МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Воронежский государственный технический университет»

Кафедра радиоэлектронных устройств и систем

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ №11-12
для студентов специальности 11.05.01
«Радиоэлектронные системы и комплексы»
очной формы обучения

Воронеж 2024

УДК 681.3.06(07)
ББК 32.97я7

Составитель А. И. Сукачев

Информационные технологии: методические указания к выполнению лабораторных работ №11-12 для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы» очной формы обучения / ФГБОУ ВО «Воронежский государственный технический университет»; сост.: А. И. Сукачев. – Воронеж: Изд-во ВГТУ, 2024. – 33 с.

В соответствии с рабочими учебными программами дисциплин приведены описания методов измерений и методик выполнения лабораторных работ, изложены теоретические сведения, лежащие в основе программирования на языке С++. По каждой лабораторной работе в описание включены: цель, основные теоретические сведения, порядок подготовки и проведения работы, перечень положений, которые необходимо отразить в выводах.

Предназначены для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы» очной формы обучения.

Методические указания подготовлены в электронном виде и содержатся в файле МУ_ИТ_ЛР11-12.pdf.

Ил. 36. Табл. 2. Библиогр.: 3 назв.

УДК 681.3.06(07)
ББК 32.97я7

Рецензент – А. В. Останков, д-р техн. наук, профессор кафедры радиотехники ВГТУ

Издается по решению редакционно-издательского совета Воронежского государственного технического университета

1. ЛАБОРАТОРНАЯ РАБОТА № 11
ОСНОВЫ ПРОЕКТИРОВАНИЯ АСИНХРОННОГО СЕРВЕРНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. ОБЩИЕ УКАЗАНИЯ

1.1.1. ЦЕЛЬ РАБОТЫ

Получить знания по проектированию простейшего асинхронного сервера.

1.1.2. ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Основным содержанием работы является создание простейшего асинхронного сервера. В ходе работы идёт ознакомление с приведённым примером приложения, производится анализ используемых технологий. На основе полученных знаний выполняется индивидуальное задание по проектированию асинхронного сервера.

1.2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.2.1. КЛАСС QTCP SERVER

Этот класс позволяет принимать входящие TCP-соединения. Вы можете указать порт или сделать так, чтобы QTcpServer выбрал его автоматически. Вы можете слушать по определенному адресу или по всем адресам машины.

Вызовите listen (), чтобы сервер прослушивал входящие соединения. Затем сигнал newConnection () отправляется каждый раз, когда клиент подключается к серверу.

Вызовите nextPendingConnection (), чтобы принять ожидающее соединение как подключенный QTcpSocket . Функция возвращает указатель на QTcpSocket в QAbstractSocket :: ConnectedState, который вы можете использовать для связи с клиентом.

Если возникает ошибка, serverError () возвращает тип ошибки, и errorString () может быть вызвана для получения понятного человеку описания того, что произошло.

При прослушивании соединений адрес и порт, на которых сервер прослушивает, доступны как serverAddress () и serverPort ().

Вызов close () заставляет QTcpServer прекращать прослушивание входящих соединений.

Хотя QTcpServer в основном предназначен для использования с циклом событий, его можно использовать и без него. В этом случае вы должны использовать waitForNewConnection (), который блокирует, пока соединение не станет доступным или не истечет время ожидания.

1.2.2. ДОКУМЕНТАЦИЯ ПО ФУНКЦИЯМ

QTcpServer :: QTcpServer (QObject * parent = nullptr)

Создает объект QTcpServer.

parent передается конструктору QObject .

[signal]void QTcpServer :: acceptError (QAbstractSocket :: SocketError socketError)

Этот сигнал испускается, когда принятие нового соединения приводит к ошибке. Параметр socketError описывает тип возникшей ошибки.

[signal]void QTcpServer :: newConnection ()

Этот сигнал испускается каждый раз, когда доступно новое соединение.

[virtual]QTcpServer :: ~ QTcpServer ()

Уничтожает объект QTcpServer. Если сервер прослушивает соединения, сокет автоматически закрывается.

Любые клиентские QTcpSocket-ы, которые все еще подключены, должны либо отключиться, либо быть восстановлены перед удалением сервера.

[protected]void QTcpServer :: addPendingConnection (сокет QTcpSocket *)

Эта функция вызывается с помощью QTcpServer ::comingConnection (), чтобы добавить сокет в список ожидающих входящих соединений.

void QTcpServer :: close ()

Закрывает сервер. Сервер больше не будет прослушивать входящие соединения.

QString QTcpServer :: errorString () const

Возвращает удобочитаемое описание последней произошедшей ошибки.

[virtual]bool QTcpServer :: hasPendingConnections () const

Возвращает, trueесли сервер имеет ожидающее соединение; в противном случае возвращается false.

[virtual protected]аннулируются QTcpServer :: incomingConnection (qintptr socketDescriptor)

Эта виртуальная функция вызывается QTcpServer, когда доступно новое соединение. SocketDescriptor аргумент является родным дескриптором сокета для принятого соединения.

Базовая реализация создает QTcpSocket, устанавливает дескриптор сокета и затем сохраняет QTcpSocket во внутреннем списке ожидающих соединений. Наконец newConnection () испускается.

Переопределите эту функцию, чтобы изменить поведение сервера, когда соединение доступно.

Если этот сервер использует QNetworkProxy, то socketDescriptor может не использоваться с собственными функциями сокетов и должен использоваться только с QTcpSocket :: setSocketDescriptor ().

Примечание. Если при повторной реализации этого метода создается другой сокет, его необходимо добавить в механизм Pending Connections, вызвав addPendingConnection ().

Примечание. Если вы хотите обработать входящее соединение как новый объект QTcpSocket в другом потоке, вы должны передать socketDescriptor другому потоку и создать там объект QTcpSocket и использовать его метод setSocketDescriptor ().

bool QTcpServer :: isListening () const

Возвращает, trueесли сервер в настоящее время прослушивает входящие соединения; в противном случае возвращается false.

bool QTcpServer :: listen (const QHostAddress & address = QHostAddress :: Any, quint16 port = 0)

Сообщает серверу прослушивать входящие соединения по адресу адреса и порту порта. Если порт равен 0, порт выбирается автоматически. Если address - QHostAddress :: Any , сервер будет прослушивать все сетевые интерфейсы.

Возвращает truев случае успеха; в противном случае возвращается false.

int QTcpServer :: maxPendingConnections () const

Возвращает максимальное количество ожидающих принятых соединений. По умолчанию 30.

[virtual]QTcpSocket * QTcpServer :: nextPendingConnection ()

Возвращает следующее ожидающее соединение как подключенный объект QTcpSocket .

Сокет создается как дочерний элемент сервера, что означает, что он автоматически удаляется при уничтожении объекта QTcpServer. По-прежнему рекомендуется удалять объект явно, когда вы закончите с ним, чтобы не тратить память.

nullptr возвращается, если эта функция вызывается, когда нет ожидающих соединений.

Примечание: Возвращенный QTcpSocket объект не может быть использован из другого потока. Если вы хотите использовать входящее соединение из другого потока, вам необходимо переопределить значение входящего соединения ().

void QTcpServer :: pauseAccepting ()

Паузы принятия новых подключений. Подключения в очереди останутся в очереди.

QNetworkProxy QTcpServer :: proxy () const

Возвращает сетевой прокси для этого сокета. По умолчанию используется QNetworkProxy :: DefaultProxy .

void QTcpServer :: resumeAccepting ()

Возобновляет прием новых соединений.

QHostAddress QTcpServer :: serverAddress () const

Возвращает адрес сервера, если сервер прослушивает соединения; в противном случае возвращает QHostAddress :: Null.

QAbstractSocket :: SocketError QTcpServer :: serverError () const

Возвращает код ошибки для последней возникшей ошибки.

quint16 QTcpServer :: serverPort () const

Возвращает порт сервера, если сервер прослушивает соединения; в противном случае возвращает 0.

void QTcpServer :: setMaxPendingConnections (int numConnections)

Устанавливает максимальное количество ожидающих принятых соединений для numConnections . QTcpServer будет принимать не более numConnections входящих соединений до вызова nextPendingConnection (). По умолчанию ограничение составляет 30 ожидающих подключений.

Клиенты по-прежнему могут подключаться после того, как сервер достиг максимального числа ожидающих подключений (т. Е. QTcpSocket все еще может излучать сигнал connected ()). QTcpServer перестанет принимать новые соединения, но операционная система все еще может держать их в очереди.

void QTcpServer :: setProxy (const QNetworkProxy & networkProxy)

Устанавливает явный сетевой прокси для этого сокета как networkProxy.

Чтобы отключить использование прокси для этого сокета, используйте тип прокси QNetworkProxy :: NoProxy :

bool QTcpServer :: setSocketDescriptor (qintptr socketDescriptor)

Устанавливает дескриптор сокета, который этот сервер должен использовать при прослушивании входящих соединений с socketDescriptor . Возвращает, trueесли сокет установлен успешно; в противном случае возвращается false.

Предполагается, что сокет находится в состоянии прослушивания.

qintptr QTcpServer :: socketDescriptor () const

Возвращает собственный дескриптор сокета, который сервер использует для прослушивания входящих инструкций, или -1, если сервер не прослушивает.

Если сервер использует QNetworkProxy, возвращенный дескриптор может не использоваться с собственными функциями сокетов.

BOOL QTcpServer :: waitForNewConnection (INT Mc = 0, BOOL * timedOut = nullptr)

Ожидание не более миллисекунды мсек или пока не будет доступно входящее соединение. Возвращает, trueесли соединение доступно; в противном случае возвращается false. Если тайм-аут операции и timedOut нет nullptr, * timedOut будет установлен в true.

Это блокирующий вызов функции. Его использование игнорируется в однопоточном приложении с графическим интерфейсом, так как все приложение перестает отвечать до тех пор, пока функция не вернется. waitForNewConnection () в основном полезна, когда нет доступного цикла событий.

