Министерство образования и науки, молодежи и спорта Украины
Севастопольский национальный технический университет

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Лабораторная работа №5

«Основы разработки приложений для работы с сетью в среде C++*Builder* на примере программы «ЧАТ»»

по дисциплине

«**Персональные компьютеры в инженерной практике**»

для студентов дневной и заочной формы обучения

по направлению 6.050901 — Радиотехника.

Севастополь
2013 г.

УДК.519.72

Методические указания «Лабораторная работа №5 Основы разработки приложений для работы с сетью в среде C++*Builder* на примере программы «ЧАТ»» по дисциплине «Персональные компьютеры в инженерной практике» для студентов дневной и заочной форм обучения по направлению 6.050901 — Радиотехника / Сост. А.В. Лукьянчиков — Севастополь: Изд-во СевНТУ, 2013. — 19 с.

Методические указания рассмотрены и утверждены на заседании кафедры радиотехники и телекоммуникаций (протокол № _ от __ _____ 2013 г.)

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент: канд. техн. наук, доцент Савочкин А.А.

Ответственный за выпуск: заведующий кафедрой радиотехники и телекоммуникаций, доктор техн. наук, профессор Гимпилевич Ю.Б.,

5. ЛАБОРАТОРНАЯ РАБОТА № 5
ОСНОВЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ ДЛЯ РАБОТЫ С СЕТЬЮ В СРЕДЕ C++*BUILDER* НА ПРИМЕРЕ ПРОГРАММЫ «ЧАТ»

5.1. Цель работы: На примере простейшего приложения «чат» разобраться в принципах разработки приложений для работы с сетью в среде C++ *Builder*.

5.2. Теоретические сведения

Так как большинство современных компьютеров объединены в сети, то и задачи программирования передачи и получения данных по сети возникают часто. Существует множество высокоуровневых средств обмена, но иногда их бывает недостаточно, и тогда приходится использовать самые низкоуровневые средства сетевого программирования — сокеты. Однако существует целый ряд причин, по которым овладеть этой технологией непросто. Во-первых, она сложна сама по себе из-за большого количества возможностей. Во-вторых, существующие библиотеки отягощены совместимостью со старыми версиями, которые, с современной точки зрения, не всегда развивались по правильному пути. В-третьих, у программистов на *Delphi* возникают дополнительные трудности, связанные как с тем, что сама реализация библиотеки сокетов ориентирована на язык C, так и с тем, что стандартный модуль *winsock* почему-то "застрял" на первой версии этой библиотеки, в то время как начиная с *Windows NT* 4 существует более удобная вторая.

Сначала рассмотрим классические методы работы с сокетами, которые не учитывают ни существования окон и оконных сообщений, ни возможности распараллеливания работы программы на несколько нитей. Это, впрочем, не означает, что программа, использующая эти методы, должна быть безоконной и однонитевой, оконные и многонитевые программы есть среди примеров этого раздела. Просто приспосабливать стандартные сокеты к окнам и распараллеливанию приходится вручную, а не за счет средств самой библиотеки. Тем не менее, из-за своей простоты стандартные сокеты нередко оказываются более удобными, чем сокеты *Windows*, даже в оконных приложениях.

Сокетом (от англ. *socket* — гнездо, розетка) называется специальный объект, создаваемый для отправки и получения данных через сеть. Отметим, что под термином "объект" в данном случае подразумевается не объект в терминах объектно-ориентированного программирования, а некоторая сущность, внутренняя структура которой скрыта от нас, поэтому с этой сущностью мы можем оперировать только как с единым и неделимым (атомарным) объектом. Этот объект создаётся внутри библиотеки сокетов, а программист, использующий эту библиотеку, получает уникальный номер (дескриптор) этого сокета. Конкретное значение этого дескриптора не несёт для программиста никакой полезной информации и может быть использовано только для того, чтобы при вызове функции из библиотеки сокетов указать, с каким сокетом требуется выполнить операцию. В этом смысле тип *TSocket* полностью аналогичен дескрипторам окон, графических объектов и т. п., с которыми мы встречались в предыдущей главе.

Чтобы две программы могли общаться друг с другом через сеть, каждая из них должна создать сокет. Каждый сокет обладает двумя основными характеристиками: протоколом и адресом, к которым он привязан. Протокол задается при создании сокета и не может быть изменен впоследствии. Адрес сокета задается позже, но обязательно до того, как через сокет пойдут данные. В некоторых случаях привязка сокета к адресу может быть неявной.

Формат адреса сокета определяется конкретным протоколом. В частности, для протоколов *TCP* и *UDP* адрес состоит из *IP*-адреса сетевого интерфейса и номера порта.

Каждый сокет имеет два буфера: для входящих и для исходящих данных. При отправке данных они сначала помещаются в буфер исходящих, и лишь затем отправляются в фоновом режиме. Программа в это время продолжает свою работу. При получении данных сокет помещает их в буфер для входящих, откуда они затем могут извлекаться программой.

Сеть может связывать разные аппаратные платформы, поэтому требуется согласование форматов передаваемых данных, в частности форматов целых чисел. Двухбайтные целые числа

хранятся в памяти в двух последовательно расположенных байтах. При этом возможны два варианта: в первом байте хранится младший байт числа, а во втором — старший, и наоборот. Способ хранения определяется аппаратной частью платформы. Процессоры *Intel* используют первый вариант, т. е. первым хранится младший байт, а другие процессоры (например, *Motorola*) — второй вариант. То же касается и четырехбайтных чисел: процессоры *Intel* хранят их, начиная с младшего байта, а некоторые другие процессоры — начиная со старшего. Сетевой формат представления таких чисел совпадает с форматом процессора *Motorola*, т. е. на платформах с процессором *Intel* необходимо переставлять байты при преобразовании чисел в сетевой формат.

Библиотека сокетов разрабатывалась для ОС *Unix*, в которой традиционно высоко ценилась переносимость между платформами, поэтому она содержит функции, позволяющие не задумываться о порядке байтов в числах: *ntohs*, *ntohl*, *htons* и *htonl*. Первая буква в названии этих функций показывает, в каком формате дано исходное число (*n* — *Network* — сетевой формат, *h* — *Host* — формат платформы), четвертая буква — формат результата, последняя буква — разрядность (*s* — *Short* — двухбайтное число, *I* — *Long* — четырехбайтное число). Например, функция *htons* принимает в качестве параметра число типа *u_short* (*Word*) в формате платформы и возвращает то же число в сетевом формате. Реализация этих функций для каждой платформы своя: где-то они переставляют байты, где-то они возвращают в точности то число, которое было им передано. Благодаря этим функциям программы становятся переносимыми. Хотя для программиста на *Delphi* вопросы переносимости не столь актуальны, приходится прибегать к этим функциям хотя бы потому, что байты переставлять нужно, а никакого более удобного способа для этого не существует.

Сетевые протоколы. Семиуровневая модель *OSI*

Сетевым протоколом называется набор соглашений, следование которым позволяет обеим сторонам одинаково интерпретировать принимаемые и отправляемые данные. Сетевой протокол можно сравнить с языком: два человека понимают друг друга тогда, когда говорят на одном языке. Причем если люди, говорящие на похожих, но немного разных языках, всё же могут понимать друг друга, то компьютеры для нормального обмена данными должны поддерживать в точности одинаковый протокол.

