include <QGridLayout>

class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow(QWidget *parent = nullptr);
   ~MainWindow();
private:
   QUdpSocket *socket;
   QPushButton *sendButton;
```

```
    QCheckBox *dateCheckBox, *timeCheckBox;
    QGridLayout *layout;
    QLineEdit *ipEdit;
private slots:
    void queryUDPDateTime();
    void recDateTime();
};
#endif // MAINWINDOW_H


//**********MainWindow.cpp******************
#include "mainwindow.h"
#include <QWidget>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QWidget *w = new QWidget(this);
    this->setCentralWidget(w);
    layout = new QGridLayout(this);
    w->setLayout(layout);
    ipEdit  = new QLineEdit(this);
    layout->addWidget(ipEdit, 0, 0, 1, 2);
    dateCheckBox = new QCheckBox(this);
    layout->addWidget(dateCheckBox, 1, 0, 1, 1);
    timeCheckBox= new QCheckBox(this);
    layout->addWidget(timeCheckBox, 1, 1, 1, 1);
    dateCheckBox->setText("Получить дату");
    timeCheckBox->setText("Получить время");
    sendButton = new QPushButton(this);
    sendButton->setText("Запросить");

    layout->addWidget(sendButton, 2, 1, 1, 1);
    socket = new QUdpSocket(this);
    socket->bind(QHostAddress::LocalHost, 8085);
    connect(sendButton, &QPushButton::clicked, this,
&MainWindow::queryUDPDateTime);
}

MainWindow::~MainWindow()
{
}

void MainWindow::queryUDPDateTime()
{
    QString s = "";
    if (dateCheckBox->isChecked()) s += "Date";
    if (timeCheckBox->isChecked()) s += "Time";
    if (s.length())
    {
        socket->writeDatagram(s.toUtf8(), QHostAddress(ipEdit->text()), 8085); //
Отправляем запрос
```

```
    }
}

void MainWindow::recDateTime()
{
    while (socket->hasPendingDatagrams())
    {
        char data[512] = {0};
        socket->readDatagram(data, sizeof data);
        this->setWindowTitle(data);
    }
}
```

Для проверки этих примеров необходимы два компьютера, соединенные по локальной сети

Подключение к базам данных на сервере

Воспользуемся примером из предыдущего урока и добавим программе режим сервера. Выбрать, какая БД будет использоваться — локальная (она же — сервер для копий на других ПК) или клиентская — можно будет в главном окне.

Для обмена данными между ПК создадим структуру с уникальным номером задачи и основными полями (**UDPStruct.h**):

```
#ifndef UDPSTRUCT_H
#define UDPSTRUCT_H
#include <QString>
enum DBQuery {INIT_DB = 0x0000, GET_NEXT_TASK = 0x0001, ADD_NEW_TASK = 0x0002,
ADD_USER = 0x0003, GET_NEXT_USER = 0x0004,
            AUTORIZATION = 0x0005, ADMIN_MODE = 0x0006, USER_MODE = 0x0007,
UPDATE_TASK = 0x0008, USER_TASK_NEXT = 0x0009,
            UPDATE_USER_TASK = 0x000A, UPDATE_REPORT = 0x000B, READ_TASK =
0x000C};
struct UDPTask
{
    union{
        struct {
            DBQuery taskUDP;
            unsigned int amountTask;     // Количество задач
            bool correctTask;            // Успешность операции
            unsigned __int8 progress;    // Прогресс выполнения задачи
            unsigned int idtask;         // ID таблицы TaskList
            unsigned pos;                // Считываемая позиция задачи или
                                         // пользователя
            char task[1024];             // Текст задачи
            char describe[8192];         // Описание задачи
            char adminpass[128];         // Пароль администратора для добавления
```

```
                                        // новых задач и пользователей
            char s_data[16];            // Дата начала
            char e_data[16];            // Дата планируемого окончания
            char fio[64];               // ФИО пользователя
            char login[64];             // Логин для входа в окно задачи
            char usertask[4096];        // Задача пользователя
            char userreport[4096];      // Отчет пользователя
        }udp;
        char bytes[sizeof udp];
    };
    UDPTask()
    {
        memset(bytes, 0, sizeof bytes); // При инициализации очищаем структуру
    }
};
#endif // UDPSTRUCT_H
```

Изменим главную QML-страницу так, чтобы можно было выбрать использовать локальную БД SQLite или соединиться с другим экземпляром программы по сети Ethernet.

Файл **main.qml**:

```
import QtQuick 2.13
import QtQuick.Window 2.13
import com.gb.BaseTask 1.0
import QtQuick.Controls 2.5

Window {
    property int amountTasks: 0
    property string newtableline: 'TaskElement{width:800; height:30;'
    property UserList userlist: null
    property TaskWindow taskWindow:null
    function loadnext()
    {
        basereader.getNext();
    }
    BaseReader
    {
        id: basereader
        onStatusLoadBase:
        {
            if (correct == false)
            {
                Qt.quit()
                console.log(error)
            }
            else
            {
                serverclientrect.destroy() // Удаляем панель выбора
            }
```

