ШАГ компьютерная академия

СЕТЕВОЕ ПРОГРАММИРОВАНИЕ

СЕТЕВОЕ ПРОГРАММИРОВАНИЕ В .NET FRAMEWORK

TCP И UDP СОКЕТЫ, UNICAST, BROADCAST, MULTICAST

ИСПОЛЬЗОВАНИЕ СЕТЕВЫХ ПРОТОКОЛОВ HTTP, SMTP, FTP

Урок №3

TCP и UDP сокеты, unicast, broadcast, multicast

Содержание

1. Обзор TCP протокола

TCP или *Transmission Control Protocol* (протокол управления передачей) применяется как надежный протокол, обеспечивающий взаимодействие через сеть компьютеров. TCP проверяет, доставляются ли данные адресату надлежащим образом.

TCP является протоколом, ориентированным на соединения. Это означает, что два процесса или приложения, прежде чем начать обмениваться данными должны установить TCP — соединение. В этом протокол TCP отличается от протокола UDP. Последний является протоколом «без организации соединения», такой протокол позволяет выполнять широковещательную передачу данных неопределенному числу клиентов. Протокол TCP был специально разработан для обеспечения надежного сквозного байтового потока по ненадежной интрасети. Такая сеть отличается от типовой локальной сети тем, что ее различные участки могут обладать сильно различающейся топологией, пропускной способностью, временами задержки размерами пакетов и т.п. При разработке TCP основное внимание уделялось способности протокола адаптироваться к свойствам объединенной сети и отказоустойчивости при возникновении различных проблем при передаче данных.

Протокол TCP описан в RFC 793. Со временем были обнаружены различные ошибки и неточности и по некоторым пунктам требования стандарта были изменены.

Подробное описание этих уточнений дается в стандарте RFC 793. расширения протокола описаны в RFC 1323.

TCP используется в таких прикладных протоколах, как HTTP, FTP, SMTP и Telnet.

1.1. Терминология TCP

- ***Сегмент*** — «порция» данных, который TCP отправляет IP
- ***Дейтаграмма*** — «порция»данных, которую IP отправляет на уровень сетевого интерфейса
- ***Порядковый номер*** — каждому сегменту TCP, которое отправляется через соединение, присваивается номер, для того чтобы приложение клиент могло, в конечном счете, получить данные в правильном порядке.

Каждая машина, поддерживающая протокол TCP, обладает т.н. транспортной сущностью TCP. Это либо библиотечная процедура, либо пользовательский процесс, либо часть ядра системы. В любом случае транспортная сущность управляет TCP-потоками и обменом данными с IP-уровнем (IP-уровень в модели OSI находится на 3-м, сетевом уровне). TCP-сущность принимает от локальных процессов пользовательские потоки данных, разбивает их на фрагменты, не более 64 Кбайт (на практике это число обычно равно 1460 байт, что позволяет поместить их в один кадр Ethernet с заголовками IP и TCP) и посылает их в виде отдельных IP-дейтаграмм. Когда IP-дейтаграммы с TCP-данными прибывают на машину, они передаются TCP-сущности, которая восстанавливает исходный байтовый поток. Иногда слово

TCP употребляется для обозначения транспортной сущности TCP, т.е. части программного обеспечения, а иногда для обозначения протокола TCP, т.е. набора правил для передачи данных.

Уровень IP не гарантирует правильной доставки дейтаграмм, поэтому именно TCP приходится следить за истекшими интервалами ожидания и в случае необходимости повторно передавать пакеты. Часто дейтаграммы прибывают в неправильном порядке. Восстановить сообщения из таких дейтаграмм также должен протокол TCP. Иными словами, протокол TCP призван обеспечить надежность, которую подразумевают современные приложения и которую не гарантирует протокол IP.

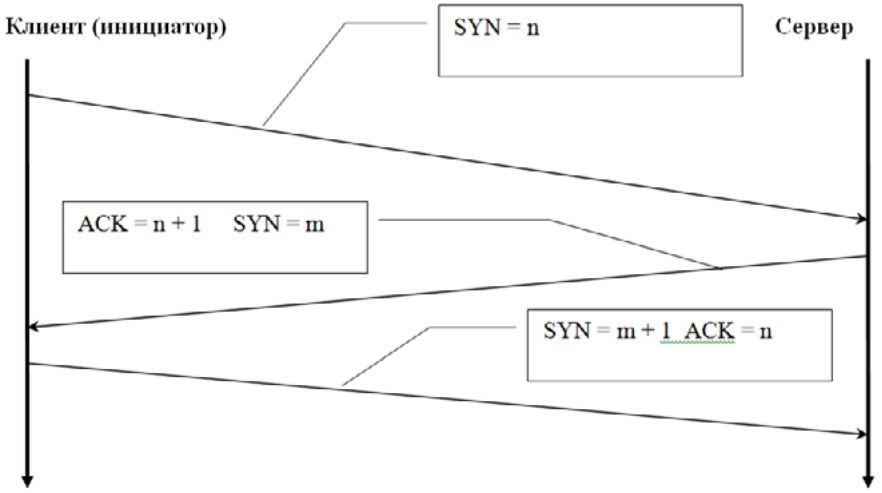
Соединения TCP

Для установки соединения TCP применяется последовательность действий, которую называют «трехфазным квитированием» — Three-Phase Handshake. Последовательность состоит из таких шагов:

1. Клиент инициализирует взаимодействие с сервером, посылая сегмент с установленным битом SYN (синхронизация). Этот сегмент данных содержит начальный порядковый номер клиента.
2. В ответ сервер отвечает отправкой сегмента с установленными битами SYN (синхронизация) и ACK (подтверждение). Этот сегмент содержит начальный порядковый номер сервера (не связанный с порядковым номером клиента) и номер подтверждения, на единицу больший порядкового номера клиента. Иными словами, номер подтверждения равен

следующему порядковому номеру, ожидаемому от клиента.

3. Клиент должен подтвердить получение сегмента отправкой обратно сегмента с установленным битом ACK(подтверждение). Номер подтверждения будет на единицу больше порядкового номера сервера, а порядковый номер будет равен номеру подтверждения сервера.

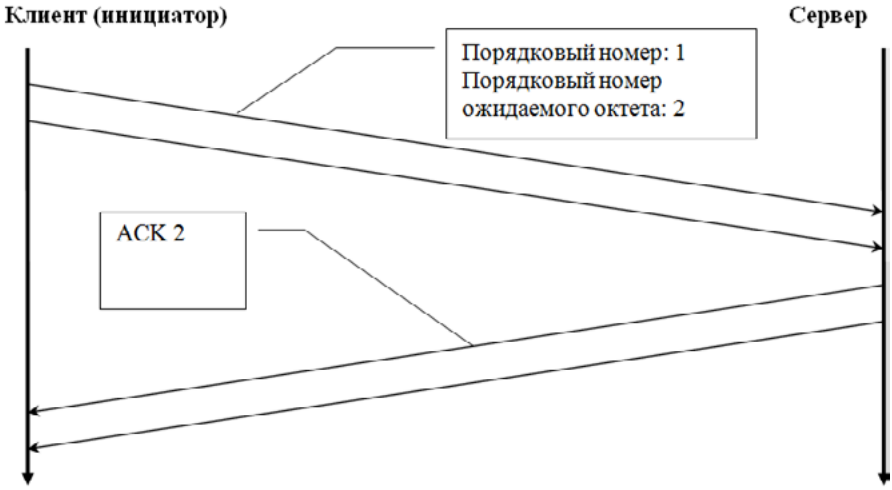


Потоковая передача данных в TCP

Протокол TCP передает данные порциями, которые называются сегментами. Чтобы гарантировать получение сегментов в правильном порядке каждому из них назначается порядковый номер. Получатель отправляет подтверждение получения сегмента. Если подтверждение не получено до истечения интервала тайм-аута, данные будут отправлены еще раз. Каждым восьми битам

данных (т.н. октету) присваивается порядковый номер. Порядковый номер сегмента равен порядковому номеру первого октета данных в сегменте, это число отправляется в заголовке TCP данного сегмента. В сегменте может также присутствовать номер подтверждения, равный порядковому номеру следующего ожидаемого сегмента данных

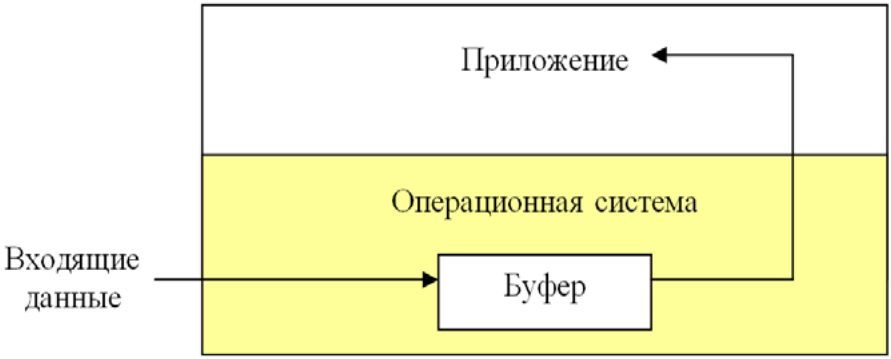
TCP использует порядковые номера, чтобы гарантировать, что дублирующие данные не будут преданы приложению, и данные будут доставлены в правильном порядке. Заголовок TCP содержит контрольную сумму, чтобы гарантировать корректность данных, доставленных получателю. Если же к получателю доставлен сегмент с неверной контрольной суммой, то такой сегмент отбрасывается, и подтверждение о получении этого сегмента не отравляется. Это приводит к тому, что после истечения тайм-аута отправитель вынужден, будет повторить передачу недоставленного сегмента.



