А.О. Семкин, С.Н. Шарангович

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Разработка сетевых приложений»

2017

Министерство образования и науки Российской федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

(ТУСУР)

Кафедра сверхвысокочастотной и квантовой радиотехники
(СВЧиКР)

А.О. Семкин, С.Н. Шарангович

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Разработка сетевых приложений»

2017

УДК 004.4+519.6

Рецензент:
профессор каф.СВЧиКР, А.Е. Мандель

А.О. Семкин, С.Н. Шарангович

Информатика: Руководство к лабораторной работе «Библиотека Qt. Разработка сетевых приложений» / А.О. Семкин, С.Н. Шарангович. – Томск: ТУСУР, 2017. – 17 с.

В данном руководстве изложены принципы создания сетевого приложения (мессенджера) в среде разработки *Qt Creator*. Приведено описание модуля работы с сетью *QtNetwork*. Представлены методические материалы компьютерной лабораторной работы, посвященной разработке сетевых приложений.

Предназначено для студентов очной, заочной и вечерней форм обучения по направлению подготовки бакалавриата 11.03.02 «Инфокоммуникационные технологии и системы связи», 11.03.01 «Радиотехника».

УДК 004.4+519.6

Оглавление

1. Цель работы

Целью данной работы является изучение принципов разработки сетевых приложений при помощи средств разработки *Qt*.

2. Введение

Qt представляет собой комплексную рабочую среду, предназначенную для разработки на C++ межплатформенных приложений с графическим пользовательским интерфейсом по принципу «написал программу – компилируй ее в любом месте». *Qt* позволяет программистам использовать дерево классов с одним источником в приложениях, которые будут работать в системах Windows, Mac OS X, Linux. Solaris, HP-UX и во многих других.

В рамках данной лабораторной работы будут рассмотрены принципы разработки сетевых приложений в среде *Qt Creator*.

3. Программирование поддержки сети в Qt

Для того, чтобы облегчить создание сетевых кроссплатформенных приложений, разработчики фреймворка *Qt* предусмотрели модуль работы с сетью *QtNetwork*. Модуль *QtNetwork* содержит как высокоуровневые классы, такие как *QHttp* или *QFtp*, так и классы *QAbstractSocket*, *QTcpServer*, *QUdpSocket* и другие, с помощью которых можно работать с сетью на низком уровне.

Для того, чтобы у разработчика появился доступ к классам модуля *QtNetwork*, необходимо в файл проекта (**.pro*) внести следующую строку:

```
QT       += core network
```

3.1. Сокетное соединение

Сокет — это устройство пересылки данных с одного конца связи на другой. Другой конец может принадлежать процессу, работающему на локальном компьютере, а может располагаться и на удаленном компьютере, подключенному к Интернету и расположенному в другом полушарии Земли. Сокетное соединение — это соединение типа точка-точка (*point-to-point*), которое производится между двумя процессами.

Сокеты разделяют на дейтаграммные (*datagram*) и поточные. Дейтаграммные сокеты осуществляют обмен пакетами данных. Поточные сокеты устанавливают связь и производят потоковый обмен данными через установленную ими связь. На практике, поточные сокеты используются гораздо чаще, чем дейтаграммные из-за того, что они предоставляют дополнительные механизмы, направленные против искажения и потери данных. Поточные сокеты работают в обоих направлениях, то есть то, что один из процессов записывает в поток, может быть считано процессом на другом конце связи, и наоборот.

Для дейтаграммных сокетов *Qt* предоставляет класс *QUdpSocket*, а для поточных — класс *QTcpSocket*.