```
            if (amount > 0)
            {
                loadnext()
            }
        }
        onLogMistake:
        {
            console.log(msg)
        }
        onPrintData:
        {
            var itog = newtableline + 'taskTxt:"' + task + '";begTxt:"'
                    + begDate + '";endTxt:"' + endDate +'";prgTxt:"'
                    + Number(progress) + '%"; y:' + amountTasks * 25 + ';idt:' +
id +
                    ';dbreader:basereader' + '}';
            console.log(itog)
            Qt.createQmlObject(itog, table, "lineTask" + amountTasks++)
            getNext()
        }
        onEmitNextLine: getNext()        // Добавление нового компонента
        onAddUserToList:
        {
            if (userlist != null)
            {
                userlist.addUser(fio)
                basereader.getNextUser()
            }
        }
        onTaskFullInformation:
        {
            var comp = Qt.createComponent("qrc:///TaskWindow.qml")
            taskWindow = comp.createObject(TaskWindow)
            taskWindow.bdreader = basereader
            taskWindow.idt = id
            taskWindow.taskname = task
            taskWindow.describe = descr
            taskWindow.startDate = begDate
            taskWindow.endDate = endDate
            taskWindow.progress = progress
            taskWindow.show()
        }
        onSetAdminModeEditTask: {
            if (taskWindow != null)
            {
                taskWindow.setAdminMode()
            }
        }
        onAdminUserTask:
        {
            if (taskWindow != null)
            {
```

```
                taskWindow.adminmode.adminModeInfoUserTask(active, fio, login,
usertask, report)
            }
        }
        onSetWorkMode: {
            if (taskWindow != null)
            {
                taskWindow.setWorkMode(fio, bigTask, report)
            }
        }
    }
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")
    Component.onCompleted:
    {
        table.width = width
        table.height = height - 50;


        rect.width = width
        rect.height = 50;
        rect.y = height - 50
    }

    ScrollView
    {
        id:table
        width: 600
        ScrollBar.horizontal.policy: ScrollBar.AlwaysOff
        ScrollBar.vertical.policy: ScrollBar.AlwaysOn
    }
    Rectangle
    {
        id: rect
        height: 30
        Button
        {
            id:addButton
            text: "Добавить"
            onClicked:
            {
                console.log("THIS")
                var component = Qt.createComponent("qrc:///NewTask.qml")
                console.log(component)
                var obj = component.createObject(NewTask)
                console.log(obj)
                obj.baseRender = basereader
                obj.show()
            }
        }
```

```
        Button{
            id:userButton
            height: 30
            x:150
            text: "Пользователи"
            onClicked:
            {
                var comp = Qt.createComponent("qrc:///UserList.qml")
                userlist = comp.createObject(UserList)
                userlist.baseRender = basereader
                userlist.show();
                basereader.getFirstUser()
            }
        }
        Rectangle
        {
            id:serverclientrect
            x:300
            y:userButton.y
            width: 400
            height: 100
            color: 'red'
            Button {
                id:localeBD
                height: 30
                text: "Локальная"
                onClicked:
                {
                    basereader.initBase() // Работает как в примере из 11 урока
                }
            }
            Rectangle
            {
                anchors.fill: ipServer
            }
            TextEdit
            {
                id:ipServer
                x: localeBD.x + localeBD.width + 100
                width: 100
                text: "192.168.1.14"
            }
            Button
            {
                id:clientMode
                x:ipServer.x
                height: 30
                y:ipServer.y + ipServer.height + 10
                text: "Подключиться"
                onClicked: {
                    basereader.initBaseClient(ipServer.text)
                }
```

```
            }
        }
    }

}
```

Теперь внесем правки в класс **BaseTask**. При инициализации класса создадим создадим сокет и установим по умолчанию флаги инициализации и режим работы БД.

Конструктор класса:

```
#include "basetask.h"
#include <QSqlError>
#include <QVariant>
#include "UDPStruct.h"

const QString adminPassword = "JeFiHi"; // Пароль администратора

BaseTask::BaseTask(QObject *parent) : QObject(parent), isInitDB(false),
isClient(false)
{
    socket = new QUdpSocket(this);
}
```

При успешной инициализации локальной БД устанавливаем флаг **isInitDB** в значение **true**. Также добавим новый метод **initBaseClient** для сетевой инициализации. Установим связь сигнала от класса **QUDPSocket** с соответствующим слотом (в зависимости от способа инициализации БД):

```
void BaseTask::initBase()
{
    database = QSqlDatabase::addDatabase("QSQLITE");           // SQLite driver
    database.setDatabaseName("./dbtasks.db");                  // Файл БД
    if (!database.open())
    {
        emit statusLoadBase(false, 0, tr("Couldn't open DB")); // Не удалось
                                                               // подключиться
                                                               // к БД
    }
    else
    {
        query = QSqlQuery(database);
        bool p = query.exec("create table if not exists TaskList(id int not
null,task varchar(256) not null,"
                            "describe varchar(4096),"
                  "begdata varchar(32) not null,enddata varchar(32) not
null,prgress int(8) not null)");
        if (!p)
        {
```

