ГОСУДАРСТВЕННЫЙ КОМИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИИ И ТЕЛЕКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

Ташкентский университет информационных технологий

Факультет "Информационные технологии"

КУРСОВАЯ РАБОТА

По предмету "Системное программное обеспечение"

ТЕМА: Разработка программного обеспечения для работы сети Интернет.

Выполнил: студент группы 225-10 ИТр

Исматуллаев С.О.

Ташкент – 2013 г.

Содержание

Введение

Работая в сети Internet мы очень часто встречаемся с разного рода многопользовательскими программами. Ими могут быть почтовые клиенты, чаты, форумы, FTP клиенты и т.п. Все эти приложения используют для своей работы разного рода протокола: FTP, POP, SMTP, HTTP, и т.д. Но базовым для них является единый протокол - TCP/IP. Типичное же приложение TCP/IP построено на клиент-серверной архитектуре.

На протяжении последних десяти лет специалисты по вычислительной технике работают над усовершенствованием приложений клиент-сервер. В результате были построены приложения, поддерживающие совместную работу множества пользователей с единственным источником данных в сети.

Архитектура клиент-сервер стала общераспространенной при общении с компьютером или с системой на его основе. Любой человек, подключающийся к диалоговой информационной системе с помощью телефонной связи, использует архитектуру клиент-сервер. Пользуясь автоматическим кассовым аппаратом, считывая штриховые коды своих покупок на проверочном устройстве магазина или расплачиваясь за них с помощью кредитной карточки, идет взаимодействие с компьютерной системой клиент-сервер.

Примером приложения построенного на данной архитектуре, является чат реального времени. В данной курсовой работе я буду создавать простейшее клиент серверное приложение. Для этого я использую кроссплатформенную среду разработки Qti язык программирования C++.

Программирование для сети.

В области компьютеризации понятие **программирования сетевых задач** или иначе называемого **сетевого программирования** (англ. *networkprogramming*),довольно сильно с хожего с понятиями *программирование сокетов* и *клиент-серверное программирование*, включает в себя написание компьютерных программ, взаимодействующих с другими программами посредством компьютерной сети. Программа или процесс, инициирующие установление связи, называются клиентским процессом, а программа, ожидающая инициации связи, называется серверным процессом. Клиентский и серверный процессы вместе образуют распределенную систему.

Архитектура файл-сервер

Самой простой архитектурой для реализации является архитектура "файл-сервер" (рисунок 1), но она же обладает и самым большим количеством недостатков, ограничивающих спектр решаемых ею задач. Простейшим случаем является случай, когда данные располагаются физически на том же компьютере, что и само приложение.

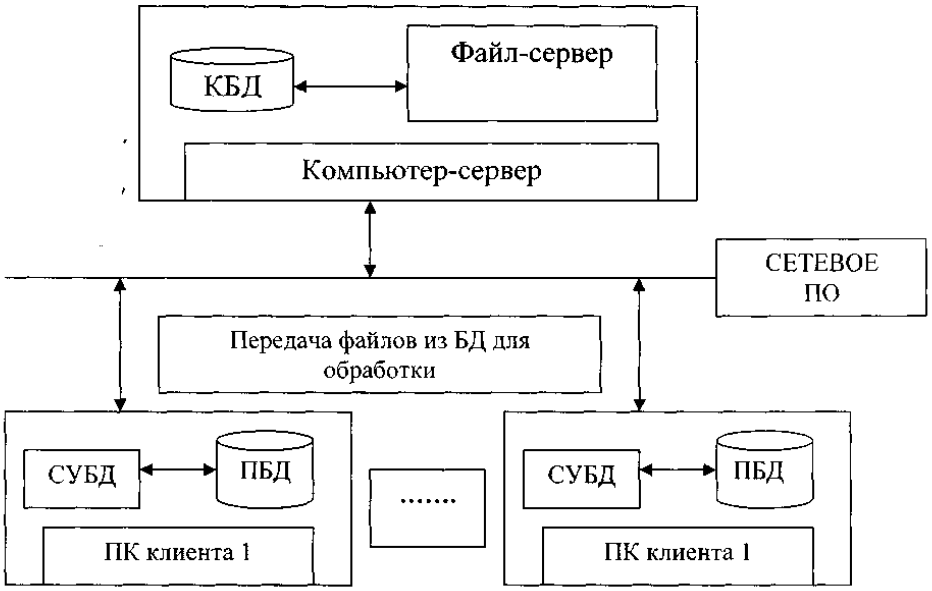


Рисунок 1 Структура информационной системы с файл-сервером

К существенным неудобствам, возникающим при работе с системой, построенной по такой архитектуре, можно отнести следующее:

- трудности при обеспечении непротиворечивости и целостности данных;

- существенная загрузка локальной сети передаваемыми данными;

- в целом, невысокая скорость обработки и представления информации;

- высокие требования к ресурсам компьютеров. При этом возникают следующие ограничения.