Для установления взаимодействия между компьютерами должен быть согласован целый ряд вопросов, начиная от напряжения в проводах и заканчивая форматом пакетов. Реализуются эти соглашения на разных уровнях, поэтому логичнее иметь не один протокол, описывающий все и вся, а набор протоколов, каждый из которых охватывает только вопросы одного уровня. Организация *Open Software Interconnection* (*OSI*) предложила разделить все вопросы, требующие согласования, на семь уровней. Это разделение известно как семиуровневая модель *OSI*.

Семейство протоколов, реализующих различные уровни, называется стеком протоколов. Стеки протоколов не всегда точно следуют модели *OSI*, некоторые протоколы решают вопросы, связанные сразу с несколькими уровнями.

Первый уровень в модели *OSI* называется физическим. На нем согласовываются физические, электрические и оптические параметры сети: напряжение и форма импульсов, кодирующих 0 и 1, какой разъем используется и т. п.

Второй уровень носит название канального. На этом уровне решаются вопросы конфигурации сети (шина, звезда, кольцо и т. п.), приема и передачи кадров, допустимости и методов разрешения коллизий (ситуаций, когда сразу два компьютера пытаются передать данные).

Третий уровень — сетевой. Здесь определяется, как адресуются компьютеры. Большинство сетей используют широковещательный способ передачи: пакет, переданный одним компьютером, получают все остальные. Протокол сетевого уровня описывает критерии, на основании которых каждый компьютер может выбирать из сети только те пакеты, которые предназначены ему, и игнорировать все остальные. На этом же уровне определяется, как пакеты проходят через маршрутизатор.

Четвертый уровень называется транспортным. На этом уровне единый физический поток данных разбивается на независимые логические потоки. Это позволяет нескольким программам независимо друг от друга использовать сеть, не опасаясь, что их данные смешаются. Кроме того,

на транспортном уровне решаются вопросы, связанные с подтверждением доставки пакета и упорядочиванием пакетов.

Пятый уровень известен как уровень сессии. Он определяет процедуру установления, завершения связи и ее восстановления после разрыва. Расписывается последовательность действий каждой стороны и пакеты, которые они должны друг другу отправить для инициализации и завершения связи. Определяются соглашения о том, как единый поток разбивается на логические пакеты.

Шестой уровень называется уровнем представлений. На этом уровне определяется то, в каком формате данные передаются по сети. Под этим подразумевается, в первую очередь, внутренняя структура пакета, а также способ представления данных разных типов. Например, для двух- и четырехбайтных целых чисел должен быть согласован порядок байтов, для логических величин — какие значения соответствуют *True*, какие — *False*, для строк — кодировка и способ задания конца строки и т. п.

Седьмой уровень называется уровнем приложений. Соглашения этого уровня позволяют работать с ресурсами (файлами, принтерами и т. д.) удаленного компьютера как с локальными, осуществлять удаленный вызов процедур и т. п.

Чтобы получить данные через сеть, должны быть реализованы все уровни, за исключением, может быть, седьмого. Для каждого уровня должен быть определен свой протокол. В идеале механизмы взаимодействия между протоколами разных уровней должны иметь столь высокую степень абстракции, чтобы один протокол на любом из уровней можно было заменить любым другим протоколом того же уровня, не внося каких-либо изменений в выше- и нижележащие уровни.

Стек *TCP/IP*

Физический и канальный уровни полностью реализуются сетевой картой или модемом (или другим устройством, выполняющим ту же функцию) и ее драйвером. Здесь действительно достигнута настолько полная абстракция, что программист обычно не задумывается о том, какая используется сеть. Поэтому мы также не будем останавливаться на этих двух уровнях.

В реальной жизни не все протоколы, особенно старые, соответствуют модели *OSI*. Существует такое понятие, как стек протоколов — набор протоколов разных уровней, которые совместимы друг с другом. Эти уровни не всегда точно соответствуют тем, которые предлагает модель *OSI*, но определенное разделение задач на уровни в них присутствует. Здесь мы сосредоточимся на стеке протоколов, который называется *TCP*/*IP* (нередко можно услышать словосочетание "протокол *TCP*/*IP*", что не совсем корректно; *TCP*/*IP* не протокол, а стек протоколов). Название этот стек получил по наименованию двух самых известных своих протоколов: *TCP* и *IP*.

Протокол сетевого уровня *IP* расшифровывается как *Internet Protocol*. Это название иногда ошибочно переводят как "протокол Интернета" или "протокол для Интернета". На самом деле, когда разрабатывался этот протокол, никакого Интернета еще и в помине не было, поэтому правильный перевод — межсетевой протокол. История появления этого протокола связана с особенностями работы сети *Ethernet*. Эта сеть строится по принципу шины, когда все компьютеры подключены, грубо говоря, к одному проводу. Если хотя бы два компьютера попытаются одновременно передавать данные по общей шине, возникнет неразбериха, поэтому все шинные сети строятся по принципу "один говорит — все слушают". Очевидно, что требуется какая-то защита от так называемых коллизий (ситуаций, когда два узла одновременно пытаются передавать данные).

Разные сети решают проблему коллизий по-разному. В промышленных сетях, например, обычно имеется маркер — специальный индикатор, который показывает, какому узлу разрешено сейчас передавать данные. Узел, называемый мастером, следит за тем, чтобы маркер вовремя передавался от одного узла к другому. Маркер исключает возможность возникновения коллизий. *Ethernet* же является одноранговой сетью, в которой нет мастера, поэтому в ней реализован другой подход: коллизии допускаются, но существует механизм их разрешения, заключающийся в том, что, во-первых, узел не начинает передачу данных, если видит, что другой узел уже что-то передает, а во-вторых, если два узла одновременно пытаются начать передачу, то оба прекращают попытку и повторяют ее через случайный промежуток времени. У кого этот

промежуток окажется меньше, тот и захватит сеть (или за этот промежуток времени сеть будет захвачена кем-то еще).

При большом числе компьютеров, сидящих на одной шине, коллизии становятся слишком частыми, и производительность сети резко падает. Для борьбы с этим служат специальные устройства — маршрутизаторы — специализированные узлы, подключенные одновременно к нескольким сетям. Пока остальные узлы каждой из этих сетей взаимодействуют только между собой, маршрутизатор никак себя не проявляет, и эти сети существуют независимо друг от друга. Но если компьютер из одной сети посылает пакет компьютеру другой сети, этот пакет получает маршрутизатор и переправляет его в ту сеть, в которой находится адресат, или в которой находится другой маршрутизатор, способный передать этот пакет адресату.

На канальном уровне существует адресация узлов, основанная на так называемом MAC-адресе сетевой карты (*MAC* — это сокращение *Media Access Control*). Этот адрес является уникальным номером карты, присвоенной ей производителем. Очевидно неудобство такого способа адресации, т. к. по MAC-адресу невозможно определить положение компьютера в сети, т. е. выяснить, куда направлять пакет. Кроме того, при замене сетевой карты меняется адрес компьютера, что также не всегда удобно. Поэтому на сетевом уровне определяется собственный способ адресации, не связанный с аппаратными особенностями узла. Отсюда следует, что маршрутизатор должен понимать протокол сетевого уровня, чтобы принимать решение о передаче пакета из одной сети в другую, а протокол, в свою очередь, должен учитывать наличие маршрутизаторов в сети и предоставлять им необходимую информацию. *IP* был одним из первых протоколов сетевого уровня, который решал такую задачу, и с его помощью стала возможной передача пакетов между сетями. Поэтому он и получил название межсетевого протокола. Впрочем, название прижилось: в некоторых статьях *MSDN*, сетевой уровень (*network layer*) называется межсетевым уровнем (*internet layer*). В протоколе *IP*, в частности, вводится важный параметр для каждого пакета: максимальное число маршрутизаторов, которое он может пройти, прежде чем попадет к адресату (этот параметр носит не совсем удачное название *TTL — Time* To *Live*, время жизни). Это позволяет защититься от бесконечного блуждания пакетов по сети.