Класс *QTcpSocket* содержит набор методов для работы с TCP (Transmition Control Protocol, протокол управления передачей данных) — это сетевой протокол низкого уровня, являющийся одним из основных протоколов в Интернете. С его помощью можно реализовать поддержку для стандартных сетевых протоколов, таких как: HTTP, FTP, POP3, SMTP, и даже для своих собственных протоколов. Этот класс унаследован от класса *QAbstractSocket*, который, в свою очередь, наследует класс *QIODevice*. А это значит, что для доступа (чтения и записи) к его объектам необходимо применять все методы класса *QIODevice* и использовать классы потоков *QDataStream* или *QTextStream*.

Работа класса *QTcpSocket* асинхронна, что дает возможность избежать блокирования приложения в процессе его работы. Но если в этом нет необходимости, то есть возможность воспользоваться серией методов, начинающихся со слова *waitFor*. Вызов этих методов приведет к ожиданию выполнения операции и заблокирует, на определенное время, исполнение программы. Не рекомендуется вызывать эти методы в потоке графического интерфейса.

3.2. Модель "клиент-сервер"

Сценарий модели "клиент-сервер" выглядит очень просто: сервер предлагает услуги, а клиент ими пользуется. Программа, использующая сокеты, может выполнять либо роль сервера, либо роль клиента.

Для того чтобы клиент мог взаимодействовать с **сервером, ему нужно знать его IP-адрес и номер порта**, через который клиент, желающий воспользоваться этими услугами сервера, должен сообщить о себе. Когда клиент устанавливает соединение с сервером, система назначает данному соединению отдельный сокет. После этого устанавливается связь между двумя этими сокетами, по которой высылаются данные запроса к серверу. А сервер высылает клиенту по этому соединению готовые, обработанные результаты согласно его запросам. Сервер не ограничен связью только с одним клиентом, на самом деле он может обслуживать многих клиентов.

Каждому сокету соответствует уникальный номер порта. Некоторые номера зарезервированы для так называемых стандартных служб:

Порт/Протокол	Описание
0/TCP,UDP	резерв (допустимо использование в качестве значения порта источника, если отправляющий процесс не ожидает ответных сообщений)
7/TCP,UDP	ECHO — предназначен для тестирования связи путём отправки данных на сервер и получения от него их же в неизменном виде
9/TCP,UDP	DISCARD — предназначен для тестирования связи путём отправки данных на сервер, который отбрасывает принятое, не отправляя никакого ответа
11/TCP,UDP	SYSTAT — выдаёт список активных пользователей в операционной системе
13/TCP,UDP	DAYTIME — предназначен для тестирования связи путём получения от сервера текущих даты и времени в текстовом виде

Порт/Протокол	Описание
20/TCP	FTP-DATA — для передачи данных FTP
21/TCP	FTP — для передачи команд FTP
22/TCP,UDP	SSH (Secure SHell) — криптографический сетевой протокол для безопасной передачи данных
23/TCP,UDP	Telnet — применяется для передачи текстовых сообщений в незашифрованном виде
25/TCP,UDP	SMTP (Simple Mail Transfer Protocol) — применяется для пересылки почтовых сообщений в виде незашифрованного текста
80/TCP,UDP	HTTP (HyperText Transfer Protocol); ранее — WWW
110/TCP,UDP	POP3 (Post Office Protocol Version 3)
143/TCP,UDP	IMAP (Internet Message Access Protocol) — используется для получения и синхронизации сообщений электронной почты
161/TCP,UDP	SNMP (Simple Network Management Protocol) — используется как порт прослушивания агентами удалённого мониторинга
194/TCP,UDP	IRC (Internet Relay Chat)
443/TCP,UDP	HTTPS (HyperText Transfer Protocol Secure) — HTTP с шифрованием по SSL или TLS

4. Реализация сервера с помощью класса QTcpServer

Для реализации сервера *Qt* предоставляет удобный класс *QTcpServer*, который предназначен для управления входящими TCP-соединениями. Программа, описанная ниже, является реализацией простого сервера, который принимает и подтверждает получение запросов клиентов.

