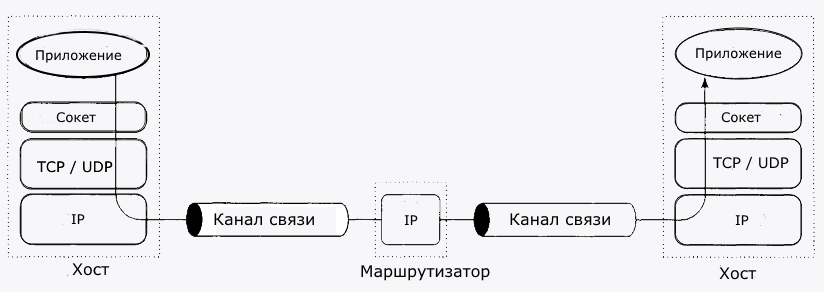
**Основы работы с сетями в C#**

*Введение в сети и протоколы*

**Протокол** – соглашение о том, как пакеты данных будут передаваться по каналам коммуникации. Таким образом, протокол упорядочивает взаимодействие.

Существует множество различных протоколов. Протоколы, которые осуществляют передачу данных по сети, составляют семейство **TCP/IP**. Основные из них: **Internet Protocol** (**IP**), **Transmission Communication Protocol** (**TCP**) и **User Data Protocol** (**UDP**).



**IP** – сетевой уровень, который использует физические каналы коммуникации – интернет-кабели, Wi-Fi и т.д., для передачи пакетов данных другому хосту.

**TCP** и **UDP** – транспортные уровни, используют определенные порты для передачи данных. **TCP** позволяет отследить потерю пакетов и их дублирование при передаче. Поэтому, TCP работает медленнее, чем UDP. **UDP** же подобного не предоставляет и нацелен на простую передачу данных.

Однако приложение взаимодействует с **TCP/UDP** напрямую, а через специальный API, который предоставляют сокеты. **Сокет ­**– это интерфейс для создания сетевых приложений, который опирается на встроенные возможности операционной системы.

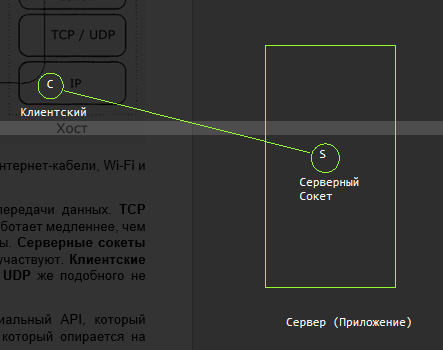
Различают два вида сокетов: потоковые сокеты (**stream socket**) и дейтаграмные сокеты (**datagram socket**). Потоковые сокеты используют протокол **TCP**, а дейтаграмные – **UDP**.

Для того, чтобы обменяться данными, необоходимо установить соединение. Для этого вызывающий сокет должен знать **IP-адрес** того сокета, с которым хочет установить соединение (аналогично сотовой связи: для того, чтобы один телефон должен набрать другому телефону). IP-адрес аналогичен телефонному номеру. В телефонной сети после набора номера вызывающая сторона чувствует вызов, то есть запрос на установление соединения и подтверждает этот запрос, снимая трубку. Аналогично и **TCP**: ожидающий запроса, на установление связи, сокет получает этот запрос и подтверждает его. После этого начинает происходить обмен данных. Как только обмен данными завершен, любой из сокетов разрывает соединение. В **TCP** протоколе необходимо установить соединение.

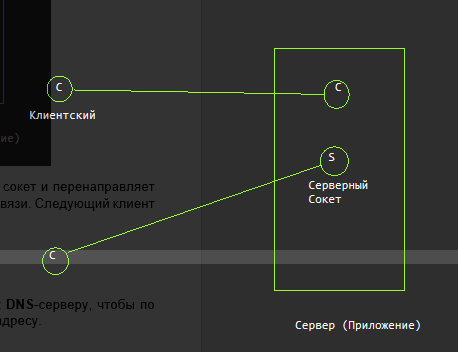
Чтобы уникально определять хосты в сети каждый хост имеет адрес. Самым распространенным протоколом адресов является **IPv4**, который предполагает представление адреса в 32-битного числа. Также набирает популярности протокол **IPv6**, адреса которого представляют 128-битное число. Чтобы не запоминать все эти числа существует служба **DNS** (**Domain Name System**), которая сопоставляет интернет-адреса в виде **IPv4** и **IPv6** с доменными именами.