Здесь следует заметить, что сеть *Ethernet* ушла далеко вперед по сравнению с моментом создания протокола *IP* и теперь организована сложнее, поэтому не следует думать, что в предыдущих абзацах изложены все принципы работы этой сети (это выходит за рамки данной книги). Тем не менее, протокол *IP* по-прежнему используется, а компьютеры по-прежнему видят в сети не только свои, но и чужие пакеты. На этом основана работа так называемых снифферов — программ, позволяющих одному компьютеру читать пакеты, пересылаемые между двумя другими компьютерами.

Для адресации компьютера протокол *IP* использует уникальное четырехбайтное число, называемое *IP*-адресом. Впрочем, более распространена форма записи этого числа в виде четырех однобайтных значений. Система назначения этих адресов довольно сложна и призвана оптимизировать работу маршрутизаторов, обеспечить прохождение широковещательных пакетов только внутри определенной части сети и т. п. Мы здесь не будем подробно останавливаться на этом, потому что в правильно настроенной сети программисту не нужно знать всех этих тонкостей: достаточно помнить, что каждый узел имеет уникальный *IP*-адрес, для которого принята запись в виде четырех цифровых полей, разделенных точками, например, 192.168.200.217. Также следует знать, что адреса из диапазона 127.0.0.—127.255.255.255 задают так называемый локальный узел: через эти адреса могут связываться программы, работающие на одном компьютере. Таким образом, обеспечивается прозрачность местонахождения адресата. Кроме того, один компьютер может иметь несколько *IP*-адресов, которые могут использоваться для одного и того же или разных сетевых интерфейсов.

Кроме *IP*, в стеке *TCP*/*IP* существует еще несколько протоколов — *ICMP*, *IGMP* и *ARP*, — решающих задачи сетевого уровня. Эти протоколы не являются полноценными и не могут заменить *IP*. Они служат только для решения некоторых частных задач.

Протокол *ICMP* (*Internet Control Message Protocol* — протокол межсетевых управляющих сообщений) обеспечивает диагностику связи на сетевом уровне. Многим знакома утилита *ping*, позволяющая проверить связь с удаленным узлом. В основе ее работы лежат специальные запросы и ответы, определяемые в рамках протокола *ICMP*. Кроме того, этот же протокол

определяет сообщения, которые получает узел, отправивший *IP*-пакет, если этот пакет по каким-то причинам не доставлен.

Протокол называется надежным (*reliable*), если он гарантирует, что' пакет будет либо доставлен, либо отправивший его узел получит уведомление о том, что доставка невозможна. Кроме того, надежный протокол должен гарантировать, что пакеты доставляются в том же порядке, в каком они отправлены, и дублирования сообщений не происходит. Протокол *IP* в чистом виде не является надежным протоколом, т. к. в нем вообще не предусмотрены средства уведомления узла о проблемах с доставкой пакета. Добавление *ICMP* также не делает *IP* надежным, т. к. *ICMP*-пакет является частным случаем *IP*-пакета, и также может не дойти до адресата, поэтому возможны ситуации, когда пакет не доставлен, а отправитель об этом не подозревает.

Протокол *IGMP* (*Internet Group Management Protocol* — протокол управления межсетевыми группами) предназначен для управления группами узлов, которые имеют один групповой *IP*-адрес. Отправку пакета по такому адресу можно рассматривать как нечто среднее между адресной и широковещательной рассылкой, т. к. такой пакет будет получен сразу всеми узлами, входящими в группу.

Протокол *ARP* (*Address Resolution Protocol* — протокол разрешения адресов) необходим для установления соответствия между *IP*- и MAC-адресами. Каждый узел имеет таблицу соответствия. Исходящий пакет содержит два адреса узла: MAC-адрес для канального уровня и *IP*-адрес для сетевого. Отправляя пакет, узел находит в своей таблице MAC-адрес, соответствующий *IP*-адресу получателя, и добавляет его к пакету. Если в таблице такой адрес не найден, отправляется широковещательное сообщение, формат которого определяется протоколом *ARP*. Получив такое сообщение, узел, чей *IP*-адрес соответствует искомому, отправляет ответ, в котором указывает свой MAC-адрес. Этот ответ также широковещательный, поэтому его получают все узлы, а не только отправивший запрос, и все узлы обновляют свои таблицы соответствия.

Строго говоря, обеспечение правильного функционирования протоколов сетевого уровня — задача администратора системы, а не программиста. В своей работе программист чаще всего сталкивается с более высокоуровневыми протоколами и не интересуется деталями реализации сетевого уровня.

Протоколами транспортного уровня в стеке *TCP*/*IP* являются протоколы *TCP* и *UDP*. Строго говоря, они решают не только задачи транспортного уровня, но и небольшую часть задач уровня сессии. Тем не менее, они традиционно называются транспортными. Эти протоколы мы рассмотрим детально в следующих разделах.

Уровни сессии, представлений и приложений в стеке *TCP*/*IP* не разделены: протоколы *HTTP*, *FTP*, *SMTP* и т. д., входящие в этот стек, решают задачи всех трех уровней. Мы здесь не будем рассматривать эти протоколы, потому что при использовании сокетов они в общем случае не нужны: программист сам определяет формат пакетов, отправляемых с помощью *TCP* или *UDP*.

Новички нередко думают, что фраза "программа поддерживает соединение через *TCP*/*IP*" полностью описывает то, как можно связаться с программой и получить данные. На самом деле необходимо знать формат пакетов, которые эта программа может принимать и отправлять, т. е. должны быть согласованы протоколы уровня сессии и представлений. Гибкость сокетов дает программисту возможность самостоятельно определить этот формат, т. е., по сути дела, придумать и реализовать собственный протокол поверх *TCP* или *UDP*. И без описания этого протокола организовать обмен данными с программой невозможно.

Протокол *UDP*

Протокол *UDP* (*User Datagram Protocol* — протокол пользовательских дейтаграмм) встречается реже, чем его "одноклассник" *TCP*, но он проще для понимания, поэтому мы начнем изучение транспортных протоколов с него. Коротко *UDP* можно описать как ненадежный протокол без соединения, основанный на дейтаграммах. Теперь рассмотрим каждую из этих характеристик подробнее. *UDP* не имеет никаких дополнительных средств управления пакетами по сравнению с *IP*. Это значит, что пакеты, отправленные с помощью *UDP*, могут теряться, дублироваться и менять порядок следования. В сети без маршрутизаторов ничего этого с

пакетами почти никогда не происходит, и *UDP* может условно считаться надежным протоколом. Сети с маршрутизаторами строятся, конечно же, таким образом, чтобы подобные случаи происходили как можно реже, но полностью исключить их, тем не менее, нельзя. Происходит это из-за того, что передача данных может идти несколькими путями через разные маршрутизаторы. Например, пакет может пропасть, если короткий путь к удаленному узлу временно недоступен, а в длинном приходится пройти больше маршрутизаторов, чем это разрешено. Дублироваться пакеты могут, если они ошибочно передаются двумя путями, а порядок следования может изменяться, если пакет, посланный первым, идет по более длинному пути, чем пакет, посланный вторым.

