СОДЕРЖАНИЕ

Введение

В настоящее время происходит автоматизация всех областей человеческой деятельности. Так рутинную работу по учету и обработке различной информации теперь можно возложить на компьютер, который будет справляться с этой задачей в разы быстрее и качественнее человека. Однако подобные задачи давно были возложены не на одну вычислительную машину, а на их группу, соединенную посредством информационных каналов. Такие группы называются «вычислительными сетями».

Сетевое программирование является одной из центральных задач при разработке уровня бизнес-приложений – потребность в эффективном и безопасном взаимодействии разных компьютеров, находящихся в одном здании или разбросанных по всему миру, остается основной для успеха многих систем.

Программирование сетевых приложение для разнообразных вычислительных сетей одно из прогрессивнейших направлений прикладного программирования. К таким приложениям относятся службы распределенных хранилищ данных. Службы обмена данными и коммуникационные службы.

1 МЕТОДЫ И СРЕДСТВА РЕШЕНИЯ ЗАДАЧИ

1.1 Краткий обзор использованных средств

Для разработки сетевого приложения «классификация объектов на цифровом изображении» будет выбран один из объектно-ориентированных языков программирования.

Для задач тестирования сетевых каналов был использован протокол *ICMP*. Данный протокол необходимо использовать для развертывания сервера, а именно для тестирования соединения с сервером от удаленных узлов сети.

ICMP – протокол межсетевых управляющих сообщений) – сетевой протокол, входящий в стек протоколов *TCP/IP*. В основном *ICMP* используется для передачи сообщений об ошибках и других исключительных ситуациях, возникших при передаче данных, например, запрашиваемая услуга недоступна, или хост, или маршрутизатор не отвечают. Также на *ICMP* возлагаются некоторые сервисные функции.

Для непосредственного сетевого взаимодействия на сетевом уровне был использован протокол *IP*. Он служит для адресации узлов сети. Так, например, клиенты обращаются к серверу посредством адресующей связки *IP* адрес/номер порта.

IP – маршрутизируемый протокол сетевого уровня стека *TCP/IP*. Именно *IP* стал тем протоколом, который объединил отдельные компьютерные сети во всемирную сеть Интернет. Неотъемлемой частью протокола является адресация сети.

IP объединяет сегменты сети в единую сеть, обеспечивая доставку пакетов данных между любыми узлами сети через произвольное число промежуточных узлов (маршрутизаторов). Он классифицируется как протокол третьего уровня по сетевой модели *OSI*. IP не гарантирует надёжной доставки пакета до адресата – в частности, пакеты могут прийти не в том порядке, в котором были отправлены, продублироваться (приходят две копии одного пакета), оказаться повреждёнными (обычно повреждённые пакеты уничтожаются) или не прийти вовсе. Гарантию безошибочной доставки пакетов дают некоторые протоколы более высокого уровня – транспортного уровня сетевой модели *OSI,* например, *TCP*, которые используют *IP* в качестве транспорта.

Для межсетевого взаимодействия на транспортном уровне был использован *TCP*. Данный протокол был использован по той причине, что нагрузка на сеть со стороны клиента – сервера будет невысокой. Высокая пропускная способность канала необязательна. Однако, данный протокол обеспечивает высокую надежность сетевого взаимодействия.

TCP – один из основных протоколов передачи данных интернета, предназначенный для управления передачей данных. Сети и подсети, в

которых совместно используются протоколы *TCP* и *IP* называются сетями *TCP/IP*.

Механизм *TCP* предоставляет поток данных с предварительной установкой соединения, осуществляет повторный запрос данных в случае потери данных и устраняет дублирование при получении двух копий одного пакета, гарантируя тем самым, в отличие от *UDP*, целостность передаваемых данных и уведомление отправителя о результатах передачи.

Когда осуществляется передача от компьютера к компьютеру через Интернет, *TCP* работает на верхнем уровне между двумя конечными системами, например, браузером и веб-сервером. *TCP* осуществляет надежную передачу потока байтов от одной программы на некотором компьютере к другой программе на другом компьютере (например, программы для электронной почты, для обмена файлами). *TCP* контролирует длину сообщения, скорость обмена сообщениями, сетевой трафик.