Кроме адреса при сетевых взаимодействиях используются порты. **Порт** представляет 16-битное число в диапазоне от 1 до 65 025. Использование портов позволяет разграничить несколько запущенных приложений на одном хосте.

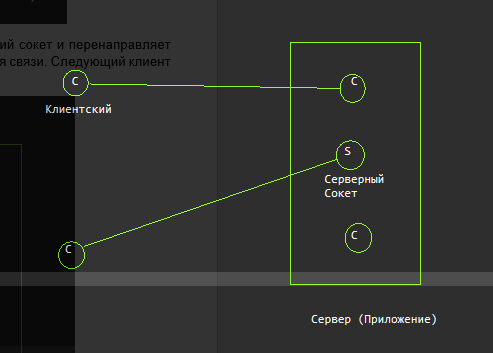
В TCP сокеты делятся на 2 вида сокетов: клинетские сокеты и серверные сокеты. **Серверные сокеты** только принимают запросы на установление соединения. В обмене данными они не участвуют. **Клиентские сокеты** инициируют запрос на установление связи и участвуют в обмене данными.



Серверный сокет, получив запрос на установление связи, создает клиенсткий сокет и перенаправляет соединения на него. Сам сервер возвращается к состоянию ожидния на установления связи. Следующий клиент посылает запрос на установление связи серверному сокету.



Серверный сокет, почувствовав запрос на установление связи, подтверждает этот запрос, путем создания клиентского сокета, на который перенаправляет соединение от удаленного сокета.



Сам сервер возвращается к состоянию ожидния на установления связи и т.д. Если закрыть серверный сокет, то это не приводит к разрыву всех сокетных соединений. На компьютере может исполнятся несколько серверных приложений. При этом IP-адрес компьютера может быть один. Чтобы система различала пакеты, предназаначенные для одного сокета, от пакетов, предназначенных для другого сокета, существуют порты. Соответственно, клиентскому сокету для установления соединения необходимо знать не только IP-адрем, но и номер порта. Любое сетевое приложение должно иметь возможность конфигурирования изменения номера порта и

*Адреса в .NET*

Прежде чем отправить запрос какому-либо ресурсу, компьютер обращается к **DNS**-серверу, чтобы по имени ресурса получить его IP-адрес, и только потом обращается по полученному IP-адресу.

*Класс IPAdress*

* Parse() – преобразует строковое представление IP-адреса в IPAdress
* Loopack – возвращает объект IPAdress для адреса 127.0.0.1
* Any – возвращает объект IPAdress для адрес 0.0.0.0
* Broadcast – возвращает объест IPAdress для адрес 255.255.255.255

*Класс IPHostEntry*

Данный класс содержит информацию об определенном компьютере-хосте.

* HostName – возвращает имя хоста
* AdressesList – возвращает все IP-адреса хоста, так как один компьютер может иметь в сети несколько IP-адресов

*Класс Dns*

* GetHostEntry() – возвращает информацию о хосте компьютера и его IP-адреса
* IPAddress[] GetHostAdresses(string hostNameOrAddress) – по названию хоста возвращает список IP-адресов
* GetHostName() – возвращает имя текущего комьютера

**Отправка запросов**

*Класс TcpListener – серверный сокет*

* TcpListener(int port) *//deprecated* – указывается номер порта, который должен прослушиваться серверным сокетом
* TcpListener(IPAddress localaddr, int port) – указывается IP-адрес, который он прослушивает, и порт
* TcpListener(IPEndPoint localeEP) – то же самое, что и прошлое, только IP-адрес и порт инкапсулируется в IPEndPoint
* Socket Server – реализует функциональность сокета Беркли.
* TcpClient AcceptTcpClient() – ожидает запросов на установление связи. Метод блокирующий: то есть, поток будет приостановлен до тех пор, пока не будет получен запрос на установление связи. Метод создаст клиентский сокет и вернет на него ссылку. Север должен как можно быстрее вернутся в вызов этого метода для ожидания следующих запросов на установление связи. Этот метод рекомендуется вызывать во вторичных потоках.
* Start() – начинает прослушивание
* Start(int backlog) – начинает прослушивание и позволяет указать размер очереди запросов на установление соединения
* Stop() – останавливает прослушивание, но ранее созданные соединения не прерываются