- невозможность организации равноправного одновременного доступа; пользователей к одному и тому же участку базы данных;

- количество одновременно работающих с системой пользователей не превышает пяти человек для ЛВС, построенной в соответствии со спецификацией 1 OBaseT (скорость обмена данными до 10Мб/с);

При всем этом система обладает одним очень важным преимуществом - низкой стоимостью.

Архитектура "файл-сервер" предусматривает концентрацию обработки на рабочих станциях. Основным преимуществом этого варианта является простота и относительная дешевизна. Подобное решение приемлемо, пока число пользователей, одновременно работающих с базой данных, не превышает 5-10 человек. При увеличении количества пользователей система может "захлебнуться" из-за перегруженности ЛВС большими потоками необработанной информации.

Сервер, как правило, — самый мощный и самый надежный компьютер. Он обязательно подключается через источник бесперебойного питания, в нем предусматриваются системы двойного или даже тройного дублирования. В особо ответственных случаях можно подключить вместе несколько серверов так, что при выходе из строя одного из них в работу автоматически включится "дублер". Таким образом, при концентрации обработки данных на сервере надежность системы в целом ограничивается только материальными средствами, которые заказчики готовы вложить в техническое оснащение.

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 IP-адрес сервера и номер порта, через который клиент должен сообщить о себе. Когда клиент начинает соединение с сервером, его система назначает данному соединению отдельный сокет, а когда сервер принимает соединение, сокет назначается со стороны сервера. После этого устанавливается связь между двумя этими сокетами, по которой высылаются данные запроса к серверу. А сервер высылает клиенту, по тому же соединению, готовые результаты согласно его запросу. Сервер не ограничен связью только с одним клиентом, на самом деле он может обслуживать многих клиентов.

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

Таблица некоторых зарезервированных номера портов:

Порт	Сервис	Описание
11	systat	Показ зарегистрированных в системе пользователей
21	FTP	Доступ к файлам по сети
22	SSH	Зашифрованное соединение к удаленному компьютеру
23	Telnet	Удаленное соединение с компьютером
25	SMTP	Отсылка электронной почты
80	HTTP	Веб-сервер (иногда применяются порты 8080 или 8000)
110	POP3	Получение электронной почты
139	Netbios-SSN	Разделение сетевых ресурсов
143	IMAP	Пересылка электронной почты
194	IRC	Internet Relay Chat
442	HTTPS	Зашифрованный HTTP

Сокеты

Сокеты (англ. *socket* — разъём) — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Принципы сокетов

Каждый процесс может создать *слушающий* сокет (серверный сокет) и *привязать* его к какому-нибудь порту операционной системы (в UNIX непривилегированные процессы не могут использовать порты меньше 1024).

Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. При этом сохраняется возможность проверить наличие соединений на данный момент, установить тайм-аут для операции и т.д.

Каждый сокет имеет свой адрес. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX-адрес. Если привязать сокет к UNIX-адресу, то будет создан специальный файл (*файл сокета*) по заданному пути, через который смогут сообщаться любые локальные процессы путём чтения/записи из него. Сокеты типа INET доступны из сети и требуют выделения номера порта.

Обычно клиент явно *подсоединяется* к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером.

Типы сокетов

Сокеты Беркли — интерфейс программирования приложений (API), представляющий собой библиотеку для разработки приложений на языке Си с поддержкой межпроцессного взаимодействия (IPC), часто применяемый в компьютерных сетях.

Сокеты Беркли (также известные как API сокетов BSD), впервые появились как API в операционной системе 4.2BSD Unix (выпущенной в 1983 году). Тем не менее, только в 1989 году Калифорнийский университет в Беркли смог начать выпускать версии операционной системы и сетевой библиотеки без лицензионных ограничений AT&T, действующих в защищённой авторским правом Unix.