1.3. ПРИМЕР ЛАБОРАТОРНЫХ ЗАДАНИЙ С МЕТОДИЧЕСКИМИ УКАЗАНИЯМИ ПО ИХ ВЫПОЛНЕНИЮ

Задание № 1

Выполнение лабораторной работы

1. Запускаем Qt Creator, выбираем пункт меню Файл –> Новый файл или проект. В открывшемся окне Приложение –> Приложение Qt Widgets, далее кнопка Выбрать (рис. 1).

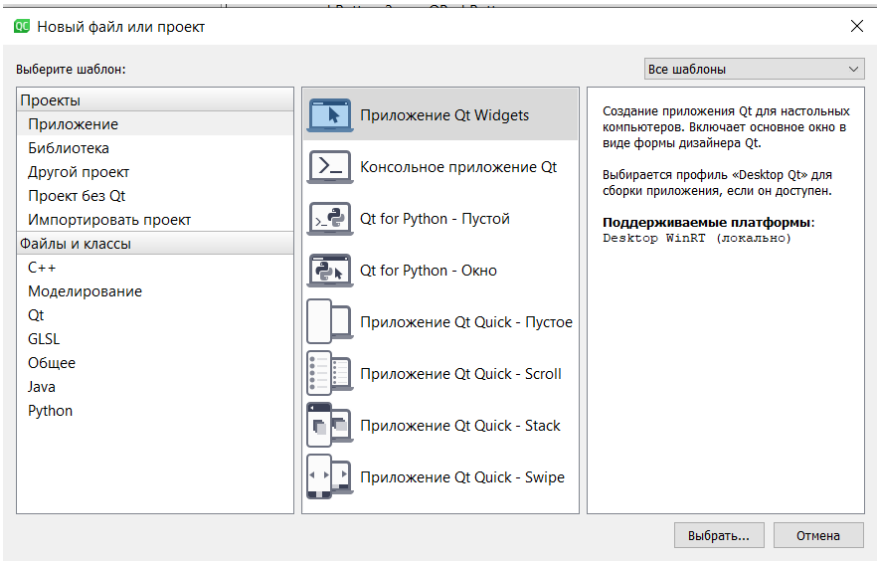


Рис. 1. Создание нового проекта

2. Задаём имя и проекта и путь к нему (рис. 2).

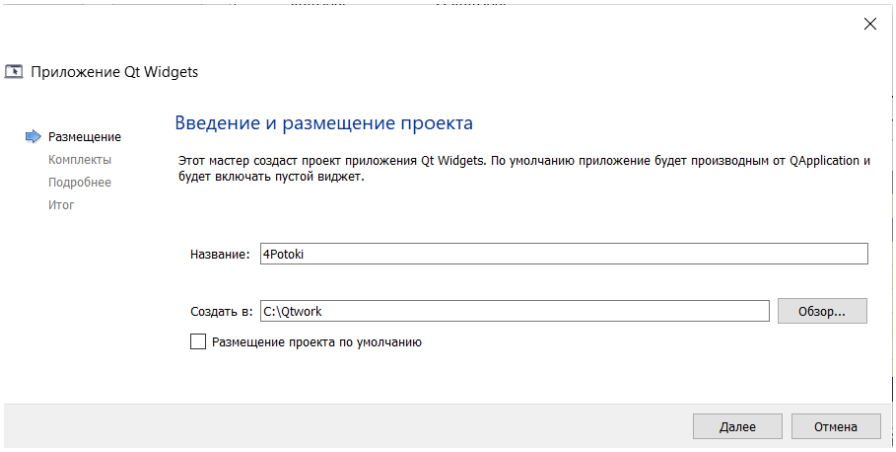


Рис. 2. Создание нового проекта

3. Выбираем комплекты (рис. 3):

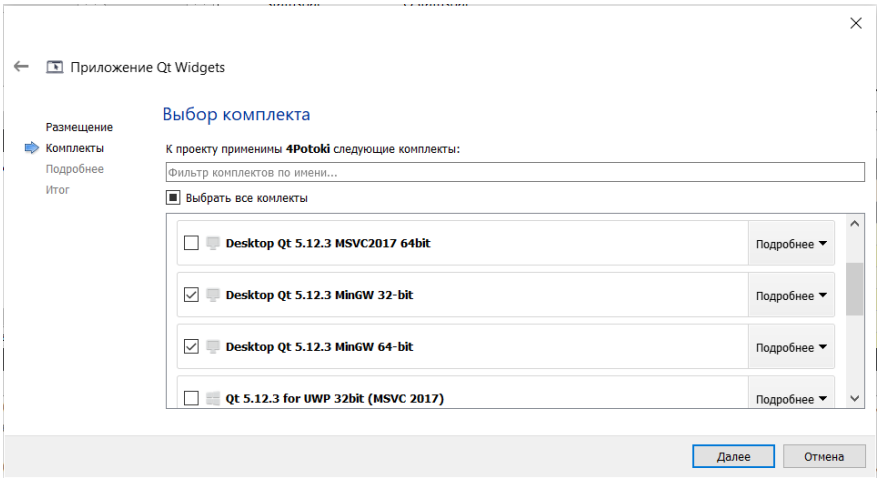


Рис. 3. Создание нового проекта

4. В качестве базового класса задаём QMainWindow (рис. 4).\

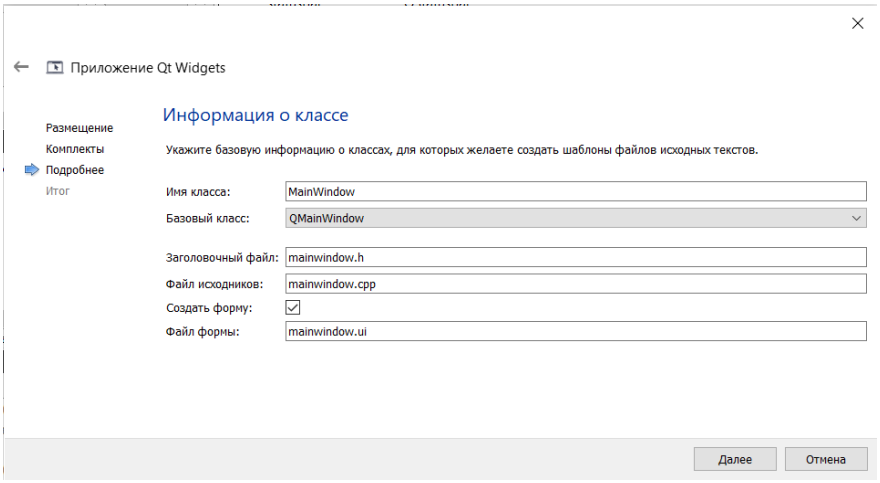


Рис. 4. Создание нового проекта

5. Завершаем создание проекта (рис. 5).

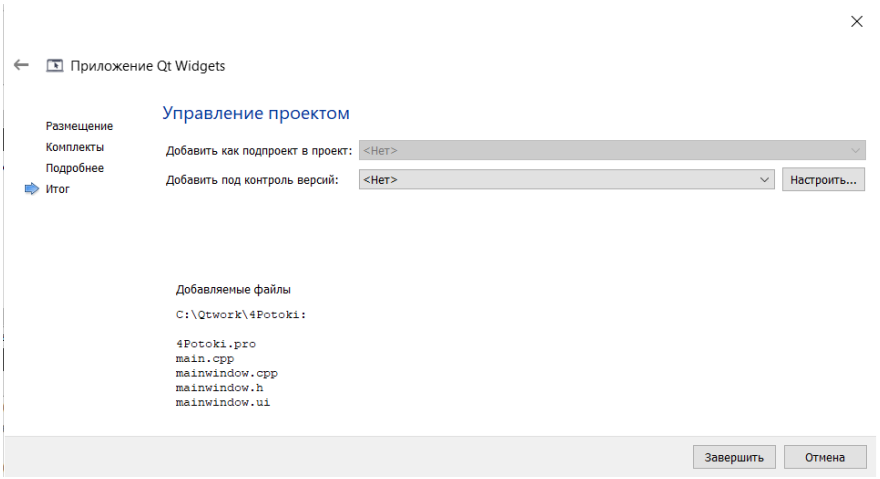


Рис. 5. Создание нового проекта

6. Создали консольное приложение serverHardware.
7. В файле serverHardware.pro подключили сборки sql и network (рис. 6).

```
1 QT -= gui
2 QT += network sql
3
4 CONFIG += c++11 console
5 CONFIG -= app_bundle
```

Рис. 6. Листинг файла serverHardware.pro (подключение сборок sql и network)

8. Для подключения к БД и реализации обмена с клиентом создали класс-одиночку DatabaseAccessor. В заголовочном файле databaseaccessor.h подключили библиотеки <QtSql> и <QThreadPool>, объявили следующие методы и переменные (рис. 7):

а) закрытый конструктор;

б) закрытую переменную QSqlDatabase db;

в) открытый статичный метод getInstance() (возвращает ссылку на объект-одиночку);

г) открытые статичные переменные QString dbHost (название хоста), QString dbName (название базы данных), QString dbUser (имя владельца БД), QString dbPass (пароль для доступа к БД);

д) перечисляемый тип Request, состоящий из идентификаторов: Authorization (идентификатор авторизации, приравнивается к 1), InsertItem (идентификатор добавления какого-либо значения в БД), UpdateItem (идентификатор обновления какого-либо значения в БД), DeleteItem (идентификатор удаления какого-либо значения из БД);