Управление потоком данных

Протокол TCP управляет объемом направляемых ему данных при помощи указания о т.н. размере фрейма (окна) в каждом подтверждении. Размер фрейма — это объем данных, который может принять получатель. Между приложением и потоком данных в сети располагается специальный буфер данных. Размер фрейма фактически представляет собой разность между размером буфера и объемом сохраненных в нем данных. Это число отправляется в заголовке, чтобы информировать удаленный хост о текущем размере окна. Этот прием называется «раздвижным окном» — *sliding window*.

Алгоритмы раздвижного окна управляют потоком данных, передаваемых по сети:



Полученные данные сохраняются в этом буфере, приложение при этом может обращаться к буферу со свойственной ему скоростью. Эта скорость зависит от множества факторов (загрузка ЦПУ, приоритет самого приложения, приоритет потока, в котором происходит чтение буфера и т.п.). По мере чтения данных из буфера он опустоша-

ется и получает возможность записывать новые данные, поступившие из сети.

Мультиплексирование

TCP позволяет нескольким процессам на одной и той же машине одновременно использовать сокет TCP. Сокет TCP состоит из адреса хоста и уникального номера порта, а TCP-соединение включает два сокета на разных концах сети. Порт может использоваться для нескольких соединений одновременно, то есть сокет на одном конце соединения может использоваться для работы нескольких соединений с разными сокетами на другом конце. Типовым примером такой ситуации может быть web-сервер, слушающий на 80-м порту и отвечающий на запросы нескольких машин.

1.2. TCP заголовки

Структура заголовка TCP рассмотрена на схеме:

Заголовок TCP

<table>
<tr><td colspan="8">0 бит</td><td>16 бит
32 бита</td></tr>
<tr><td colspan="8">Порт-источник</td><td>Порт-адресат</td></tr>
<tr><td colspan="9">Порядковый номер</td></tr>
<tr><td colspan="9">Номер подтверждения</td></tr>
<tr><td>Длина TCP заголовка</td><td>Резерв</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Размер фрейма (окна)</td></tr>
<tr><td colspan="8">Контрольная сумма</td><td></td></tr>
<tr><td colspan="8">Опции</td><td>Заполнение</td></tr>
</table>

Порядковый номер и номер подтверждения используются TCP, чтобы гарантировать, что все данные прибывают в правильном порядке.

В четвертом байте после длины TCP заголовка и резервных битов находятся биты управления (флаги):

- ***URG*** — означает, что сегмент содержит срочные (urgent) данные
- ***ACK*** — означает, что сегмент содержит номер подтверждения (acknowledgement)
- ***PSH*** — означает, что данные нужно протолкнуть (push) к получателю
- ***RST*** — сброс соединения (reset)
- ***SYN*** — применяется при синхронизации порядковых номеров
- ***FIN*** — конец данных.

Когда приложение отправляет данные, используя TCP, они перемещаются в них по стеку протоколов. Данные проходят по всем уровням и, в конечном счете, передаются через сеть как поток битов. При этом на самом нижнем — физическом уровне определяется, какой сигнал соответствует 0 и 1. Каждый уровень в наборе протоколов TCP/IP добавляет к исходным данным дополнительную информацию в форме заголовков (*header*) или окончаний (*tailer*). Когда пакет прибывает на конечный узел в сети, он снова проходит все уровни снизу доверху, достигая, в конечном счете, прикладного уровня. Каждый уровень анализирует данные, отделяя от пакета свою информацию в заголовке и/или в окон-

чании и после всех преобразований данные достигают приложение-сервер в той самой форме, в какой они покинули приложение-клиент.

Пример передачи данных между уровнями OSI для TCP/IP

	Данные пользователя				
Прикладной уровень	Заголовок приложения	Данные пользователя			
Транспортный уровень	Заголовок TCP	Прикладные данные = заголовок приложения + данные пользователя			
Сетевой уровень	Заголовок IP	Заголовок TCP	Прикладные данные		
Канальный уровень	Заголовок Ethermet	Заголовок IP	Заголовок TCP	Прикладные данные	Окончание Ethermet

1.3. Преимущества и недостатки TCP

Преимущества TCP

TCP — это сложный, требующий больших затрат времени протокол, что объясняется его механизмами установки соединения, но при этом он берет на себя заботу о гарантированной доставке пакетов, избавляя

программиста от необходимости включать эту функциональность в прикладной протокол.

TCP осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета. В отличие от протокола UDP он гарантирует, что приложение получит данные точно в такой же последовательности, в какой они были отправлены, и без потерь.

Реализация TCP, как правило, встроена в ядро современной операционной системы, хотя есть и реализации TCP в контексте приложения

Известные недостатки TCP

TCP требует явного указания максимального размера сегмента (MSS) в случае, если виртуальное соединение осуществляется через сегмент сети, где максимальный размер блока (MTU) менее чем стандартный MTU Ethernet (1500 байт).

В протоколах туннелирования, таких как GRE, IPIP максимальный размер блока туннеля меньше чем стандартный, поэтому сегмент TCP максимального размера имеет длину пакета больше, чем MTU. Поскольку фрагментация в подавляющем большинстве случаев запрещена, то такие пакеты отбрасываются.

Проявление этой проблемы выглядит как «зависание» соединений. При этом «зависание» может происходить в произвольные моменты времени, а именно тогда, когда отправитель использовал сегменты длиннее допустимого размера.

Для решения этой проблемы на маршрутизаторах применяются правила Firewall-а, добавляющие параметр

MSS во все пакеты, инициирующие соединения, чтобы отправитель использовал сегменты допустимого размера.

Обнаружение ошибок при передаче данных

Хотя протокол осуществляет проверку контрольной суммы по каждому сегменту, используемый алгоритм считается относительно слабым. Так в 2008 году не обнаруженная сетевыми средствами ошибка в передаче одного бита, привела к остановке серверов системы Amazon Web Services.

В общем случае распределенным сетевым приложениям рекомендуется использовать дополнительные программные средства для гарантирования целостности передаваемой информации.

Атаки на протокол

Недостатки протокола проявляются в успешных теоретических и практических атаках, при которых злоумышленник может получить доступ к передаваемым данным, выдать себя за другую сторону или привести к отказу в работе системы.

2. Обзор протокола UDP

UDP *или User Datagram Protocol* (протокол дейтаграмм пользователя) — это простой, ориентированный на дейтаграммы протокол без организации соединения, предоставляющий быстрое, не обязательно надежный способ транспортировки данных. Он поддерживает взаимодействия «один со многими» и поэтому часто применяется для широковещательной и групповой передачи дейтаграмм.

<table>
<tr><th colspan="2">Уровни OSI</th><th>TCP /IP</th><th colspan="5">Стек протоколов TCP /IP</th></tr>
<tr><td>7</td><td>Прикладной</td><td rowspan="3">Прикладной</td><td rowspan="3">HTTP</td><td rowspan="3">FTP</td><td rowspan="3">SMTP</td><td rowspan="3">RIP</td><td rowspan="3">DNS</td></tr>
<tr><td>6</td><td>Представления</td></tr>
<tr><td>5</td><td>Сеансовый</td></tr>
<tr><td>4</td><td>Транспортный</td><td>Транспортный</td><td colspan="3">TCP</td><td colspan="2">UDP</td></tr>
<tr><td>3</td><td>Сетевой</td><td>Интернет</td><td colspan="5">IP</td></tr>
<tr><td>2</td><td>Канальный</td><td rowspan="2">Сетевой</td><td colspan="5" rowspan="2">Ethernet, ATM, Frame Relay и т.п.</td></tr>
<tr><td>1</td><td>Физический</td></tr>
</table>

UDP располагается на транспортном уровне поверх IP, т.е. протокола сетевого уровня. Транспортный уровень обеспечивает взаимодействие между сетями через шлюзы. В нем используются IP — адреса для отправки пакетов данных через Интернет или другую сеть при помощи драйверов устройств. TCP и UDP входят в набор протоколов TCP/IP, каждый из которых имеет свои преимущества и недостатки.

2.1. Терминология UDP

Пакеты — при передаче данных пакетом называется последовательность двоичных цифр, представляющих данные и сигналы управления, которые передаются и коммутируются через хост. Внутри пакета информация упорядочена в соответствии со специальным форматом.

Дейтаграмма — это отдельный, независимый пакет данных, содержащий информацию, достаточную для передачи от источника к адресату. Поскольку дейтаграмма в этом смысле «самодостаточна», никакого дополнительного обмена между источником, адресатом и транспортной сетью не требуется.

MTU — это сокращение означает *Maximum Transmission Unit* (максимальный блок передачи). MTU характеризует канальный уровень и соответствует максимальному числу байт, которое можно передать в одном пакете. Иными словами, MTU — это самый большой пакет, который может переносить данная сетевая среда. Например, Ethernet имеет фиксированное значение MTU, равное 1500 байтам. В протоколе UDP, если размер дейтаграммы больше MTU, протокол выполняет фрагментацию, разбивая одну дейтаграмму на несколько фрагментов таким образом, чтобы каждый фрагмент был меньше MTU.