API сокетов Беркли сформировал *defacto* стандарт абстракции для сетевых сокетов. Большинство прочих языков программирования используют интерфейс, схожий с API языка Си.

API Интерфейса транспортного уровня (TLI), основанный на STREAMS, представляет собой альтернативу сокетному API. Тем не менее, API сокетов Беркли значительно преобладает в популярности и количестве реализаций.

Интерфейс сокета Беркли — API, позволяющий реализовывать взаимодействие между компьютерами или между процессами на одном компьютере. Данная технология может работать со множеством различных устройств ввода/вывода и драйверов, несмотря на то, что их поддержка зависит от реализации операционной системы. Подобная реализация интерфейса лежит в основе TCP/IP, благодаря чему считается одной из фундаментальных технологий, на которых основывается Интернет. Технология сокетов

впервые была разработана в Калифорнийском университете Беркли для применения на Юникс-системах. Все современные операционные системы имеют ту или иную реализацию интерфейса сокетов Беркли, так как это стало стандартным интерфейсом для подключения к сети Интернет.

Программисты могут получать доступ к интерфейсу сокетов на трёх различных уровнях, наиболее мощным и фундаментальным из которых является уровень сырых сокетов. Довольно небольшое число приложений нуждается в ограничении контроля над исходящими соединениями, реализуемыми ими, поэтому поддержка сырых сокетов задумывалась быть доступной только на компьютерах, применяемых для разработки на основе технологий, связанных с Интернет. Впоследствии, в большинстве операционных систем была реализована их поддержка, включая Windows XP

Unix domain socket (Доменный сокет Unix) или IPC-сокет (сокет межпроцессного взаимодействия) — конечная точка обмена данными, схожая с Интернет-сокетом, но не использующая сетевой протокол для взаимодействия (обмена данными). Он используется в операционных системах, поддерживающих стандарт POSIX, для межпроцессного взаимодействия. Корректным термином стандарта POSIX является POSIX Local IPC Sockets.

Доменные соединения Unix являются по сути байтовыми потоками, сильно напоминая сетевые соединения, но при этом все данные остаются внутри одного компьютера (т.е. обмен данными происходит локально). UDS используют файловую систему как адресное пространство имен, т.е. они представляются процессами как иноды в файловой системе. Это позволяет двум различным процессам открывать один и тот же сокет для взаимодействия между собой. Однако, текущее взаимодействие (обмен данными) не использует файловую систему, а только буферы памяти ядра.

В дополнение к отсылаемым данным процессы могут отсылать файловые дескрипторы через соединение на основе UDS (включая файловые дескрипторы для доменных сокетов), используя системные вызовы sendmsg() и recvmsg(). Это означает, что доменные сокеты могут быть использованы как объектно-возможностная коммуникационная система.

Windows Sockets API (WSA), название которого было укорочено до Winsock. Это техническая спецификация, которая определяет, как сетевое программное обеспечение Windows будет получать доступ к сетевым сервисам, в том числе, TCP/IP. Он определяет стандартный интерфейс между клиентским приложением (таким как FTP клиент или веб-

браузер) и внешним стеком протоколов TCP/IP. Он основывается на API модели сокетов Беркли, использующейся в BSD для установки соединения между программами.

Ранние операционные системы Microsoft, такие как MS-DOS и MicrosoftWindows имели ограничения по работе с сетью, которые были связаны с использованием протокола NetBIOS. В частности, Microsoft в то время не поддерживал работу со стеком протоколов TCP/IP. Несколько университетских групп и коммерческих фирм, включая MIT, FTP Software, SunMicrosystems, Ungermann-Bass, и Excelan, представляли свои решения для работы с TCP/IP в MS-DOS, часто как часть программно-аппаратного комплекса. После выпуска MicrosoftWindows 2.0 к этим разработчикам присоединились и другие, такие как Distinct и NetManage, которые помогли в организации поддержки протоколов TCP/IP для Windows. Недостаток, с которыми столкнулись все вышеперечисленные разработчики, состоял, в том, что каждый из них использовал свои собственные API (ApplicationProgrammingInterface). Без единой стандартной модели программирования, трудно было убедить независимых разработчиков программного обеспечения в создании сетевых программ, которые могли бы работать на основе реализации стека протоколов TCP/IP любого из разработчиков. Стало понятно, что необходима стандартизация.