1.2 Архитектура клиент-сервер

К добавлению внешнего сервера рано или поздно приходит любой сложный проект. Причины, при этом, бывают совершенно различные. Одни, загружают дополнительные сведения из сети, другие, синхронизируют данные между клиентскими устройствами, третьи- переносят логику выполнения приложения на сторону сервера. Как правило, к последним относятся большинство «деловых» приложений. По мере отхода от парадигмы «песочницы», в которой все действия выполняются только в рамках исходной системы, логика выполнения процессов переплетается, сплетается, завязывается узлами настолько, что становится трудно понять, что является исходной точкой входа в процесс приложения. В этом момент, на первое место выходит уже не функциональные свойства самого приложения, а его архитектура, и, как следствие, возможности к масштабированию.

Архитектура клиент-сервер предъявляет специфические требования, как к клиенту, так и к серверу. Программа, удовлетворяющая этим требованиям, может считаться клиент-серверным приложением, выполняющим распределенную обработку данных. Под распределенной обработкой понимается выполнение серверной частью программы запросов клиентской части. Серверная часть приложения обеспечивает хранение данных и их обработку, и отправку данных клиенту, а клиентская часть передает серверу соответствующие запросы.

К преимуществам клиент-серверных систем можно отнести:

– клиент-серверный подход – модульный, причем серверные программные компоненты компактны и автономны;

– поскольку каждый компонент выполняется в отдельном защищенном процессе пользовательского режима, сбой сервера не повлияет на остальные компоненты операционной системы;

– автономность компонентов делает возможным их выполнение на нескольких процессорах на одном компьютере (симметричная многопроцессорная обработка) или на нескольких компьютерах сети (распределенные вычисления);

– обязанность клиента, как правило, – предоставлять пользовательские сервисы и, прежде всего, пользовательский интерфейс, то есть средства для приема, отображения и редактирования данных, введенных пользователем, которые служат основой для запроса серверу. Кроме того, клиент можно настроить на обработку части данных, чтобы уменьшить нагрузку на ресурсы сервера.

Недостатками являются:

– неработоспособность сервера может сделать неработоспособной всю вычислительную сеть. Неработоспособным сервером следует считать сервер, производительности которого не хватает на обслуживание всех клиентов, а также сервер, находящийся на ремонте, профилактике и т.п.;

– поддержка работы данной системы требует отдельного специалиста системного администратора;

– высокая стоимость оборудования.

В любой сети, построенной на современных сетевых технологиях, присутствуют элементы клиент-серверного взаимодействия, чаще всего на основе двухзвенной архитектуры. Двухзвенной она называется из-за необходимости распределения трех базовых компонентов между двумя узлами (клиентом и сервером). На рисунке 1.1 предстален пример двухзвенной архитектуры.