*Класс TcpListener – серверный сокет*

* TcpClient() – создает клиентский сокет и для установления связи нужно вызвать метод Connect
* TcpClient(IPEndPoint locale) – создает клиентский сокет и сразу препдринимают попітку установления соединения
* TcpClient(string hostname, int port) – создает клиентский сокет и сразу препдринимают попітку установления соединения
* Available – возвращает количество байт, доступных для чтения
* Socket Client – вернет ссылку на объект сокета, для которого TcpClient является адаптером
* Bool Connected (не рекомендуется) – возвращает, установлено ли соединение на момент последнего обмена данными
* Close() – закрывает соединение
* Connect(IPAddress address, int port) – устанавливает соединение
* NetworkStream GetStream() – вернет ссылку на поток, с помощью которого мы обмениваемся данными по сети

*Класс WebClient (System.Net)*

Данный класс предназаначен для загрузки файлов с определенных ресурсов.

* DownloadFile(String resource, String pathToSave) – самый простой способ загрузки файла.
* OpenRead(String resource) – возвращает запрошеный файл в качестве потока, тем самым позволяет манипулировать этим потоком в процессе загрузки.

*Классы WebRequest и WebResponse*

Для отправки запроса предназначен класс WebRequest. Класс WebResponse позволяет получить ответ на запрос.

*Класс WebRequest*

* Create() – создание объекта класс WebRequest.
* GetResponse() – отправка и получение запроса.

*Класс WebResponse*

* CharacterSet – возвращает
* ContentLength – размер данных ответа
* ContentEncoding – метод, который используется для кодирования основного текста ответа
* ContentType – тип данных ответа
* Cookies - куки
* Headers - заголовки
* Method – метод отправки данныъ
* StatusCode
* StatsDescription
* GetResponseHeader()
* GetResponseStream() – получение потока ответа.
* Close() – закрытие потока.

*Классы HttpWebRequest и HttpWebResponse*

Данные классы предназначены для получения информации специфичной для протокола **HTTP** и наследуются от классов WebReqeust и WebResponse.

*Класс HttpWebRequest*

* TimeOut – время ожидание отклика от сервера (ожидание выполнения методов GetResponse() и GetRequestStream()). По умолчанию 100 секунд.
* KeppAlive – позволяет устанавливать постоянное подключение к серверу. В итоге для нескольких запросов к одному серверу можно использовать одно и то же подключение. По умолчанию true.
* AllowAutoRedirect – указывает, следовать ли переадресации. По умолчанию true.
* MaximumAutoRedirects – максимальное количество переадресаций.
* Credentials – представляет собой объект класса NetworkCredential, который идентифицирует пользователя.

*Класс HttpWebResponse*

* ContentLength – возвращает длину содержимого в запросе (в байтах). Является значением заголовка Content-Length, возвращаемого с ответом. Длина заголовков в этот объем не включается.
* Cookies – представляет объект класса CookieCollection и возвращает файлы куки, которые связаны с ответом.
* Headers – возвращает заголовки, связанные с данным ответом.
* LastModified – возвращает дату и время последнего изменения содержимого ответа.

**Архитектура Master-Slave**

Архитектура Master-Slave позволяет создавать устойчивые к ошибкам и надежне клиент-серверные приложения. Основные концепции этой архитектуры:

* Инициатором отправки запросов является мастер. Запросы мастера представляют из себя команды, на которые Slave должен ответить
* Мастер никогда не пошлет следующую команду, не получив ответ на предыдущую команду. Если мастер не получил ответ на свою команду, то это является поводом к разрыву соединения и попытки установить новое соединение, то есть переустановить соединение.
* Slave никогда не пошлет пакет с данніми мастеру первым. Slave посылает только ответ на команду мастера.
* Мастер является инициатором соединений. И он же является инициатором разрыва соединений. Slave никогда не будет разрывать соединения первым. С точки зрения программирования, как мастер так и Slave должны разрабатываться с учетом возможного разрыва связи в любой момент времени.

**Понятие протокола данных**

Протокол данных, кроме самих данных, содержит информацию о том, что нужно сделать с этими данными.