е) открытый слот обмена данными setbuff_in(QJsonDocument str_in).

```
#ifndef DATABASEACCESSOR_H
#define DATABASEACCESSOR_H

#include <QObject>
#include <QtSql>
#include <QThreadPool>

class DatabaseAccessor : public QObject
{
    Q_OBJECT
public:

    static DatabaseAccessor* getInstance();
    static QString dbHost;
    static QString dbName;
    static QString dbUser;
    static QString dbPass;
    typedef enum {
        Authorization = 1,
        InsertItem,
        UpdateItem,
        DeleteItem,
        }Request;
private:
    DatabaseAccessor();

//    DatabaseAccessor& operator=(const DatabaseAccessor& );
    QSqlDatabase db;


signals:

public slots:


    void setbuff_in(QJsonDocument str_in);
};

#endif // DATABASEACCESSOR_H
```

Рис. 7. Листинг файла databaseaccessor.h

9. Создали класс задач MyTask. В заголовочном файле mytask.h подключили databaseaccessor.h, а также библиотеки (рис. 8):

а) <QDebug>;

б) <QRunnable>;

в) <QByteArray>;

г) <QJsonObject>;

д) <QJsonDocument>;

е) <QJsonParseError>;

ж) <QFile>;
з) <QtSql>.

```
#ifndef MYTASK_H
#define MYTASK_H
#include <QDebug>
#include <QObject>
#include <QRunnable>
#include <QByteArray>
#include <QJsonObject>
#include <QJsonDocument>
#include <QJsonParseError>
#include <QFile>
#include <QtSql>
#include "databaseaccessor.h"
```

Рис. 8. Листинг файла mytask.h (подключение библиотек)

10. В заголовочном файле mytask.h объявили следующие методы и переменные (рис. 9):

а) открытый метод чтения данных get_QByteArray(QJsonDocument buff);

б) открытый метод идентификации данных get_id(QString id_in);

в) закрытые переменные QJsonDocument buff_in, QSqlDatabase db_in, QString id;

г) сигналы Result(QJsonDocument), Resultarr(QString), setbuff_in(QJsonDocument);

д) защищённый метод запуска экземпляра класса run();

е) открытый слот получения данных из БД getResultQuery(QJsonObject).

```
class MyTask : public QObject, public QRunnable
{
    Q_OBJECT
public:
    MyTask(QObject * parent = nullptr);
    void get_QByteArray(QJsonDocument buff);
    void get_id(QString id_in);

private:
    QJsonDocument buff_in;
    QSqlDatabase db_in;
    QString id;

signals:
    void Result(QJsonDocument);
    void Resultarr(QString);

   void setbuff_in(QJsonDocument);

protected:
    void run();
public slots:
    void getResultQuery(QJsonObject);
   // void getResultQuery(QString);

};

#endif // MYTASK_H
```

Рис. 9. Листинг файла mytask.h (объявление методов и переменных)

11. В исходнике mytask.cpp реализовали объявленные методы и переменные. Связали сигнал &MyTask::setbuff_in со слотом &DatabaseAccessor::setbuff_in (рис. 10).

```cpp
#include "mytask.h"

MyTask::MyTask(QObject * parent)
{
    connect(this,&MyTask::setbuff_in,DatabaseAccessor::getInstance(),&DatabaseAccessor::setbuff_in);
}

void MyTask::get_QByteArray(QJsonDocument buff)
{
    buff_in = buff;
}


void MyTask::get_id(QString id_in)
{
    id = id_in;
}
void MyTask::run()
{
    qDebug()<<"yes";
    emit setbuff_in(buff_in);

}
void MyTask::getResultQuery(QJsonObject res)
{
    QJsonDocument json;
    json.setObject(res);
   emit Result(json);
}

```

Рис. 10. Листинг файла mytask.cpp

12. В исходнике databaseaccessor.cpp реализовали объявленные методы и переменные. С помощью функции QMetaObject::invokeMethod связали методы DatabaseAccessor::setbuff_in и MyTask::getResultQuery (рис. 11, 12, 13).

```cpp
#include "databaseaccessor.h"


QString DatabaseAccessor::dbHost;
QString DatabaseAccessor::dbName;
QString DatabaseAccessor::dbUser;
QString DatabaseAccessor::dbPass;
DatabaseAccessor::DatabaseAccessor()
{

    db = QSqlDatabase::addDatabase("QPSQL");
    db.setHostName(dbHost);
    db.setDatabaseName(dbName);
    db.setUserName(dbUser);
    db.setPassword(dbPass);
    if (db.open())
    {
        qDebug("connected to database");
    }
    else
    {
        qDebug("Error occured in connection to database");
    }
}

DatabaseAccessor* DatabaseAccessor::getInstance()
{
    static DatabaseAccessor instance;
    return &instance;
}
```

Рис. 11. Листинг файла databaseaccessor.cpp

```
void DatabaseAccessor::setbuff_in(QJsonDocument str_in)
{
    QSqlQuery query(db);

    int type = str_in.object().value("type").toInt();

    QJsonObject obj = str_in.object().value("main").toObject();
    qDebug()<<obj;
    switch (type) {
    case Authorization:
    {

        QString login = obj.value("login").toString();
        QString pass = obj.value("password").toString();
        query.exec("SELECT id,login, pass FROM public.login where login ='"+login+"' and password='"+pass+"';");

       if(query.first())
       {

           QString idQuery=query.value(query.record().indexOf("id")).toString();
         //   QString passQuery=query.value(query.record().indexOf("pass")).toString();
           QJsonObject obj;
           obj.insert("type",Authorization);
           QJsonObject obj1;
           obj1.insert("result","yes");
           obj1.insert("idUser",idQuery);
           obj.insert("main",obj1);
           qDebug()<<obj1;
           QMetaObject::invokeMethod(sender(), "getResultQuery", Qt::QueuedConnection, Q_ARG(QJsonObject, obj));
           break;
       }
       QJsonObject obj;
       obj.insert("type",Authorization);
       QJsonObject obj1;
       obj1.insert("result","no");
       obj.insert("main",obj1);
       QJsonDocument docRes;
       docRes.setObject(obj);
       QMetaObject::invokeMethod(sender(), "getResultQuery", Qt::QueuedConnection, Q_ARG(QJsonObject, obj));
       qDebug()<<docRes;
       break;
```

Рис. 12. Листинг файла databaseaccessor.cpp (продолжение)

```
        }
        case InsertItem:
        {
            break;
        }
        case DeleteItem:
        {
            break;
        }
        case UpdateItem:
        {
            break;
        }
        }

}
```

Рис. 13. Листинг файла databaseaccessor.cpp (окончание)

13. Создали класс MyClient. В заголовочном файле myclient.h подключили mytask.h, а также библиотеки <QTcpSocket>, <QDebug>, <QThreadPool>. Объявили следующие методы и переменные (рис. 14):

а) открытый метод для создания сокета SetSocket(int Descriptor);

б) сигнал finished();

в) открытые слоты connected(), disconnected(), readyRead(), TaskResult(QJsonDocument);

г) закрытую переменную сокета socket.

```
#ifndef MYCLIENT_H
#define MYCLIENT_H

#include <QObject>
#include <QTcpSocket>
#include <QDebug>
#include <QThreadPool>
#include "mytask.h"

class Myclient : public QObject
{
    Q_OBJECT
public:

    explicit Myclient(QObject *parent = nullptr);
    void SetSocket(int Descriptor);
   // static QJsonDocument formJSON(RequestDriver type, QJsonValue val);
private:
    //QThreadPool* pool;

signals:
    void finished();
public slots:
    void connected();
    void disconnected();
    void readyRead();
    void TaskResult(QJsonDocument);
   // void UpadateResult(QString);
private:
    QTcpSocket * socket;
};

#endif // MYCLIENT_H
```

Рис. 14. Листинг файла myclient.h

14. В исходнике myclient.cpp реализовали объявленные методы и переменные. Связали сигнал MyTask::Result(QJsonDocument) со слотом MyClient::TaskResult(QJsonDocument) (рис. 15).

```
#include "myclient.h"

Myclient::Myclient(QObject *parent) : QObject(parent)
{
    QThreadPool::globalInstance()->setMaxThreadCount(4);
}
void Myclient::SetSocket(int Descriptor)
{
    socket = new QTcpSocket(this);
    connect(socket,SIGNAL(connected()),this,SLOT(connected()));
    connect(socket,SIGNAL(disconnected()),this,SLOT(disconnected()));
    connect(socket,SIGNAL(readyRead()),this,SLOT(readyRead()));

    socket->setSocketDescriptor(Descriptor);
    qDebug()<<"client connected " << Descriptor;
}
void Myclient::connected()
{
    qDebug()<<"client connected event";
}
void Myclient::disconnected()
{
    qDebug()<<"client disconnected";
}
void Myclient::readyRead()
{
    MyTask  * mytask = new MyTask(this);
  //  qDebug()<<" "<<QThread::currentThread();
    mytask->get_QByteArray(QJsonDocument::fromJson(socket->readAll()));
  //  mytask->get_id(QString::number(socket->socketDescriptor()));
    mytask->setAutoDelete(false);
    connect(mytask,SIGNAL(Result(QJsonDocument)),this,SLOT(TaskResult(QJsonDocument)),Qt::QueuedConnection);
  //  connect(mytask,SIGNAL(Resultarr(QJsonArray)),this,SLOT(UpadateResult(QString)),Qt::QueuedConnection);
    connect(this,&Myclient::finished,mytask,&MyTask::deleteLater);
    QThreadPool::globalInstance()->start(mytask);
}
void Myclient::TaskResult(QJsonDocument str)
{

    //QByteArray arr;
  //  arr.append(str);
    qDebug()<<"result= "<<str;
    socket->write(str.toJson());
emit finished(); }
```

Рис. 15. Листинг файла client.cpp

10) Создали класс MyServer. В заголовочном файле myserver.h подключили myclient.h, а также библиотеки <QTcpServer>, <QTcpSocket>, <QAbstractSocket>. Объявили следующие методы (рис. 16):

а) открытый метод запуска сервера StartServer();