Модель Sockets API Windows была предложена Мартином Холлом из JSB Software (позднее StardustTechnologies), в рамках информационной группы BOF (Birdsof a Feather) и согласована в сети CompuServe на BBS в октябре 1991 года. Первое издание спецификации было написано Мартином Холлом, Марком Товфиком из Microdyne (позднее SunMicrosystems), Джеффом Арнольдом (SunMicrosystems), Генри Сандерс и Дж. Алардом из Microsoft, и при участии многих других разработчиков. Возникли вопросы о том, кому присвоить авторские права, права интеллектуальной собственности. В конце концов, было решено, что авторские права на спецификацию будут принадлежать пяти авторам, как физическим лицам.

Начинаяс Windows 2000 Winsock работает через Transport Driver Interface

Windows 8 включает в себя RIO (Registered IO), расширяющий возможности Winsock.

Практическая часть

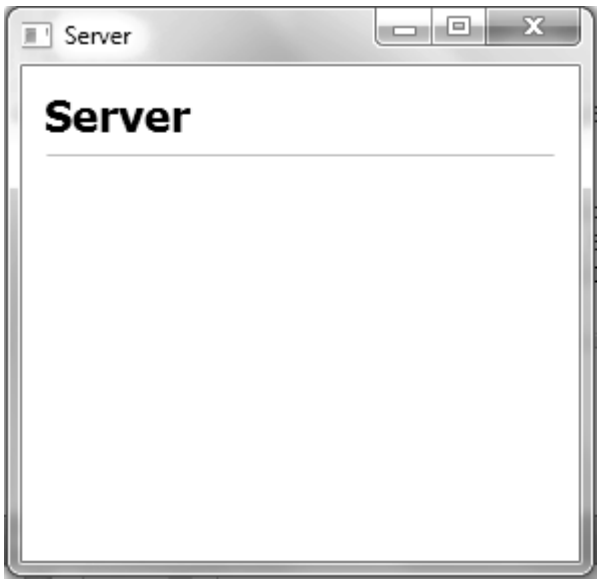
Как уже было сказано, сокеты разделяют на дейтаграммные (datagram) и поточные. Дейтаграммные сокеты осуществляют обмен пакетами данных. Поточные сокеты устанавливают связь и производят потоковый обмен данными через установленную ими линию связи.

Для дейтаграмных сокетов Qt предоставляет класс QUdpSocket, а для поточных — класс QTcpSocket. В этой главе мы ограничимся рассмотрением только поточных сокетов. Класс QTcpSocket содержит набор методов для работы с TCP (TransmitionControlProtocol, протокол управления передачей данных) — сетевым протоколом низкого уровня, который является одним из основных протоколов в Интернете. С его помощью можно реализовать поддержку для стандартныхсетевых 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 можно применять все методы класса QIODevice и использовать классы потоков QDataStream или QTextStream.

Работа класса QTcpSocket асинхронна, что дает возможность избежать блокировки приложения в процессе его работы. Но если вам это не нужно, то можете воспользоваться серией методов, начинающихся со слова waitFor. Вызов этих методов приведет к ожиданию выполнения операции и заблокирует, на определенное время, исполнение вашей программы. Не рекомендуется вызывать эти методы в потоке графического интерфейса.

Реализация сервера

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



Листинг 1. Файл main.cpp:

```
#include<QtGui/QApplication>
#include"testserver.h"
intmain(intargc,char*argv[])
{
        QApplicationa(argc,argv);
        testserverServer(2323);
        Server.show();
        returna.exec();
}
```

В листинге 1 создается объект сервера. Чтобы запустить сервер, мы создаем объект определенного нами в листингах 2—3класса testserver, передав в конструктор номер порта, по которому должен осуществляться нужный сервис. В нашем случае передается номер порта, равный 2323.