Порт — чтобы поставить в соответствии входящим данным конкретный процесс, выполняемый на компьютере, UDP использует порты. UDP направляет пакет адресату, используя номер порта, указанный

в UDP — заголовке дейтаграммы. Порты представлены 16-битными номерами и, следовательно, принимают значения в диапазоне от 0 до 65535. порты, которые также называют конечными точками логических соединений, разделены на три категории:

- ***Общеизвестные порты*** — от 0 до 1023
- ***Регистрируемые порты*** — от 1024 до 49 151
- ***Динамические / частные порты*** — от 49 152 до 65 535

UDP использует следующие общеизвестные порты

Номер порта UDP	Описание
15	NETSTAT — состояние сети
53	DNS — сервер доменных имен
69	TFTP — простейший протокол передачи файлов
137	Служба имен NetBIOS
138	Служба дейтаграмм NetBIOS
161	SNMP

Следует отметить, что порты UDP могут получать более одного сообщения в одно и то же время. В некоторых случаях сервисы TCP и UDP могут использовать одни и те же номера портов, например 7 (Echo) и 23 (Telnet).

Список портов UDP и TCP можно найти в Интернете на странице http://www.iana.org/assignments/port-numbers

IP — адреса

Дейтаграмма IP состоит из 32-битных IP адресов источника и назначения. IP адрес назначения задает конечную точку для дейтаграммы UDP, а IP — адрес источника используется для получения информации об

отправителе. В точке назначения пакеты фильтруются и те из них, адреса источников которых не входят в допустимый набор адресов, автоматически отбрасываются без уведомления отправителя.

Однонаправленный IP — адрес однозначно идентифицирует хост в сети, тогда как групповой IP адрес определяет конкретную группу адресов в сети. Широковещательные IP адреса получаются и обрабатываются всеми хостами локальной сети или данной подсети.

IP адрес	Диапазон IP адресов	Использование
Class A	От 0.0.0.0 до 127.255.255.255	В сетях с большим числом хостом, например, сети крупных корпораций
Class B	От 128.0.0.0 до 191.255.255.255	Сети со средним числом хостов
Class C	От 192.0.0.0 до 223.255.255.255	Сети с небольшим числом хостов
Class D	От 224.0.0.0 до 239.255.255.255	В сетях с групповой рассылкой
Class E	От 240.0.0.0 до 247.255.255.255	Зарезервирован для экспериментов

Несколько IP адресов зарезервированы для специальных случаев применения:

IP адрес	Использование
0.0.0.0	IP адрес хоста по умолчанию
127.0.0.1	IP адрес локального хоста
255.255.255.255	Широковещательный IP адрес для всей локальной сети

TTL — (*time to live*) значение времени жизни. Позволяет установить верхний предел числа маршрутизаторов, через которые может пройти дейтаграмма. Применение TTL не позволяет пакетам бесконечно передаваться по сети. Оно инициализируется отправителем и уменьшается на единицу каждым маршрутизатором, обрабатывающим дейтаграмму. Когда значение TTL падает до нуля, дейтаграмма аннулируется.

Групповая рассылка — это открытый, базирующийся на стандартах метод одновременного распространения идентичной информации нескольким пользователям. Групповая рассылка является основным средством протокола UDP, она невозможна для протокола TCP. Групповая рассылка позволяет добиться взаимодействия одного со многими, например, делает возможными такие возможности, как рассылка новостей и почты нескольким получателям, Интернет-радио или демонстрационные программы реального времени. Групповая рассылка не так сильно нагружает сеть, как широковещательная передача, поскольку данные отправляются сразу определенным пользователям, остальные пользователи не получают данных.

2.2. Принципы работы UDP

Когда приложение, базирующееся на UDP, отправляет данные другому хосту в сети, UDP дополняет их восмибайтовым заголовком, содержащим номерами портов адресата и отправителя, общую длину данных и контрольную сумму. Поверх дейтаграммы UDP свой заголовок добавляет IP, формирую дейтаграмму IP.

Формат заголовка UDP:

<table>
<tr><td rowspan="8">IP</td><td rowspan="6">Заголовок IP</td><td colspan="2">Версия (4 бита)</td><td>Длина заголовка (4 бита)</td><td>Тип обслуживания (8 битов)</td><td colspan="2">Общая длина (16 битов)</td></tr>
<tr><td colspan="4">Идентификация (16 бит)</td><td>Флаги (3 бита)</td><td>Смещение фрагмента (13 бит)</td></tr>
<tr><td colspan="2">Время жизни (8 бит)</td><td colspan="2">Протокол (8 бит)</td><td colspan="2">Контрольная сумма заголовка (16 бит)</td></tr>
<tr><td colspan="6">IP – адрес источника (32 бита)</td></tr>
<tr><td colspan="6">IP – адрес получателя (32 битa)</td></tr>
<tr><td rowspan="3">Дейтаграмма UDP</td><td rowspan="2">Заголовок UDP</td><td colspan="4">Номер порта источника (16 бит)</td><td colspan="2">Номер порта получателя (16 бит)</td></tr>
<tr><td colspan="4">Длина UDP (16 бит)</td><td colspan="2">Контрольная сумма (16 бит)</td></tr>
<tr><td>Данные</td><td colspan="6">Данные + байты заполнители</td></tr>
</table>

Общая длина заголовка UDP составляет восемь байт. Максимальный возможный размер дейтаграммы IP равен 65535 байт. С учетом двадцати байт заголовка IP и восьми байт заголовка UDP длина данных пользователя может достигать 65535–28 = 65507 байт. Однако большинство программ работают с данными меньшего размера. Например, для большинства приложений длина дейтаграммы 8192 байта, поскольку именно такой объем данных пользователя по умолчанию считывается и записывается сетевой файловой системой NFS.

Если длина пакета превышает установленный по умолчанию MTU, то на сетевом уровне (в данном случае IP)

пакет разбивается на фрагменты. Заголовок IP содержит всю необходимую информацию о фрагментах.

Когда программа — отправитель посылает дейтаграмму в сеть, она направляется по IP адресу назначения. Этот адрес указывается в заголовке IP. При проходе через маршрутизатор значение времени жизни (*time to live, TTL*) уменьшается на единицу.

Когда дейтаграмма прибывает к заданному пункту назначения и порту, на сетевом уровне проверяется заголовок IP и определяется, фрагментирована ли дейтаграмма. Если да, то дейтаграмма собирается в соответствии с информацией, записанной в заголовке. Наконец на прикладном уровне заголовок удаляется и приложение — адресат получает отправленные данные.

2.3. Преимущества UDP

- ***Нет установки соединения.*** Поскольку UDP является протоколом без организации соединений, он не требует накладных расходов, связанных с настройкой и проверкой соединений. Соответственно и задержек на установки соединений удается избежать. Эту особенность UDP использует служба DNS, работала с установкой соединений значительно замедлила бы ее работу.
- Как следствие отсутствия соединения UDP работает ***быстрее*** TCP.
- ***Топологическое разнообразие.*** UDP поддерживает возможности взаимодействия «один со многими» и «один с одним», в то время как TCP поддерживает только взаимодействие «один с одним».

- ***Размер заголовка.*** Для каждого пакета заголовок UDP имеет длину восемь байт, в то время как TCP имеет длину заголовка двадцать байт.

2.4. Недостатки UDP

- ***Отсутствие сигналов квитирования.*** Поскольку отсутствует установка соединения, у отправителя нет возможности узнать, достигла ли дейтаграмма адресата. В результате UDP не может гарантировать доставку в отличие от TCP, который использует сигналы квитирования.
- ***Отсутствие сеансов.*** В отличие от TCP в протоколе UDP нет возможности поддерживать сеансы. В TCP применяется специальный идентификатор сеанса.
- ***Порядок доставки.*** UDP не гарантирует, что адресату будет доставлена только одна копия данных. UDP не гарантирует также, что данные будут получены в том же порядке, в каком они создавались источником. Протокол TCP вместе с номерами портов использует порядковые номера и постоянно отправляемые подтверждения, что гарантирует упорядоченную доставку данных.
- ***Безопасность.*** TCP более защищен, чем UDP. Часто брандмауэры и маршрутизаторы не пропускают пакеты UDP. Это делается из соображений безопасности, поскольку хакеры могут воспользоваться портами UDP без установки явных соединений.
- ***Управление потомком.*** В UDP управление потоком отсутствует, поэтому плохо спроектированное UDP приложение может сильно загрузить сеть.

Сводная таблица характеристик протоколов UDP и TCP:

Характеристика	UDP	TCP
Установка соединений	Нет	Да
Использование сеансов	Нет	Да
Наличие подтверждений	Нет	Да
Упорядоченная передача	Нет	Да
Управление потоком	Нет	Да
Безопасность	Ниже	Выше
Относительная скорость	Выше	Ниже
Взаимодействие	Один со многими, один с одним	один с одним
Длина заголовка	8 байт	20 байт

3. Работа с протоколом TCP на платформе .NET

До создания платформы .NET средств для высокоуровневого программирования TCP практически не было. Так в Visual C++ приходилось использовать классы CSocket и CAsyncSocket. В .NET классы для работы с сокетами размещены в отдельном пространстве имен System. Net.Sockets. Это пространство имен содержит не только низкоуровневые классы (например, Socket) но и классы TcpListener и TcpClient для организации взаимодействия приложений при помощи протокола TCP. Эти классы имеют простые интерфейсы, в отличие от класса Socket, в котором при получении/отправке данных применяется побайтовый подход, классы TcpListener и TcpClient используют потоковую модель. В них все взаимодействие между клиентом и сервером основывается на потоке с применением класса NetworkStream. При необходимости можно работать и с отдельными байтами.

3.1. Класс TCPListener

Приложение — сервер начинает работу, устанавливая связь с локальной конечной точкой и находясь в состоянии ожидания входящих запросов от клиентов. Как только на сервер приходит запрос клиента, связанный с соответствующим портом, приложение — сервер активизируется. Приложение — сервер создает канал, предназначенный для взаимодействия с этим клиентом.