б) защищённый метод входящего подключения incomingConnection(int handle).

```
#ifndef MYSERVER_H
#define MYSERVER_H

#include <QTcpServer>
#include <QTcpSocket>
#include <QAbstractSocket>
#include "myclient.h"

class MyServer : public QTcpServer
{
    Q_OBJECT
public:
    explicit MyServer(QObject *parent = nullptr);
    void StartServer();
protected:
    void incomingConnection(int handle);
signals:

public slots:
};

#endif // MYSERVER_H
```

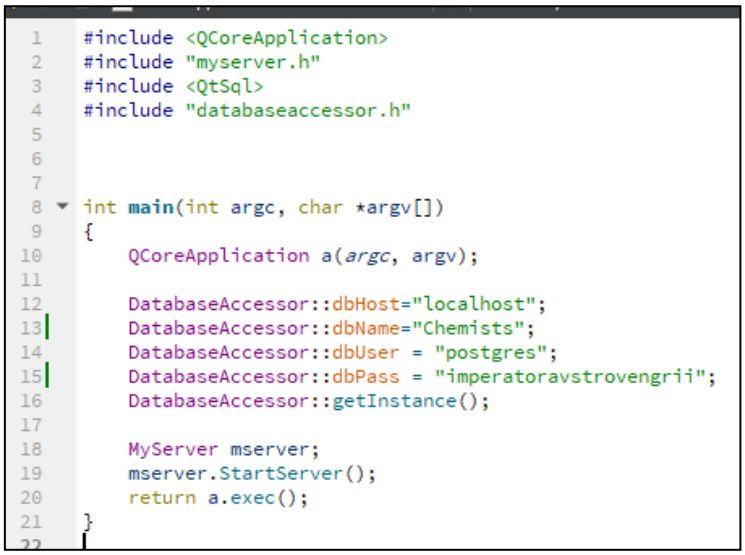
Рис. 16. Листинг файла myserver.h

11) В исходнике myserver.cpp реализовали объявленные методы (рис. 17).

```
#include "myserver.h"

MyServer::MyServer(QObject *parent) : QTcpServer(parent)
{

}
void MyServer::StartServer()
{

    if (listen(QHostAddress::Any,2323))
    {
        qDebug()<<"started";
    }
    else {
        qDebug()<<"not started";
    }


}
void MyServer::incomingConnection(int handle)
{
    Myclient* client = new Myclient(this);
    client->SetSocket(handle);
}
```

Рис. 17. Листинг файла myserver.cpp

12) В файле main.cpp подключили myserver.h, databaseaccessor.h, а также библиотеку <QtSql>. Выполнили подключение к БД посредством класса-одиночки (рис. 18).



```
#include <QCoreApplication>
#include "myserver.h"
#include <QtSql>
#include "databaseaccessor.h"



int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    DatabaseAccessor::dbHost="localhost";
    DatabaseAccessor::dbName="Chemists";
    DatabaseAccessor::dbUser = "postgres";
    DatabaseAccessor::dbPass = "imperatoravstrovengrii";
    DatabaseAccessor::getInstance();

    MyServer mserver;
    mserver.StartServer();
    return a.exec();
}
```

Рис. 18. Листинг файла main.cpp

13) Алгоритм программы serverHardware выглядит следующим образом: когда к серверу подключаются клиенты, образуется пул потоков, в которых выполняются задачи. При передаче запроса клиентом данные в формате JSON-документа поступают в класс задач, откуда они идут в класс-одиночку. Одиночка обрабатывает полученную от клиента информацию и формирует ответ (в виде JSON), который направляется в класс задач, а впоследствии – в класс клиента.

Вывод: мы получили базовые навыки проектирования асинхронного серверного ПО.

1.4. ЛАБОРАТОРНЫЕ ЗАДАНИЯ

Ознакомиться с примером выполнения лабораторного задания. Опираясь на приведённый пример, написать самостоятельно асинхронное серверное программное обеспечение.

1.5. УКАЗАНИЯ ПО ОФОРМЛЕНИЮ ОТЧЕТА

Отчет оформляется в виде пояснительной записки на листах формата А4 (210 х 297 мм). Необходимо дома подготовить заготовку по всей работе. Заготовка должна содержать цель и содержание работы, основные теоретические сведения, все пункты лабораторных заданий, листинги проектов с комментариями. В конце отчета необходимо привести выводы по лабораторной работе и ответы на контрольные вопросы.

1.6. КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНЫМ ЗАДАНИЯМ

1. Что позволяет реализовывать класс QTcp Server?
2. Расскажите об основных функциях (методах) класса QTcp Server.
3. Расскажите о назначении метода startServer.

2. ЛАБОРАТОРНАЯ РАБОТА № 12
ПОЛУЧЕНИЕ БАЗОВЫХ ЗНАНИЙ ПО РАЗРАБОТКЕ ПРОТОКОЛА ОБМЕНА КЛИЕНТ-СЕРВЕРНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. ОБЩИЕ УКАЗАНИЯ

2.1.1. ЦЕЛЬ РАБОТЫ

Познакомиться с основами разработки протокола обмена данными между клиентом и серверным ПО. Применяя полученные знания, создать протокол обмена.

2.1.2. ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Основным содержанием работы является создание простого протокола обмена между клиентом и серверным ПО с использованием возможностей Qt Framework. В ходе работы идёт ознакомление с приведённым примером протокола, производится анализ используемых технологий и методологии. На основе полученных знаний выполняется индивидуальное задание по проектированию протокола обмена.

2.2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.2.1. СЕТЕВОЕ ПРОГРАММИРОВАНИЕ С QT

Модуль Qt Network предлагает классы, которые позволяют вам писать клиенты и серверы TCP / IP. Он предлагает классы низкого уровня, такие как QTcpSocket , QTcpServer и QUdpSocket, которые представляют концепции сети низкого уровня, и классы высокого уровня, такие как QNetworkRequest , QNetworkReply и QNetworkAccessManager, для выполнения сетевых операций с использованием общих протоколов. Он также предлагает классы, такие как QNetworkConfiguration , QNetworkConfigurationManager и QNetworkSession, которые реализуют управление однонаправленным каналом.

Qt Network предоставляет набор API для программирования приложений, использующих TCP / IP. Такие операции, как запросы, файлы cookie и отправка данных по HTTP, обрабатываются различными классами C++.

Чтобы использовать классы Qt Network, добавьте эту директиву в файлы C++:

```
#include < QtNetwork >
```

Рис. 19. Добавление QtNetwork

Чтобы связать с модулем Qt Network, добавьте эту строку в файл проекта:

```
QT += network
```

Рис. 20. Связь с Qt network

2.2.2. КЛАССЫ QT ДЛЯ СЕТЕВОГО ПРОГРАММИРОВАНИЯ

Таблица 1

Список классов Qt для сетевого программирования

Класс	Назначение
QAbstractNetworkCache	Интерфейс для реализации кеша
QNetworkCacheMetaData	Информация о кеше
QHstsPolicy	Указывает, что хост поддерживает строгую политику безопасности транспорта HTTP (HSTS)
QHttp2Configuration	Управляет параметрами и настройками HTTP / 2
QHttpMultiPart	Напоминает многокомпонентное сообщение MIME для отправки по HTTP
QHttpPart	Используется внутри составного HTTP-сообщения MIME.
QNetworkAccessManager	Позволяет приложению отправлять сетевые запросы и получать ответы
QNetworkCookie	Содержит один сетевой файл cookie
QNetworkCookieJar	Реализует простой jar объектов QNetworkCookie
QNetworkDiskCache	Реализует очень простой дисковый кеш
QNetworkReply	Содержит данные и заголовки для запроса, отправленного с QNetworkAccessManager
QNetworkRequest	Содержит запрос для отправки с QNetworkAccessManager
QNetworkConfigurationManager	Управляет конфигурациями сети, предоставляемыми системой
QNetworkConfiguration	Абстракция одной или нескольких конфигураций точки доступа
QNetworkSession	Контроль над точками доступа системы и управление сеансами в случаях, когда несколько клиентов получают доступ к одной точке доступа
QHostAddress	IP-адрес
QNetworkDatagram	Данные и метаданные UDP-дейтаграммы
QNetworkAddressEntry	Хранит один IP-адрес, поддерживаемый сетевым интерфейсом, а также связанную с ним маску сети и широковещательный адрес
QNetworkInterface	Список IP-адресов хоста и сетевых интерфейсов
QAbstractSocket	Базовый класс для всех типов сокетов

Продолжение табл. 1

Класс	Назначение
QLocalServer	Локальный сокет-сервер
QLocalSocket	Локальный сокет
QSctpServer	SCTP-сервер
QSctpSocket	SCTP-сокет
QTcpServer	TCP-сервер
QTcpSocket	TCP-сокет
QUdpSocket	UDP-сокет
QDtls	Этот класс обеспечивает шифрование для сокетов UDP
QDtlsClientVerifier	Этот класс реализует генерацию и проверку cookie DTLS на стороне сервера.
QDtlsClientVerifier :: GeneratorParameters	Этот класс определяет параметры для генератора файлов cookie DTLS
QOcspResponse	Этот класс представляет ответ протокола состояния сертификата в сети
QSslCertificate	Удобный API для сертификата X509
QSslCertificateExtension	API для доступа к расширениям сертификата X509
QSslCipher	Представляет криптографический шифр SSL
QSslConfiguration	Содержит конфигурацию и состояние соединения SSL
QSslDiffieHellmanParameters	Интерфейс для параметров Диффи-Хеллмана для серверов
QSslEllipticCurve	Представляет эллиптическую кривую для использования алгоритмами шифрования эллиптической кривой
QSslError	Ошибка SSL
QSslKey	Интерфейс для закрытых и открытых ключей
QSslPreSharedKeyAuthenticator	Данные аутентификации для наборов шифров с предварительными общими ключами (PSK)
QSslSocket	Зашифрованный сокет SSL для клиентов и серверов
QAuthenticator	Объект аутентификации
QDnsDomainNameRecord	Хранит информацию о записи доменного имени
QDnsHostAddressRecord	Хранит информацию о записи адреса хоста
QDnsLookup	Представляет поиск DNS

Окончание табл. 1

Класс	Назначение
QDnsMailExchangeRecord	Хранит информацию о DNS-записи MX
QDnsServiceRecord	Хранит информацию о записи DNS SRV
QDnsTextRecord	Хранит информацию о записи TXT DNS
QHostInfo	Статические функции для поиска имени хоста
QNetworkProxy	Прокси сетевого уровня
QNetworkProxyFactory	Выбор прокси
QNetworkProxyQuery	Используется для запроса настроек прокси для сокета

2.2.3. СЕТЕВЫЕ ОПЕРАЦИИ ВЫСОКОГО УРОВНЯ ДЛЯ HTTP И FTP

API доступа к сети представляет собой набор классов для выполнения общих сетевых операций. API обеспечивает уровень абстракции для определенных используемых операций и протоколов (например, получение и публикация данных по HTTP) и предоставляет только классы, функции и сигналы для общих понятий или понятий высокого уровня.