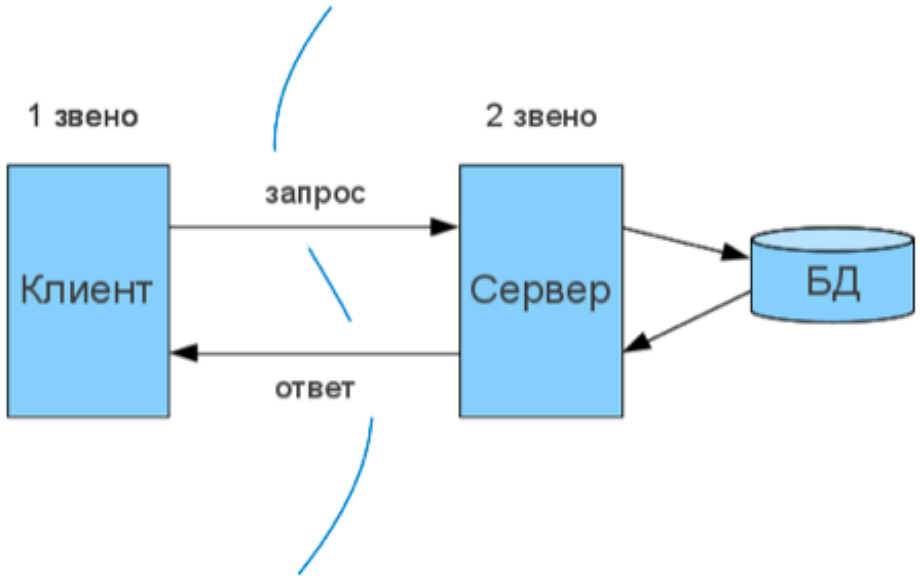

Рисунок 1.1 – двухзвенная клиент-серверная архитектура

Двухзвенная архитектура используется в клиент-серверных системах, где сервер отвечает на клиентские запросы напрямую и в полном объеме, при этом используя только собственные ресурсы. Т.е. сервер не вызывает сторонние сетевые приложения и не обращается к сторонним ресурсам для выполнения какой-либо части запроса.

Еще одна тенденция в клиент-серверных технологиях связана с все большим использованием распределенных вычислений. Они реализуются на основе модели сервера приложений, где сетевое приложение разделено на две и более частей, каждая из которых может выполняться на отдельном компьютере. Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее согласованном формате. В этом случае двухзвенная клиент-серверная архитектура становится трехзвенной. Как правило, третьим звеном в трехзвенной архитектуре становится сервер приложений. Пример трехзвенной архитектуры представлен на рисунке 1.2.

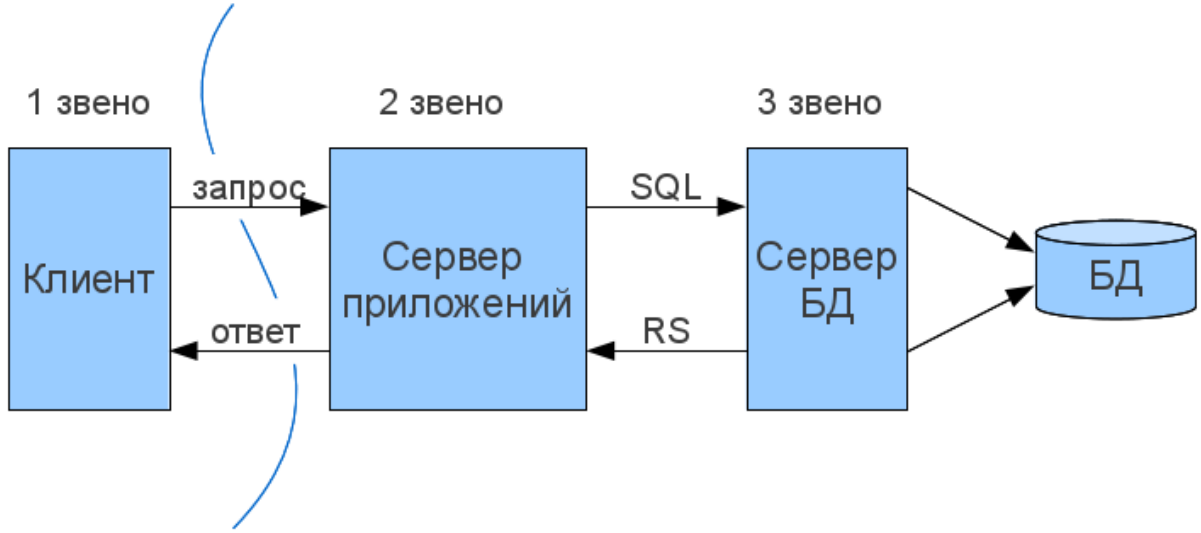

Рисунок 1.2 – трехзвенная клиент-серверная архитектура

Трехзвенная архитектура может быть расширена до многозвенной путем выделения дополнительных серверов, каждый из которых будет представлять собственные сервисы и пользоваться услугами прочих серверов разного уровня.