Все сказанное отнюдь не означает, что на основе *UDP* нельзя построить надежный обмен данными, просто заботу об этом должно взять на себя само приложение. Каждый исходящий пакет должен содержать порядковый номер, и в ответ на него должен приходить специальный пакет — квитанция, которая уведомляет отправителя, что пакет доставлен. При отсутствии квитанции пакет высылается повторно (для этого необходимо ввести тайм-ауты на получение квитанции). Принимающая сторона по номерам пакетов восстанавливает их исходный порядок.

UDP не поддерживает соединение. Это означает, что при использовании этого протокола можно в любой момент отправить данные по любому адресу без необходимости каких-либо предварительных действий, направленных на установление связи с адресатом. Это напоминает процесс отправки обычного письма: на нем пишется адрес, и оно опускается в почтовый ящик без каких-либо предварительных действий. Такой подход обеспечивает большую гибкость, но лишает систему возможности автоматической проверки исправности канала связи.

Дейтаграммами называются пакеты, которые передаются как единое целое. Каждый пакет, отправленный с помощью *UDP*, составляет одну дейтаграмму. Принятые дейтаграммы складываются в буфер принимающего сокета и могут быть получены только раздельно: за одну операцию чтения из буфера программа, использующая сокет, может получить только одну дейтаграмму. Если в буфере лежит несколько дейтаграмм, потребуется несколько операций чтения, чтобы прочитать все. Кроме того, одну дейтаграмму нельзя получить из буфера по частям: она должна быть прочитана целиком за одну операцию.

Чтобы данные, передаваемые разным сокетам, не перемешивались, каждый сокет должен получить уникальный в пределах узла номер от 0 до 65 535, называемый номером порта. При отправке дейтаграммы отправитель указывает *IP*-адрес и порт получателя, и система принимающей стороны находит сокет, привязанный к указанному порту, и помещает данные в его буфер. По сути дела, *UDP* является очень простой надстройкой над *IP*, все функции которой заключаются в том, что физический поток разделяется на несколько логических с помощью портов, и добавляется проверка целостности данных с помощью контрольной суммы (сам по себе протокол *IP* не гарантирует отсутствия искажений данных при передаче).

Максимальный размер одной дейтаграммы *IP* равен 65 535 байтам. Из них не менее 20 байтов занимает заголовок *IP*. Заголовок *UDP* имеет размер 8 байтов. Таким образом, максимальный размер одной дейтаграммы *UDP* составляет 65 507 байтов.

Типичная область применения *UDP* — программы, для которых потеря пакетов некритична. Например, некоторые сетевые 3*D*-игры в локальной сети используют *UDP*, т. к. очень часто посылают пакеты, информирующие о действиях игрока, и потеря одного пакета не приведет к существенным проблемам: следующий пакет доставит необходимые данные. Достоинства *UDP* — простота установления связи, возможность обмена данными с несколькими адресами через один сокет и отсутствие необходимости возобновлять соединение после разрыва связи. В некоторых задачах также очень удобно то, что дейтаграммы не смешиваются, и получатель всегда знает, какие данные были отправлены одной дейтаграммой, а какие — разными.

Еще одно преимущество *UDP* — возможность отправки широковещательных дейтаграмм. Для этого нужно указать широковещательный *IP*-адрес (обычно это 255.255.255.255, но в некоторых случаях допустимы адреса типа 192.168.100.225 для вещания в пределах сети 192.168.100.XX и т. п.), и такую дейтаграмму получат все сокеты в локальной сети, привязанные к заданному порту. Эту возможность нередко используют программы, которые заранее не знают, с какими компьютерами они должны связываться. Они посылают широковещательное

сообщение и связываются со всеми узлами, которые распознали это сообщение и прислали на него соответствующий ответ. По умолчанию для широковещательных пакетов число маршрутизаторов, через которые они могут пройти (*TTL*), устанавливается равным нулю, поэтому такие пакеты не выходят за пределы подсети.

Протокол *TCP*

Протокол *TCP* (*Transmission Control Protocol* — протокол управления передачей) является надежным потоковым протоколом с соединением, т. е. полной противоположностью *UDP*. Единственное, что у этих протоколов общее, — это способ адресации: в *TCP* каждому сокету также назначается уникальный номер порта. Уникальность номера порта требуется только в пределах протокола: два сокета могут иметь одинаковые номера портов, если один из них работает через *TCP*, а другой — через *UDP*.

В *TCP* предусмотрены так называемые хорошо известные (*well-known*) порты, которые зарезервированы для нужд системы и не должны использоваться программами. Стандарт *TCP* определяет диапазон хорошо известных портов от 0 до 255, в *Windows* и в некоторых других системах этот диапазон расширен до 0—1023. Часть портов *UDP* тоже выделена для системных нужд, но зарезервированного диапазона в *UDP* нет. Кроме того, некоторые системные утилиты используют порты за пределами диапазона 0—1023. Полный список системных портов для *TCP* и *UDP* содержится в *MSDN*, в разделе *Resource Kits*/*Windows* 2000 *Server Resource Kit*/*TCP*/*IP Core Networking*/*Appendixes*/*TCP and UDP Port Assignment*.

Для отправки пакета с помощью *TCP* отправителю необходимо сначала установить соединение с получателем. После выполнения этого действия соединенные таким образом сокеты могут использоваться только для отправки сообщений друг другу. Если соединение разрывается (самой программой или из-за проблем в сети), эти сокеты уже непригодны для установления нового соединения: они должны быть уничтожены, а вместо них созданы новые сокеты.

Механизм соединения, принятый в *TCP*, подразумевает разделение ролей соединяемых сторон: одна из них пассивно ждет, когда кто-то установит с ней соединение, и называется сервером, другая самостоятельно устанавливает соединение и называется клиентом. Действия клиента по установке связи заключаются в следующем: создать сокет, привязать его к адресу и порту, вызвать функцию для установки соединения, передав ей адрес сервера. Если все эти операции выполнены успешно, то связь установлена, и можно начинать обмен данными. Действия сервера выглядят следующим образом: создать сокет, привязать его к адресу и порту, перевести в режим ожидания соединения и дождаться соединения. При соединении система создаст на стороне сервера специальный сокет, который будет связан с соединившимся клиентом, и обмениваться данными с подключившимся клиентом сервер будет через этот новый сокет. Старый сокет останется в режиме ожидания соединения, и другой клиент сможет к нему подключиться. Для каждого нового подключения будет создаваться новый сокет, обслуживающий только данное соединение, а исходный сокет будет по-прежнему ожидать соединения. Это позволяет нескольким клиентам одновременно соединяться с одним сервером, а серверу — не путаться в своих клиентах. Точное число клиентов, которые могут одновременно работать с сервером, в документации не приводится, но оно достаточно велико.

Установление такого соединения позволяет осуществлять дополнительный контроль прохождения пакетов. В рамках протокола *TCP* выполняется проверка доставки пакета, соблюдения очередности и отсутствия дублей. Механизмы обеспечения надежности достаточно сложны, и мы их здесь рассматривать не будем. Программисту для начала достаточно знать, что данные, переданные с помощью *TCP*, не теряются, не дублируются и доставляются в том порядке, в каком были отправлены. В противном случае отправитель получает сообщение об ошибке. Соединенные сокеты время от времени обмениваются между собой специальными пакетами, чтобы убедиться в наличии соединения.