Сетевые запросы представлены классом QNetworkRequest , который также действует как общий контейнер для информации, связанной с запросом, такой как любая информация заголовка и используемое шифрование. URL, указанный при создании объекта запроса, определяет протокол, используемый для запроса. В настоящее время для загрузки и скачивания поддерживаются HTTP, FTP и локальные URL-адреса файлов.

Координация сетевых операций выполняется классом QNetworkAccessManager . После того, как запрос был создан, этот класс используется для его отправки и выдачи сигналов, чтобы сообщить о его ходе. Менеджер также координирует использование файлов cookie для хранения данных на клиенте, запросов на аутентификацию и использования прокси-серверов.

Ответы на сетевые запросы представлены классом QNetworkReply ; они создаются QNetworkAccessManager при отправке запроса. Сигналы, предоставляемые QNetworkReply, могут использоваться для индивидуального мониторинга каждого ответа, или разработчики могут использовать вместо этого сигналы менеджера и отбрасывать ссылки на ответы. Поскольку QNetworkReply является подклассом QIODevice , ответы могут обрабатываться синхронно или асинхронно; т.е. как блокирующие или неблокирующие операции.

Каждое приложение или библиотека может создать один или несколько экземпляров QNetworkAccessManager для обработки сетевого взаимодействия.

2.2.4. ИСПОЛЬЗОВАНИЕ TCP С QTCPSOCKET И QTCPSERVER

TCP (Transmission Control Protocol) - это сетевой протокол низкого уровня, используемый большинством интернет-протоколов, включая HTTP и FTP, для передачи данных. Это надежный, ориентированный на поток, ориентиро-

ванный на соединение транспортный протокол. Это особенно хорошо подходит для непрерывной передачи данных.

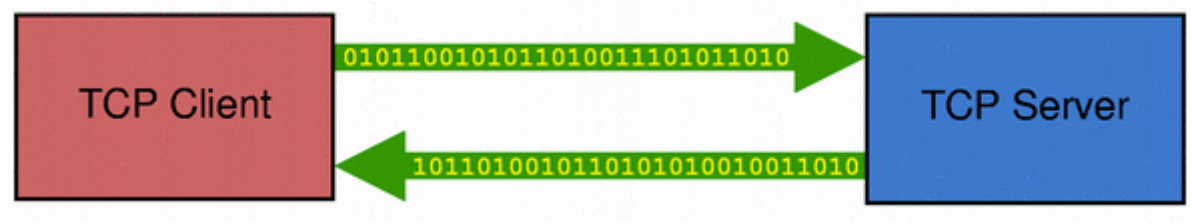


Рис. 21. Протокол TCP

Класс QTcpSocket предоставляет интерфейс для TCP. Вы можете использовать QTcpSocket для реализации стандартных сетевых протоколов, таких как POP3, SMTP и NNTP, а также пользовательских протоколов.

Перед началом любой передачи данных необходимо установить TCP-соединение с удаленным хостом и портом. Как только соединение установлено, IP-адрес и порт узла доступны через QTcpSocket :: peerAddress () и QTcpSocket :: peerPort (). В любое время одноранговый узел может закрыть соединение, и тогда передача данных немедленно прекратится.

QTcpSocket работает асинхронно и излучает сигналы для сообщения об изменениях статуса и ошибках, как QNetworkAccessManager . Он основан на цикле событий для обнаружения входящих данных и автоматической очистки исходящих данных. Вы можете записывать данные в сокет, используя QTcpSocket :: write (), и читать данные, используя QTcpSocket :: read (). QTcpSocket представляет два независимых потока данных: один для чтения и один для записи.

Поскольку QTcpSocket наследует QIODevice , вы можете использовать его с QTextStream и QDataStream . При чтении из QTcpSocket вы должны убедиться, что достаточно данных доступно, предварительно вызвав QTcpSocket :: bytesAvailable ().

Если вам нужно обрабатывать входящие TCP-соединения (например, в серверном приложении), используйте класс QTcpServer . Вызовите QTcpServer :: listen () для настройки сервера и подключитесь к сигналу QTcpServer :: newConnection (), который посылается один раз для каждого подключающегося клиента. В вашем слоте вызовите QTcpServer :: nextPendingConnection (), чтобы принять соединение и использовать возвращенный QTcpSocket для связи с клиентом.

Хотя большинство его функций работают асинхронно, возможно использовать QTcpSocket синхронно (т.е. блокировать). Чтобы получить блокирующее поведение, вызовите функции QTcpSocket 's waitFor ... (); они приостанавливают вызывающий поток до тех пор, пока не будет издан сигнал. Например, после вызова неблокирующей функции QTcpSocket :: connectToHost () вызовите QTcpSocket :: waitForConnected (), чтобы заблокировать поток до тех пор, пока не будет получен сигнал connected () .

Синхронные сокеты часто приводят к коду с более простым потоком управления. Основным недостатком подхода waitFor ... () является то, что события не будут обрабатываться, пока функция waitFor ... () блокируется. При использовании в потоке графического интерфейса это может привести к зависанию пользовательского интерфейса приложения. По этой причине мы рекомендуем использовать синхронные сокеты только в потоках без GUI. При синхронном использовании QTcpSocket не требует цикла обработки событий.

2.2.5. ИСПОЛЬЗОВАНИЕ UDP С QUDPSOCKET

UDP (User Datagram Protocol) - это легкий, ненадежный, ориентированный на дейтаграммы протокол без установления соединения. Его можно использовать, когда надежность не важна. Например, сервер, который сообщает время суток, может выбрать UDP. Если датаграмма с указанием времени суток потеряна, клиент может просто сделать еще один запрос.

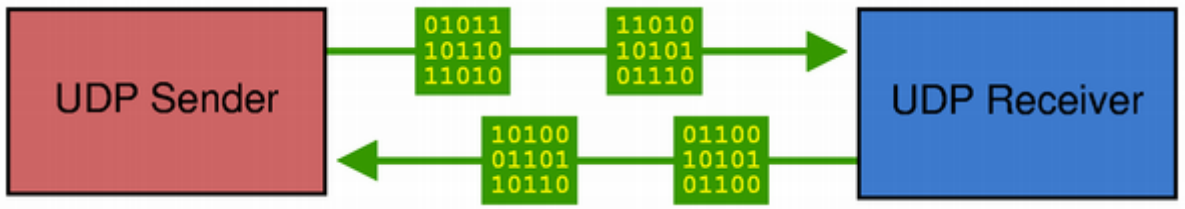


Рис. 22. Протокол UDP

Класс QUdpSocket позволяет отправлять и получать дейтаграммы UDP. Он наследует QAbstractSocket и поэтому разделяет большую часть интерфейса QTcpSocket . Основное отличие состоит в том, что QUdpSocket передает данные в виде дейтаграмм, а не в виде непрерывного потока данных. Вкратце, дейтаграмма - это пакет данных ограниченного размера (обычно меньше 512 байт), содержащий IP-адрес и порт отправителя и получателя дейтаграммы в дополнение к передаваемым данным.

QUdpSocket поддерживает трансляцию IPv4. Широковещание часто используется для реализации сетевых протоколов обнаружения, например, для определения, какой хост в сети имеет больше всего свободного места на жестком диске. Один хост передает в сеть дейтаграмму, которую получают все остальные хосты. Каждый хост, который получает запрос, затем отправляет ответ отправителю с его текущим объемом свободного дискового пространства. Отправитель ожидает, пока он не получит ответы от всех хостов, и затем может выбрать сервер с наибольшим количеством свободного места для хранения данных. Чтобы передать дейтаграмму, просто отправьте ее на специальный адрес QHostAddress :: Broadcast (255.255.255.255) или на широковещательный адрес вашей локальной сети.

QUdpSocket :: bind () подготавливает сокет для приема входящих дейтаграмм, подобно QTcpServer :: listen () для TCP-серверов. Всякий раз, когда приходит одна или несколько дейтаграмм, QUdpSocket испускает сигнал readyRead () . Вызовите QUdpSocket :: readDatagram (), чтобы прочитать дейтаграмму.

2.2.6. РАЗРЕШЕНИЕ ИМЕН ХОСТОВ С ИСПОЛЬЗОВАНИЕМ QHOSTINFO

Перед установлением сетевого соединения QTcpSocket и QUdpSocket выполняют поиск имени, переводя имя хоста, к которому вы подключаетесь, в IP-адрес. Эта операция обычно выполняется с использованием протокола DNS (службы доменных имен).