В основном потоке приложение-сервер ожидает другие входящие запросы от клиентов. Класс TCPListener предназначен для прослушивания запросов клиентов и создания нового экземпляра класса Socket или класса TCPClient, которые можно использовать для дальнейшего взаимодействия с клиентом. Как и TCPClient, класс TCPListener также инкапсулирует внутри себя экземпляр класса Socket в виде поля m_Serversoket с уровнем доступа protected, доступный только для его подклассов. Основные методы класса представлены в таблице:

Возвращаемый результат	Название метода	Описание
Socket	AcceptSocket()	Разрешает создать соединение и возвращает объект Socket, используемый для взаимодействия с клиентом
TcpClient	AcceptTcpClient()	Разрешает создать соединение и возвращает объект TcpClient, используемый для взаимодействия с клиентом
bool	Pending()	Показывает, есть ли запросы, ожидающие соединения
void	Start() / Start(int)	Заставляет данный экземпляр TCPListener начать прослушивать запросы соединения
void	Stop()	Закрывает данный экземпляр TCPListener, находящийся в режиме прослушивания

Возвращаемый результат	Название метода	Описание
IAsyncResult	BeginAcceptTcpClient (AsyncCallback *callback*, object state)	Начинает асинхронную операцию для приема входящего запроса
TcpClient	EndAcceptTcpClient (IAsyncResult *asyncResult*)	Асинхронно принимает входящий запрос на соединение и создает новый экземпляр класса TcpClient для работы с клиентом

Основные свойства класса TCPListener:

Тип	Имя свойства	Описание
IPEndpoint	LocalEndpoint	Возвращает объект, реализующий интерфейс IPEndpoint, который содержит информацию о локальном сетевом интерфейсе и номере порта, используемую для ожидания входящих запросов от клиентов
bool	Active	Показывает, слушает ли в настоящий момент, данный экземпляр класса TCPListener запросы соединения
Server	Socket	Возвращает объект базового класса Socket, ипользуемый объектом TCP для прослушивания запросов

Создание экземпляра класса TCPListener

Для создания экземпляра класса TCPListener применяются следующие конструкторы:

- public TcpLitener(int port) // считается устаревшим
- public TcpLitener(IPEndPoint endPoint)
- public TcpLitener(IPAddress ipAddr, int port)

В первый конструктор передается номер используемого порта для прослушивания запросов. В этом случае IP

адрес равен IPAddress.Any, т.е. сервер принимает действия клиентов на всех сетевых интерфейсах, такое значение эквивалентно IP-адресу 0.0.0.0. Во второй конструктор передается объект IPEndPoint, определяющий IP адрес и порт, на котором выполняется прослушивание:

```
IPAddress adr = IPAddress.Parse(«127.0.0.1»);
IPEndPoint endp = new IPEndPoint(adr, 11000);
TcpListener tcpListener = new TcpListener(endp);
```

Последний перегруженный конструктор принимает объект IPAddress и номер порта:

```
IPAddress adr = IPAddress.Parse(«127.0.0.1»);
int port = 11000;
TcpListener tcpListener = new TcpListener(adr, port);
```

Прослушивание клиентов

На следующем шаге после создания сокета можно приступать к прослушиванию запросов клиентов. В классе TCPListener есть метод Start(), который выполняет при вызове такие шаги:

1. Сначала он связывает сокет, используя IP-адрес и порт, преданные в конструктор TCPListener
2. Затем, вызывая метод Listen() базового класса Socket, он начинает прослушивать запросы соединения от клиентов:

```
TcpListener tcpListener = new TcpListener(adr, port);
tcpl.Start();
```

Приступив к прослушиванию сокета, можно вызвать метод Pending() для проверки, нет ли ожидающих запросов на установку соединения в очереди приложения. Этот метод позволяет проверить наличие ожидающих клиентов до вызова метода Accept(), который блокирует выполняющийся поток:

```
if (tcpListener.Pending())
{
    this.Title = «в очереди программы есть запросы
                  на соединение»;
}
```

Установка соединений с клиентами

В обычной ситуации программа — сервер работает с двумя сокетами: один используется классом TCPListener, а второй — для взаимодействия с отдельным клиентом. Чтобы дать согласие на любой запрос, ожидающий в настоящий момент в очереди, можно воспользоваться методом AcceptSocket() или AcceptTcpClient(). Эти методы возвращают соответственно объекты Socket или TCPListener и дают согласие на запросы клиентов.

```
Socket socket = tcpListener.AcceptSocket();
```

Или

```
TcpClient tcpClient = tcpListener.AcceptTcpClient();
```

Отправка и получение сообщений

В зависимости от типа сокета, созданного при установке соединения, реальный обмен данными между клиентами и сокетом сервера выполняется методами Send() и Receive() объекта Socket. Также можно использовать методы записи/чтения класса NetworkStream.

Остановка прослушивания программы — сервера

После завершения взаимодействия с клиентом следует вызвать метод Stop() класса TCPListener для остановки прослушивания сокета.

3.2. Класс TCP Client

Класс TcpClient обеспечивает TCP сервисы для соединений на стороне клиента. Этот класс инкапсулирует экземпляр класса Socket и обеспечивает TCP-сервисы на более высоком, чем Socket уровне. Экземпляр класса Socket — это private поле m_ClientSoket, используемый для взаимодействия с сервером TCP. Класс TcpClient содержит следующие свойства:

Тип	Имя свойства	Описание
LingerOption	LingerState	Разрешает создать соединение и возвращает объект Socket, ипользуемый для взаимодействия с клиентом
bool	NoDelay	Разрешает создать соединение и возвращает объект TcpClient, ипользуемый для взаимодействия с клиентом
int	ReceiveBufferSize	Показывает, есть ли запросы, ожидающие соединения

Тип	Имя свойства	Описание
int	ReceiveTimeout	Задает время ожидания (мсек) которое класс ожидает получения данных после инициирования этой операции. По окончании этого времени генерируется исключение SocketException
int	SendBufferSize	Задает размер буфера для исходящих данных
int	SendTimeout	Задает время ожидания (мсек) которое класс ожидает подтверждения получения данных. По окончании этого времени генерируется исключение SocketException
bool	Active	Указывает, есть ли активное соединение с удаленным хостом
Socket	Client	Задает объект Socket , инкапсулированный TcpClient

Основные методы класса TCP Client:

Возвращаемый результат	Название метода	Описание
void	Close()	Закрывает TCP соединение
void	Connect()	Устанавливает соединение с удаленным TCP хостом
NetworkStream	GetStream()	Возвращает объект NetworkStream, используемый для передачи данных между клиентом и удаленным хостом

Конструкторы класса TCP Client

В классе определено четыре конструктора

- TCP Client()
- TCP Client(IPEndPoint ipEnd)

- TCP Client(string hostName, int port)
- TCP Client(AddressFamily family)

Конструктор по умолчанию создает объект TcpClient таким образом, что объект получает от сервис-провайдера IP адрес и номер порта. Затем в обязательном порядке для установки соединения с удаленным хостом следует вызвать метод Connect(). Этот конструктор предназначен для работы только с адресами типа IPv4.

Второй перегруженный конструктор принимает один параметр типа IPEndPoint. Он инициализирует новый экземпляр класса TcpClient, связанный с указанной конечной точкой. Следует отметить, что это не удаленная, а локальная конечная точка (EndPoint). Если попытаться передать конструктору удаленную конечную точку, будет выброшено исключение с сообщением о неверном IP адресе в данном контексте.

Если использовать конструктор с экземпляром класса IPEndPoint, то после создания экземпляра TcpClient нужно вызвать метод Connect():

```
//создание локальной конечной точки
IPAddress adr = IPAddress.Parse(«127.0.0.1»);
//создание экземпляра класса IPEndPoint
IPEndPoint endPoint = new IPEndPoint(adr, 9876);
//создание экземпляра класса TcpClient
TcpClient client = new TcpClient(endPoint);
//попытка установить соединение с сервером
client.Connect(«192.168.1.52», 9877);
```

Параметр, переданный конструктору TcpClient, является локальной конечной точкой, в то время как метод

Connect() устанавливает соединение между клиентом и сервером и поэтому принимает в качестве параметра удаленную конечную точку.