Листинг 2. testserver.h

```
#ifndefTESTSERVER_H
#defineTESTSERVER_H

#include<QtGui>
#include<QWidget>
#include<QtNetwork/QTcpServer>
#include<QtNetwork/QTcpSocket>

class QLineEdit;
class QTextEdit;
class testserver : public QWidget
{
Q_OBJECT
```

```
private:
QTcpServer*m_ptcpServer;
QTextEdit*m_ptxt;
quint16m_nNextBlockSize;

private:
voidsendToClient(QTcpSocket*pSocket,constQString&str);

public:
testserver(intnPort,QWidget*pwgt=0);

publicslots:
virtualvoidslotNewConnection();
voidslotReadClient();
};
#endif//TESTSERVER_H
```

В классе testserver, определяемом в листинге 2, мы объявляем атрибут m_ptcpServer, который и является основой управления нашим сервером. Атрибут m_nNextBlockSize служит для хранения длины следующего полученного от сокета блока. Многострочное текстовое поле m_ptxt предназначено для информирования о происходящих соединениях.

Листинг 3. testserver.cpp:

```
#include"testserver.h"

testserver::testserver(intnPort,QWidget*pwgt):QWidget(pwgt),
m_nNextBlockSize(0)
{
m_ptcpServer=newQTcpServer(this);

if(!m_ptcpServer->listen(QHostAddress::Any,nPort)){
QMessageBox::critical(0,"ServerError","Unabletostartthe
                        server:"+m_ptcpServer->errorString());
m_ptcpServer->close();
return;
}
connect(m_ptcpServer,SIGNAL(newConnection()),this,
            SLOT(slotNewConnection()));
m_ptxt=newQTextEdit;
m_ptxt->setReadOnly(true);
//Layoutsetup
QVBoxLayout*pvbxLayout=newQVBoxLayout;
pvbxLayout->addWidget(newQLabel("<H1>Server</H1>"));
pvbxLayout->addWidget(m_ptxt);
setLayout(pvbxLayout);
}
```

```
/* Для запуска сервера нам необходимо вызвать в конструкторе метод listen(). В этот метод
необходимо передать номер порта, который мы получили в конструкторе. При возникновении
ошибочных ситуаций,  например, невозможности захвата порта, этот метод возвратит
значение false, на которое мы отреагируем показом окна сообщения об ошибке. Если ошибки не
произошло, мы соединяем определенный нами слот slotNewConnection()ссигналомnewConnection (),
который высылается при каждом присоединении нового клиента. Для отображения информации
мы создаем виджетмногострочного  текстового поля (указатель m_ptxt) и устанавливаем в нем
режим, в котором  возможен только просмотр информации, вызовом метода setReadOnly ().
* /

voidtestserver::slotNewConnection()
{
QTcpSocket*pClientSocket=m_ptcpServer->nextPendingConnection();
connect(pClientSocket,SIGNAL(disconnected()),pClientSocket,
        SLOT(deleteLater()));
connect(pClientSocket,SIGNAL(readyRead()),this,SLOT(slotReadClient()));
sendToClient(pClientSocket,"ServerResponse:Connected!");
}

/* Метод slotNewConnection()вызывается каждый раз при соединении с новым клиентом. Для
подтверждения соединения с клиентом необходимо вызвать метод nextPendingConnection(),
который возвратит сокет, посредством которого можно осуществлять дальнейшую связь с
клиентом. Дальнейшая работа сокета обеспечивается сигнально- слотовыми связями. Мы
соединяем сигнал disconnected(), высылаемый сокетом при отсоединении клиента, со
стандартным слотом QObject::deleteLater(), предназначенным для его последующего
уничтожения. При поступлении запросов от клиентов высылается сигнал readyToRead (),
который мы  соединяемсослотом slotReadClient ().
* /

voidtestserver::slotReadClient()
{
QTcpSocket*pClientSocket=(QTcpSocket*)sender();
QDataStreamin(pClientSocket);
in.setVersion(QDataStream::Qt_4_8);
for(;;){
if(!m_nNextBlockSize){
if(pClientSocket->bytesAvailable()<sizeof(quint16)){
break;
}
in>>m_nNextBlockSize;
}
if(pClientSocket->bytesAvailable()<m_nNextBlockSize){
break;
}
QTimetime;
QStringstr;
in>>time>>str;
QStringstrMessage=time.toString()+""+"Clienthassent-"+
                  str;
m_ptxt->append(strMessage);
m_nNextBlockSize=0;
sendToClient(pClientSocket,"ServerResponse:Received\""+str+
                  "\"");
```