Со стороны клиента нет никакой разницы между двухзвенной и трезвенной архитектурой. Здесь важно понимание двух вещей:

– может быть множество клиентов, использующих один аккаунт для общения с севером;

– каждый клиент, как правило, имеет свое собственное локальное хранилище.

В данной работе была использована двухзвенная архитектура для решения поставленной задачи. Однако следует отметить, то, что в данном приложении небыли использованы системы управления базами данных, так

как в данном случае их использование было бы избыточным. Хоть и допустимым.

В ряде случаев, локальное хранилище может быть синхронизировано с облаком, и, соответственно, с каждым из клиентов.

Следует отметить, что поскольку, некоторые разработчики стремятся избавиться от «серверной части» некоторые приложения построены вокруг синхронизации их хранилищ в «облаке». Т. е. фактически, имеют так же, двухзвенную систему, но с переносом архитектуры её развертывания на уровень операционной системы. В некоторых случаях такая структура оправдана, но такая система не так легко масштабируется, и её возможности весьма ограничены.

На самом примитивном уровне абстракции приложение, ориентированное на работу с сервером, состоит из следующих архитектурных слоев:

- ядро приложения;
- графический пользователь интерфейс;
- компоненты повторного использования;
- файлы окружения;
- ресурсы приложения.

1.3 Сокеты. Протоколы TCP и UDP

Класс *Socket* обеспечивает широкий набор методов и свойств для сетевых взаимодействий. Класс *Socket* позволяет выполнять как синхронную, так и асинхронную передачу данных с использованием любого из коммуникационных протоколов, имеющихся в перечислении *ProtocolType* [4].

Класс *Socket* придерживается шаблона имен платформы *.NET Framework* для асинхронных методов. Например, синхронный метод *Receive* соответствует асинхронным методам *BeginReceive* и *EndReceive*.

Если приложению при его исполнении требуется только один поток, воспользуйтесь приведенными ниже методами, которые разработаны для работы в синхронном режиме:

– если используется протокол, ориентированный на установление соединения, такой как протокол *TCP*, сервер должен выполнять прослушивание подключений, используя метод *Listen*. Метод *Accept* обрабатывает любые входящие запросы на подключение и возвращает объект *Socket*, который может использоваться для передачи данных с удаленного узла. Используйте этот возвращенный объект *Socket* для вызова метода *Send* или *Receive*. Вызовите метод *Bind*, прежде чем производить обращение к методу *Listen*, если необходимо указать локальный *IP*-адрес или номер порта. Используйте нулевое значение для номера порта, если требуется, чтобы свободный порт был назначен основным поставщиком услуг. Если требуется произвести подключение к прослушивающему узлу, вызовите метод *Connect*. Для обмена данными вызовите метод *Send* или *Receive*;

– если используется протокол, не ориентированный на установление соединения, такой как протокол *UDP*, нет необходимости в отслеживании подключений. Для приема всех поступающих дейтаграмм вызовите метод *ReceiveFrom*. Для посылки дейтаграмм на удаленный узел воспользуйтесь методом *SendTo*.

Логику работы «клиент-сервер» можно увидеть на рисунке 1.3.

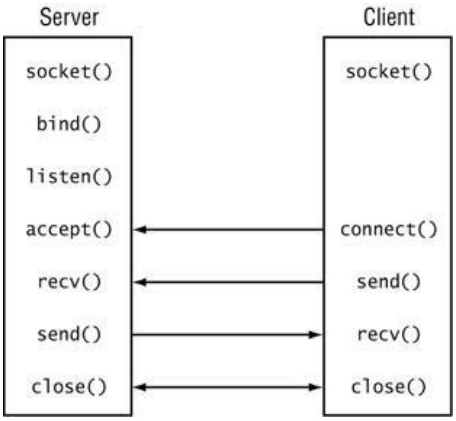

Рисунок 1.3 – Логика работы «клиент-сервер»

Чтобы выполнить передачи с использованием отдельных потоков во время исполнения, воспользуйтесь следующими методами, предложенными для работы в асинхронном режиме:

– если применяется протокол, ориентированный на установление соединения, такой как протокол *TCP*, используйте методы *Socket*, *BeginConnect* и *EndConnect* для подключения к прослушивающему узлу. Для асинхронного обмена данными воспользуйтесь методами *BeginSend* и *EndSend* или методами *BeginReceive* и *EndReceive*. Входящие запросы на подключение могут быть обработаны с помощью методов *BeginAccept* и *EndAccept*;

– если используется протокол без установления соединения, такой как протокол *UDP*, можно воспользоваться для посылки дейтаграмм методами *BeginSendTo* и *EndSendTo*, а для получения дейтаграмм можно применить методы *BeginReceiveFrom* и *EndReceiveFrom*.

Если на сокете выполняется несколько асинхронных операций, они не обязательно должны завершаться в том же порядке, в котором эти операции запускаются.

Когда прием и отправка данных завершены, используйте метод *Shutdown* для того, чтобы отключить объект *Socket*. После вызова метода *Shutdown* обратитесь к методу *Close*, чтобы освободить все связанные с объектом *Socket* ресурсы.

Класс *Socket* позволяет выполнить настройку объекта *Socket* с использованием метода *SetSocketOption*. Извлеките эти параметры, используя метод *GetSocketOption*.

TCP (*Transmission Control Protocol*) – обмен данными, ориентированный на соединения, может использовать надежную связь, для

обеспечения которой протокол уровня 4 посылает подтверждения о получении данных и запрашивает повторную передачу, если данные не получены или искажены. Протокол *TCP* использует именно такую надежную связь. *TCP* используется в таких прикладных протоколах, как *HTTP, FTP, SMTP* и *Telnet* [5].

Протокол *TCP* требует, чтобы перед отправкой сообщения было открыто соединение. Серверное приложение должно выполнить так называемое «пассивное открытие (*passive open*)», чтобы создать соединение с известным номером порта, и, вместо того чтобы отправлять вызов в сеть, сервер переходит в ожидание поступления входящих запросов. Клиентское приложение должно выполнить «активное открытие (*active open*)», отправив серверному приложению, синхронизирующий порядковый номер (*SYN*), идентифицирующий соединение. Клиентское приложение может использовать динамический номер порта в качестве локального порта.

Сервер должен отправить клиенту подтверждение (*ACK*) вместе с порядковым номером (*SYN*) сервера. В свою очередь клиент отвечает *ACK*, и соединение устанавливается.

После этого может начаться процесс отправки и получения сообщений. При получении сообщения в ответ всегда отправляется сообщение *ACK*. Если до получения *ACK* отправителем истекает тайм-аут, сообщение помещается в очередь на повторную передачу.

TCP – это сложный, требующий больших затрат времени протокол, что объясняется его механизмом установления соединения, но он берет на себя заботу о гарантированной доставке пакетов, избавляя нас от необходимости включать эту функциональную возможность в прикладной протокол.

Протокол *TCP* имеет встроенную возможность надежной доставки. Если сообщение не отправлено корректно, мы получим сообщение об ошибке.

Логику работы протокола *TCP* можно увидеть на рисунке 1.4.

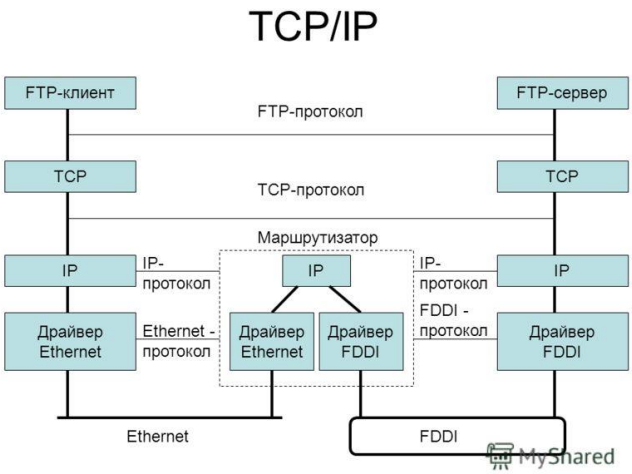

Рисунок 1.4 – Логика работы протокола *TCP*

В отличие от *TCP, UDP* – очень быстрый протокол, поскольку в нем определен самый минимальный механизм, необходимый для передачи данных. Конечно, он имеет некоторые недостатки. Сообщения поступают в любом порядке, и то, которое отправлено первым, может быть получено последним. Доставка сообщений *UDP* вовсе не гарантируется, сообщение может потеряться, и могут быть получены две копии одного и того же сообщения. Последний случай возникает, если для отправки сообщений в один адрес использовать два разных маршрута [5].