QHostInfo предоставляет статическую функцию, которая позволяет вам выполнять такой поиск самостоятельно. Вызвав QHostInfo :: lookupHost () с именем хоста, указателем QObject и подписью слота, QHostInfo выполнит поиск имени и вызовет данный слот, когда результаты будут готовы. Фактический поиск выполняется в отдельном потоке, использующем собственные методы операционной системы для выполнения поиска имен.

QHostInfo также предоставляет статическую функцию QHostInfo :: fromName (), которая принимает имя хоста в качестве аргумента и возвращает результаты. В этом случае поиск имени выполняется в том же потоке, что и вызывающая сторона. Эта перегрузка полезна для приложений без GUI или для поиска имен в отдельном потоке без GUI. (Вызов этой функции в потоке графического интерфейса пользователя может привести к зависанию вашего пользовательского интерфейса, пока функциональные блоки выполняют поиск.)

2.2.7. ПОДДЕРЖКА СЕТЕВЫХ ПРОКСИ

Сетевое взаимодействие с Qt может осуществляться через прокси, которые направляют или фильтруют сетевой трафик между локальными и удаленными соединениями.

Отдельные прокси представлены классом QNetworkProxy , который используется для описания и настройки соединения с прокси. Поддерживаются типы прокси, которые работают на разных уровнях сетевого взаимодействия, с поддержкой SOCKS 5, позволяющей проксировать сетевой трафик на низком уровне, а прокси HTTP и FTP работают на уровне протокола. Смотрите QNetworkProxy :: ProxyType для получения дополнительной информации.

Прокси может быть включен для каждого сокета или для всех сетевых коммуникаций в приложении. Вновь открытый сокет можно заставить использовать прокси, вызывая его функцию QAbstractSocket :: setProxy () перед подключением. Проксирование в масштабе всего приложения может быть включено для всех последующих соединений сокетов с помощью функции QNetworkProxy :: setApplicationProxy ().

Прокси-фабрики используются для создания политик использования прокси. QNetworkProxyFactory предоставляет прокси на основе запросов для определенных типов прокси. Сами запросы кодируются в объектах QNetworkProxyQuery, которые позволяют выбирать прокси на основе ключевых критериев, таких как назначение прокси (TCP, UDP, TCP-сервер, запрос URL-адреса), локальный порт, удаленный хост и порт, а также протокол в использовать (HTTP, FTP и т. д.).

QNetworkProxyFactory :: proxyForQuery () используется для непосредственного запроса фабрики. Политика всего приложения для прокси может быть реализована путем передачи фабрики в QNetworkProxyFactory :: setApplicationProxyFactory (), а пользовательская политика прокси может быть создана путем создания подкласса QNetworkProxyFactory.

2.2.8. КЛАСС QTCPSERVER

Класс QTcpServer предоставляет сервер на основе TCP.

Этот класс позволяет принимать входящие TCP-соединения. Вы можете указать порт или сделать так, чтобы QTcpServer выбрал его автоматически. Вы можете слушать по определенному адресу или по всем адресам машины.

Вызовите listen (), чтобы сервер прослушивал входящие соединения. Затем сигнал newConnection () отправляется каждый раз, когда клиент подключается к серверу.

Вызовите nextPendingConnection (), чтобы принять ожидающее соединение как подключенный QTcpSocket . Функция возвращает указатель на QTcpSocket в QAbstractSocket :: ConnectedState, который вы можете использовать для связи с клиентом.

Если происходит ошибка, serverError () возвращает тип ошибки, и errorString () может быть вызван для получения понятного для человека описания того, что произошло.

При прослушивании соединений адрес и порт, на которых сервер прослушивает, доступны как serverAddress () и serverPort ().

Вызов close () заставляет QTcpServer прекращать прослушивание входящих соединений.

Хотя QTcpServer в основном предназначен для использования с циклом событий, его можно использовать и без него. В этом случае вы должны использовать waitForNewConnection (), который блокирует, пока соединение не станет доступным или не истечет время ожидания.

Некоторые функции класса:

1. void QTcpServer :: incomingConnection (qintptr socketDescriptor). Эта виртуальная функция вызывается QTcpServer, когда доступно новое соединение. SocketDescriptor аргумент является родным дескриптором сокета для принятого соединения.

Базовая реализация создает QTcpSocket , устанавливает дескриптор сокета и затем сохраняет QTcpSocket во внутреннем списке ожидающих соединений. Наконец newConnection () испускается.

Переопределите эту функцию, чтобы изменить поведение сервера, когда соединение доступно.

2. void QTcpServer :: newConnection (). Этот сигнал испускается каждый раз, когда доступно новое соединение.

3. void QTcpServer :: addPendingConnection. Эта функция вызывается с помощью QTcpServer ::comingConnection (), чтобы добавить сокет в список ожидающих входящих соединений.

2.3. ЛАБОРАТОРНЫЕ ЗАДАНИЯ И МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ИХ ВЫПОЛНЕНИЮ

Начнем с написания клиентского ПО.

1. Создать новый проект с базовым классом QWidget и подключить в pro файле модуль Qt Network (рис. 23)

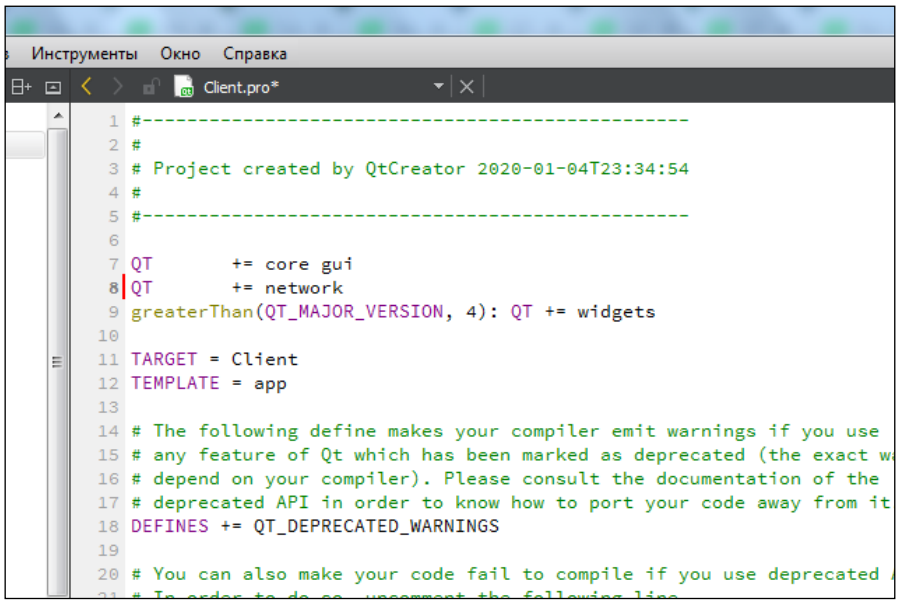


Рис. 23. Созданный проект

2. В режиме «Дизайн» создать следующий интерфейс (рис. 24). Два верхних поля понадобятся для ввода логина и пароля. Левое нижнее поле для ввода данных, отправляемых на сервер. Правое верхнее – для отображения принятых от сервера данных.

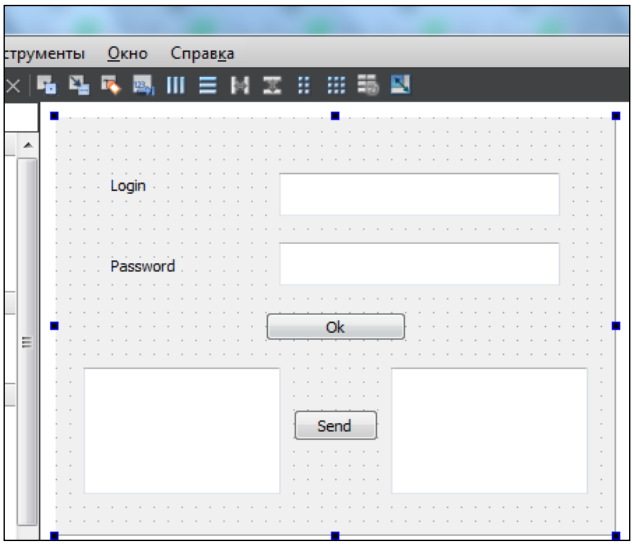


Рис. 24. Интерфейс

3. В файле widget.h объявить функцию, выполняющуюся при нажатии клавиши «Ок», и подключить классы, необходимые для работы с сокетом и json

(рис. 25, строки 5-8, 27). Также необходимо объявить объект класса QTcpSocket (рис. 25, строка 17).

Рис. 25. Листинг файла *widget.h*

4. Реализовать функцию on_buttonOk_clicked() (рис. 26). Здесь создается сокет и его сигналы соединяются с соответствующими слотами, объявленными в файле widget.h. Сигнал connected испускается, когда соединение с сервером успешно установлено, disconnected – разорвано, readyRead – пришли данные с сервера.