**UDP**

User Datagram Protocol. UDP можно сравнить с радио-вещанием: передающий, то есть вещающий сокет, ведет передачу данных по аналогии с радиостанцией и не знает, кто его слушает, а кто нет. Принимающий сокет аналогичен радиоприемнику: принимать будет только тогда, когда будет слушать. Протокол UDP не гарантирует доставку данных. Также, возможна ситуация, что пакет, отправленный позже, придет раньше. Соответственно UDP является быстрым протоколом.

*UdpClient*

* UdpClient() – создает вещающий сокет. Когда произойдет вещание, будет указано в методе Send
* UdpClient(IPEndPoint locale) – создает слушающий сокет. Принимает IP-адрес и порт, который будет прослушиваться
* Socket Client { get; set; } – вернет ссылку на класс Socket
* Close()
* Int Send(byte[] dgram, int bytes, IPEndPoint endPoint) – осуществляет отправку массива байт по адресу и порту, которые указаны в параметре endPoint
* Byte[] Receive(ref IPEndPoint remoteEP) – принимает массив байт, содержит информацию, откуда пришел пакет

**Регулярные выражения в С#**

*Введение в регулярные выражения*

Язык регулярных выражений включает в себя два средства:

1. Набор управляющих кодов для идентификации специфических типов символов
2. Система для группирования частей подстрок и промежуточных результатов как действий

С помощью регулярных выражений можно выполнять достаточно сложные и высокоуровневые действия над строками:

* Находить (и, возможно, помечать к удалению) все повторяющиеся слова в строке
* Сделать заглавными первые буквы всех слов
* Преобразовать первые буквы слов длиннее трех символов в заглавные
* Выделить различные элементы в **URI**

**Метасимволы** – специальные символы, задающие команды, а также управляющие последовательности (символы, перед которыми стоит обратный слеш \ и имеющие специальное значение).

*Метасимволы, используемые в регулярных выажениях в С#*

|  |  |  |  |
| --- | --- | --- | --- |
| **Символ** | **Значение** | **Пример** | **Соответствует** |
| Классы символов | | | |
| **[…]** | Любой из символов, указанных в скобках | **[a-z]** | В исходной строке может быть любой символ английского алфавита в нижнем регистре |
| **[^…]** | Любой из символов, не указанных в скобках | **[^0-9]** | В исходной строке могут быть любые символы кроме цифр |
| **.** | Любой символ, кроме перевода строки или другого разделителя Unicode-строки |  |  |
| **\w** | Любой текстовый символ, не являющийся пробелом, символом табуляции и т.д. (Т.е. не пустой символ) |  |  |
| **\W** | Любой символ, не являющийся текстовым символом |  |  |
| **\s** | Любой пробельный символ из набора Unicode |  |  |
| **\S** | Любой непробельный символ из набора Unicode. (\w != \S) |  |  |
| **\d** | Любые ASCII-цифры. Эквивалентно [0-9] |  |  |
| **\D** | Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9] |  |  |
| Символы повторения | | | |
| **{n, m}** | Соответствует предшествующему шаблону, повторенному не менее **n** и не более **m** раз | **s{2, 4}** | “Press”, “ssl”, “progressss” |
| **{n,}** | Соответствует предшествующему шаблону, повторенному **n** раз и более | **s{1,}** | “ssl” |
| **{n}** | Соответствует в точности **n** экземплярам предшествующего шаблона | **s{2}** | “Press”, “ssl” |
| **?** | Соответствует нулю или одному экземпляру предшествующего шаблона  предшествующий шаблон является необязательным | Эквивалентно  **s{0, 1}** |  |
| **+** | Соответствует одному или более экземпляру предшествующего шаблона | Эквивалентно  **s{1,}** |  |
| **\*** | Соответствует нулю или более экземплярам предшествующего шаблона | Эквивалентно  **s{0,}** |  |
| Символы регулярных выражений выбора | | | |
| **|** | Логическое ИЛИ. Соответствует выражению либо слева либо справа |  |  |
| **(...)** | Группирует элементы в единое целое, которое может использоваться с символами \*, +, ?, | и т.п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках. |  |  |
| **(?:…)** | Только группировка. Группирует, но не запоминает соответствующие этой группе символы. |  |  |
| Якорные символы регулярных выражений | | | |
| **^** | Соответствует началу строкового выражения или началу строки при многострочном поиске | ^Hello | “Hello, world”, но не “OK. Hello, world” т.к. в этой строке Hello находится не в начале строки |
| **$** | Соответствет концу строкового выражения или концу строки при многострочном поиске | Hello$ | “World, Hello” |
| **\b** | Соответствует границе слова, т.е. соответствует позиции между символами **\w** и символом **\W** или между символом **\w** и началом или концом строки | \b(my)\b | Из строки “Hello, my world” выберет слово “my” |
| **\B** | Соответствует позиции, не являющейся границей слов | \B(ld)\b | Найдется в слове “World”, но не в слове “ld” |