```
            emit statusLoadBase(false, 0, query.lastError().text());
            return;
        }
        p = query.exec("create table if not exists UserList(fio varchar(128)not
null,login varchar(128) not null,pass varchar(128) not null)");
        p = query.exec("create table if not exists TaskUser(id int not null,
login varchar(128)not null, usertask varchar(4096), userreport varchar(4096))");
        query.exec("select count (*) from TaskList");
        if (query.next())
        {
            quint32 amount = query.value(0).toUInt();
            listIDTask.clear();
            listlogins.clear();
            if (query.exec("select id from TaskList"))
            {
                while (query.next())
                {
                    listIDTask.append(query.value(0).toUInt());
                }
            }
            pos = 0;

            if (query.exec("select login from UserList"))
            {
                while (query.next())
                {
                    listlogins.append(query.value("login").toString());
                }
            }
            upos = 0;

            emit statusLoadBase(true, amount, "");
            socket->bind(QHostAddress::LocalHost, 8888);
            connect(socket, &QUdpSocket::readyRead, this,
&BaseTask::serverUDPReceive);
            isInitDB = true;

        } else
        {
            emit statusLoadBase(false, 0, query.lastError().text());
        }

    }
}

void BaseTask::initBaseClient(QString ip)
{
    socket->bind(QHostAddress::LocalHost, 8888);
    serverHost = QHostAddress(ip);
    connect(socket, &QUdpSocket::readyRead, this, &BaseTask::clientUDPReceive);
    isClient = true; // Содержимое БД будет запрашиваться через сеть Ethernet
    UDPTask udptask;
```

```
    udptask.udp.taskUDP = DBQuery::INIT_DB;
    socket->writeDatagram(udptask.bytes, sizeof udptask.bytes, serverHost, 8888);
}
```

Для добавления, редактирования, считывания данных таблиц будем проверять флаг **isClient**, чтобы определять, откуда брать данные (локальная БД, запрос через сеть).

Код для считывания таблицы задач (метод **getNext**) примет следующий вид:

```
void BaseTask::getNext()
{
    if (!isInitDB) return;
    if (isClient)
    {
        UDPTask udp;
        udp.udp.taskUDP = DBQuery::GET_NEXT_TASK; // Операция, которую нужно
                                                  // выполнить над БД
        udp.udp.pos = pos++;                      // Увеличиваем счетчик
                                                  // Отправляем датаграмму на
                                                  // серверный экземпляр
                                                  // программы
        socket->writeDatagram(udp.bytes, sizeof udp.bytes, serverHost, 8888);
        return;
    }
    if (pos >= listIDTask.size())return;
    bool p = query.exec("select * from TaskList where id=" +
QString::number(listIDTask[pos++]));
    if (query.next())
    {

        emit printData(query.value(0).toUInt(), query.value(1).toString(),
                       query.value(2).toString(),query.value(3).toString(),
                       query.value(4).toString(),query.value(5).toUInt());
    }
}
```

Добавление новой задачи:

```
void BaseTask::addNew(QString task, QString descr, QString begData, QString
endData, QString admpass)
{
    if (!isInitDB) return;
    if (isClient)
    {
        UDPTask udp;
        udp.udp.taskUDP = DBQuery::ADD_NEW_TASK;
        QByteArray ba = admpass.toUtf8();
        memcpy(udp.udp.adminpass, ba.data(), ba.size());
```

```
        ba = task.toUtf8();
        memcpy(udp.udp.task, ba.data(), ba.size());
        ba = descr.toUtf8();
        memcpy(udp.udp.describe, ba.data(), ba.size());
        ba = begData.toUtf8();
        memcpy(udp.udp.s_data, ba.data(), ba.size());
        ba = endData.toUtf8();
        memcpy(udp.udp.e_data, ba.data(), ba.size());
        socket->writeDatagram(udp.bytes, sizeof udp.bytes, serverHost, 8888);
        return;
    }
    if (admpass != adminPassword) {
        return;
    }
    quint32 id = 0;
    for (;;id++) // Определяем свободный ID
    {
        query.exec("select count (*) from TaskList where id=" +
QString::number(id));
        if (query.next())
        {
            if (!query.value(0).toInt()) break;
        }
    }
    bool corr = query.exec("insert into TaskList VALUES(" +
                QString::number(id) + ",'" +
                task + "','" +
                descr + "','" +
                begData + "','" +
                endData + "',0)"
                );
    if (!corr) emit logMistake(query.lastError().text());
    else
    {
        listIDTask.append(id);
        emit emitNextLine();
    }
}
```

Считывание пользователей: основные изменения коснутся метода **getNextUser**, хотя для проверки успешной инициализации БД дополним и метод **getFirstUser**.