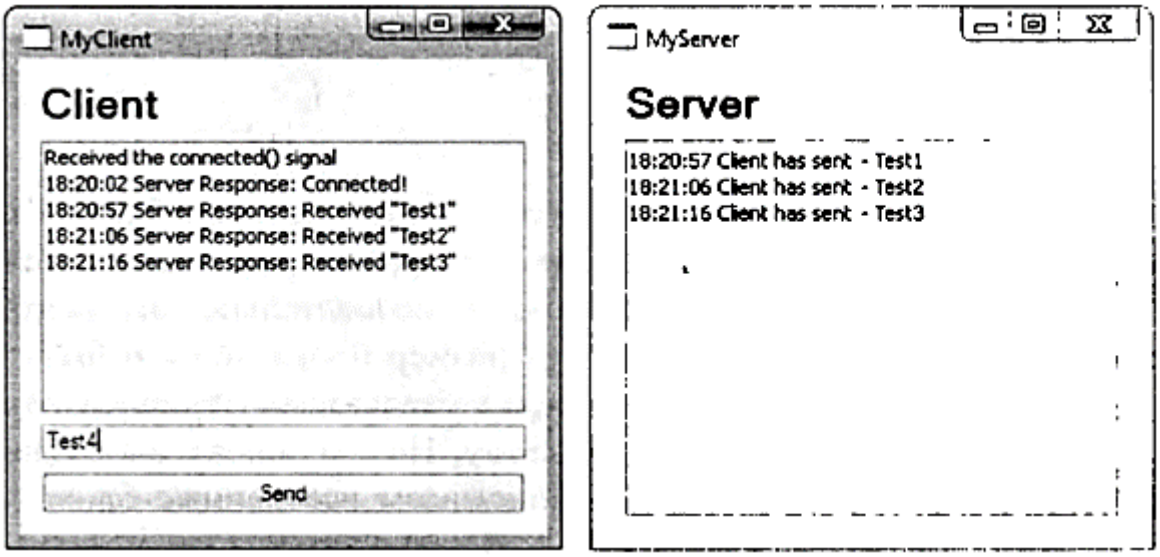
```
}
}

voidtestserver::sendToClient(QTcpSocket*pSocket,constQString&str)
{
QByteArrayarrBlock;
QDataStreamout(&arrBlock,QIODevice::WriteOnly);
out.setVersion(QDataStream::Qt_4_8);
out<<quint16(0)<<QTime::currentTime()<<str;
out.device()->seek(0);
out<<quint16(arrBlock.size()-sizeof(quint16));
pSocket->write(arrBlock);
}
```

В методе sendToClient() мы формируем данные, которые будут отосланы клиенту. Есть небольшая деталь, которая заключается в том, что нам заранее не известен размер блока, а следовательно, мы не можем записывать данные сразу в сокет, так как размер блока должен быть выслан в первую очередь. Поэтому мы прибегаем к следующему трюку. Сначала мы создаем объект arrBlock класса QByteArray. На его основе мы создаем объект класса QDataStream, в который записываем все данные блока, причем вместо реального размера записываем 0. После этого мы перемещаем указатель на начало блока вызовом метода seekO, вычисляем размер блока как размер arrBiock, уменьшенный на sizeof (quinti6), и записываем его в поток (out) с текущей позиции, которая уже перемещена в начало блока. После этого созданный блок записывается в сокет вызовом метода write ().

Реализация клиента

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 установить связь с сервером и, в случае успеха, вышлет сигнал connectedo. 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 запущен сервер или не соответствует номер порта. После установления соединения, объект класса QTcpSocket может высылать или считывать данные сервера. Рисунок иллюстрирует взаимодействие клиента с сервером. Высылка на сервер информации, введенной в однострочном текстовом поле окна клиента, производится после нажатия кнопки Send (Выслать).



В функции main (), мы создаем объект клиента. Если сервер и клиент запускаются на одном компьютере, то в качестве имени компьютера можно передать строку "localhost". Номер порта в нашем случае равен 2323, так как это тот порт, который используется нашим сервером.