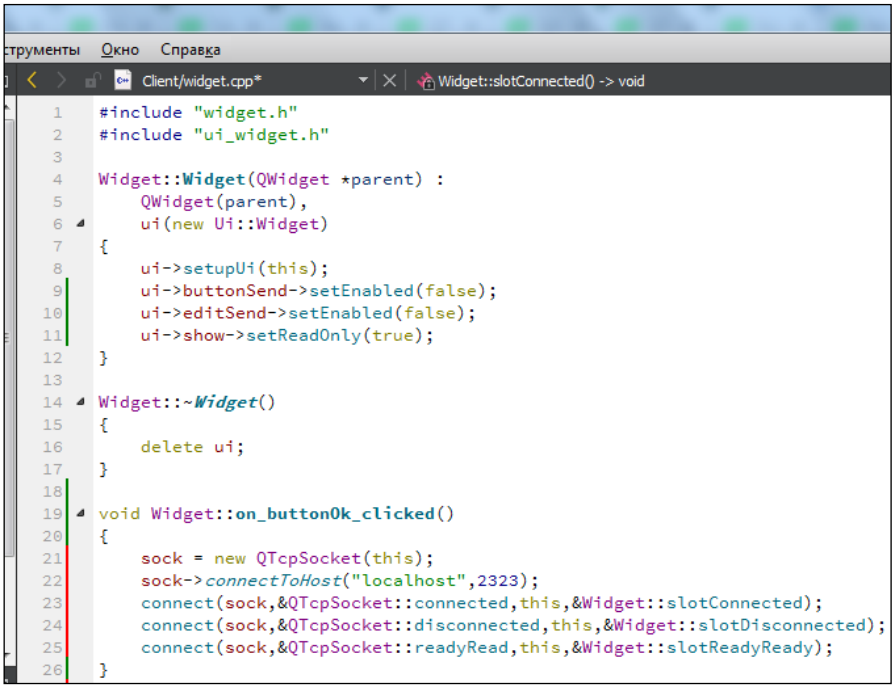


Рис. 26. Реализация функции on_buttonOk_clicked()

5. Реализовать слот slotConnected() (рис. 27). Здесь формируется JSON документ, посредством которого клиент и сервер будут осуществлять обмен данными. С ключом type отправляем тип сообщения – Autorization, которому соответствует цифра 1 в созданном нами перечислении typeMessage (рис. 25, строки 21-24). С ключом main отправляем объект, содержащий основные данные сообщения. В нашем случае это логин и пароль пользователя. Далее JSON документ записывается в сокет. Фактически это означает отправку на сервер.

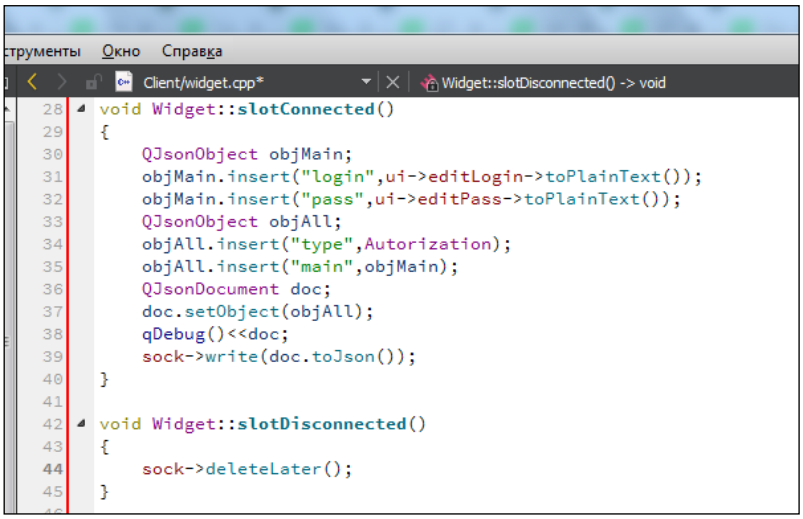


Рис. 27. Реализация слота slotConnected()

На этом этапе необходимо создать сервер.

6. Открыть однопоточный сервер, написанный на прошлой лабораторной работе (рис. 28-29). Добавить в него перечисление typeMessage.

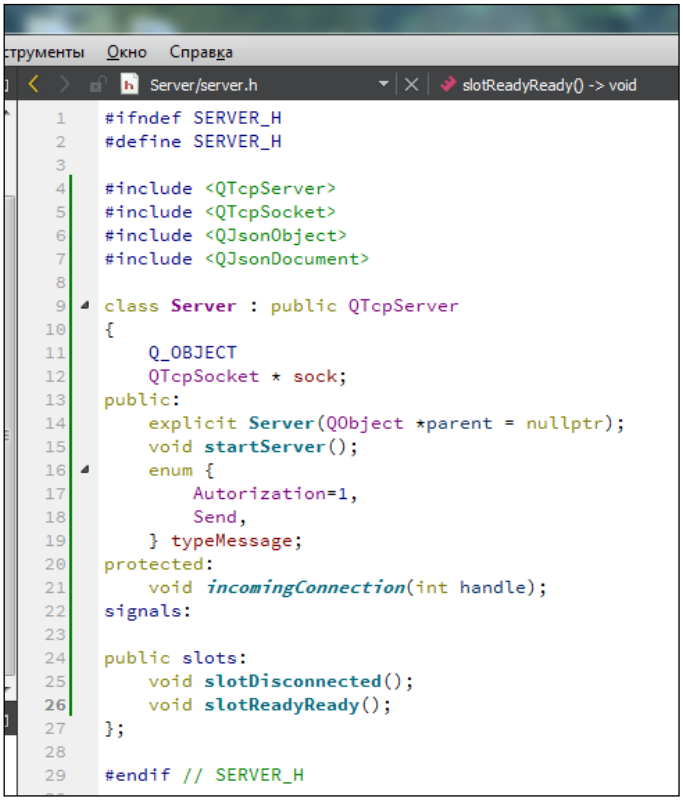


Рис. 28. Листинг файла *server.h*

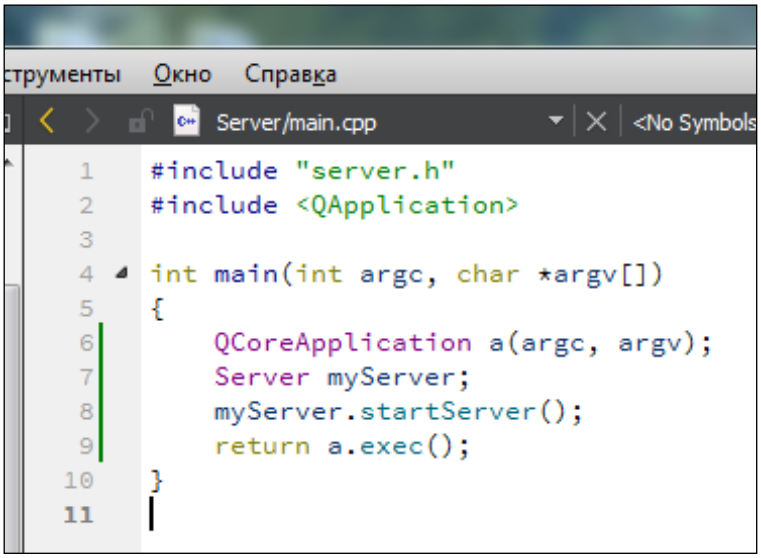


Рис. 29. Листинг файла *main.cpp*

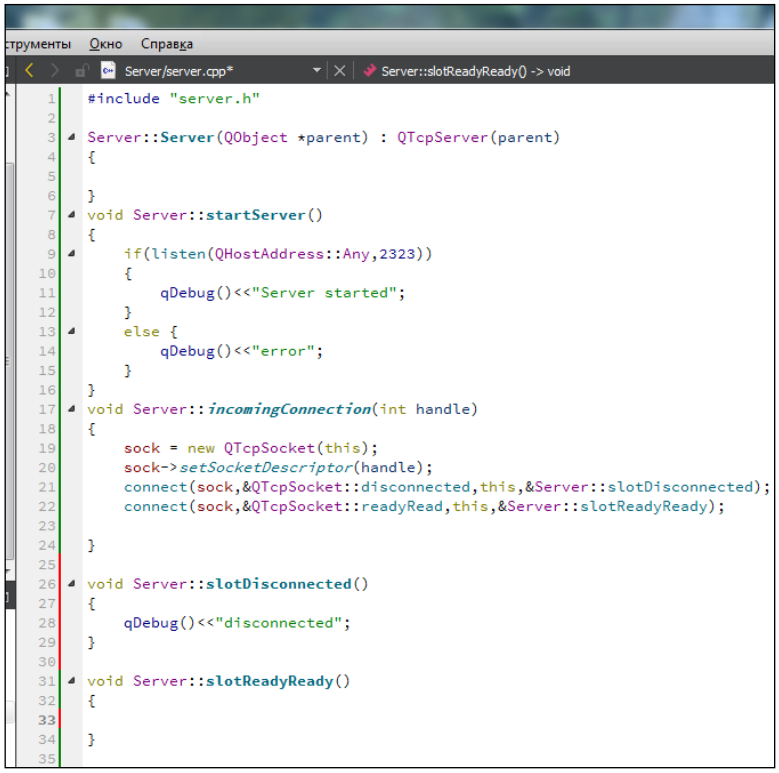


Рис. 30. Листинг файла *server.cpp*

7. Более подробно рассмотрим слот slotReadyRead() (рис. 31). Здесь мы читаем данные из сокета и записываем их в JSON документ. По значению ключа type определяем дальнейшие действия. В нашем случае это Autorization. Мы проверяем, соответствует ли введенный логин и пароль заданному и, в зависимости от результата, отправляем ответ клиенту: yes или no.