```
void BaseTask::getFirstUser()
{
    if (!isInitDB) return;
    upos = 0;
    getNextUser();
}
```

```
void BaseTask::getNextUser()
{
   if (!isInitDB) return;
   if (isClient)
   {
       UDPTask udp;
       udp.udp.taskUDP = DBQuery::GET_NEXT_USER;
       udp.udp.pos = upos++;
       socket->writeDatagram(udp.bytes, sizeof udp.bytes, serverHost, 8888);
       return;
   }
   if (upos < listlogins.size())
   {
       bool p = query.exec("select fio from UserList where login='" +
listlogins.at(upos++) + "'");
       if (p)
       {
           if (query.next())
           {
               emit addUserToList(query.value("fio").toString());
           }
       } else emit logMistake("Ошибка SQL-запрос");
   }
}
```

Добавление нового пользователя:

```
void BaseTask::addNewUser(QString fio, QString login, QString passwrd, QString
apasswrd)
{
   if (!isInitDB) return;
   if (isClient)
   {
       UDPTask udp;
       udp.udp.taskUDP = DBQuery::ADD_USER;
       QByteArray ba = apasswrd.toUtf8();
       memcpy(udp.udp.adminpass, ba.data(), ba.size());
       ba = fio.toUtf8();
       memcpy(udp.udp.fio, ba.data(), ba.size());
       ba = login.toUtf8();
       memcpy(udp.udp.login, ba.data(), ba.size());
       ba = passwrd.toUtf8();
       memcpy(udp.udp.pass, ba.data(), ba.size());
       socket->writeDatagram(udp.bytes, sizeof udp.bytes, serverHost, 8888);
       return;
   }
   if (fio.length() == 0 || login.length() == 0 || passwrd.length() == 0 ||
apasswrd != adminPassword || login == "admin") return;
   bool user = query.exec("select * from UserList where login='" + login + "'");
// Проверяем есть ли такой пользователь
```

```
    if (!user)
    {
        logMistake("Ошибка SQL");
        return;
    }

    if (query.next())
    {
        logMistake("Такой пользователь есть");
        return;
    }

    user = query.exec("insert into UserList values('" + fio +
                      "','" + login + "','" + passwrd + "')"
                      );
    if (user)
    {
        listlogins.append(login);
        getNextUser();
    }
    else
    {
        logMistake("Не удалось добавить");
    }
}
```

Запрос на просмотр задачи:

```
void BaseTask::getLine(int id)
{
    if (!isInitDB) return;
    if (isClient)
    {
        UDPTask udp;
        udp.udp.taskUDP = DBQuery::READ_TASK;
        udp.udp.idtask = id;
        socket->writeDatagram(udp.bytes, sizeof udp.bytes, serverHost, 8888);
        return;
    }
    if (query.exec("select * from TaskList where id = " + QString::number(id)))
    {
        if (query.next())
        {
            curID = query.value(0).toUInt();
            tupos = 0;
            emit taskFullInformation(curID, query.value(1).toString(),

query.value(2).toString(),query.value(3).toString(),

query.value(4).toString(),query.value(5).toUInt());
        }
```

```
    } else emit logMistake("Ошибка SQL-запроса");
}
```

Авторизация пользователя:

```
void BaseTask::autorization(QString login, QString passwrd)
{
    if (!isInitDB) return;
    if (isClient)
    {
        UDPTask udp;
        udp.udp.taskUDP = DBQuery::AUTORIZATION;
        QByteArray ba = passwrd.toUtf8();
        memcpy(udp.udp.pass, ba.data(), ba.size());
        ba = login.toUtf8();
        memcpy(udp.udp.login, ba.data(), ba.size());
        socket->writeDatagram(udp.bytes, sizeof udp.bytes, serverHost, 8888);
        return;
    }
    if (login == "admin" && passwrd == adminPassword) //режим администратирования
    {
        emit setAdminModeEditTask();
        return;
    }
    if (query.exec("select * from UserList where login='" + login + "' and
pass='" + passwrd + "'"))
    {
        if (query.next())
        {

            QString fio = query.value("fio").toString();
            query.exec("select * from TaskUser where id=" +
QString::number(curID) + " and login='" + login + "'");
            if (query.next())
            {
                curlogin = login;
                QString usertask = query.value("usertask").toString();
                QString report = query.value("userreport").toString();
                QString mtask = "", describe = "";
                if (query.exec("select * from TaskList where id=" +
QString::number(curID)))
                {
                    if (query.next())
                    {
                        mtask = query.value("task").toString();
                        describe = query.value("describe").toString();
                    }
                } else return;
                QString info = "Основная задача:" + mtask + "\nОписание:" +
describe + "\nВаша задача:" + usertask;
                emit setWorkMode(fio, info, report);
```

```
                return;
            }
        }emit logMistake("Нет такого пользователя");
    }else emit logMistake("Ошибка SQL-запроса");
}
```

Обновление администратором текущей задачи:

```
void BaseTask::updateTaskData(quint32 id, QString task, QString describe,
QString stDate, QString endDate, QString progress)
{
    if (!isInitDB) return;
    if (isClient)
    {
        UDPTask udp;
        udp.udp.taskUDP = DBQuery::UPDATE_TASK;
        udp.udp.idtask = curID;
        QByteArray ba = task.toUtf8();
        memcpy(udp.udp.task, ba.data(), ba.size());
        ba = describe.toUtf8();
        memcpy(udp.udp.describe, ba.data(), ba.size());
        ba = stDate.toUtf8();
        memcpy(udp.udp.s_data, ba.data(), ba.size());
        ba = endDate.toUtf8();
        memcpy(udp.udp.e_data, ba.data(), ba.size());
        int prog = progress.toUInt();
        if (prog < 0) prog = 0;
        if (prog > 100) prog = 100;
        udp.udp.progress = prog;
        socket->writeDatagram(udp.bytes, sizeof udp.bytes, serverHost, 8888);
        return;
    }
    int prog = progress.toUInt();
    if (prog < 0) prog = 0;
    if (prog > 100) prog = 100;
    QString cmd = "task='" + task + "',";
    cmd += "describe='" + describe + "',";
    cmd += "begdata='" + stDate + "',";
    cmd += "enddata='" + endDate+ "',";
    cmd += "prgress=" + QString::number(prog);
    bool res = query.exec("update TaskList set " + cmd + " where id=" +
QString::number(id));
    if (!res)
    {
        emit logMistake(query.lastError().text());
    }
}
```

Обновление задачи для исполнителей:

```
void BaseTask::changeTaskUser(QString login, QString task)
{
   if (!isInitDB) return;
   if (isClient)
   {
       UDPTask udp;
       udp.udp.taskUDP = DBQuery::UPDATE_USER_TASK;
       udp.udp.idtask = curID;
       QByteArray ba = curlogin.toUtf8();
       memcpy(udp.udp.login, ba.data(), ba.size());
       ba = task.toUtf8();
       memcpy(udp.udp.usertask, ba.data(), ba.size());
       return;
   }
   bool res = query.exec("select * from TaskUser where id=" +
QString::number(curID) + " and login='" + login + "'");
   if (!res)
   {
       emit logMistake(query.lastError().text());
       return;
   }
   if (query.next())
   {
       if (!query.exec("update TaskUser set usertask='" + task + "' where id=" +
QString::number(curID) + " and login='" + login + "'"))
       {
           emit logMistake(query.lastError().text());
       }

   } else
   {
       // p = query.exec("create table if not exists TaskUser(id int not null,
login varchar(128)not null, usertask varchar(4096), userreport varchar(4096))");
       if (!query.exec("insert into TaskUser values(" + QString::number(curID) +
                       ",'" + login + "','" + task + "','')"
                       ))
       {
           emit logMistake(query.lastError().text());
       }
   }
}
```

Обновление отчета исполнителем:

```
void BaseTask::saveReport(QString report)
{
   if (!isInitDB) return;
   if (isClient)
   {
       UDPTask udp;
       udp.udp.taskUDP = DBQuery::UPDATE_REPORT;
```

```
        udp.udp.idtask = curID;
        QByteArray ba = curlogin.toUtf8();
        memcpy(udp.udp.login, ba.data(), ba.size());
        ba = report.toUtf8();
        memcpy(udp.udp.userreport, ba.data(), ba.size());
        socket->writeDatagram(udp.bytes, sizeof udp.bytes, serverHost, 8888);
    }
    if (!query.exec("update TaskUser set userreport = '" + report + "' where id="
+ QString::number(curID) + " and login='" + curlogin + "'"))
    {
        emit logMistake(query.lastError().text());
    }
}
```

Обработка запросов со стороны сервера:

```
void BaseTask::serverUDPReceive()
{
    UDPTask udp;
    if (!socket->hasPendingDatagrams())
    {
        return;
    }
    socket->readDatagram(udp.bytes, sizeof udp, &clienHost, &clientport);
    if (udp.udp.taskUDP = DBQuery::INIT_DB)
    {
        udp.udp.correctTask = isInitDB;
        udp.udp.amountTask = listIDTask.size();
        socket->writeDatagram(udp.bytes, sizeof udp, clienHost, clientport);
        return;
    }
    if (udp.udp.taskUDP == DBQuery::GET_NEXT_TASK)
    {

        if (pos >= listIDTask.size())return;
        bool p = query.exec("select * from TaskList where id=" +
QString::number(listIDTask[udp.udp.pos]));
        if (query.next())
        {
            udp.udp.idtask = query.value(0).toInt();
            QByteArray ba = query.value(1).toString().toUtf8();
            memcpy(udp.udp.task, ba.data(), ba.size());
            ba = query.value(2).toString().toUtf8();
            memcpy(udp.udp.describe, ba.data(), ba.size());
            ba = query.value(3).toString().toUtf8();
            memcpy(udp.udp.s_data, ba.data(), ba.size());
            ba = query.value(4).toString().toUtf8();
            memcpy(udp.udp.e_data, ba.data(), ba.size());
            udp.udp.progress = query.value(5).toInt();
            socket->writeDatagram(udp.bytes, sizeof udp, clienHost, clientport);
        }
```