Третий перегруженный конструктор создает новый экземпляр класса TcpClient и устанавливает соединение с использованием двух параметров — DNS-имени и номера порта , например

```
TcpClient client = new TcpClient(«192.168.1.52», 80);
```

В этом случае указывается IP адрес удаленного хоста и порт на удаленном хосте. Таким образом, в этом случае нельзя указать адрес локального порта. Еще одни конструктор позволяет передать в качестве параметра экземпляр типа-перечисления AddressFamily, возможные значения перечисления представлены ниже:

Имя	Описание
Unknown	Семейство неизвестных адресов
Unspecified	Семейство неуказанных адресов
InterNetwork	Адрес для IP версии 4
Iso	Адрес для протоколов ISO
Osi	Адрес для протоколов OSI
Ecma	Адрес ассоциации ECMA
DataLink	Прямой адрес интерфейса связи с данными (data-link interface)
NetBios	Адрес NetBios
FireFox	Адрес FireFox
Banyan	Адрес Banyan
InterNetworkV6	Адрес для IP версии 6

Установка соединения с хостом

После создания экземпляра класса TcpClient, на следующем шаге следует вызвать метод Connect(). Он имеет несколько перегруженных вариантов:

Имя	Описание
Connect(IPEndPoint)	Устанавливает соединение клиента с удаленным TCP хостом используя IPEndPoint.
Connect(IPAddress, Int32)	Устанавливает соединение клиента с удаленным TCP хостом, используя указанные IP адрес и номер порта
Connect(IPAddress[], Int32)	Устанавливает соединение клиента с удаленным TCP используя несколько указанных IP адресов (в массиве) и номер порта
Connect(String, Int32)	Устанавливает соединение клиента с удаленным TCP используя имя хоста и номер порта, например newClient. Coonect("127.0.0.1", 80)

Поскольку установка соединения зависит от множества факторов, в том числе и от качества работы сети, наличия работающего приложения — сервера, рекомендуется вызывать этот метод в блоке try/catch:

```
try
{
    TcpClient client = new TcpClient(«localhost», 80);
}
catch (SocketException sockEx)
{
    MessageBox.Show(«Ошибка сокета:» + sockEx.Message);
}
catch (Exception Ex)
```

```
{
        MessageBox.Show(«Ошибка:» + Ex.Message);
    }
```

Отправка и получение сообщений

Для организации взаимодействия между двумя соединенными приложениями как канал используется экземпляр класса NetworkStream. Прежде чем отправлять и получать любые данные, нужно определить базовый поток. Класс TcpClient предоставляет метод GetStream() для этих целей. При помощи сокета он создает экземпляр класса NetworkStream и возвращает его вызывающей программе. Получив поток, следует использовать методы класса потока Write() и Read() для чтения из приложения хоста и записи. Пример использования метода Write():

```
try
{
    TcpClient client = new TcpClient(«localhost», 80);
    NetworkStream nwStream = client.GetStream();
    byte []bt = Encoding.ASCII.GetBytes(«test of
                connection!»);
    nwStream.Write(bt, 0, bt.Length);
}
catch (SocketException sockEx)
{
    MessageBox.Show(«Ошибка сокета:» + sockEx.Message);
}
catch (ArgumentNullException nullEx)
{
    MessageBox.Show(«Объект не создан»);
```

```
}
catch (IOException ioEx)
{
    MessageBox.Show(«Ошибка чтения/записи» +
                ioEx.Message);
}
```

Метод Write принимает три параметра — массив байт, позицию в массиве, начиная с которой следует читать данные и число байт, которые необходимо записать в поток. В вышеуказанном примере в поток записывается целиком весь массив байт, получившийся в результате преобразования строки в кодировке ASCII.

Метод Read() имеет те же три параметра — массив байт для хранения прочитанных данных, позиция в массиве, начиная с которой следует размещать данные и число байт, которое должен прочесть метод:

```
TcpClient client = new TcpClient(«localhost», 80);
NetworkStream nwStream = client.GetStream();
client.ReceiveBufferSize = 1000;
byte []btRead = new byte[client.ReceiveBufferSize];
nwStream.Read(btRead, 0, client.ReceiveBufferSize);
```

Свойство ReceiveBufferSize класса TcpClient служит для задания длины буфера для чтения.

Закрытие сокета TCP

После окончания взаимодействия с клиентом следует вызвать метод Close(). Это позволяет освободить ресурсы, связанные с сокетом TCP.

3.3. Пример использования классов для работы с TCP

Для иллюстрации работы с классами TcpListener и TcpClient можно построить два оконных приложения — сервер и клиент. Сервер получает сообщение от клиента и добавляет его в специальный список полученных сообщений.

Код серверного приложения:

```
using System;
using System.Threading;
using System.IO;
using System.Net.Sockets;
using System.Net;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace TCP_1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        TcpListener list;
        private void button1_Click(object sender,
             EventArgs e)
        {
            try
            {
               //создание экземпляра класса TcpListener
```

```
            //данные о хосте и порте читаются
            //из текстовых окон
            list = new TcpListener(
                IPAddress.Parse(textBox1.Text),
                Convert.ToInt32(textBox2.Text));
            //начало прослушивания клиентов
            list.Start();
            //создание отдельного потока
            //для чтения сообщения
            Thread thread = new Thread(
                    new ThreadStart(ThreadFun)
                    );
            thread.IsBackground = true;
            //запуск потока
            thread.Start();
        }
        catch (SocketException sockEx)
        {
            MessageBox.Show(«Ошибка сокета:» +
                        sockEx.Message);
        }
        catch (Exception Ex)
        {
            MessageBox.Show(«Ошибка:» +
                        Ex.Message);
        }
    }

    void ThreadFun()
    {
        while (true)
        {
            //сервер сообщает клиенту о готовности
            //к соединению
            TcpClient cl = list.AcceptTcpClient();
            //чтение данных из сети в формате
            //Unicode
```

```
                StreamReader sr = new StreamReader(
                    cl.GetStream(), Encoding.Unicode
                );
                string s = sr.ReadLine();
                //добавление полученного сообщения
                //в список
                messageList.Items.Add(s);
                cl.Close();

                //при получении сообщения EXIT
                //завершить приложение
                if (s.ToUpper() == «EXIT»)
                {
                    list.Stop();
                    this.Close();
                }
            }
        }

        private void Form1_FormClosed(object sender,
             FormClosedEventArgs e)
        {
            if (list != null)
                list.Stop();
        }
    }
}
```

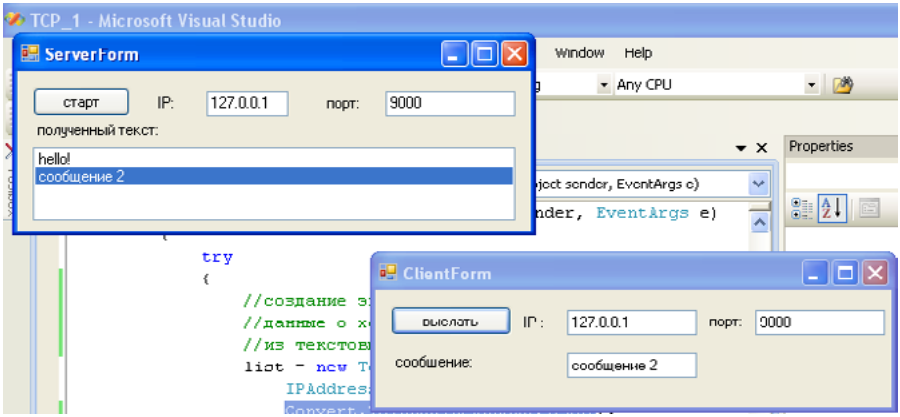
Код клиентского приложения:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```

```
namespace tcp_client
{
    public partial class Form1 : Form
    {
        TcpClient client;
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender,
            EventArgs e)
        {
            try
            {
                //создание экземпляра класса
                //IPEndPoint
                IPEndPoint endPoint = new IPEndPoint(
                    IPAddress.Parse(textBox1.Text),
                    Convert.ToInt32(textBox3.Text)
                );
                client = new TcpClient();
                //установка соединения
                //с использованием
                //данных IP и номера порта
                client.Connect(endPoint);
                //получение сетевого потока
                NetworkStream nstream = client.
                     GetStream();
                //преобразование строки сообщения
                //в массив байт
                byte[] barray = Encoding.Unicode.
                     GetBytes(textBox2.Text);
                //запись в сетевой поток всего массива
                nstream.Write(barray, 0, barray.
                     Length);
                //закрытие клиента
```

```
                client.Close();
            }
            catch (SocketException sockEx)
            {
                MessageBox.Show(«Ошибка сокета:» +
                    sockEx.Message);
            }
            catch (Exception Ex)
            {
                MessageBox.Show(«Ошибка:» +
                    Ex.Message);
            }
        }
        private void Form1_FormClosed(object sender,
                FormClosedEventArgs e)
        {
            if (client != null)
                client.Close();
        }
    }
}
```

Пример работы приложений во взаимодействии сервер-клиент.



4. Работа с протоколом UDP на платформе .NET

В отличие от протокола TCP, в котором используется квитирование (контроль отправки / получения данных) в протоколе UDP подобный механизм отсутствует, что позволяет ему работать гораздо быстрее. UDP часто используют для передачи мультимедийных данных (например, видео), поскольку точный порядок прибытия пакетов в пункт назначения может часто является несущественным.

На платформе .NET существует специальный класс UdpClient для реализации User datagramm Protocol (UDP). Существуют также другие возможности для реализации UDP — класс Socket, элемент управления Winsock и API Winsock. Последние две возможности используют технологию COM. Класс UdpClient построен поверх класса Socket и используя UdpClient программист фактически работает с экземпляром Socket'а.

В .NET классы для работы с протоколами TCP и UDP находятся в пространстве имен System.Net.Sockets.

4.1. Класс UdpClient

Схема использования класса UdpClient довольно проста. Сначала следует создать экземпляр класса UdpClient. Затем посредством вызова метода Connect() следует установить соединение с удаленным хостом. При этом, поскольку UDP не ориентирован на установку соединений

метод Connect() до отправки или получения данных не устанавливает соединение с удаленным хостом. Когда отправляется дейтаграмма, то пункт назначения должен быть известен, для чего необходимо указать IP адрес и номер порта. Третий шаг состоит в отправке и получении данных с использованием методов Send() и Receive(). В конце работы следует вызвать метод Close() для закрытия соединения с UDP.