Ниже приведено содержимое файла *main.cpp*

```
#include <QApplication>
#include "MyServer.h"
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    MyServer     server(2323);

    server.show();

    return app.exec();
}
```

Создается объект сервера. Чтобы запустить сервер, нужно создать объект класса *MyServer*, передав в конструктор номер порта, по которому должен осуществляться нужный сервис. В данном случае передается номер порта, равный 2323.

Состав класса *MyServer* описан в заголовочном файле *MyServer.h*

```
#ifndef MYSERVER_H
#define MYSERVER_H

#include <QWidget>
#include <QTcpServer>
#include <QTextEdit>
#include <QTcpSocket>


class QTcpServer;
class QTextEdit;
```

```
class QTcpSocket;

class MyServer : public QWidget {
Q_OBJECT
private:
    QTcpServer* m_ptcpServer;
    QTextEdit*  m_ptxt;
    quint16     m_nNextBlockSize;

private:
    void sendToClient(QTcpSocket* pSocket, const QString& str);

public:
    MyServer(int nPort, QWidget* pwgt = 0);

public slots:
    virtual void slotNewConnection();
            void slotReadClient   ();
};

#endif // MYSERVER_H
```

В классе *MyServer* объявляется элемент *m_ptcpServer*, который и является основной частью для управления сервером. Элемент *m_nNextBlockSize* служит для хранения длины следующего, полученного от сокета блока.

Реализация конструктора класса, а также его методов приводится в файле *MyServer.cpp*

```
#include "MyServer.h"
#include <QtGui>
#include <QMessageBox>
#include <QVBoxLayout>
#include <QLabel>

MyServer::MyServer(int nPort, QWidget* pwgt /*=0*/) : QWidget(pwgt)
                                                    , m_nNextBlockSize(0)
{
    m_ptcpServer = new QTcpServer(this);
    if (!m_ptcpServer->listen(QHostAddress::Any, nPort)) {
        QMessageBox::critical(0,
                              "Server Error",
                              "Unable to start the server:"
                              + m_ptcpServer->errorString()
                             );
        m_ptcpServer->close();
        return;
    }
    connect(m_ptcpServer, SIGNAL(newConnection()),
            this,         SLOT(slotNewConnection())
           );

    m_ptxt = new QTextEdit;
    m_ptxt->setReadOnly(true);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(new QLabel("<H1>Server</H1>"));
    pvbxLayout->addWidget(m_ptxt);
    setLayout(pvbxLayout);
}
```

Для установки сервера необходимо вызвать в конструкторе метод *listen()*. В этот метод необходимо передать номер порта, который объект класса получает в конструкторе. При возникновении ошибочных ситуаций, например невозможности захвата порта, этот метод возвратит значение *false*, на которое объект класса *MyServer* отреагирует показом окна сообщения об ошибке.

Далее производится соединение с сигналом *newConnection()*, который высылается при каждом присоединении нового клиента.

Для отображения информации создается виджет многострочного текстового поля (указатель *m_ptxt*) и в нем вызовом метода *setReadOnly()* устанавливается режим, делающий возможным только просмотр информации.

```
/*virtual*/ void MyServer::slotNewConnection()
{
    QTcpSocket* pClientSocket = m_ptcpServer->nextPendingConnection();
    connect(pClientSocket, SIGNAL(disconnected()),
            pClientSocket, SLOT(deleteLater())
           );
    connect(pClientSocket, SIGNAL(readyRead()),
            this,          SLOT(slotReadClient())
           );

    sendToClient(pClientSocket, "Server Response: Connected!");
}
```

Метод *slotNewConnection()* вызывается каждый раз при соединении с новым клиентом. Из этого метода выполняются соединения с сигналами *disconnected()* и *readyRead()*, которые сигнализируют об отсоединении клиента и его готовности к передаче данных соответственно. В завершение вызывается метод *sendToServer()* для отсылки приветствия присоединенному клиенту. В этом методе вторым параметром передается строка. Внутри самого метода будет сгенерирован временной штамп, который будет отослан клиенту вместе с переданной строкой.