```
        return;
    }
    if (udp.udp.taskUDP == DBQuery::ADD_NEW_TASK)
    {
        int curAmountTasks = listIDTask.size();
        addNew(QString::fromUtf8(udp.udp.task),
QString::fromUtf8(udp.udp.describe), QString::fromUtf8(udp.udp.s_data),
               QString::fromUtf8(udp.udp.e_data),
QString::fromUtf8(udp.udp.adminpass));
        if (curAmountTasks < listIDTask.size())
        {
            bool p = query.exec("select * from TaskList where id=" +
QString::number(listIDTask[curAmountTasks]));
            if (query.next())
            {
                udp.udp.idtask = query.value(0).toInt();
                QByteArray ba = query.value(1).toString().toUtf8();
                memcpy(udp.udp.task, ba.data(), ba.size());
                ba = query.value(2).toString().toUtf8();
                memcpy(udp.udp.describe, ba.data(), ba.size());
                ba = query.value(3).toString().toUtf8();
                memcpy(udp.udp.s_data, ba.data(), ba.size());
                ba = query.value(4).toString().toUtf8();
                memcpy(udp.udp.e_data, ba.data(), ba.size());
                udp.udp.progress = query.value(5).toInt();
                udp.udp.taskUDP == DBQuery::GET_NEXT_TASK;
                socket->writeDatagram(udp.bytes, sizeof udp, clienHost,
clientport);
            }
        }
        return;
    }
    if (udp.udp.taskUDP == DBQuery::GET_NEXT_USER)
    {
        if (udp.udp.pos < listlogins.size())
        {
            bool p = query.exec("select fio from UserList where login='" +
listlogins.at(udp.udp.pos) + "'");
            if (p)
            {
                if (query.next())
                {
                    QByteArray ba = udp.udp.fio;
                    memcpy(udp.udp.fio, ba.data(), ba.size());
                    socket->writeDatagram(udp.bytes, sizeof udp, clienHost,
clientport);
                }
            }
        }
        return;
    }
    if (udp.udp.taskUDP == DBQuery::ADD_USER)
```

```
    {
        udp.udp.taskUDP = DBQuery::GET_NEXT_USER;
        QString fio = QString::fromUtf8(udp.udp.fio);
        QString login = QString::fromUtf8(udp.udp.login);
        QString passwrd = QString::fromUtf8(udp.udp.pass);
        QString apasswrd = QString::fromUtf8(udp.udp.adminpass);
        int curAmountUser = listlogins.size();
        addNewUser(fio, login, passwrd, apasswrd);
        if (curAmountUser = listlogins.size())
        {
            socket->writeDatagram(udp.bytes, sizeof udp, clienHost, clientport);
        }
        return;
    }
    if (udp.udp.taskUDP == DBQuery::READ_TASK)
    {
        if (query.exec("select * from TaskList where id = " +
QString::number(udp.udp.idtask)))
        {
            if (query.next())
            {
                QByteArray ba = query.value(1).toString().toUtf8();
                memcpy(udp.udp.task, ba.data(), ba.size());
                ba = query.value(2).toString().toUtf8();
                memcpy(udp.udp.describe, ba.data(), ba.size());
                ba = query.value(3).toString().toUtf8();
                memcpy(udp.udp.s_data, ba.data(), ba.size());
                ba = query.value(4).toString().toUtf8();
                memcpy(udp.udp.e_data, ba.data(), ba.size());
                udp.udp.progress = query.value(5).toInt();
                socket->writeDatagram(udp.bytes, sizeof udp, clienHost,
clientport);
            }
        }
        return;
    }
    if (udp.udp.taskUDP == DBQuery::AUTORIZATION)
    {
        QString login = QString::fromUtf8(udp.udp.login);
        QString passwrd = QString::fromUtf8(udp.udp.pass);
        if (login == "admin" && passwrd == adminPassword)
        {
            udp.udp.taskUDP = DBQuery::ADMIN_MODE;
            socket->writeDatagram(udp.bytes, sizeof udp, clienHost, clientport);
            return;
        }
        if (login == "admin" || login.length() == 0 || passwrd.length() == 0)
return;
        if (query.exec("select * from UserList where login='" + login + "' and
pass='" + passwrd + "'"))
        {
            if (query.next())
```

```
            {

                QString fio = query.value("fio").toString();
                query.exec("select * from TaskUser where id=" +
QString::number(udp.udp.idtask) + " and login='" + login + "'");
                if (query.next())
                {
                    QString usertask = query.value("usertask").toString();
                    QString report = query.value("userreport").toString();
                    QString mtask = "", describe = "";
                    if (query.exec("select * from TaskList where id=" +
QString::number(udp.udp.idtask)))
                    {
                        if (query.next())
                        {
                            mtask = query.value("task").toString();
                            describe = query.value("describe").toString();
                        }
                    } else return;
                    QString info = "Основная задача:" + mtask + "\nОписание:" +
describe + "\nВаша задача:" + usertask;
                    // emit setWorkMode(fio, info, report);
                    udp.udp.taskUDP = DBQuery::USER_MODE;
                    QByteArray ba = fio.toUtf8();
                    memcpy(udp.udp.fio, ba.data(),  ba.size());
                    ba = info.toUtf8();
                    memcpy(udp.udp.describe, ba.data(),  ba.size());
                    ba = report.toUtf8();
                    memcpy(udp.udp.userreport, ba.data(),  ba.size());
                    socket->writeDatagram(udp.bytes, sizeof udp, clienHost,
clientport);
                    return;
                }
            }emit
        }
        return;
    }
    if (udp.udp.taskUDP == DBQuery::USER_TASK_NEXT)
    {
        if (listlogins.size() <= udp.udp.pos) return;
        /*********************************************************/
        bool res = query.exec("select * from TaskUser where id=" +
QString::number(udp.udp.idtask));
        if (res)
        {
            QString task = "";
            QString report = "";
            QString fio = "";
            bool act= false;
            while(query.next())
            {
                if (query.value("login").toString() ==
```