*Свойства перечисления RegexOptions*

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| *CultureInvariant* | Предписывает игнорировать национальные установки строки |
| *ExplicitCapture* | Модифицирует способ поиска соответствия, обеспечивая только буквальные совпадения |
| *IgnoreCase* | Игнорирует регистр символов во входной строке |
| *IgnorePatternWhiteSpace* | Удаляет из входной строки незащищенные управляющими символами пробелы и разрешает комментарии |
| *Multiline* | Изменяет значание **^** и **$** так, что они применяются к началу и концу каждой строки, а не к началу и концу всего входного текста |
| *RightToLeft* | Читает входную строку справа налево |
| *Singleline* | Специфицирует однострочный режим, в котором точка символизирует соответствуие любому символу |

*Класс Regex*

* Regex(String)
* Regex(String, RegexOptions)
* IsMatch(String) – возвращает логическое значение, соответствует ли входная строка заданному щаблону
* Match(String) – возвращает объект класса Match, который содержит в себе **первое** найденое соответствие из входной строки
* Matches(String) – возвращает все найденые совпадения в виде MatchCollection
* static Replace() – позволяет заменять найденые совпадения на собственные значения

*Метасимволы замены в регулярных выражениях*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Символ** | **Описание** | **Пример шаблона** | **Пример шаблона замены** | **Результат** |
| **$number** | Замещает часть строки, соответствующей группе **number** | \b(\w+)(\s)(\w+)\b | $3$2$1 | “один два” => “два один” |
| **$$** | Представляет литерал **$** | \b(\d+)\s?USD | $$$1 | “103 USD” => “$103” |
| **$&** | Замещает копией полного соответствия | (\$\*(\d\*(\.+\d)?){1}) | \*\*$& | “$1.03” => “\*\*$1.03\*\*” |
| **$`** | Замещает весь текст входной строки до соответствия | B+ | $` | “AABBCC” => “AAAACC” |
| **$’** | Замещает весь текст входной строки после соответствия | B+ | $’ | “AABBCC” => “AACCCC” |
| **$+** | Замещает последнюю захваченную группу | B+(C+) | $+ | “AABBCCDD” => “AACCDD” |
| **$\_** | Замещает всю исходную строку | B+ | $\_ | “AABBCC” => “AAAABBCCCC" |

**Почему ты решил хранить наименования данных для отправки запросов И тэги, из которых нужно парсить характеристики товара в БД, а не прописывать их прямо в коде?**

− У меня было два варианта. Первый из них – это реализация отдельного класса для каждого сайта. То есть, у меня должен быть базовый класс, например Site, а от него будут наследоваться классы для каждого сайта, например PromUaSite. И этот самый PromUaSite будет иметь свою реализацию каждого из процессов (вход на сайт, парсинг товара, добавления товара и т.д.). Соответственно каждый сайт, грубо говоря, будет иметь свой класс и в нем будут прописаны данные, которые требуются для отправки запросов и парсинга. Второй вариант – это хранить эти данные в базе данных. В таком случае таблица должна состоять из кучи таких столбцов, как идентификатор сайта, тэг идентификатора пользователя, тэг успешного входа на сайт, тэг навигации страниц, тэг кабинета пользователя, тэг наименования товара, тэг его описания и т.д. В общем, достаточно большое количество данных, которые проще прописать сразу в коде. Именно поэтому я уже было решил прописывать все эти данные в коде. Но в какой-то момент я понял, что это будет неправильно в том плане, что в случае изменения одного из требуемых тэгов, пользователю понадобиться обновлять всю программу. А это уж точно хуже, чем идти за этими данными в базу данных. Я посчитал, что такие изменения должны быть незаметны для пользователя и решил хранить эти данные в БД.