Основные методы класса представлены в таблице:

Возвращаемый результат	Название метода	Описание
void	Connect (IPEndPoint endPoint) Connect (IPAddress, int) Connect (string, int)	Настраивает параметры удаленного хоста
byte[]	Receive (ref IPEndPoint remoteEP)	Возвращает UDP дейтаграмму, посланную удаленным хостом
int	Send (array<Byte>[], Int32, IPEndPoint) Send (byte[] dgram,int bytes)	Высылает UDP дейтаграмму удаленному хосту
void	Close()	Закрывает данный экземпляр UdpClient
void	JoinMulticastGroup (IPAddress) JoinMulticastGroup (Int32, IPAddress)	Присоединение к группе многоадресатнойрассылки (групповой рассылки), определяемой параметром IPAddress. Параметр Int32 определяет значение TTL (time-to-live)
void	DropMulticastGroup (IPAddress) DropMulticastGroup (IPAddress, int)	Исключение из группы многоадресатной рассылки, определяемой параметром IPAddress

Основные свойства класса UdpClient :

Тип	Имя свойства	Описание
bool	Active	Возвращает или устанавливает переменную, определяющую, были ли настроены параметры удаленного хоста для соединения. Это protected свойство
Socket	Client	Возвращает инкапсулированное в классе UdpClient поле типа Socket. Это protected свойство
Bool	DontFragment	Возвращает или устанавливает переменную, определяющую, можно ли фрагментировать IP дейтаграммы

Создание экземпляра класса UdpClient

Для создания экземпляра класса TCPListener применяются следующие конструкторы

Имя	Описание
UdpClient()	Создает экземпляр класса UdpClient. Сервис-провайдер назначает ему IP адрес и номер порта. Затем следует или вызвать метод Connect() или предавать информацию о соединении при отправке данных
UdpClient (AddressFamily)	Создает экземпляр класса UdpClient с использованием перечисления AddressFamily
UdpClient(Int32)	Создает экземпляр класса UdpClient и связывает его с номером локального порта. Пределы допустимых адресов хранятся в полях MaxPort и MinPort класса IPEndPoint. Если указанный порт занят, генерируется исключение SocketException
UdpClient (IPEndPoint)	Создает экземпляр класса UdpClient на основе IPEndPoint. IPEndPoint инкапсулирует данные о IP адресе и номере порта. В свою очередь IP адрес можно описать как экземпляр класса IPAddress.

Имя	Описание
UdpClient(Int32, AddressFamily)	Создает экземпляр класса UdpClient с использованием перечисления AddressFamily и номера порта
UdpClient (String, Int32)	Создает экземпляр класса UdpClient на основе имени удаленного хоста и номера порта. После использования этого конструктора можно не вызывать метод Connect(), поскольку он вызывается непосредственно из конструктора.

Определение информации о соединении

После создания объекта UdpClient можно установить соединение для взаимодействия с удаленным хостом. Фактически устанавливать данные об удаленном хосте можно в некоторых конструкторах UdpClient, или явно вызывать метод Connect() или можно предавать эти данные в метод Send() при фактической передаче данных.

Перегруженные методы Connect ():

Сигнатура метода	Описание
Connect (IPEndPoint)	Устанавливает данные об удаленном хосте, используя экземпляр класса IPEndPoint, которые инкапсулирует информацию об IP адресе и номере порта.
Connect (IPAddress, Int32)	Устанавливает данные об удаленном хосте, используя IP адрес и номер порта.
Connect (String, Int32)	Устанавливает данные об удаленном хосте, используя имя хоста и номер порта.

Пример установки соединения с использованием экземпляра класса IPEndPoint:

```
UdpClient client = new UdpClient();
    IPAddress address = IPAddress.Parse(«192.168.1.51»);
    IPEndPoint endPoint = new IPEndPoint(address, 1234);
```

```
    try
    {
        client.Connect(endPoint);
    }
    catch(Exception ex)
    {
        MessageBox.Show(«Ошибка» + ex.Message);
    }
```

Пример установки соединения с использованием IP адреса и номера порта:

```
UdpClient client = new UdpClient();
IPAddress address = IPAddress.Parse(«192.168.1.51»);
try
{
    client.Connect(address, 1234);
}
catch(Exception ex)
{
    MessageBox.Show(«Ошибка» + ex.Message);
}
```

Отправка данных при помощи объекта UdpClient

Создав экземпляр класса UdpClient и установив данные об удаленном хосте, можно приступать к отправке данных. Для этого используется метод Send(). Он высылает дейтаграмму удаленному хосту. Поскольку протокол UDP не использует квитирования, никаких подтверждений от удаленного хоста. Как и метод Connect(), метод Send() существует в нескольких перегруженных вариантах, при этом все методы возвращают число высланных байт:

Сигнатура метода	Описание
Send (Byte[], Int32)	Высылает UDP дейтаграмму удаленному хосту. Первый параметр — массив байт, определяющий дейтаграмму, второй — длина дейтаграммы.
Send (Byte[], Int32, IPEndPoint)	Высылает UDP дейтаграмму удаленному хосту. Первый параметр — массив байт, определяющий дейтаграмму, второй — длина дейтаграммы, третий — параметры удаленного хоста, которому посылается дейтаграмма
Send (Byte[], Int32, String, Int32)	Высылает UDP дейтаграмму удаленному хосту. Первый параметр — массив байт, определяющий дейтаграмму, второй — длина дейтаграммы, третий — имя удаленного хоста, четвертый — номер порта.

Пример посылки сообщения с помощью метода Send() без использования метода Connect():

```
UdpClient client = new UdpClient();
byte[] bArray = Encoding.ASCII.GetBytes(«test message»);
try
{
    client.Send(bArray, bArray.Length,
          «remoHostNumberOne», 9001);
}
catch(Exception ex)
{
     MessageBox.Show(«Ошибк»» + ex.Message);
}
```

Получение данных с использованием класса UdpClient

Для получения данных от удаленного хоста применяется метод Receive(). Этот метод имеет следующую сигнатуру:

```
byte[] Receive(ref IPEndPoint remoteEP)
```

Метод возвращает в виде массива байт принятые значения. В качестве единственного параметра используется экземпляр класса IPEndPoint, который инкапсулирует данные об IP адресе и порте удаленного хоста. Этот метод рекомендуется выполнять в отдельном потоке, поскольку он запрашивает у базового сокета наличие дейтаграмм и блокирует потока, пока дейтаграмма не будет получена. Если же вызвать этот метод на основном потоке, то выполнение программы будет приостановлено.

После получения дейтаграммы этот метод возвращает данные в массиве байт, предварительно удалив информацию заголовка и заполняет экземпляр класса IPEndPoint, который передается в качестве единственного параметра.

Пример использования метода Receive():

```
//переменная LocalPort предварительно инициализирована
UdpClient uClient = new UdpClient(LocalPort);
//переменная ipEnd будет передана в метод Receive
IPEndPoint ipEnd = null;
byte[] responce = uClient.Receive(ref ipEnd);
//преобразование в строку
string strResult = Encoding.Unicode.
       GetString(responce);
Console.WriteLine(strResult);
```

Закрытие соединения

При работе с классом UdpClient последним шагом является вызов метода без параметров Close(). Если при закрытии соединения возникают ошибки, генерируется исключение SocketException.

4.2. Пример использования: написание приложения — чата

Чтобы проиллюстрировать использование класса UdpClient можно создать консольное приложение, которое при запуске запросит параметры для настройки сокета, а затем позволит передавать тестовые данные другому экземпляру этого приложения, таким образом, работая как простейший чат. В приложении существуют два потока — один для чтения данных, другой для передачи. Данные получаются как цепочка байт и затем преобразуются в строку, которая и выводится на экран. Для удобства пользователя сообщения собеседника выводятся другим цветом.