UDP не требует открывать соединение, и данные могут быть отправлены сразу же, как только они подготовлены. *UDP* не отправляет подтверждающие сообщения, поэтому данные могут быть получены или потеряны. Если при использовании *UDP* требуется надежная передача данных, ее следует реализовать в протоколе более высокого уровня.

Логику работы протокола *UDP* можно увидеть на рисунке 1.5.

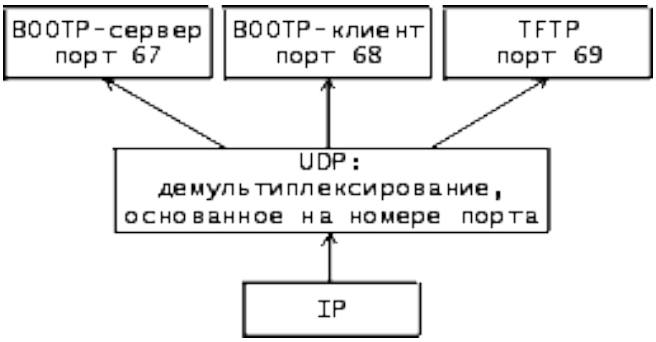

Рисунок 1.5 – Логика работы протокола *UDP*

Так в чем же преимущества *UDP*, зачем может понадобиться такой ненадежный протокол? Чтобы понять причину использования *UDP*, нужно различать однонаправленную передачу, широковещательную передачу и групповую рассылку.

Однонаправленное (*unicast*) сообщение отправляется из одного узла только в один другой узел. Это также называется связью «точка-точка». Протокол *TCP* поддерживает лишь однонаправленную связь. Если серверу нужно с помощью *TCP* взаимодействовать с несколькими клиентами, каждый клиент должен установить соединение, поскольку сообщения могут отправляться только одиночным узлам.

Широковещательная передача (*broadcast*) означает, что сообщение отправляется всем узлам сети.

Групповая рассылка (*multicast*) – это промежуточный механизм: сообщения отправляются выбранным группам узлов.

UDP может использоваться для однонаправленной связи, если требуется быстрая передача, например, для доставки мультимедийных данных, но главные преимущества *UDP* касаются широковещательной передачи и групповой рассылки. *UDP* не требует открывать соединение, и

данные могут быть отправлены сразу же, как только они подготовлены. *UDP* не отправляет подтверждающие сообщения, поэтому данные могут быть получены или потеряны. В отличие от *TCP, UDP* – очень быстрый протокол, поскольку в нем определен самый минимальный механизм, необходимый для передачи данных.

Обычно, когда мы отправляем широковещательные или групповые сообщения, не нужно получать подтверждения из каждого узла, поскольку тогда сервер будет наводнен подтверждениями, а загрузка сети возрастет слишком сильно. Примером широковещательной передачи является служба времени. Сервер времени отправляет широковещательное сообщение, содержащее текущее время, и любой хост, если пожелает, может синхронизировать свое время со временем из широковещательного сообщения.

UDP – это быстрый протокол, не гарантирующий доставки. Если требуется поддержание порядка сообщений и надежная доставка, нужно использовать *TCP* [5].

В ходе анализа первой главы было принято решение использовать клиент-серверную технологию, сокеты и протокол *TCP*. Так как использование *TCP* предоставляет надежную передачу в ущерб скорости. Однако, с учётом специфики разрабатываемого приложения высокая скорость передачи данных не является существенным фактором, определяющий работоспособность клиент-сервера.