Если из-за неполадок в сети произошел разрыв связи, при попытке отправить данные или прочитать их клиент получит отказ, а соединение будет разорвано. После этого клиент должен уничтожить сокет, создать новый и повторить подключение. Сервер также получает ошибку на сокете, обслуживающем данное соединение, но существенно позже (эта задержка может достигать часа). При обнаружении ошибки сервер просто уничтожает сокет и ждет нового

подключения от клиента. Возможна ситуация, когда клиент уже подключился заново и для него создан новый сокет, а старый сокет еще не закрыт. Это не является существенной проблемой — на старом сокете рано или поздно будет получена ошибка, и он будет закрыт. Тем не менее, сервер может учитывать такую ситуацию и уничтожать старый сокет, не дожидаясь, пока на нем будет получена ошибка, если новое соединение с тем же клиентом уже установлено. На исходный сокет, находящийся в режиме ожидания подключения, физические разрывы связи никак не влияют, после восстановления связи никаких действий с ним проводить не нужно.

Если на клиентской стороне не удалось для нового сокета установить соединение с сервером с первого раза (из-за отсутствия связи или неработоспособности сервера), этот сокет не обязательно уничтожать: он может использоваться при последующих попытках установления связи неограниченное число раз, пока связь не будет установлена.

Протокол *TCP* называется потоковым потому, что он собирает входящие пакеты в один поток. В частности, если в буфере сокета лежат 30 байтов, принятые по сети, не существует возможности определить, были ли эти 30 байтов отправлены одним пакетом, 30 пакетами по 1 байту или еще как-либо. Гарантируется только то, что порядок байтов в буфере совпадает с тем порядком, в котором они были отправлены. Принимающая сторона также не ограничена в том, как она будет читать информацию из буфера: все сразу или по частям. Это существенно отличает *TCP* от *UDP*, в котором дейтаграммы не объединяются и не разбиваются на части.

Склеивание пакетов осуществляется не только принимающей, но и отправляющей стороной. Библиотека сокетов может придержать в выходном буфере то, что кладет программа, и потом отправить одним пакетом данные, которые программа складывала в буфер постепенно. И наоборот, данные большого объема могут быть отправлены по частям, поэтому возможна ситуация, когда принимающая сторона находит в своем буфере только часть сообщения, посланного своим визави. Это значит, что оставшаяся часть сообщения придет позже. Будут ли пакеты сливаться или разбиваться на части, зависит от пропускной способности и текущей загрузке сети, определяется это алгоритмом, который называется алгоритм Нагля.

TCP применяется там, где программа не хочет заботиться о проверке целостности данных. За отсутствие этой проверки приходится расплачиваться более сложной процедурой установления и восстановления связи. Если при использовании *UDP* сообщение не будет отправлено из-за проблем в сети или на удаленной стороне, никаких действий перед отправкой следующего сообщения выполнять не нужно и можно использовать тот же сокет. В случае же *TCP*, как уже было сказано, необходимо сначала уничтожать старый сокет, затем создать новый и подключить его к серверу, и только потом можно будет снова отправлять сообщения. Другим недостатком *TCP* по сравнению с *UDP* является то, что через один TCP-сокет все пакеты отправляются только по одному адресу, в то время как через *UDP*-сокет разные пакеты могут быть отправлены по разным адресам. И наконец, *TCP* не позволяет рассылать широковещательные сообщения. Но, несмотря на эти неудобства, *TCP* применяется существенно чаще *UDP*, потому что автоматическая проверка целостности данных и гарантия их доставки является очень важным преимуществом. Кроме того, *TCP* гарантирует более высокую безопасность в Интернете (это связано с тем, что обеспечить безопасность при передаче данных легче при наличии соединения, а не в ситуации, когда пакеты могут передаваться от кого угодно кому угодно).

То, что *TCP* склеивает данные в один поток, не всегда удобно. Во многих случаях пакеты, приходящие по сети, обрабатываются отдельно, поэтому читать их из буфера желательно тоже по одному. Это просто сделать, если все пакеты имеют одинаковую длину. Но при различной длине пакетов принимающая сторона заранее не знает, сколько байтов нужно прочитать из буфера, чтобы получить ровно один пакет и ни байта больше. Чтобы обойти эту ситуацию, в пакете можно предусмотреть обязательный заголовок фиксированной длины, одно из полей которого хранит длину пакета. В этом случае принимающая сторона может читать пакет по частям: сначала заголовок известной длины, а потом тело пакета, размер которого стал известен благодаря заголовку. Другой способ разделения пакетов — вставка между ними заранее оговоренной последовательности байтов, которая не может появиться внутри пакета.

Но самое неудобное то, что пакеты не только склеиваются, но и разбиваются на части. Принимающая сторона может получить пакет меньшего размера, чем отправленный, если этот

пакет был послан по частям, и на момент его чтения принимающей стороной еще не все части были получены. Тогда приходится повторять операцию чтения данных, пока не будет получено все, что нужно.

В отличие от *UDP*, в *TCP* данные, которые программа отправляет одной командой, могут разбиваться на части и отправляться несколькими *IP*- пакетами. Поэтому ограничение на длину данных, отправляемых за один раз, в *TCP* отсутствует (точнее, определяется доступными ресурсами системы). Количество данных, получаемое отправителем за одну операцию чтения, ограничено размером низкоуровнего буфера сокета и может быть разным в различных реализациях. Следует иметь в виду, что при переполнении буфера принимающей стороны протокол *TCP* предусматривает передачу отправляющей стороне сигнала, по которому она приостанавливает отправку, причем этот сигнал прерывает всю передачу данных между этими двумя компьютерами с помощью *TCP*, т. е. это может повлиять и на другие программы. Поэтому желательно не допускать таких ситуаций, когда у принимающей стороны в буфере накапливается много данных.

Некоторые порты *TCP* зарезервированы для стандартного использования конкретными протоколами высокого уровня и службами. Другими словами, следует использовать эти номера портов при реализации этих служб и избегать их использования во всех остальных случаях.

Таблица 5.1 — Зарезервированные порты

Протокол	Порт
HTTP (*Hypertext Transfer Protocol*)	80
FTP (*File Transfer Protocol*)	21
SMTP (*Simple Mail Transfer Protocol*)	25
*POP*3 (*Post Office Protocol, version* 3)	110
Telnet	23

В файле *C*:*WINDOWS**system*32*drivers**etc**services* перечислены стандартные порты, используемые службами. Сокеты клиентов всегда указывают номер порта или имя службы сокета сервера, с которым им требуется установить соединение.

Протокол — это набор правил для клиента и сервера, определяющий коммуникационный поток. Низкоуровневые протоколы Интернета, такие как *TCP*/*IP*, обычно реализуются операционной системой. Однако термин «протокол» также применяется и для высокоуровневых стандартных протоколов Интернета (таких как *HTTP*, *FTP* или *SMTP*). Прикладные протоколы расположены на более высоком уровне, чем протоколы передачи, поскольку они абстрагированы от транспортного механизма, предоставляемого *TCP*/*IP*. Это делает протоколы независимыми не только от операционной системы и аппаратного обеспечения, но также и от физической структуры сети.

В данной работе будем рассматривать протокол *TCP* и работать с ним посредством компонентов *ServerSocket* и *ClientSocket*.

5.3. Методические рекомендации к выполнению работы

Коммуникация через сокет происходит в несколько этапов. Сначала начинает работать программа-сервер, но она просто ожидает запроса от клиента. Программа-клиент запрашивает соединение у сервера, с которым ей требуется установить соединение. Когда клиент отправляет запрос, сервер может принять соединение, открыв специальный сокет на стороне сервера, который образует соединение с сокетом на стороне клиента.