Для подтверждения соединения с клиентом необходимо вызвать метод *nextPendingConnection()*, который возвратит сокет, посредством которого можно осуществлять дальнейшую связь с клиентом. Сигнал *disconnected()*, высылаемый сокетом при отсоединении клиента, связывается со слотом *deleteLater()*, предназначенным для его последующего уничтожения. При поступлении запросов от клиентов высылается сигнал *readyToRead()*, который соединяется со слотом *slotReadClient()*.

```
void MyServer::slotReadClient()
{
    QTcpSocket* pClientSocket = (QTcpSocket*)sender();
    QDataStream in(pClientSocket);
    in.setVersion(QDataStream::Qt_4_2);
    for (;;) {
        if (!m_nNextBlockSize) {
            if (pClientSocket->bytesAvailable() < sizeof(quint16)) {
                break;
            }
            in >> m_nNextBlockSize;
        }
```

```
            if (pClientSocket->bytesAvailable() < m_nNextBlockSize) {
                break;
            }
            QTime   time;
            QString str;
            in >> time >> str;

            QString strMessage =
                time.toString() + " " + "Client has sended - " + str;
            m_ptxt->append(strMessage);

            m_nNextBlockSize = 0;

            sendToClient(pClientSocket,
                         "Server Response: Received \"" + str + "\""
                        );
        }
    }
```

Сначала производится преобразование указателя, возвращаемого методом *sender()*, к типу *QTcpSocket*. Цикл *for* нужен, так как не все высланные клиентом данные могут прийти одновременно. Поэтому сервер должен быть в состоянии получать как весь блок целиком, так и только часть блока, а так же и все блоки сразу. Каждый переданный сокетом блок начинается полем, описывающим размер блока. Размер блока, который считывается при условии того, что размер полученных данных не меньше двух байт и атрибут *m_nNextBlockSize* равен нулю (то есть размер блока неизвестен). Если доступных данных для чтения больше или равно размеру блока, тогда производится считывание из потока данных в переменные *time* и *str*. После этого значение переменной *time* преобразуется вызовом метода *toString()* в строку и используется вместе со строкой *str* для строки сообщения *strMessage*. Строка сообщения добавляется в виджет текстового поля вызовом метода *append()*. Анализ блока данных завершается присваиванием атрибуту *m_nNextBlockSize* значения 0, которое говорит о том, что размер очередного блока данных неизвестен. Вызовом метода *sendToClient()* сервер сообщает клиенту о том, что ему успешно удалось прочитать высланные им данные.

```
void MyServer::sendToClient(QTcpSocket* pSocket, const QString& str)
{
    QByteArray  arrBlock;
    QDataStream out(&arrBlock, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_2);
    out << quint16(0) << QTime::currentTime() << str;

    out.device()->seek(0);
    out << quint16(arrBlock.size() - sizeof(quint16));

    pSocket->write(arrBlock);
}
```

В методе *sendToClient()* формируются данные, которые будут высланы клиенту. Но есть один нюанс, который заключается в том, что зарнее

неизвестен размер блока, и, следовательно, нет возможности записывать данные сразу в сокет, так как размер блока должен быть выслан в первую очередь. Поэтому выполняются следующие действия. Сначала создается объект *QByteArray*. В него записываются все данные блока, причем записывается размер, равный 0. После этого указатель перемещается на начало блока вызовом метода *seek()*, вычисляется и записывается размер блока в поток (*out*) (вычисляется как размер *arrBlock* с вычитанием из него *sizeof(quint16)*). После этого созданный блок записывается в сокет вызовом метода *write()*.

Примечание. Для пересылки обычных строк можно было бы использовать класс потока ввода *QTextStream*. Причиной, по которой используется класс *QDataStream*, является то, что пересылка бинарных данных представляет собой общий случай, это необходимо для того, чтобы переслать объект класса *QTime*. То есть, нельзя отправлять не только строки, но и еще растровые изображения, объекты палитры и т. д. Далее будет использоваться класс *QDataStream*.