Функция main():

```
#include<QtGui/QApplication>
#include"testclient.h"
intmain(intargc,char*argv[])
{
QApplicationa(argc,argv);
testclientclient("localhost",2323);
client.show();
returna.exec();
}
```

Файлtestclient.h:

```
#ifndefTESTCLIENT_H
#defineTESTCLIENT_H
#include<QtGui>
#include<QWidget>
#include<QtNetwork/QTcpSocket>
classQLineEdit;
classQTextEdit;
classtestclient:publicQWidget{
Q_OBJECT
private:
QTcpSocket*m_pTcpSocket;
QTextEdit*m_ptxtInfo;
QLineEdit*m_ptxtInput;
quint16m_nNextBlockSize;
```

```
public:
testclient(constQString&strHOst,intnPort,QWidget*pwgt=0);
privateslots:
voidslotReadyRead();
voidslotError(QAbstractSocket::SocketError);
voidslotSendToServer();
voidslotConnected();
};
#endif//TESTCLIENT_H
```

В классе testclient, мы объявляем атрибут m_pTcpSocket, который необходим для управления нашим клиентом, и атрибутm_nNextBlockSize, необходимый нам для хранения длины следующего полученного от сокета блока. Остальные два атрибута m_txtInfo и m_txtInput используются для отображения и ввода информации соответственно.

Файл testclient.cpp:

```
#include"testclient.h"

testclient::testclient(constQString&strHost,
intnPort,
QWidget*pwgt):QWidget(pwgt),m_nNextBlockSize(0)
{
m_pTcpSocket=newQTcpSocket(this);
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=newQTextEdit;
m_ptxtInput=newQLineEdit;
m_ptxtInfo->setReadOnly(true);
QPushButton*pcmd=newQPushButton("&Send");
connect(pcmd,SIGNAL(clicked()),SLOT(slotSendToServer()));
connect(m_ptxtInput,SIGNAL(returnPressed()),
this,SLOT(slotSendToServer()));
QVBoxLayout*pvbxLayout=newQVBoxLayout;
pvbxLayout->addWidget(m_ptxtInput);
pvbxLayout->addWidget(m_ptxtInfo);
pvbxLayout->addWidget(pcmd);
setLayout(pvbxLayout);
}

voidtestclient::slotReadyRead()
{
QDataStreamin(m_pTcpSocket);
in.setVersion(QDataStream::Qt_4_5);
for(;;){
if(!m_nNextBlockSize){
if(m_pTcpSocket->bytesAvailable()<sizeof(quint16)){
```

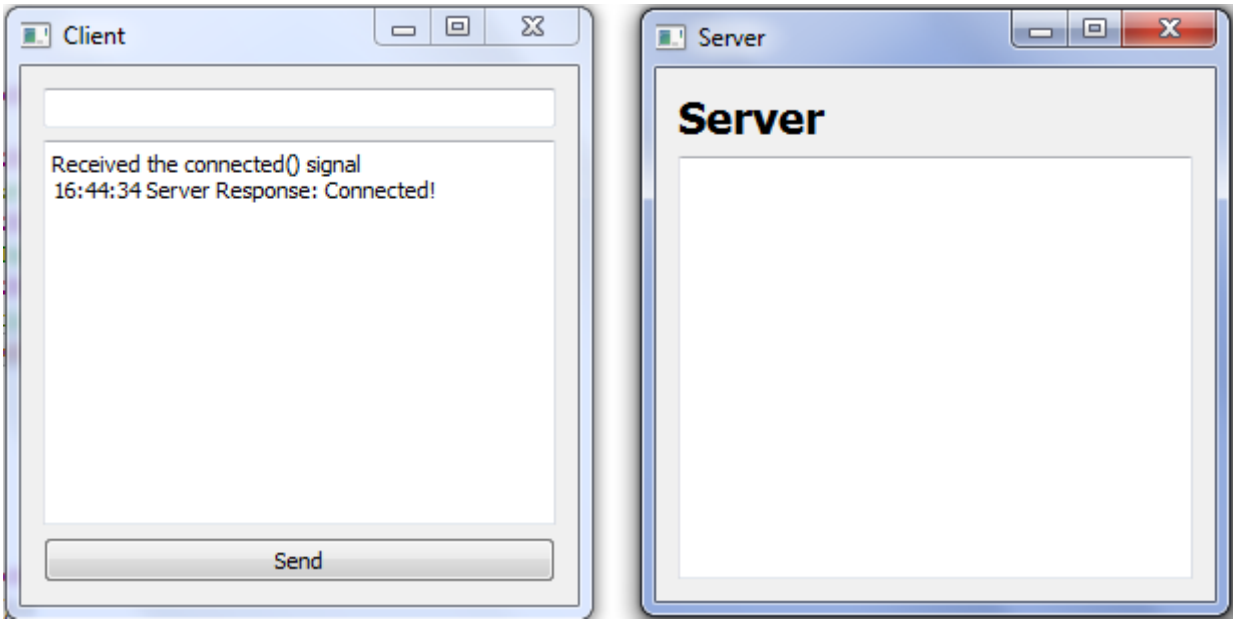
```
break;
}
in>>m_nNextBlockSize;
}
if(m_pTcpSocket->bytesAvailable()<m_nNextBlockSize){
break;
}
QTimetime;
QStringstr;
in>>time>>str;
m_ptxtInfo->append(time.toString()+""+str);
m_nNextBlockSize=0;
}
}

voidtestclient::slotError(QAbstractSocket::SocketErrorerr){
QStringstrError=
"Error"+(err==QAbstractSocket::HostNotFoundError?
"Thehostwasnotfound.":
err==QAbstractSocket::RemoteHostClosedError?
"Theremotehostisclosed.":
err==QAbstractSocket::ConnectionRefusedError?
"Theconnectionwasrefused.":
QString(m_pTcpSocket->errorString())
);
m_ptxtInfo->append(strError);
}

voidtestclient::slotSendToServer(){
QByteArrayarrBlock;
QDataStreamout(&arrBlock,QIODevice::WriteOnly);
out.setVersion(QDataStream::Qt_4_5);
out<<quint16(0)<<QTime::currentTime()<<m_ptxtInput->text();
out.device()->seek(0);
out<<quint16(arrBlock.size()-sizeof(quint16));
m_pTcpSocket-
>write("<stream:streamxmlns='jabber:client'xmlns:stream='http://etherx.jabber.org/streams'to='o
ffice'version='1.0'>");
m_ptxtInput->setText("");
}

voidtestclient::slotConnected(){
m_ptxtInfo->append("Receivedtheconnected()signal");
}
```