Для поддержки этой модели существуют три типа сокетных соединений:

- клиентские соединения инициируются клиентом и связывают локальный сокет клиента с удаленным сокетом сервера. Клиентские сокеты должны описывать сервер, с которым им требуется установить соединение, предоставив либо имя узла, либо *IP*-адрес и его порт;
- слушающие соединения (*listening connections*) — это пассивные серверные сокеты, ожидающие клиента. Как только клиент делает новый запрос, сервер активизирует новый сокет, выделенный для этого конкретного соединения, и затем возвращается к слушанию.

Слушающие серверные сокеты должны указывать порт, представляющий реализуемую ими службу (фактически клиент собирается подключаться через этот порт);

- серверные соединения — это соединения, активизируемые серверами, поскольку они принимают запрос от клиента.

Различные типы соединений имеют значение только для установления канала от клиента к серверу. После того как канал связи установлен, обе стороны могут обмениваться данными и запросами друг с другом в произвольном порядке.

При работе с сокетами в *Windows* возможны несколько подходов. Чтение данных из сокета или запись в него могут осуществляться асинхронно, поэтому это не блокирует выполнение другого кода в сетевом приложении. Такое соединение называется неблокирующим (*nonblocking connection*). Неблокирующие соединения производят чтение и запись асинхронно. При работе, например, с компонентами *ClientSocket* и *ServerSocket*, система генерирует на клиенте события *OnRead* или *OnWrite*, а события *OnClientRead* или *OnClientWrite* серверов информируют сокет в тот момент, когда другая сторона соединения пытается прочитать или записать данные.

Альтернатива асинхронному подходу — блокирующие соединения, при которых приложение ожидает завершения операций чтения или записи перед выполнением следующей строки кода. В этом случае следует согласованно писать код для обеих сторон, поскольку иначе не будут генерироваться события. При работе с блокирующим соединением необходимо создать на сервере поток и, как правило, работать с потоком также и на стороне клиента.

Рассмотрим работу с сокета на простейшем примере.

Для работы с сокетами в поставку *Builder* входят наборы сокетных компонентов, при помощи которых можно читать и записывать информацию поверх соединения *TCP/IP*. На странице *Internet* палитры компонентов расположены компоненты *ClientSocket* и *ServerSocket*, а также новые компоненты *TcpClient* и *TcpServer*.

Для использования сокетного компонента необходимо предоставить хост и службу. На стороне сервера хост представляет собой адрес текущего компьютера, на стороне клиента вы можете указать либо имя домена, либо *IP*-адрес.

Начнем с написания серверной части. Для этого на форму вынесем поле *Memo*1 и компонент *ServerSocket*.

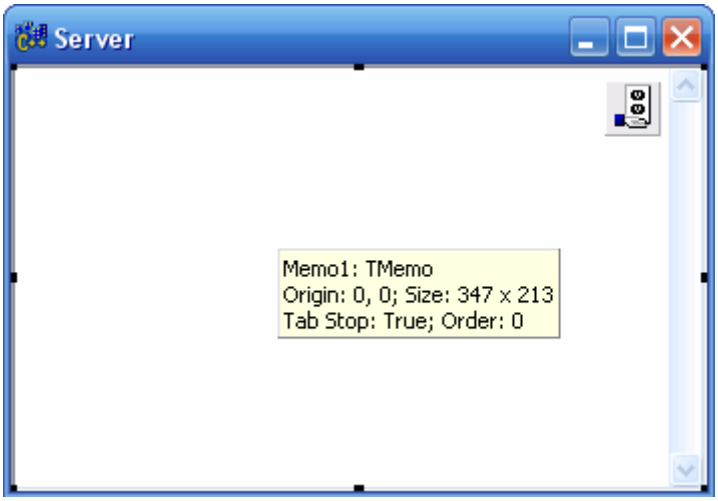


Рисунок 5.1 — Главное окно серверного приложения

Активацию серверного сокета будем производить при запуске формы, для этого в событии *Activate* пишем код:

```
void __fastcall TForm1::FormActivate(TObject *Sender)
{
 ServerSocket1->Port=5555;
 ServerSocket1->Active=true;
}
```

Таким образом, мы задали свой порт сервера (5555) и активировали сокет. Для обработки событий, возникающих при работе сокета, обратимся к испектору объектов.

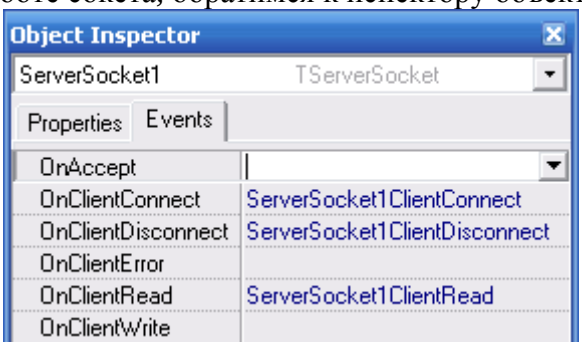


Рисунок 5.2 — События компонента *ServerSocket*

В данном примере мы воспользуемся простейшими событиями сокета:
OnClientConnect — возникает при подключении клиента
OnClientDisconnect — соответственно, возникает при завершении работы с клиентом
OnClientRead — возникает при приеме данных от клиента

Код для каждого из этих событий может выглядеть примерно так:

```
void __fastcall TForm1::ServerSocket1ClientConnect(TObject *Sender,
      TCustomWinSocket *Socket)
{
 Memo1->Lines->Add(DateTimeToStr(Now())+
                   " Connect: " + Socket->RemoteAddress);
}
//---------------------------------------------------------------------
void __fastcall TForm1::ServerSocket1ClientRead(TObject *Sender,
      TCustomWinSocket *Socket)
{
 Memo1->Lines->Add(Socket->RemoteAddress+" -> "+
                   Socket->ReceiveText());
 Socket->SendText("OK");
}
//---------------------------------------------------------------------
void __fastcall TForm1::ServerSocket1ClientDisconnect(TObject *Sender,
      TCustomWinSocket *Socket)
{
 Memo1->Lines->Add(DateTimeToStr(Now())+" Disconnect: " +
                   Socket->RemoteAddress);
}
```

Обратите внимание, что при приеме данных от клиента они считываются функцией *ReceiveText*(), после которой идет отправка сообщения “OK” клиенту, чтобы тот был к курсе доставки сообщения. Также не забываем отключить сокет при завершении работы сервера (в *FormClose*) *ServerSocket*1*->Active=false*;

На этом создание простейшего сервера завершено, переходим к клиенту.

Вынесем на форму поле *Edit*1 и 3 кнопки - для соединения/отключения и передачи данных, а также сам *ClientSocket*.

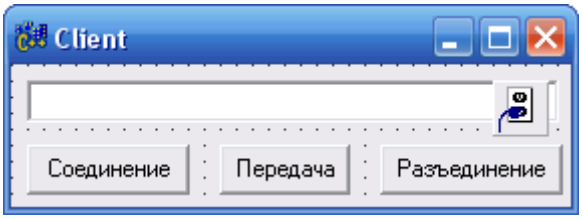


Рисунок 5.3 — Главное окно клиентского приложения

Компонент *ClientSocket* для адресации использует два различных свойства (*Host* и *Address*), а служба указывается в свойстве *Port* или *Service*. В данном примере запуск и клиента и сервера будет происходить на одной машине, поэтому в качестве адреса сервера вставляем *localhost* (127.0.0.1) и указываем 5555 порт сервера.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 ClientSocket1->Address="127.0.0.1";
 ClientSocket1->Port=5555;
 ClientSocket1->Active=true;
}
```

Настройки сокета и его активация производятся при нажатии кнопки “Соединение“. Для кнопок “Передача” и “Разъединение” используется код:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
 ClientSocket1->Socket->SendText(Edit1->Text);
}
//---------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
 ClientSocket1->Active=false;
}
```

При этом функцией *SendText*() от клиента на сервер передается содержимое поля ввода *Edit*1.