5. Реализация клиента с помощью класса QTcpSocket

Приложение-клиент должно быть создано в виде отдельного проекта *Qt*.

Для реализации клиента нужно создать объект класса *QTcpSocket*, а затем вызвать метод *connectToHost()*, передав в него первым параметром имя компьютера (или его IP-адрес), а вторым — номер порта сервера. Объект класса *QTcpSocket* сам попытается произвести установку связи с сервером и, в случае успеха, вышлет сигнал *connected()*. В противном случае будет выслан сигнал *error(int)* с кодом ошибки, определенным в перечислении *QAbstractSocket::SocketError*. Это может произойти, например, в том случае, если на указанном компьютере не установлен сервер или не соответствует номер порта. После установления соединения объект класса *QTcpSocket* может высылать или считывать данные сервера.

Высылка на сервер информации, введенной в однострочном текстовом поле окна клиента, производится после нажатия на кнопку Send (Выслать).

Ниже приведено содержимое файла *main.cpp*

```
#include <QApplication>
#include "MyClient.h"

int main(int argc, char** argv)
{
   QApplication app(argc, argv);
    MyClient client("localhost", 2323);
        client.show();
    return app.exec();
}
```

В функции *main()* создается объект клиента. Если запуск сервера и клиента производится на одном и том же компьютере, то в качестве имени компьютера можно передать строку "localhost". Номер порта, в данном случае, равен 2323, так как это тот порт, который используется нашим сервером.

Состав класса *MyClient* приведен в файле *MyClient.h*

```
#ifndef _MyClient_h_
#define _MyClient_h_

#include <QWidget>
#include <QTcpSocket>
#include <QLabel>
#include <QTextLayout>
#include <QTextEdit>
#include <QLineEdit>
#include <QVBoxLayout>
#include <QTime>

class MyClient : public QWidget {
Q_OBJECT
private:
    QTcpSocket* m_pTcpSocket;
    QTextEdit*  m_ptxtInfo;
    QLineEdit*  m_ptxtInput;
    quint16     m_nNextBlockSize;

public:
    MyClient(const QString& strHost, int nPort, QWidget* pwgt = 0) ;

private slots:
    void slotReadyRead   ( );
    void slotError(QAbstractSocket::SocketError);
    void slotSendToServer( );
    void slotConnected   ( );
};

#endif  //_MyClient_h_
```

В классе *MyClient* объявляется элемент *m_pTcpSocket*, который необходим для управления клиентом, и элемент *m_nNextBlockSize*, необходимый для хранения длины следующего полученного от сокета блока. Остальные два элемента - *m_txtInfo* и *m_txtInput* - используются для отображения и ввода информации соответственно.

Реализация конструктора класса, а также его методов приводится в файле *MyClient.cpp*

```
MyClient::MyClient(const QString &strHost,
                   int nPort,
                   QWidget* pwgt /*=0*/
                  ) : QWidget(pwgt)
                    , m_nNextBlockSize(0)
{
    m_pTcpSocket = new QTcpSocket(this);
     this->setWindowTitle("Client");
    m_pTcpSocket->connectToHost(strHost, nPort);
    connect(m_pTcpSocket, SIGNAL(connected()), SLOT(slotConnected()));
    connect(m_pTcpSocket, SIGNAL(readyRead()), SLOT(slotReadyRead()));
    connect(m_pTcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),
            this,         SLOT(slotError(QAbstractSocket::SocketError))
           );


    m_ptxtInfo  = new QTextEdit;
```

```
    m_ptxtInput = new QLineEdit;
    connect(m_ptxtInput, SIGNAL(returnPressed()),
            this,        SLOT(slotSendToServer())
           );
    m_ptxtInfo->setReadOnly(true);

    QPushButton* pcmd = new QPushButton("&Send");
    connect(pcmd, SIGNAL(clicked()), SLOT(slotSendToServer()));

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(new QLabel("<H1>Client</H1>"));
    pvbxLayout->addWidget(m_ptxtInfo);
    pvbxLayout->addWidget(m_ptxtInput);
    pvbxLayout->addWidget(pcmd);
    setLayout(pvbxLayout);
}
```