Успешно запущенные приложения Clientи Server. На окне Clientприложения выводиться сообщение об успешном завершении.


Client
Received the connected() signal
16:44:34 Server Response: Connected!
Send
Server
Server

Заключение

В приведенной работе было разработано очень простое клиент-серверное приложение. Написав программу “Client” и “Server”, я столкнулся со сложностью представления функциональности всей системы, так как очень сложно представить всю работу системы в голове или даже на бумаге.

В ходе выполнения и прохождения технологической практики я научился созданию клиент – серверных приложений с использованием технологии «сокетов». Разработав программу я научился использовать кроссплатформенную среду разработки Qti получил опыт в использовании технологии сокетов на языке программирования C++.

Узнал очень ценные информации о сетевом программировании на Qt. Класс QTcpSocket содержит набор методов для работы с TCP (TransmitionControlProtocol, протокол управления передачей данных) — сетевым протоколом низкого уровня, который является 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.

Многоуровневая архитектура клиент-сервер позволяет существенно упростить распределенные вычисления, делая их не только более надежными, но и более доступными. Появление таких средств, как Java и PHP, упрощает связь сервера приложений с клиентами, а объектно-ориентированные менеджеры транзакций обеспечивают согласованную работу сервера приложений с базами данных. В результате создаются все предпосылки для создания сложных распределенных информационных систем, которые эффективно используют все преимущества современных технологий.

Может этот приложения я доработаю к защите своего диплома.

Литературы

1. Избачков Ю.С. Информационные системы: Учебник для вузов/ Ю.С.Избачков, Петров В.Н. – Санкт-Петербург, 2006. - 656 с.

2. Марченко А.Л. C#. Введение в программирование: Учебное пособие – М., 2005. - 258с.

3. Орлик С. Многоуровневые модели в архитектуре клиент-сервер [http://ods.com.ua/win/rus/db/kbd97/22.htm] / С. Орлик

4. Полякова Л.Н. Лекция: Хранимые процедуры

[http://www.intuit.ru/department/database/sql/12/2.html]/ Л.Н.Полякова

5. Трёхуровневая архитектура [http://ru.wikipedia.org/wiki]

6. Шлее М. - Qt4.5. Профессиональное программирование на C++ (В подлиннике). «БХВ-ПЕТЕРБУРГ» 2010.

7. Жасмин Бланшет - QT 4 программирование CUI на C++. «КУДИЦ-ПРЕСС» Москва 2007.

8.http://www.programmersclub.ru/

9. http://habrahabr.ru/post/111239/

10. http://habrahabr.ru/post/131585/