Для проверки доставки воспользуемся событием *ClientSocket*1*Read,* которое возникает при получении данных клиентским сокетом. Принимаемое от сервера сообщение “ОК” с помощью функции *ReceiveText*() помещается обратно в *Edit*1.

```
void __fastcall TForm1::ClientSocket1Read(TObject *Sender,
      TCustomWinSocket *Socket)
{
 Edit1->Text=Socket->ReceiveText();
}
```

Программа готова, запускаем сервер и клиент, наблюдаем корректный процесс работы (если есть межсетевой экран, включаем разрешение для клиента)

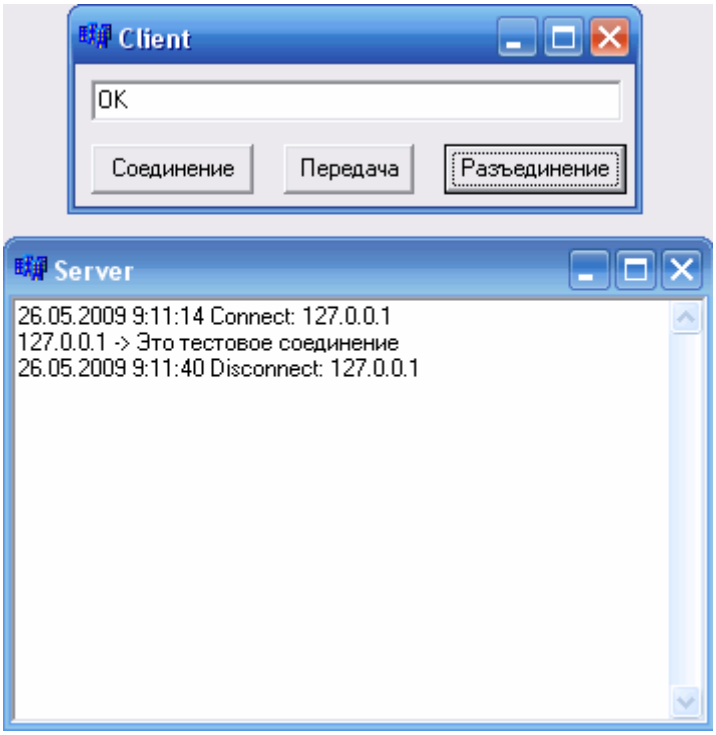
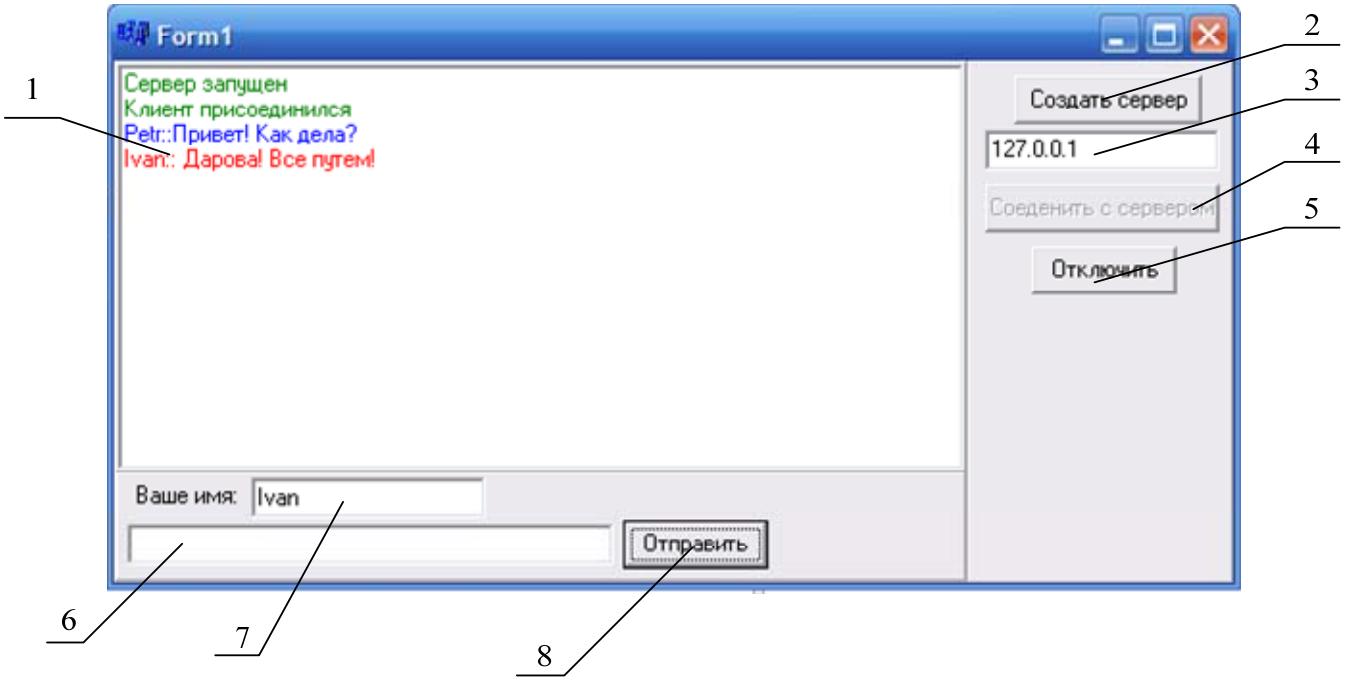


Рисунок 5.4 — Пример работы программы

Следует учитывать, что если программа должна иметь минимальный размер и ее код не должен сильно зависеть от языка, то необходимо пользоваться не стандартными компонентами, а специальными *API*-функциями для работы с сетью. В среде *Windows* есть отдельный раздел таких функций - *WINSOCKETS API*.

В данной работе в отличии от примера приведенного выше разрабатываемое приложение должно совмещать функции сервера и клиента. Рекомендуемый вид приложения изображен на рисунке 5.5.



1 — компонент *Memo*1 для вывода сообщений собеседника и сообщений программы;
2 — компонент *Button*2 для открытия сервеного сокет и прослушивания порта номер которого задается в индивидуальном задании;
3 — компонент *Edit*3 для указания *IP* адреса удаленного сервера с которым требуется соеденится с помощью компонента 4;
4 — компонент *Button*3 для соеденения с удаленным севером адрес которого указан в компоненте 3;

5 — компонент *Button*4 для закрытия всех соеденений;
6 — компонент *Edit*1 сообщение чата которое требуется передать собеседнику посредством компонента 8;
7 — компонент *Edit*2 для ввода собственного имени которое будет отображаться в чате;
8 — компонент *Button*1 для отправки сообщения собеседнику;

Рисунок 5.5 — Главное окно программы чат

Для начала необходимо создадать новый проект(*file->new->application*), поместим на форму компоненты:

tclientsocket и *tserversocket* , чтобы программа могла быть и клиентом и сервером (не одновременно).