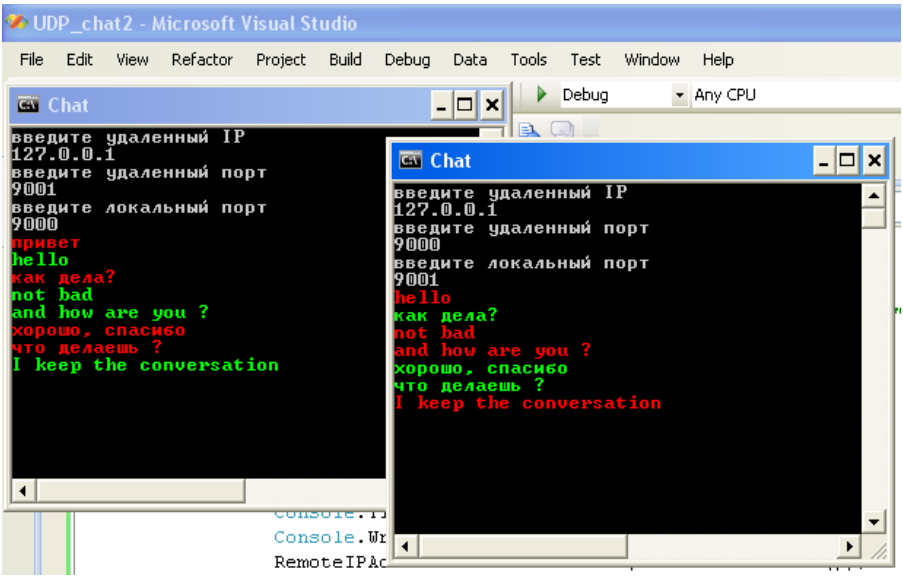
```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Threading;

class Program
{
    //переменные, необходимые для настройки
    //подключения:
    //удаленный хост и порты - удаленный и локальный
    static int RemotePort;
    static int LocalPort;
    static IPAddress RemoteIPAddr;

   [STAThread]
   static void Main(string[] args)
   {
      try
      {
```

```
        Console.SetWindowSize(40, 20);
        Console.Title = «Chat»;
        Console.WriteLine(«введите удаленный IP»);
        RemoteIPAddr = IPAddress.Parse(Console.
            ReadLine());
        Console.WriteLine(«введите удаленный порт»);
        RemotePort = Convert.ToInt32(Console.
            ReadLine());
        Console.WriteLine(«введите локальный порт»);
        LocalPort = Convert.ToInt32(Console.ReadLine());
        //отдельный поток для чтения в методе
        //ThreadFuncReceive
        //этот метод вызывает метод Receive()
        //класса UdpClient,
        //который блокирует текущий поток, поэтому
        //необходим отдельный поток
        Thread thread = new Thread(
               new ThreadStart(ThreadFuncReceive)
        );
        //создание фонового потока
        thread.IsBackground = true;
        //запуск потока
        thread.Start();
        Console.ForegroundColor = ConsoleColor.Red;
        while (true)
        {
           SendData(Console.ReadLine());
        }
    }

    catch (FormatException formExc)
    {
        Console.WriteLine(«Преобразование
                невозможно:» + formExc);
    }
    catch (Exception exc)
    {
```

```
            Console.WriteLine(«Ошибка:» + exc.Message);
        }
    }

    static void ThreadFuncReceive()
    {
        try
        {
            while (true)
            {
                //подключение к локальному хосту
                UdpClient uClient = new
                UdpClient(LocalPort);
                IPEndPoint ipEnd = null;
                //получание дейтаграммы
                byte[] responce = uClient.Receive(ref ipEnd);
                //преобразование в строку
                string strResult = Encoding.Unicode.
                        GetString(responce);
                Console.ForegroundColor = ConsoleColor.
                        Green;
                //вывод на экран
                Console.WriteLine(strResult);
                Console.ForegroundColor = ConsoleColor.Red;
                uClient.Close();
            }
        }

        catch (SocketException sockEx)
        {
            Console.WriteLine(«Ошибка сокета:» +
                    sockEx.Message);
        }

        catch (Exception ex)
        {
            Console.WriteLine(«Ошибка:» + ex.Message);
```

```
        }
    }

    static void SendData(string datagramm)
    {
        UdpClient uClient = new UdpClient();
        //подключение к удаленному хосту
        IPEndPoint ipEnd = new IPEndPoint(RemoteIPAddr,
            RemotePort);

        try
        {
            byte[] bytes = Encoding.Unicode.
                GetBytes(datagramm);
            uClient.Send(bytes, bytes.Length, ipEnd);
        }

        catch(SocketException sockEx)
        {
            Console.WriteLine(«Ошибка сокета:» + sockEx.
                Message);
        }

        catch(Exception ex)
        {
            Console.WriteLine(«Ошибка:» + ex.Message);
        }

        finally
        {
            //закрытие экземпляра класса UdpClient
            uClient.Close();
        }
    }
}
```

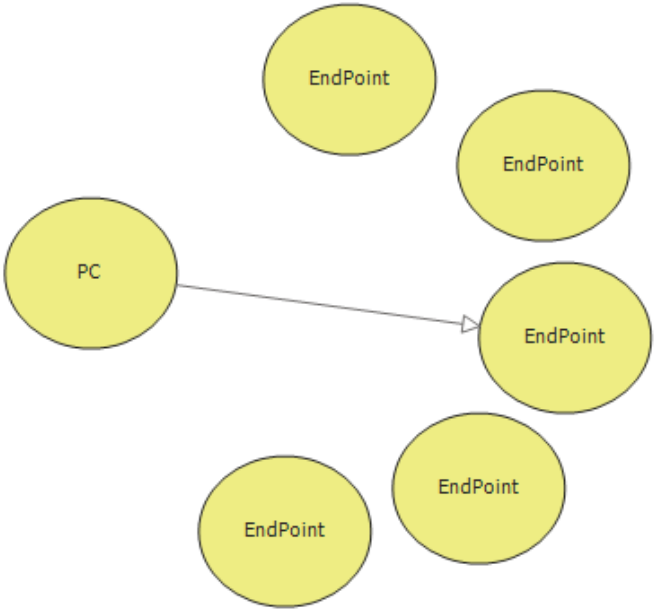
Пример работы приложения — чата:



5. Что такое Unicast

Схема маршрутизации *unicast* предполагает, что пакет отправляется только одному сетевому устройству от другого сетевого устройства напрямую.

Эта схема маршрутизации может быть использована как через TCP, так и через UDP-протокол.



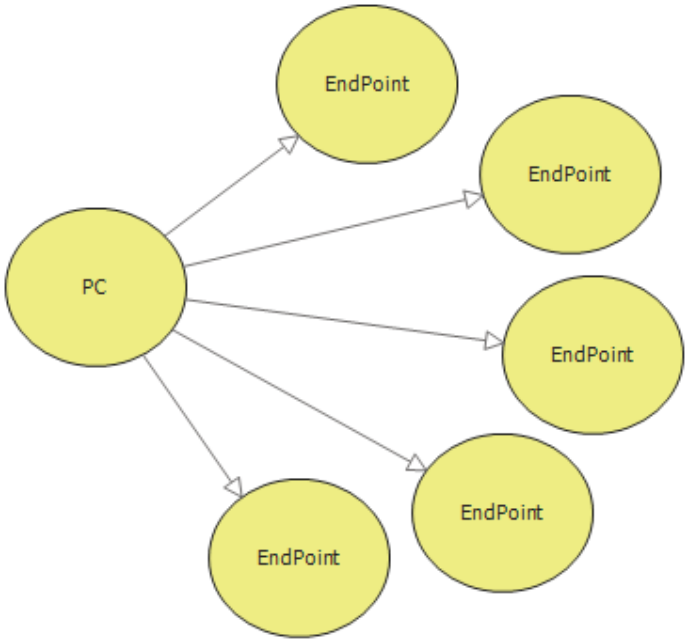
Смысл состоит в том, что данные передаются одному реципиенту. Но это, в свою очередь не означает, что они передаются всегда на один и тот же компьютер, поскольку, один компьютер может иметь несколько сетевых устройств, а значит в один момент времени обладать несколькими IP-адресами, и даже находиться в различных подсетях.

И, также, может в различный момент времени менять IP-адрес.

Также необходимо понимать, что используя схему маршрутизации unicast при отправке одного и того же пакета многим получателям, Вам придётся вручную отправлять данные каждому реципиенту отдельно.

6. Что такое Broadcast

Broadcast (разновидность широковещательного канала) схема маршрутизации — это специальный подвид сетевого взаимодействия, который предполагает что после получения пакета, роутер передаёт этот пакет всем компьютерам, которые соответствуют некоторому адресному пространству (подмножеству). Например, все компьютеры подсети, или вообще всем доступным компьютерам.



Однако, несмотря на то, что broadcast схема маршрутизации, может показаться очень привлекательной, она сильно расходует ресурсы сети, а значит использовать её нужно как можно уже, ведь дейтаграмма отправляется не

только тем компьютерам, которые ожидают сообщения, а вообще всем. Так же многие виды сетевых атак, например, такая, как широковещательный шторм, используют широковещательный канал. Поэтому на современном этапе многие роутеры блокируют широковещательные запросы в целях безопасности. И, если вы решите разработать чат, используя широковещательный канал, то в рамках локальной подсети он работать будет, но из неё, скорее всего пакеты выходить не будут.

Для того, чтобы отправить broadcast-пакет нужно создать UDP-сокет и отправить дейтаграмму используя broadcast-адрес. Для определения broadcast-адреса нужно применить побитовую операцию конъюнкции к первому адресу подсети (например, в подсети с маской 255.255.0.0 первым адресом будет 192.168.0.0) и дополнению маски подсети (0.0.255.255). Таким образом получается, что 192.168.0.0 | 0.0.255.255 = 192.168.255.255.

Или в бинарном виде это будет выглядеть следующим образом:

```
11000000101010000000000000000000
OR
    00000000000000001111111111111111
    --------------------------------
= 11000000101010001111111111111111
```

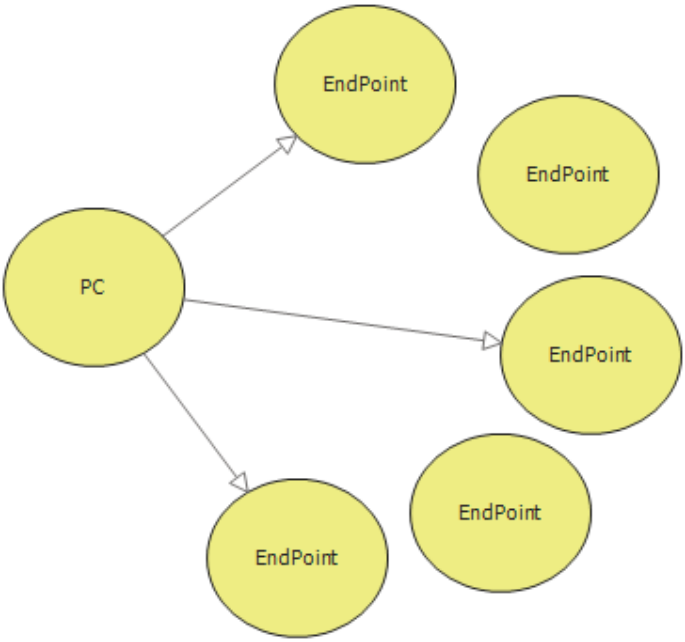
После того, как был определён широковещательный адрес мы создаём сокет и отправляем запрос.

```
Socket soc = new Socket(
AddressFamily.InterNetwork, SocketType.Dgram,
       ProtocolType.Udp);
```

```
IPEndPoint ipep = new IPEndPoint(IPAddress.
      Parse(«192.168.255.255»), 1234);
soc.Connect(ipep);
string message = «Hello network!!!»;
soc.Send(Encoding.Default.GetBytes(message));
```

7. Что такое Multicast

Multicast — это такая технология сетевой адресации, при которой сообщение доставляются сразу группе получателей. При этом используется стратегия наиболее эффективного способа доставки данных: сообщение передаётся через одни сегмент сети только один раз; данные копируются только тогда, когда направление к получателям разделяется.