В конструкторе происходит создание объекта сокета (указатель *m_pTcpSocket*). Из объекта сокета вызывается метод *connectToHost()*, осуществляющий связь с сервером. Первым параметром в этот метод передается имя компьютера, а вторым - номер порта. Связь между сокетами асинхронна. Сокет высылает сигнал *connected()* как только будет произведено соединение, а также высылает сигнал *readyRead()* о готовности предоставить данные для чтения. Сигналы *connected()*, *readyRead()* соединяются со слотами *slotConnected()* и *slotReadyRead()* соответственно. В случаях возникновения ошибок сокет высылает сигнал *error()*, который соединяется со слотом *slotError()*, в котором производится отображение ошибок.

Затем создается пользовательский интерфейс программы, состоящий из надписи, кнопки нажатия, однострочного и многострочного текстовых полей. Сигнал *clicked()* виджета кнопки нажатия соединяется со слотом *slotSendToServer()* класса *MyClient*, ответственным за отправку данных на сервер. Для того чтобы к аналогичному действию приводило и нажатие клавиши <Enter>, сигнал *returnPressed()* виджета текстового поля (*m_ptxtInput*) соединяется с тем же слотом *slotSendToServer()*.

```
void MyClient::slotReadyRead()
{
    QDataStream in(m_pTcpSocket);
    in.setVersion(QDataStream::Qt_4_2);
    for (;;) {
        if (!m_nNextBlockSize) {
            if (m_pTcpSocket->bytesAvailable() < sizeof(quint16)) {
                break;
            }
            in >> m_nNextBlockSize;
        }

        if (m_pTcpSocket->bytesAvailable() < m_nNextBlockSize) {
            break;
        }
        QTime time;
        QString str;
        in >> time >> str;

        m_ptxtInfo->append(time.toString() + " " + str);
```

```
            m_nNextBlockSize = 0;
        }
    }
```

Слот *slotReadyToRead()* вызывается при поступлении данных от сервера. Цикл *for* нужен, так как не все данные с сервера могут прийти одновременно. Поэтому клиент должен быть в состоянии получить как весь блок целиком, так и только часть блока или даже все блоки сразу. Каждый переданный блок начинается полем, хранящим размер блока.

После того как клиент будет уверен, что блок получен целиком, то он может использовать оператор >> объекта потока *QDataStream* (переменная *in*). Чтение данных из сокета осуществляется при помощи объекта потока данных. Полученная информация добавляется в виджет многострочного текстового поля (указатель *m_ptxtInfo*) с помощью метода *append()*.

В завершение анализа блока данных элементу *m_nNextBlockSize* присваивается значение 0, которое указывает на то, что размер очередного блока данных неизвестен.

```
void MyClient::slotError(QAbstractSocket::SocketError err)
{
    QString strError =
        "Error: " + (err == QAbstractSocket::HostNotFoundError ?
                     "The host was not found." :
                     err == QAbstractSocket::RemoteHostClosedError ?
                     "The remote host is closed." :
                     err == QAbstractSocket::ConnectionRefusedError ?
                     "The connection was refused." :
                     QString(m_pTcpSocket->errorString())
                    );
    m_ptxtInfo->append(strError);
}
```