Далее разместим компонент *Memo*1 (закладка *Standart*) — в нем будет отображаться текст чата.

Следующим на форму нужно добавит компонент *Edit*1 (*Standart*) — в него мы будем писать текст, который нужно отправить собеседнику.

Ну и конечно тяжело обойтись без кнопки отправить – добавляем на форму *Button*1 . Кроме того что уже есть на форме, нам еще понадобится три кнопки *Button* и два компонента *Edit* (для ввода имени собеседника и адреса сервера) .

Итак, на форме :

- *ClientSocket*1 и *ServerSocket*1
- *Memo*1
- *Edit*1,*Edit*2,*Edit*3
- *Button*1,*Button*2,*Button*3,*Button*4

Теперь изменяем свойства:
*Button*1->*Caption* на "Отправить"
*Button*2->*Caption* на "Создать сервер"
*Button*3->*Caption* на "Соединить с сервером" и
*Button*4->*Caption* на "Отключить" .
Убираем текст во всех компонентах *Edit*. Свойство *Memo*1->*Readonly = true* ,

*Clientsocket*1->*Host* — нужно написать *ip*-адрес сервера к которому вы будете присоеденяться (*ip*-адресс устанавливается в настройках соединения *windows*), если прописать 127.0.0.1 , то вы будете конектиться к себе на компьютер (так удобно делать, когда проверяешь на работоспособность свою программу. Запустив ее дважды, одна клиент с 127.0.0.1 , а другая сервер) если же вы коннектитесь к другому компьютеру, то заранее договоритесь какой будет адрес (143.0.0.5 — например). Но для того чтобыадресс можно было легко сменить, мы и добавили на форму *Edit*3, его текст при коннекте и будет отвечать свойству *ClientSocket*1->*Host* и *ClientSocket*1->*Address* .

В свойстве *ClientSocket*1->*Port* и *ServertSocket*1->*Port* — должны стоять одинаковые значения, чтобы Сервер и Клиент прослушивали и работали на один порт . Число выбирается согласно индивидуального задания (1024 например).

Кнопку "Отключиться" изначально нужно сделать недоступной(*Enabled = false*)так как вначале отсоеденяться нам нет от кого.

Дальше опишем обработчики событий для кнопок "Создать", "Соединиться", "Отключить" .

Кнопка "Создать" — активизирует сервер. Он начинает прослушивать порт на коннект со стороны клиента .

```
void __fastcall tform1::Button2Click(tobject *sender)
{
ServerSocket1->Active = true ;
// Делаем недоступную "Соединиться" (так как мы уже сервер)
Button3->Enabled = false;
// Делаем доступную "Отключиться"
Button4->Enabled = true;
Memo1->Lines->Add("Сервер создан") ;
}
```

Так программа стала сервером.

Теперь опишем клиента (Кнопка "Соединить с сервером")

В *Edit*3->*Text* впишите 127.0.0.1 — предполагается что тестироваться будет на одном компьютере

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
ClientSocket1->Host= Edit3->Text;   // Присваиваем Клиенту IP из Edit3
ClientSocket1->Address=Edit3->Text ;
ClientSocket1->Active = true ;
// Делаем недоступную "Создать" (так как мы коннектимся)
Button2->Enabled = false ;
// Делаем доступную "Отключиться"
Button4->Enabled = true ;
}
```

Дальше будем описывать события компонентов Клиента и Сервера *OnConnect* (когда присоединился) .

```
void __fastcall TForm1::ServerSocket1ClientConnect(TObject *Sender,
     TCustomWinSocket *Socket)
{
Memo1->Lines->Add("Клиент присоединился");
}
```

Это когда вы сервер и к Вам присоединились, на Мемо появится надпись. Аналогично для клиента:

```
void __fastcall TForm1::ClientSocket1Connect(TObject *Sender,
     TCustomWinSocket *Socket)
{
Memo1->Lines->Add("Подключен к серверу");
}
```

Итак, *Edit*2 нужен для Вашего ника, так как требуется идентификация. Теперь самое главное — описание кнопки "Отправить :

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
if(Edit2->Text == "") {
ShowMessage("Введите Ваш ник !");
return ;
}
if(Edit1->Text == "")
{
ShowMessage("Введите текст который надо отправить");
return ;
}
//Это была обработка исключительных ситуаций , типа пустых строк ввода ;
Memo1->Lines->Add(Edit2->Text+":: "+ Edit1->Text) ; //добавляем свое сообщение себе в Мемо
if (ServerSocket1->Active == true) {
// Если мы сервер, то посылаем нашу строку первому в списке клиенту ( чат розщитан на двоих ) , иначе :
  ServerSocket1->Socket->Connections[0]->
   SendText(Edit2->Text+"::"+Edit1->Text); }
else // Посылаем строку серверу
```

```
{ ClientSocket1->Socket->SendText(Edit2->Text+"::"+Edit1->Text);}
Edit1->Text = "" ; // Очищаем Edit1
}
```

Также надо описать прием информации и занесение ее в Memo1. Делается это обработчиком события *OnRead* у *ClientSocket* и *ServerSocket* :

```
void __fastcall TForm1::ClientSocket1Read(TObject *Sender,
    TCustomWinSocket *Socket)
{
Memo1->Lines->Add(Socket->ReceiveText()) ;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::ServerSocket1ClientRead(TObject *Sender,
    TCustomWinSocket *Socket)
{
Memo1->Lines->Add(Socket->ReceiveText()) ;
}
```

Обработчик события кнопки отключить выглядит следующим образом:

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
if(ServerSocket1->Active) Memo1->Lines->Add("Сервер остановлен"); else
Memo1->Lines->Add("Отсоеденен от сервера");
ClientSocket1->Active=false;
ServerSocket1->Active=false;
Button4->Enabled = false ;
Button2->Enabled = true ;
Button3->Enabled = true ;
}
```

5.4. Индивидуальные задания

Согласно варианту полученному у преподавателя выбрать номер порта для написания приложения из табл. 5.2.

Таблица 5.2 — Индивидуальное задание

Вариант	Номер порта для чата
0	1025
1	1027
2	1029
3	1031
4	1033
5	1035
6	1037
7	1039
8	1041
9	1043

5.5. Содержание отчета

Отчет должен содержать:

1. Титульный лист;
2. Цель работы;
3. Описание и алгоритм работы программы;
4. Текст программы и результаты разработки программы;
5. Результаты работы программы;
6. Выводы.

Перед защитой отчета студент должен продемонстрировать работу программы.

5.6. Контрольные вопросы

5.6.1. Какие свойства используются при создании приложения-сервера с использованием компонента *TServerSocket*?

5.6.2. Какие свойства используются при создании приложения-клиента с использованием компонента *TClientSocket*?

5.6.3. С помощью какого метода можно получить данные с сервера по инициативе клиента ?

5.6.4. С помощью какого метода клиент может переслать текстовые данные на сервер ?

5.6.5. Куда в приложении-клиенте поступают данные, пересылаемые с сервера?

5.6.6. Какие методы используются для передачи информации от клиента серверу?

5.6.7. С помощью какого метода можно получить данные с сервера по инициативе клиента?

5.6.8. Как организовать постоянное отслеживание информации на сервере в приложении-клиенте?

5.6.9. Какое событие в приложении-сервере может быть использовано для определения того, что от клиента серверу послана информация?

5.6.10. Какие способы установления контакта с сервером Вы знаете?

5.6.11. Что нужно сделать, чтобы при запуске приложения-клиента автоматически запускалось и приложение-сервер?