Слово multicast, как правило, используется как ссылка на IP Multicast — метод отправки IP дейтаграмм («в народе» называемых «пакетами») группе получателей за один сеанс отправки данных.

Согласно стандарта RFC 3171, адреса в диапазоне от 224.0.0.0 до 239.255.255.255 используются протоколом IPv4 как multicast-адреса и формируют класс адресов «D». Как только отправитель посылает дейтаграмму на некоторый multicast-адрес, роутер немедленно перенаправляет её всем получателям, которые зарегистрировались на получение информации с этого multicast-адреса.

Для того, чтобы осуществить отправку данных на multicast-адрес средствами .NET Framework необходимо сперва создать обычный UDP-сокет.

```
Socket someMulticastSocket = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Dgram,
    ProtocolType.Udp);
```

Затем необходимо установить опцию MulticastTimeTolive, которая влияет на время жизни пакета. Если установить её в значение 1, то пакет не выйдет за пределы локальной сети. Если же установить её в значение отличное от 1, то дейтаграмма будет проходить через несколько роутеров.

Установка этой опции осуществляется посредством вызова компонентного метода SetSocketOption класса Socket.

```
someMulticastSocket.SetSocketOption(
    SocketOptionLevel.IP,
    SocketOptionName.MulticastTimeToLive, 2);
```

Следующим действием мы создаём объект класса IPAddress, описывающий некоторый, выбранный нами, multicast-адрес. Регистрируем этот адрес, для созданного

нами сокета посредством вызова метода SetSocketOption, создаём на базе этого адреса конечную точку соединения и соединяем наш сокет с этой конечной точкой.

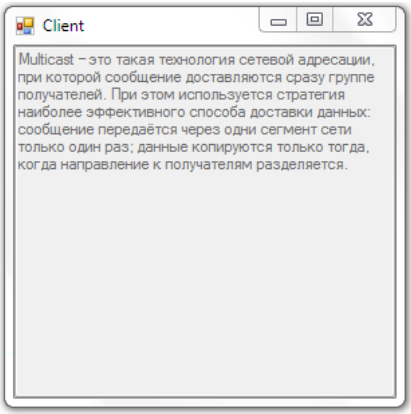
```
IPAddress dest = IPAddress.Parse(«224.5.5.5»);
someMulticastSocket.SetSocketOption(
    SocketOptionLevel.IP,
    SocketOptionName.AddMembership,
new MulticastOption(dest));
IPEndPoint ipep = new IPEndPoint(dest, 4567);
someMulticastSocket.Connect(ipep);
```

Теперь мы можем осуществлять передачу multicast-сообщений. Поэтому мы можем перейти к вызову метода Send нашего сокета.

```
string message = «Hello network!!!»;
someMulticastSocket.Send(Encoding.Default.
    GetBytes(message));
```

8. Пример реализации musticast приложения

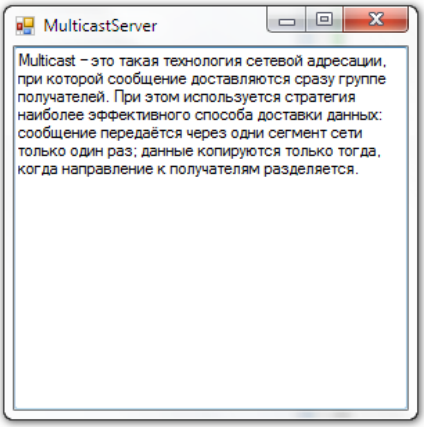
Мы рассмотрим использование схемы маршрутизации multicast на примере оконного приложения, которое осуществляет рассылку введённой пользователем информации всем зарегистрированным пользователям (то есть тем пользователям, у которых запущено клиентское приложение) и представляет собой доску объявлений. Мы реализуем клиент-серверную модель с целью унификации интерфейса пользователя.



Интерфейс клиентского приложения будет представлен окном, в котором размещено текстовое поле в режиме multiline и свойство enabled которого установлено в значение false, чтобы пользователь не мог редактировать приходящие к нему сообщения.

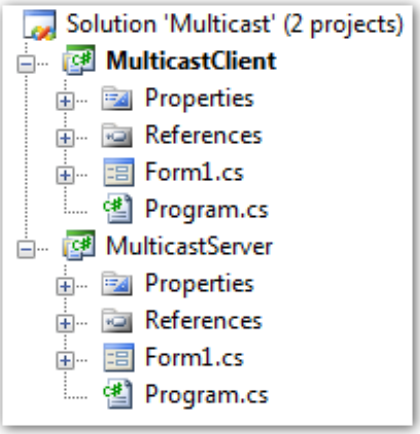
Интерфейс серверной части будет выглядеть похожим образом, с одним исключением — поле должно позволять

редактирование, чтобы пользователь мог менять сообщения, на «доске сообщений».



Создайте два оконных проекта в одном решении и отредактируйте их интерфейс так, как показано выше.

Пусть текстовое поле в серверном приложении называется textBox1 (название по умолчанию), а в клиентском — outputData.



Серверная часть приложения

Нам необходимо объявить статическую переменную, которая будет хранить отправляемое сообщение, а так же целочисленную переменную, которая определит интервал, с которым актуальные данные будут отправляться получателям. Далее необходимо описать потоковый метод, который будт «заниматься» отправкой информации.

```
static string message = «Hello network!!!»;
static int Interval = 1000;
static void multicastSend()
    {
        while (true)
        {
            Thread.Sleep(Interval);
            Socket soc = new Socket(
                AddressFamily.InterNetwork,
                SocketType.Dgram,
                ProtocolType.Udp);
            soc.SetSocketOption(
                SocketOptionLevel.IP,
                SocketOptionName.MulticastTimeToLive, 2);
            IPAddress dest = IPAddress.
                Parse(«224.5.5.5»);
            soc.SetSocketOption(SocketOptionLevel.IP,
                SocketOptionName.AddMembership,
                new MulticastOption(dest));
            IPEndPoint ipep = new
                IPEndPoint(dest, 4567);
            soc.Connect(ipep);
            soc.Send(Encoding.Default.
                GetBytes(message));
            soc.Close();
        }
    }
```

Как Вы видите, в потоковом методе нами открыт бесконечный цикл, в котором мы сначала приостанавливаем выполнение потока на определённый переменной Interval период, затем создаём сокет и регистрируем в нём широковещательный адрес. После этого мы инициируем соединение с широковещательным multicast-каналом, и осуществляем отправку информации. После чего закрываем сокет.

После объявления потокового метода необходимо создать объект потокового класса, и запустить его на исполнение в конструкторе. Не забудьте установить поле IsBackground потокового класса true, чтобы впоследствии не возникло проблем с его закрытием.

```
Thread Sender = new Thread(new
    ThreadStart(multicastSend));
public Form1()
{
    InitializeComponent();
    Sender.IsBackground = true;
    Sender.Start();
}
```

Последним пунктом — необходимо обработать событие TextChanged текстового поля, чтобы обновлять информацию, хранимую переменной message.

```
private void textBox1_TextChanged(object sender,
    EventArgs e)
{
    message = textBox1.Text;
}
```

Клиентская часть

В первую очередь, необходимо объявить делегат, чтобы иметь возможность впоследствии обновлять информацию в текстовом поле без ошибки меж-поточного доступа. Поскольку текстовое поле было создано в потоке отличном от того, в котором оно обновляется, могут возникать ошибки, связанные с блокировкой ресурсов.

```
delegate void AppendText(string text);
void AppendTextProc(string text)
{
    dataOutput.Text = text;
}

void Listner()
{
    while (true)
    {
        Socket soc = new Socket(
            AddressFamily.InterNetwork,
            SocketType.Dgram, ProtocolType.Udp);
        IPEndPoint ipep = new IPEndPoint(IPAddress.
            Any, 4567);
        soc.Bind(ipep);
        IPAddress ip = IPAddress.Parse(«224.5.5.5»);
        soc.SetSocketOption(
            SocketOptionLevel.IP,
            SocketOptionName.AddMembership,
            new MulticastOption(ip, IPAddress.Any));
        byte[] buff = new byte[1024];
        soc.Receive(buff);
        this.Invoke(new AppendText(AppendTextProc),
            Encoding.Default.GetString(buff));
        soc.Close();
    }
}
```

Как Вы видите, в потоковом методе, который будет выполнять получение информации по multicast-каналу, мы открыли бесконечный цикл. В этом цикле мы создаём обычный UDP сокет, для которого регистрируем multicast ip-адрес. После этого мы ставим сокет в ожидание получения информации, и как только он её получил, мы вызываем метод, который обновит информацию в текстовом поле.

В классе необходимо объявить объект потокового класса, проинициализировать его, и запустить поток вызовом метода Start

По выполнении описанных выше действий можно запустить оба проекта и посмотреть работающее приложение.

```
Thread listen;
public Form1()
{
    InitializeComponent();
    listen = new Thread(new ThreadStart(Listner));
    listen.IsBackground = true;
    listen.Start();
}
```

9. Домашнее задание

С использование схемы маршрутизации multicast, реализуйте без серверный чат с графическим пользовательским интерфейсом. То есть такой чат, которому не нужен сервер, и который отправлял бы сообщения только тем компьютерам, которые его ожидают. Так же предусмотрите возможность мониторинга пользователей, находящихся онлайн.