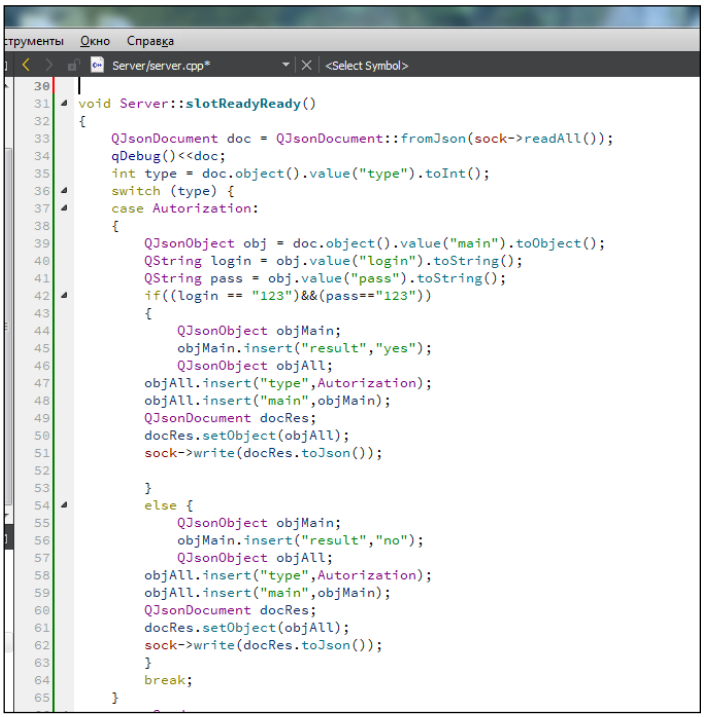
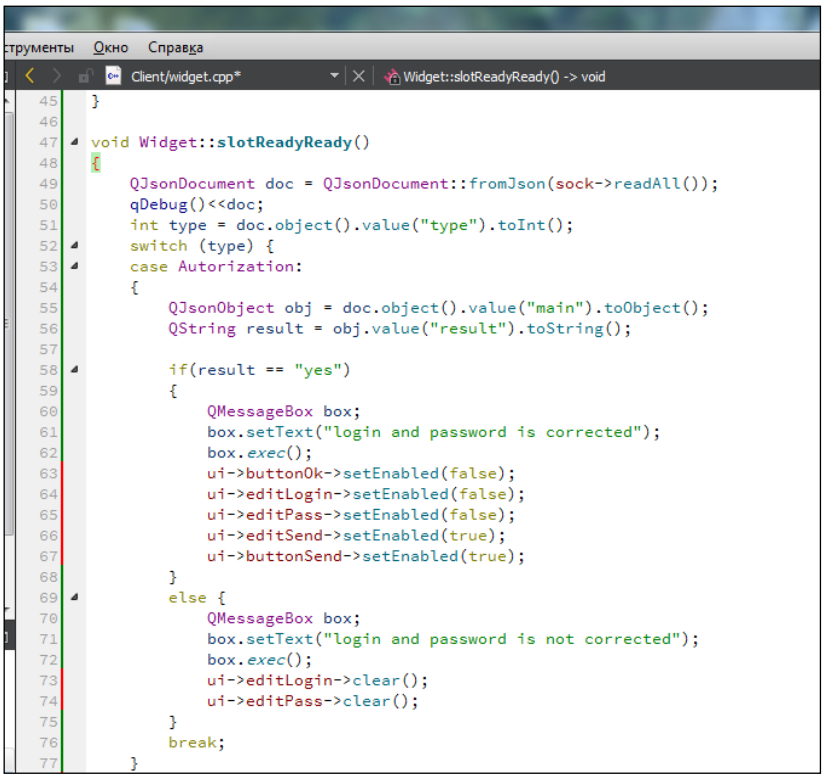


Рис. 31. Листинг файла *server.cpp* со слотом slotReadyRead()

8. Когда клиент получает данные, выполняется слот slotReadyRead() (рис. 32). Здесь также выполняется чтение документа из сокета, определяется тип сообщения. В нашем случае это тип Autorization. Если результат yes, выводим сообщение об успешном прохождении авторизации. Если no – о неуспешном и очищаем поля ввода.




```
45  }
46
47  void Widget::slotReadyReady()
48  {
49      QJsonDocument doc = QJsonDocument::fromJson(sock->readAll());
50      qDebug()<<doc;
51      int type = doc.object().value("type").toInt();
52      switch (type) {
53      case Autorization:
54      {
55          QJsonObject obj = doc.object().value("main").toObject();
56          QString result = obj.value("result").toString();
57
58          if(result == "yes")
59          {
60              QMessageBox box;
61              box.setText("login and password is corrected");
62              box.exec();
63              ui->buttonOk->setEnabled(false);
64              ui->editLogin->setEnabled(false);
65              ui->editPass->setEnabled(false);
66              ui->editSend->setEnabled(true);
67              ui->buttonSend->setEnabled(true);
68          }
69          else {
70              QMessageBox box;
71              box.setText("login and password is not corrected");
72              box.exec();
73              ui->editLogin->clear();
74              ui->editPass->clear();
75          }
76          break;
77      }
```

Рис. 32. Листинг файла widget.cpp

При успешном прохождении авторизации мы можем отправлять сообщения на сервер и получать ответы. Пусть сервер добавляет букву а к каждому сообщения клиента.

9. Объявить и реализовать слот on_buttonSend_clicked(). Здесь мы сразу будем отправлять введенный пользователем текст на сервер (рис. 33).

```
88  void Widget::on_buttonSend_clicked()
89  {
90      QJsonObject objMain;
91      objMain.insert("text",ui->editSend->toPlainText());
92      QJsonObject objAll;
93      objAll.insert("type",Send);
94      objAll.insert("main",objMain);
95      QJsonDocument doc;
96      doc.setObject(objAll);
97      sock->write(doc.toJson());
98      ui->editSend->clear();
99  }
100
```

Рис. 33. Реализация слота on_buttonSend_clicked()

10. Реализовать обработку этого типа сообщений на сервере (рис. 34)

```
    }
    case Send:
    {
        QJsonObject obj = doc.object().value("main").toObject();
        QString text = obj.value("text").toString();
        QJsonObject objMain;
        objMain.insert("text","a"+text);
        QJsonObject objAll;
        objAll.insert("type",Send);
        objAll.insert("main",objMain);
        QJsonDocument docRes;
        docRes.setObject(objAll);
        sock->write(docRes.toJson());
        break;
    }
    }
}
```

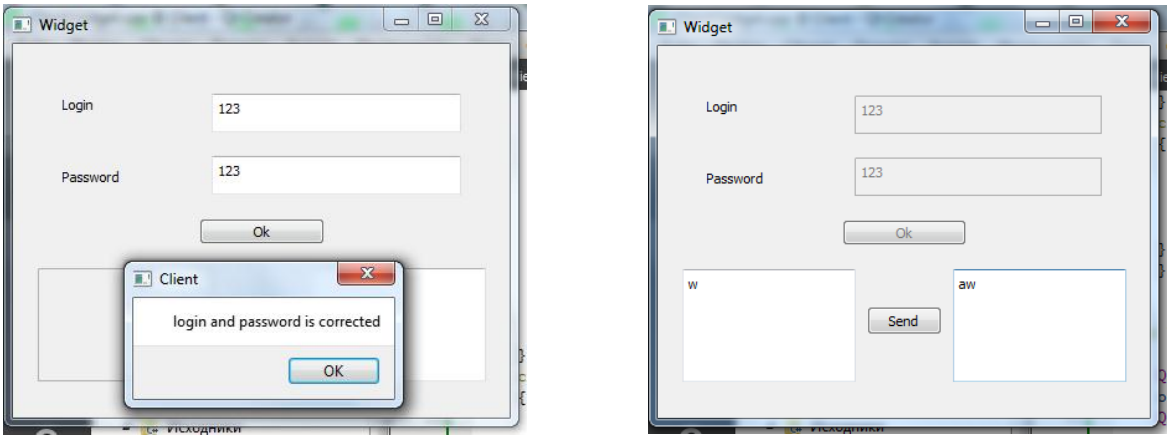
Рис. 34. Реализация обработки сообщений на сервере

11. Реализовать обработку этого типа сообщений в клиенте (рис. 35)

```
    }
    case Send:
    {
        QJsonObject obj = doc.object().value("main").toObject();
        QString result = obj.value("text").toString();
        ui->show->append(result);
        break;
    }
    }
}
```

Рис. 35. Реализация обработки сообщений в клиенте

12. Работа приложения выглядит следующим образом (рис. 36)



а) б)

Рис. 36. Пример работы приложения

2.4. ВАРИАНТЫ ЛАБОРАТОРНЫХ ЗАДАНИЙ

Создать клиент-серверное приложение с прохождением авторизации и функционалом по вариантам

Таблица 2

Варианты заданий

Вариант	Задание
1	Сервер добавляет приветствие к имени, которое вводит пользователь
2	Сервер увеличивает на единицу число, введенное пользователем
3	Сервер дублирует данные, полученные от клиента
4	Сервер удваивает число, полученное от клиента
5	Сервер отправляет строку, введенную пользователем, в обратном порядке
6	Сервер делит на 2 число, полученное от клиента
7	Сервер отправляет введенную пользователем строку без изменений
8	Сервер возвращает сумму цифр введенного числа
9	Сервер добавляет восклицательный знак в конце введенной пользователем строки
10	Сервер возвращает остаток от деления введенного числа на 3

2.5. УКАЗАНИЯ ПО ОФОРМЛЕНИЮ ОТЧЕТА

Отчет должен содержать название и цель работы, краткие теоретические сведения, а также код с комментариями. В конце отчета должен быть вывод.

2.6. КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНЫМ ЗАДАНИЯМ

1. Расскажите о модуле Qt Network.
2. Назовите несколько классов модуля Qt Network.
3. Каковы особенности протокола TCP?
4. Каковы особенности протокола UDP?
5. Каковы особенности класса QTcpServer?
6. Каковы особенности класса QTcpSocket?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Саммерфилд М. Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++ / М. Саммерфилд – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 560 с.
2. Шлее М. Qt 5.10. Профессиональное программирование на C++ / М. Шлее – СПб: БХВ-Петербург, 2018. – 1072 с.
3. Бланшет Ж. QT 4: программирование GUI на C++ / Ж. Бланшет – М.: КУДИЦ-ПРЕСС, 2007. – 546 с.

ОГЛАВЛЕНИЕ

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ №11, 12
для студентов специальности 11.05.01
«Радиоэлектронные системы и комплексы»
очной формы обучения

Составитель
Сукачев Александр Игоревич

Издается в авторской редакции

Подписано к изданию 18.03.2024.
Уч.-изд. л. 1,7.

ФГБОУ ВО «Воронежский государственный технический университет»
394006 Воронеж, ул. 20-летия Октября, 84