Слот *slotError()* вызывается при возникновении ошибок. В нем преобразуется код ошибки в текст для того, чтобы отобразить его в виджете многострочного текстового поля.

```
void MyClient::slotSendToServer()
{
    QByteArray  arrBlock;
    QDataStream out(&arrBlock, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_2);
    out << quint16(0) << QTime::currentTime() << m_ptxtInput->text();

    out.device()->seek(0);
    out << quint16(arrBlock.size() - sizeof(quint16));

    m_pTcpSocket->write(arrBlock);
    m_ptxtInput->setText("");
}
```

Следует обратить внимание на то, что записывать данные сразу в *QTcpSocket* возможности нет, потому что не известен размер блока, который должен быть выслан в первую очередь. Поэтому сначала необходимо создать объект

QByteArray, для того чтобы записывать все данные блока в него, записывая сначала размер равным 0. После того как все необходимые данные блока записаны, указатель перемещается на начало блока и вызовом метода *seek()* записывается размер блока, который вычисляется как размер *arrBlock* с вычитанием из него *sizeof(quint16)*. Это делается для исключения данных размера при подсчете байт.

После нажатия кнопки Send (Послать) вызывается слот *slotSendToServer()*, который записывает в сокет строку, введенную пользователем в виджете однострочного текстового поля (указатель *m_ptxtInput*).

```
void MyClient::slotConnected()
{
    m_ptxtInfo->append("Received the connected() signal");
}
```

Как только связь с сервером установлена, вызывается метод *slotConnected()* и в виджет текстового поля добавляется строка сообщения.

6. Методические указания по выполнению работы

Лабораторную работу необходимо выполнить в следующей последовательности:

1. Изучите теоретические разделы 2-5, выполните на компьютере все приведенные в данных разделах примеры, скомпилируйте и запустите на исполнение полученные программы. В качестве дополнительных источников информации, воспользуйтесь указанными в перечне используемой литературы (раздел 7) пособиями, а также открытыми источниками в сети Интернет.
2. Самостоятельно выполните задание, описанное ниже в разделе 6.1.
3. Подготовьте исходный код программ в Qt Creator покажите его преподавателю.
4. В случае отсутствия видимых ошибок в коде, скомпилируйте программы и запустите их на выполнение.
5. Получите оценку.

6.1. Расширение функционала сетевого приложения

Полученная на предыдущих этапах работы программа не позволяет пользователю выбирать IP-адрес и TCP-порт сервера без обращения к исходному коду программы-клиента.

Дополните программу-клиент строками ввода IP-адреса и TCP-порта сервера, к которому необходимо выполнить подключение, доработайте методы класса *MyClient* таким образом, чтобы у пользователя появилась возможность выбрать сервер и подключиться к нему.

Подключите компьютеры студентов Вашей группы к одной локальной сети. Попросите преподавателя помочь Вам, если это требуется. Запустите

программу-сервер на разных компьютерах созданной сети. Подключитесь к разным серверам с использованием доработанной Вами программы-клиента.

Для определения IP-адресов компьютеров в локальной сети воспользуйтесь встроенными средствами операционной системы. Номера TCP-портов серверов уточните у Ваших одногруппников. Попросите преподавателя помочь Вам, если это требуется.

7. Рекомендуемая литература

1. Qt. Профессиональное программирование на C++: Наиболее полное руководство / М. Шлее. - СПб. : БХВ-Петербург, 2005. – 544 с.
2. Бланшет Ж., Саммерфилд М. Qt 4: программирование GUI на C++. Пер. с англ. 2-е изд., доп. – М.: КУДИЦ-ПРЕСС, 2008. – 736 с.
3. Разработка сетевых приложений. [Электронный ресурс]. - Режим доступа: http://qt-doc.ru/qt-network.html (дата обращения: 24.07.2017)

Учебное издание

Семкин Артем Олегович

Шарангович Сергей Николаевич

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Разработка сетевых приложений»

Формат 60x84 1/16. Усл. печ. л. _____.
Тираж _____ экз. Заказ_____.
Томский государственный университет систем управления и радиоэлектроники.
634050, Томск, пр. Ленина, 40. Тел. (3822) 533018.