```
listlogins.at(udp.udp.pos))
                {
                    act = true;
                    break;
                }
            }
            if (act)
            {
                task = query.value("usertask").toString();
                report = query.value("userreport").toString();
            }
            res = query.exec("select fio from UserList where login='" +
listlogins.at(udp.udp.pos) + "'");
            if (res)
            {
                res = query.next();
                if(!res) return;
                fio = query.value("fio").toString();
            } else return;
            // emit adminUserTask(act, fio, listlogins.at(t), task, report);
            udp.udp.correctTask = act;
            QByteArray ba = fio.toUtf8();
            memcpy(udp.udp.fio, ba.data(), ba.size());
            ba = listlogins.at(udp.udp.pos).toUtf8();
            memcpy(udp.udp.login, ba.data(), ba.size());
            ba = task.toUtf8();
            memcpy(udp.udp.usertask, ba.data(), ba.size());
            ba = report.toUtf8();
            memcpy(udp.udp.userreport, ba.data(), ba.size());
            socket->writeDatagram(udp.bytes, sizeof udp, clienHost, clientport);
        }
        /*************************************************************/
        return;
    }
    if (udp.udp.taskUDP == DBQuery::UPDATE_USER_TASK)
    {
        //*****************************************************************
        bool res = query.exec("select * from TaskUser where id=" +
                              QString::number(udp.udp.idtask) + " and login='" +
QString::fromUtf8(udp.udp.login) + "'");
        if (!res)
        {
            return;
        }
        if (query.next())
        {
            if (!query.exec("update TaskUser set usertask='" +
QString::fromUtf8(udp.udp.usertask) + "' where id=" +
QString::number(udp.udp.idtask)
                            + " and login='" + QString::fromUtf8(udp.udp.login)
+ "'"))
            {
```

```
                emit logMistake(query.lastError().text());
            }

        } else
        {
            query.exec("insert into TaskUser values(" +
QString::number(udp.udp.idtask) +
                               ",'" + QString::fromUtf8(udp.udp.login) + "','" +
QString::fromUtf8(udp.udp.usertask) + "','')");
        }
        //*****************************************************************
        return;
    }
    if (udp.udp.taskUDP == DBQuery::UPDATE_TASK)
    {
        int prog = udp.udp.progress;
        QString cmd = "task='" + QString::fromUtf8(udp.udp.usertask) + "',";
        cmd += "describe='" + QString::fromUtf8(udp.udp.describe) + "',";
        cmd += "begdata='" + QString::fromUtf8(udp.udp.s_data) + "',";
        cmd += "enddata='" + QString::fromUtf8(udp.udp.e_data)+ "',";
        cmd += "prgress=" + QString::number(prog);
        query.exec("update TaskList set " + cmd + " where id=" +
QString::number(udp.udp.idtask));
        return;
    }
    if (udp.udp.taskUDP == DBQuery::UPDATE_TASK)
    {
        query.exec("update TaskUser set userreport = '" +
QString::fromUtf8(udp.udp.userreport)
                   + "' where id=" + QString::number(udp.udp.idtask) + " and
login='" + QString::fromUtf8(udp.udp.login) + "'");
        return;
    }
}
```

Для работы в режиме клиента в зависимости от задачи извлекаем данные из структуры и генерируем необходимые сигналы согласно локальному режиму работы с БД:

```
void BaseTask::clientUDPReceive()
{
   UDPTask udp;
   if (udp.udp.taskUDP = DBQuery::INIT_DB)
   {
       emit statusLoadBase(udp.udp.correctTask, udp.udp.amountTask, "");
       pos = 0;
       upos = 0;
       return;
   }
   if (udp.udp.taskUDP == DBQuery::GET_NEXT_TASK)
   {
```

```
        emit printData(udp.udp.idtask, QString::fromUtf8(udp.udp.task),
QString::fromUtf8(udp.udp.describe),
                       QString::fromUtf8(udp.udp.s_data),
QString::fromUtf8(udp.udp.e_data), udp.udp.progress);
        return;
    }
    if (udp.udp.taskUDP == DBQuery::GET_NEXT_USER)
    {
        emit addUserToList(QString(udp.udp.fio));
        return;
    }
    if (udp.udp.taskUDP == DBQuery::READ_TASK)
    {
        curID = udp.udp.idtask;
        tupos = 0;
        emit taskFullInformation(curID, QString::fromUtf8(udp.udp.task),
QString::fromUtf8(udp.udp.describe),
                                 QString::fromUtf8(udp.udp.s_data),
QString::fromUtf8(udp.udp.e_data), udp.udp.progress);
        return;
    }
    if (udp.udp.taskUDP == DBQuery::ADMIN_MODE)
    {
        emit setAdminModeEditTask();
        return;
    }
    if (udp.udp.taskUDP == DBQuery::USER_MODE)
    {
        curlogin = QString::fromUtf8(udp.udp.login);
        emit setWorkMode(QString::fromUtf8(udp.udp.fio),
QString::fromUtf8(udp.udp.describe), QString::fromUtf8(udp.udp.userreport));
        return;
    }
    if (udp.udp.taskUDP == DBQuery::USER_TASK_NEXT)
    {
        emit adminUserTask(udp.udp.correctTask, QString::fromUtf8(udp.udp.fio),
QString::fromUtf8(udp.udp.login),
                           QString::fromUtf8(udp.udp.usertask),
QString::fromUtf8(udp.udp.userreport));
        return;
    }
}
```

В объявление класса **BaseTask** добавим новые методы и слоты для обеспечения работы UDP:

```
#ifndef BASETASK_H
#define BASETASK_H

#include <QObject>
#include <QDate>
#include <QSqlDatabase>
```

```
#include <QSqlQuery>
#include <QHash>
#include <QUdpSocket>

struct BData{
   int day, mounth, year;
};
struct DataBase
{
   quint32 idTask;
   QChar task[512];
   QChar discrib[4096];
   BData beginDate, endDate;
   quint8 progress;
};

class BaseTask : public QObject
{
   Q_OBJECT
public:
   explicit BaseTask(QObject *parent = nullptr);

signals:
   void statusLoadBase(bool correct, quint32 amount, QString error);
   void printData(int id, QString task, QString descr, QString begDate, QString
endDate, quint32 progress);
   void logMistake(QString msg);
   void emitNextLine();
   void addUserToList(QString fio);
   void taskFullInformation(int id, QString task, QString descr, QString
begDate, QString endDate, quint32 progress);
   void setAdminModeEditTask();
   void setWorkMode(QString fio, QString bigTask, QString report);
   void adminUserTask(bool active, QString fio, QString login, QString usertask,
QString report);
public slots:
   void initBase();
   void initBaseClient(QString ip);
   void getNext();
   void getLine(int id);
   void addNew(QString task, QString descr, QString begData, QString endData,
QString admpass);
   void addNewUser(QString fio, QString login, QString passwrd, QString
apasswrd);
   void getFirstUser();
   void getNextUser();
   void autorization(QString login, QString passwrd);
   void updateTaskData(quint32 id, QString task, QString describe, QString
stDate, QString endDate, QString progress);

   void getFirstUserTask();
   void getNextUserTask();
```

```
    void changeTaskUser(QString login, QString task);
    void saveReport(QString report);
    void serverUDPReceive();
    void clientUDPReceive();

private:
    QSqlDatabase database;
    QSqlQuery query;
    // Список задач
    QList<quint32>listIDTask;
    quint32 pos;
    // Список пользователей
    QList<QString>listlogins;
    quint32 upos;
    //-----------------------
    quint32 curID;
    quint32 tupos;
    QString curlogin;
    //---------- Сокет ------------
    QUdpSocket *socket;
    bool isInitDB;
    bool isClient;
    QHostAddress serverHost;
    QHostAddress clienHost;
    unsigned short clientport;
};

#endif // BASETASK_H
```

Это демонстрационный пример. На практике данные, передаваемые через сеть Ethernet, необходимо шифровать (как правило ассиметричным алгоритмом, например RSA, в котором используется пара ключей — открытый и закрытый). В Qt для работы с сокетом с шифрованием используется класс **QSslSocket**. Для корректной работы **QSslSocket**, возможно, понадобится пересобрать модуль **network** с библиотекой **OpenSSL**.

Практическое задание

Разработать программу, которая будет скачивать картинки с Яндекс.Картинки и отображать их. Пользователь должен иметь возможность ввести название того, что он хочет найти (например, "кот") и программа должна вывести 3 картинки по этому запросу.

Подсказки:

1. Вам необходимо скачать html-разметку страницы с поисковым запросом. Например, для запроса "кот" необходимо скачать html страницы: https://yandex.ru/images/search?text=кот .

2. Из html-разметки вам необходимо найти теги <img>, которые представляют картинки. У этого тега есть атрибут src, в котором и содержится ссылка на картинку. Ее вам нужно вычленить. Для того, чтобы слишком часто не делать запросы к Яндексу и чтобы он вас не заблокировал, скачайте html

страницы в файл и работайте с файлом. Позже, когда отладите вашу программу, можно вернуться к скачиванию html с сайта.
3. По найденным ссылкам скачать картинки и отобразить их в окне программы. Реализация этой части рассмотрена на 12 уроке.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Составляем DNS-запрос вручную. https://habr.com/ru/post/346098/
2. Официальная документация по класса QSQLDatabase. https://doc.qt.io/qt-5/qsqldatabase.html
3. Официальная документация по класса QSQLQuery https://doc.qt.io/qt-5/qsqlquery.html

Дополнительные материалы

1. Составляем DNS-запрос вручную.
2. Официальная документация по класса QSQLDatabase.
3. Официальная документация по класса QSQLQuery.